DICE Firmware Development Environment

User Guide

For Version 4.0.x

For the latest version of this and other documents, please visit the TCAT Subversion Repository.

Doc Rev 2.1
11/5/2012

CONTENTS

1. Overview

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

The Firmware Development Environment 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, Host Software and Utilities, 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.

The source code for firmware development is available via online revision control system. Contact TCAT for a login account to the server.

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.

Some familiarity with Linux/Unix or Windows cross-development for Linux embedded targets and knowledge of 1394 technologies is assumed. A deep level of Unix experience is not essential.

The Developer should have DICE target hardware available such as the DICE EVM003.

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

3. Document Roadmap

Evaluating the DICE and Tools

The *DICE Firmware Development Environment Installation Guide* is a useful guide for understanding the Firmware 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 *TCD22xx User Guide* (for DICE Jr and DICE Mini) 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 Panels and other GUI Tools and Applications.

Project Managers and Developers can review the *DICE Development Steps Detailed Checklist* document in your Subversion tag or in the public documents area of the subversion server.

https://dev.tctechnologies.tc/tcat/tags/release/public/latest/docs/dice/developmentChecklist.pdf

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 DICE Firmware SDK should consult the *DICE Firmware Migration Guide* for information about the differences between these two major versions, and for details on how use the newest sources in an existing installation.

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 that implements the communications and audio configuration software on a DICE-based device.

Notation

Notes

> Information of particular importance is highlighted in this way, including the parts of the source code that are most relevant to the developer.

Paths

The Firmware Development Environment 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 Environment 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/myProject/make
$ ls
```

CLI prompts

RedBoot CLI prompts are indicated with:

```
RedBoot> version
```

Firmware Application prompts are shown as follows:

```
> splash
```

5. News

Components

This version of the DICE Firmware Development Environment contains an updated Cygwin environment, which runs under all versions of Windows (32-bit and 64-bit) from Windows XP through Windows 7 and new packages such as Python, Subversion, nano, etc.

Note that when using Subversion GUI programs such as Tortoise SVN, the file permissions that are created may not be compatible with the Cygwin environment. It is recommended that developers use the included command-line version of svn to check out the development sources from the TCAT repository.

Also included is version 0.4.0 of the OpenOCD JTAG debug server. This version does not work under 64-bit versions of Windows, since OpenOCD does not support this at the time. 64-bit compatible versions are under development however, and you can look online for updates (see the Resources section).

Aside from new packages and greater platform compatibility, the new installer creates an equivalent environment to the previous (3.x) version, and it will back up your Cygwin home directory. You can restore any previous settings files from your previous home directory by copying them into the new home directory that was created during the installation process.

TCAT Subversion Repository

The latest documents, Firmware source code, Host development source code, EVM binaries, and Drivers can be found online.

The repository can be browsed with a web browser, and developers should use the included command-line svn client to work with the sources.

For evaluators who do not have an account yet for the TCAT repository, the latest binaries and documents, including information about using the repository, can be found in the public area at:

https://dev.tctechnologies.tc/tcat/tags/release/public/latest

Developers will be given a password protected TAG URL to access other areas of the repository. Contact TCAT to obtain a login that grants access to the appropriate development sources.

Prebuilt Firmware Binaries

These are no longer included in the Installer. The latest binaries can always be found in your Subversion tag, and in the public area of the TCAT repository.

6. Checklist

Evaluating the DICE chip, tools and software

- ☐ Install the DICE Firmware Development Environment. Please consult the *DICE Firmware Development Environment Installation Guide* first for useful information.
- ☐ EVM Users, consult the EVM User Guide for your board for information about preparing the board for use.
- ☐ Experiment with the CLI commands, Host interfacing, and experiment with the tool-chain and source code. See the *DICE Firmware CLI Reference* for details.
- ☐ 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
- ☐ Checkout your tag from the TCAT Subversion Repository
- ☐ Build the EVM project(s).
- ☐ 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.

Application Developers

- ☐ 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
- ☐ Your new project directory can be kept in your in-house version control without being disturbed by updates from the TCAT repository, however if you make changes outside your project directory, it may be a good idea to keep a separate local tree, and merge in any changes from TCAT into your local tree.

Detailed Checklist

- ☐ Project Managers and Developers can review the *DICE Development Steps Detailed Checklist* document in your Subversion tag or in the public documents area of the subversion server.

  https://dev.tctechnologies.tc/tcat/tags/release/public/latest/docs/dice/developmentChecklist.pdf

7. File structure Orientation

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

The DICE Firmware Development Environment 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 (supplied separately) 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** and **/interface** directories.

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, you can download and use GNU Windows Emacs, and the installer includes 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 Development Environment's Cygwin distribution to do so. However, we have provided an xterm-like shell and a full emacs installation here, so it is not likely that you'll need CygwinX. See the *DICE Firmware Development Environment Installation Guide* for instructions.

Additionally, users who are more used to using IDE's such as Visual Studio for their embedded development front-end (excluding debugging) can find Visual C++ project files in the example firmware projects. 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 GNU Windows **Emacs** and the **Insight** graphical front-end for gdb. These all have support for serial debugging of DICE targets. JTAG debugging is also supported (and recommended) in all of the above gdb options.

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.

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, and is recommended.

A test utility that runs entirely in on-chip RAM is provided to help verify your custom hardware. This utility reads and writes to the memory address/data bus and performs comprehensive exercising of external RAM.

PC/Mac Drivers

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

Driver support for Windows and Macintosh computers is evolving rapidly. The latest drivers and installers will always be available in your Subversion tag, and you can always 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 a type of Host driver is referred to as the Driver Model. The driver model is selected by making defines in the build system. The default driver model is the DiceDriver model, which also includes compatible Host drivers, cross-platform driver interface libraries, utilities and debugging tools.

8. Application Development Quickstart

Overview

This section gives an introduction to development with the Firmware sources. The Developer will create a new project based on the Template, and make changes mainly in the new project directory. Visit the TCAT Subversion repository for the latest documents.

Start Here

This document covers the 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.

TCAT Subversion repository

Developers will have their own methods of course, but it is recommended that you *checkout* the entire svn tag from the TCAT repository and also *export* a duplicate local working copy somewhere. This way your svn checkout will have the subversion history, and your exported duplicate will not, so you can import it into your own local revision control. When TCAT make a new release, you can *update* your local svn tag checkout, and then *diff* the new revision with your local exported directory, which allows you to *merge* any changes from TCAT into your working copy.

Firmware Projects

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 a template using the shell scripts provided, such as

/firmware/project/new_tcd22x0_EVM003_proj.sh, run '**make install**,'

then start your edit/compile/debug routine. You'll typically run '**make dep**' from then on, unless you change dependencies or add new files. See *The Build System* section for details.

If you have Visual Studio 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 projects in **/firmware/project/** which can be created with the **install.sh** script provide a lot of examples that can be brought into your custom firmware application as needed.

Main Areas of Customization

The majority of the changes necessary fall into only 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**

When you (the vendor) are ready to start verifying production firmware and drivers, your vendor information, such as strings and 1394 vendor_id, can be added to the build tree. These changes are made in the top-level **interface** directory, in **tcat_dice_myproduct.h.** At that time, you will be given a custom Subversion tag in the TCAT repository, which will also contain any other customizations (logo's, icons, Control Panel preferences, etc.) that you wish to include. This custom tag will always include the driver binaries (and installers) that are compatible with your custom vendor information.

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**. By default, 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.

A number of example audio configurations, i.e. different combinations of audio inputs and outputs, are provided using the **myMode** functions. Production firmware applications will not usually contain these modes, as they are there to provide examples of various streaming configurations.

Communication with Inter IC peripherals

The **i2c** module provides an API for communicating with **i2c** peripherals such as nonvolatile storage, CODEC's, etc. The DICE EVM projects include examples for reading and writing the EPROM, and show an examples for configuration of CODEC's.

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.

SPI interface

The **spi** module provides an API for communicating with external peripherals such as CPLD's, CODEC's, etc. The DICE-JR and DICE-MINI have hardware SPI, and the DICE has a software SPI bitbang implementation.

Other Areas of Interest

SPS

The Simple Persistent System module provides an API for storing run-time settings in a flash file so values can persist across device reset.

Gray rotary encoder interface

The DICEII has four, and JICE JR has two, and DICE Mini has one Gray encoder interfaces built-in. The **EVM003** project 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.

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 1394 audio streaming and routing within the DICE is implemented in hardware, the ARM doesn't have to directly process audio data, and has a great deal of compute cycles available. However the ARM can optionally be used for example to mix or otherwise process 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.

Note that the DICE Jr and DICE Mini have a built-in hardware mixer and peak detector, so the mixer8 module can be used as an example for moving audio into and out of the ARM processor for any purpose.

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 **dicedriver** module shows 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 EVM's, 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 sources 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.

9. Directory Structure

The Cygwin "root" directory contains the entire environment 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 environment 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 environment in the default **c:\cygwin** location, you will see the directory structure below on disk. For the most part, only the **/home** and **/etc** directories are relevant.

/cygdrive/c/cygwin	
|-- bin	binaries installed from the Cygwin packages database 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.
|-- cygwin_mirror	the Cygwin packages database
|-- dev	unix style dev directory
|-- docs	documents linked to the Start Menu
|-- etc	**unix style etc directory** 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.
|-- gnuarm	the GNU ARM compiler, binutils, etc. 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. These tools will be used if they are selected in the **Make.params** file of your firmware project.
|-- gnutools	the ARM EABI compiler, binutils, etc. Contains the ARM EABI tools, including binutils, ARM compiler, assembler, linker, debugger. Look in the **/guntools/bin** directory for useful tools for manipulating binary files. These tools will be used if they are selected in the **Make.params** file of your firmware project.
|-- home	**contains user home directories** 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.
|-- inst	backup installation files
|-- lib	Cygwin lib directory
|-- tmp	Cygwin tmp directory
|-- uninst	Uninstaller linked to the Start Menu
|-- usr	Cygwin usr directory
`-- var	Cygwin var directory

DICE Firmware directory

DICE Firmware is all contained in the **/firmware** directory of your root checkout directory, along with the top-level **/interface** directory which includes files that are included by both firmware and Host software.

All firmware can be built in place in this directory tree. There is no need to copy any header files, libraries, or tools binaries to other locations. This is also true for the Host software (drivers, Control Panel, utilities, debug tools). Host software is discussed elsewhere.

The layouts shown below may change over time as additional functionality is added to the sources.

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. Using this, development of multiple application projects at the same time is straight-forward. Most of the files that are changed for your application code are contained in a separate project folder for each application. Also, separate kernel libraries are maintained for each project. Project templates are provided for easily starting new firmware applications.

In most cases, developers will be working with files in the main and target directories in the project directory.

When you use search tools within your favorite editor or from a command line 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. These directories contain a great deal of files that are not relevant to day-to-day development.

Locating /firmware and /interface

The **/firmware** directory is self-contained, as long as the global **/interface** directory is at the same directory level, since the build system finds it relative to the firmware sources. This **/interface** directory is also common to the Host software sources, which are also included in your checkout tag.

When you checkout or export your svn tag, the entire directory tree is self-contained and firmware and Host sources will build in place.

In all cases, you should have a top level directory which includes your entire checked-out tag. This top-level directory may be located anywhere on your local machine.

These can be located on another path on the local workstation, on a network share, etc.

See the section below *Software Updates* for more info.

Directory Structure Overview

```
/interface              the global headers, used for firmware and host development
/firmware                       firmware tree, self-contained (along with /interface)
|-- chip                        chip related modules (rarely changed by developer)
|   |-- dice2                   modules specific to DICEII
|   |   |-- avs                 Audio Video System module
|   |   |   |-- interface       all modules have this headers directory
|   |   |   `-- src             all modules have this implementation files directory
|   |   |-- dal                 DICE Abstraction Layer module
|   |   |-- dice                handles operation of the DICE audio interfaces
|   |   |-- gray                Gray coder/decoder
|   |   |-- i2c                 LLC interface
|   |   |-- internal_ram_test   memory test utility, generated by make
|   |   |-- llc                 1394 Link Layer Controller module
|   |   `-- spi                 Serial Parallel Interface
|   `-- diceJr                  modules specific to DICE-JR and DICE-MINI
|       |-- avs
|       |-- dal
|       |-- dice
|       |-- eap                 Enhanced Application configuration Protocol
|       |-- gray
|       |-- i2c
|       |-- internal_ram_test
|       |-- llc
|       `-- spi
|-- interface                   firmware-specific global headers
|-- kernel                      kernel configuration and build destination
|   |-- build                   kernel sources are copied and built here
|   |   |-- ecos_diceJr_evm003_tcd22x0      an example kernel (see below)
|   |   |   |-- devs
|   |   |   |-- error
|   |   |   |-- hal
|   |   |   |-- infra
|   |   |   |-- install
|   |   |   |-- io
|   |   |   |-- isoinfra
|   |   |   |-- kernel
|   |   |   |-- language
|   |   |   `-- services
|   `-- configurations          eCos kernel configurations used during kernel build
|-- module                      chip-independent modules
|   |-- 1394avc                 Audio Video Control protocol implementation
|   |   |-- interface           all modules have this module-specific headers directory
|   |   `-- src                 all modules have this implementation files directory
|   |-- 1394lal                 Link Abstraction Layer
|   |-- aml                     Abstract MIDI Layer
|   |-- cli                     Command Line Interface
|   |-- cms                     Connection Management System
|   |-- crb                     Configuration ROM Builder
|   |-- dicedriver              Implements the DiceDriver protocol
```

```
|   |-- fcp                 Function Control Protocol
|   |-- fis                 Flash file system access
|   |-- frmwload            Firmware Loader protocol implementation
|   |-- misc                low level utilities
|   `-- sps                 Simple Persistent Storage
|-- os                      Operating systems directory
|   `-- ecos                eCos file
|       |-- TCKernel        TCAT eCos abstraction
|       |   |-- interface
|       |   `-- src
|       `-- src             eCos distribution
|           |-- acsupport
|           |-- examples
|           |-- packages    the eCos sources
|           `-- tools       Host programs used to configure/build kernel libraries
`-- project
    |-- AudioMonitor        Firmware implementation for the original Audio Monitor
    |   |-- doxygen         HTML docs generated from source code
    |   |-- main            project-specific implementation files
    |   |-- make            make-related files and Visual Studio project file
    |   |-- target          hardware-specific implementation files
    |   |   |-- interface
    |   |   `-- src
    |   `-- test            various utilities
    |       |-- interface
    |       `-- src
    |-- evm003_tcd22x0      example project directory (see below)
    |   |-- bin             firmware executable will be built here
    |   |-- doxygen         HTML documentation generated by make
    |   |-- lib             module libraries
    |   |-- main            project-specific implementation files
    |   |-- make            make-related files and Visual Studio project file
    |   |-- obj             intermediate object files
    |   |-- target          hardware-specific implementation files
    |   |   |-- interface   target (hardware-specific) header files
    |   |   `-- src         target (hardware-specific) implementation files
    |   `-- test            various utilities
    |       |-- interface
    |       `-- src
    |-- template            example project templates
    |   |-- avc_tcd22x0     Implements an AV/C device on EVM002 with DICE
    |                       JR/MINI microboard
    |   |   |-- doxygen
    |   |   |-- main
    |   |   |-- make
    |   |   |-- target
    |   |   `-- test
    |   |-- cms_diceII_EVM002       Implements a CMS device on EVM002 with
    |                       DICEII microboard
    |   |-- cms_tcd22x0     Implements a CMS device on EVM002 with DICE
    |                       JR/MINI microboard
    |   |-- diceII          Implements a DiceDriver device on classic DICEII EVM
```

```
|   |   |-- axm20
|   |   |-- dalRemote
|   |   |-- doxygen
|   |   |-- dsp
|   |   |-- main
|   |   |-- make
|   |   |-- mixer8
|   |   |-- target
|   |   `-- test
|   |-- diceII_EVM002          Implements a DiceDriver device on EVM002 with
|                              a DICEII microboard
|   |-- eap_tcd22x0            Implements an EAP device on EVM002 with a
|                              DICE JR/MINI microboard
|   |-- eap_tcd22x0_EVM003     Implements an EAP device on EVM003
|   |-- no1394_tcd22x0         Audio-only (no 1394 interface) implementation
|   |-- tcd22x0                Implements a DiceDriver device on EVM002 with
|                              A DICE JR/MINI microboard
|   `-- tcd22x0_EVM003         Implements a DiceDriver device on EVM003
```

Project Directories

When a new project is created from a template from a bash shell, the project name is used as the directory name, and the template files are copied into it, and the relevant references are changed to the project name within the copied files.

In the directory structure above in bold type, a project was created from the source tree in /firmware/project like so:
$ ./new_tcd22x0_EVM003.sh evm003_tcd22x0

Building a firmware application from there is a two-step process. The makefile for the project is in the make directory in the project directory
/firmware/project/evm003_tcd22x0/make

From there, you will initially run make install in a bash shell
$ make install

This will create a kernel build directory in
/firmware/kernel/build/ecos_diceJr_evm003_tcd22x0

The directory name is a concatenation of the OS, CHIP variables in the Make.params file from the make directory and the project name. The kernel will be built as a library. You will normally not need to look at the kernel files, but the directory must be deleted before rebuilding the kernel. This is done for you in the ‘make install’ make target.

Then, the firmware application is built and linked to the library. Once the kernel library is built, you will not have to rebuild it unless you make a change to the kernel. To build the application any time after that, run make dep
$ make dep

from within the project make directory.

Using Visual Studio as IDE

Also note that inside the project make directory are Visual Studio solution and project files. The project files allow you to edit, browse and search the source. You can also build from within Visual Studio rather than a bash shell, if you prefer. The project settings will call the appropriate ARM compiler for you.

In that case, for the first build use Build->Rebuild <projectname>, which is equivalent to 'make install' and thereafter run Build->Build <projectname> which is equivalent to 'make dep' For more details, see *The Build System* section.

Output from the build will appear in the output window, and any errors or warnings can be clicked to go to the related file and line.

10. 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. In General, nothing gin these directories should normally be modified by the Developer. 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.

Global headers

/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
/firmware/project/template
Initially contains DICE EVM template projects. The initial installation script will create and build two projects from these templates which run on the DICE EVM's. This directory also contains Customer projects which are initially created from the templates. This allows concurrent development of multiple Applications.

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

For example, to create a new project and build it (including its kernel library) for a DICE-Jr application, the developer can use the supplied shell script.

```
$ ./new_tcd22x0_EVM003_proj.sh myProduct
$ cd myPoduct/make
$ make install
```

Chip-specific Directories

Currently, the chip directory contains implementation of modules that depend on the chip configurations for DICEII, DICE-JR and DICE-MINI. The **dice2** chip directory implements DICEII, and the **diceJr** chip directory implements DICE-JR and DICE-MINI and is functionally equivalent to the dice2. Nothing in these directories should normally be modified by the developer.

/firmware/chip/diceJr/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/diceJr/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/diceJr/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/diceJr/eap
Supports the Enhanced Application Protocol. This protocol is used to control the on-chip hardware mixer and peak detector.

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

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

/firmware/chip/diceJr/internal_ram_test
Contains a minimal application that runs entirely in the on-chip RAM in the DICE chip. This is useful for initial testing of new hardware. This is not built by the install script, but can be made at any time using **make in_ram_test_jr** in the project's **make** directory. This is detailed later in this document.

/firmware/chip/diceJr/llc
Driver for the Samsung 1394 link layer controller in DICE. 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/diceJr/spi
This exists only in the **diceJr** directory, and is for DICE-JR and DICE-MINI only. This is the serial-parallel interface implementation. Files in this directory will not normally be modified by the developer.

Kernel Directories

/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:

diceJrkernel.ecc – For DICE-JR and DICE-MINI kernel libraries
kernel.ecc – For DICE-II kernel libraries
redboot_ram.ecc – For 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
internal_ram_test.ecc – For creating the on-chip SRAM resident memory tests

/firmware/kernel/build
The kernel files produced by ecosconfig during build for DICE EVM's and similar hardware.

When the install target is built for a project, a directory will be created here which contains the sources needed to build the kernel. This allows multiple concurrent products to be developed where there is the possibility that the kernels are different for each product.

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

In general, nothing in these directories should normally be modified by the developer.

/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/cms
Implements the dicedriver Connection Management System protocol. This allows devices to be configured to stream to each other when a Host driver is not present on the 1394 bus.

/firmware/module/crb
Implements the Configuration ROM Builder, which simplifies the creation of the 1394 Configuration ROM which allows Host drivers to discover information about the DICE device.

/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. This is used by the AV/C driver model.

/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 application CLI prompt.

/firmware/module/frmwload
Files in this directory support loading of flash files from a host computer over the 1394 bus.

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.

/firmware/module/sps
Simple Persistent Storage implementation. Provides flash memory management of run-time settings which must survive across device resets.

Operating System Directories

Files in these directories will not normally be modified by the developer.

/firmware/os/ecos/src
This is the official eCos distribution with changes and additions for the DICE chip family.

/firmware/ecos/TCKernel
Implements the operating system abstraction layer. This provides a TC Applied Technologies defined kernel API, which wraps eCos kernel functions.

Project Directories

/firmware/project
This is where most of the customizations to the source code are done for custom applications. This directory contains shell scripts for creating and building projects based on the DICE EVM's, which can be modified as needed for custom designs.

/firmware/project/AudioMonitor
This project implements a firmware application for the original AudioMonitor board. This project is not intended for use with the Audio Monitor II board, which is now referred to as the EVM003.

The Template Projects

/firmware/project/template
The projects in this directory are not meant to be used or modified by the developer. They are starting points for creating new projects for starting custom product implementations. As improvements are made by TCAT to the

templates, these will be made available as updates and so any changes to these templates could be overwritten.

Developers who wish to create their own custom templates can make them here if desired, and then look at source differences in the updates from TCAT to determine if the changes should be added to the custom templates.

/firmware/project/template/diceII
This template is used when starting with applications based on the DICEII EVM development board, or similar hardware.

/firmware/project/template/diceII_EVM002
This template is used when starting with applications based on the EVM002 development board with a DICEII microboard, or similar hardware.

/firmware/project/template/tcd22x0
This template is used when starting with the EVM002 development board with either a DICE-JR or DICE-MINI microboard, or similar hardware.

/firmware/project/template/tcd22x0_EVM003
This template is used when starting with the EVM003. The EVM003 is also known as the Audio Monitor II, which is not to be confused with the original Audio Monitor board.

/firmware/project/template/no1394_tcd22x0
This template is used to create an application that does not use the 1394 stack. This is usually a starting point for firmware that only uses the audio i/o and routing in the DICE chip.

/firmware/project/template/cms_tcd22x0
This template is used when creating CMS-based firmware, starting with the EVM002 development board with either a DICE-JR or DICE-MINI microboard, or similar hardware.
See the CMS documents for more information.

/firmware/project/template/cms_tcd22x0
This template is used when creating CMS-based firmware, starting with the EVM002 development board with a DICEII microboard, or similar hardware.
See the CMS documents for more information.

/firmware/project/template/eap_tcd22x0
This template is used when creating EAP-based firmware, starting with the EVM002 development board with either a DICE-JR or DICE-MINI microboard, or similar hardware.
See the EAP documents for more information.

/firmware/project/template/eap_tcd22x0_EVM003
This template is used when creating EAP-based firmware, starting with the EVM003 development board, or similar hardware.
See the EAP documents for more information.

Other templates may appear over time. See the section *Using Project Templates* below for more information.

Template Project Directories

The directory structure is the same for the templates (where the classic DICEII EVM template has a few more modules).

/firmware/project/template/<template_name>
A starting point for new firmware applications. Use the included shell script to create your new projects. See the section *Project Template* for details.

/firmware/project/template/<template_name>/axm20 (deprecated)
Files that support the AXM20 i/o expander board, when using classic DICE II EVM.

/firmware/project/template/<template_name>/dalRemote (deprecated)
Remote Host calls to the DAL firmware module (no longer supported).

/firmware/project/template/<template_name>/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/template/<template_name>/dsp (deprecated)
Code for the Motorola DSP chip included on the classic DICEII EVM.

/firmware/project/template/<template_name>/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/template/<template_name>/make
Makefiles, batch files, and Visual Studio project files

/firmware/project/template/<template_name>/mixer8 (deprecated)
Implements software mixing in the embedded ARM processor

/firmware/project/template/<template_name>/target
Developers set up the personality of the device here, such as 1394 Configuration ROM, GPIO, etc.

/firmware/project/template/<template_name>/test
Several handy test utilities, including a memory test module that can be used to verify the memory configurations.

11. Using Project Templates

Shell scripts

In your **/firmware/project** directory are a number of shell scripts that can be used to create new starting points for development.

/firmware/project/install.sh

This script creates and builds the projects for DICE EVM's. It is normally just used when the environment is initially installed and the developer wants to create binaries for the various EVM's using the project templates. Look in this file for examples for starting your own development projects.

The other scripts are used to create individual projects from specific templates. The scripts copy the template files to a new directory, and then replace certain references in the relevant files with the project name to produce output in the firmware tree that corresponds to that project. For example, if a project called **myProject** is created using the **tcd22x0_EVM003** template, then its project files will be placed in kernel library will be built in:

/firmware/project/myProject

and the kernel for the project will be built in

/firmware/kernel/build/ecos_diceJr_myProject

The 'ecos' and 'diceJr' parts of the directory name are set by the OS and CHIP definitions in the file
/firmware/project/myProject/make/Make.params

Creating a new project from a Template

Create a project using a template as follows:

user@computername /firmware/project
$ ./new_tcd22x0_EVM003_proj.sh myProject

The new project is then a working directory for that particular project.

To initially build the new project the **install** target.

user@computername /firmware/project/make/myProject
$ make install

This initial build will make the kernel as well as the application.

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

12. 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 application intermediate and executable files are kept within the project directory in the **bin**, **lib** and **obj** directories. The kernel for the project is kept in **/firmware/kernel/build** in a directory with a name that is unique to the project.

Make Hierarchy

Kernels, Boot monitors, Applications, memory tests 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 DICE EVM003, go to its project make directory and invoke **make** for the target from there.

```
user@computername ~
$ cd /firmware/project/evm003_tcd22x0/make
user@computername /firmware/project/evm003_tcd22x0/make
$ make <target>
```

The <target> parameter is summarized in table 11.1.

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 Studio 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.

Visual Studio 'Build->Build projectXXX' calls vc_make.bat with 'dep'
Visual Studio 'Build->Rebuild projectXXX' calls vc_make.bat with 'install'
Visual Studio 'Build->Clean projectXXX' calls vc_make.bat with 'clean'

There are no other equivalents in the default Visual Studio Build Menu for the other make targets, but if you use them often then these can be added to the Visual Studio 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 Application executable files. The names are created by using the BOARD define, concatenated with "Debug" if the build is not defined with _RELEASE in the CFLAGS definition. The BOARD define is set by the template creation script project name that was specified.

For example, the default projects created by install.sh from the templates will build executables as follows:

/firmware/project/evm003_tcd22x0/bin/
evm003_tcd22x0Debug Contains debug symbols for gdb
evm003_tcd22x0Debug.bin Executable that can be written to flash

As mentioned before, the kernel source directory, which is created and built for each Application project, is named by concatenating the the OS, CHIP and BOARD defines. In this case, the kernel is created in the following directory:

/firmware/kernel/build/ecos_diceJr_evm002_tcd22x0/

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 with **make install**, its kernel library is kept in the **/firmware/kernel/build/** directory in a subfolder that is given a name as described above.

All of the application object, library, and resulting executable binaries are kept in the project's directory, for example of the default EVM003 project, in the **/firmware/project/evm003_tcd22x0/obj**, **lib** and **bin** directories.

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 resulting executable:

/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 resulting executable:

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

Compiled Memory Test Utility file locations

Build the kernel library for the Application before building these targets.

DICE-II

The **in_ram_test_dice2** target creates a kernel lib directory and RAM test:
/firmware/kernel/build/in_ram_test_dice2
/firmware/chip/dice2/internal_ram_test Executable

DICE-JR, DICE-MINI

The **in_ram_test_jr** target creates a kernel lib directory and RAM test:
/firmware/kernel/build/in_ram_test_jr
/firmware/chip/diceJr/internal_ram_test Executable

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

Compiled Documentation

The **make doxygen** target builds the html docs start page **index.html** in, for example:
/firmware/project/evm003_tcd22x0/doxygen/GeneratedDocumentation/html
If you have installed the Microsoft Help Compiler, the help file will be in:
/firmware/project/ evm003_tcd22x0/doxygen/GeneratedDocumentation/help

See the section *Self Documenting Code and Doxygen*.

When to use each make target

Scenario	Command(s)
When building the sources first time	Bash
	user@computername /firmware/project **$ source install.sh**
When creating a new project from the project Template	Bash
	user@computername /firmware/project/myProject/make **$ make install**
	VC++
	Build->Rebuild myProject
After ongoing edits to the non-kernel source code	Bash
	user@computername /firmware/project/myProject/make **$ make dep**
	VC++
	Build->Build myProject
After changing Make.params that add or remove modules, and cause changes to module makefiles	Bash
	user@computername /firmware/project/myProject/make **$ make clean** **$ make dep**
	VC++
	Build->Clean myProject Build->Build myProject
When making changes to the kernel	Bash
	user@computername /firmware/project/myProject/make **$ make cleanKernel** **$ make install**
Comprehensive clean	Bash
	user@computername /firmware/project/myProject/make **$ make cleanAll**
When making the flash-resident RedBoot boot monitor	Bash
	user@computername /firmware/project/myProject/make **$ make redboot_romram**
When making the RAM JTAG-loadable RedBoot boot monitor	Bash
	user@computername /firmware/project/myProject/make **$ make redboot_ram**
When making the in-RAM test applications	Bash
	user@computername /firmware/project/evm_dice2 **$ make in_ram_test_dice2** user@computername /firmware/project/evm_tcd22x0 **$make in_ram_test_jr**
When making the project's documentation	Bash
	user@computername /firmware/project/myProject/make **$ make doxygen**

Table 11.1 Make targets summary

How to add a new module to a project

Add the module

Typically you can copy an existing module directory (except the "**main**" directory) and replace the source and header files with your own, then edit the Makefile to reference your new sources.

Your module should have an initialization function, which is called from **cyg_main.c**. As a rule of thumb, each module should use a module #define to make itself removable.

Modify Make.params in your project directory

Add an item to the MODULE list

Choose a unique module name and add it to the list where **APP_MODULE** is specified:

```
APP_MODULE := frmwload 1394lal fcp misc fis aml ftm my_new_module
```

Add a unique #define for the module

This will allow the module to be removable with #ifdef directives where appropriate. This example uses **_MY_NEW_MODULE** which, of course, is used as –D_MY_NEW_MODULE in the compiler flags.

Add a CFLAGS entry, and include file path(s) for the module

To include the module in the build, add an if-case for it:

```
#####################################################################
#               my_new_module module (my new module)
#--------------------------------------------------------------------
ifeq (my_new_module, $(findstring my_new_module, ${MODULE}))
       CFLAGS += -D_MY_NEW_MODULE -I${PROJECT_DIR}/my_new_module/interface
endif
```

Note here that PROJECT_DIR could also be COMMON_DIR or CHIP_DIR, however it is cleaner to make new modules in the project directory.

Modify Makefile

Add the module's Makefile

Add an if-case in the **allall** make target that specifies the location of the Makefile for your module:

```
ifeq (my_new_module, $(findstring my_new_module, $(MODULE)))
       $(MAKE) -C ${PROJECT_DIR}/my_new_module/src
endif
```

Add the module's dependencies

Add an if-case in the **dep** make target. This will make sure the module is listed in the libraries that the linker needs to reference.

```
ifeq (my_new_module, $(findstring my_new_module, $(MODULE)))
       sed  -e '/GROUP/{s/libmy_new_module.a//}' ${LIB_DIR}/mytarget.ld > tmp
       sed  -e '/GROUP/{s/libtcat.a/libtcat.a libmy_new_module.a/}' tmp >
${LIB_DIR}/mytarget.ld
       rm tmp
endif
```

Modify cyg_main.c

In your own program, for example in

/firmware/project/myProject/main/cyg_main.c

call the module's initialization in **mainInitialize()** so it get's linked in, and to perform any initializations which must happen before threads are running.

```
#ifdef _MY_NEW_MODULE
       hResult = myNewModuleInitialize();
       if (hResult != NO_ERROR) return hResult;
#endif //_MY_NEW_MODULE
```

If your module has threads, also add an initializer for them in **mainInitializeTasks(void)**

Rebuild

Now that you have new dependencies, you'll need to scan them and rebuild all objects.

user@computername /firmware/project/myProject/make
$ make clean
$ make dep

If you are adding a module to a template, avoid module names which have **dice** in the name, because of match and replace operations done on files such as Make.params by the template creation scripts.

13. More on DICE Applications

Firmware on a DICE device consists of three main parts. A *Boot Loader*/*ROM Monitor*, a *kernel*, and *Application* software. The kernel and Application code are linked into a single executable image.

Boot loader/RedBoot ROM Monitor

In a device that is running from flash memory (i.e. the boot code runs from flash and loads the rest of the image to RAM where it is executed), 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. There are two boot loaders to use, 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 build system. 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, handles onboard 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. DICE firmware is also designed to be used peer-to-peer as well, which is described in other documents. TC Applied Technologies makes available this Firmware Development Environment, Host Software Development sources, EVM boards, and drivers.

Here we focus mostly on the firmware sources, 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 or none (in the case of an audio-only device or CMS device).. 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 sources. The projects provided are a complete implementation of the architecture. Contact TCAT to get the latest Host drivers, Host Software development API's and sample code.

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.

AV/C

This firmware sources provide 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 a DICE EVMs, it will present itself to the Host as a device with various i/o combinations, depending on the DIP switch settings on the EVM.

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 **frmwload** module.

Mac AV/C drivers are updated occasionally by Apple Computer, 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.

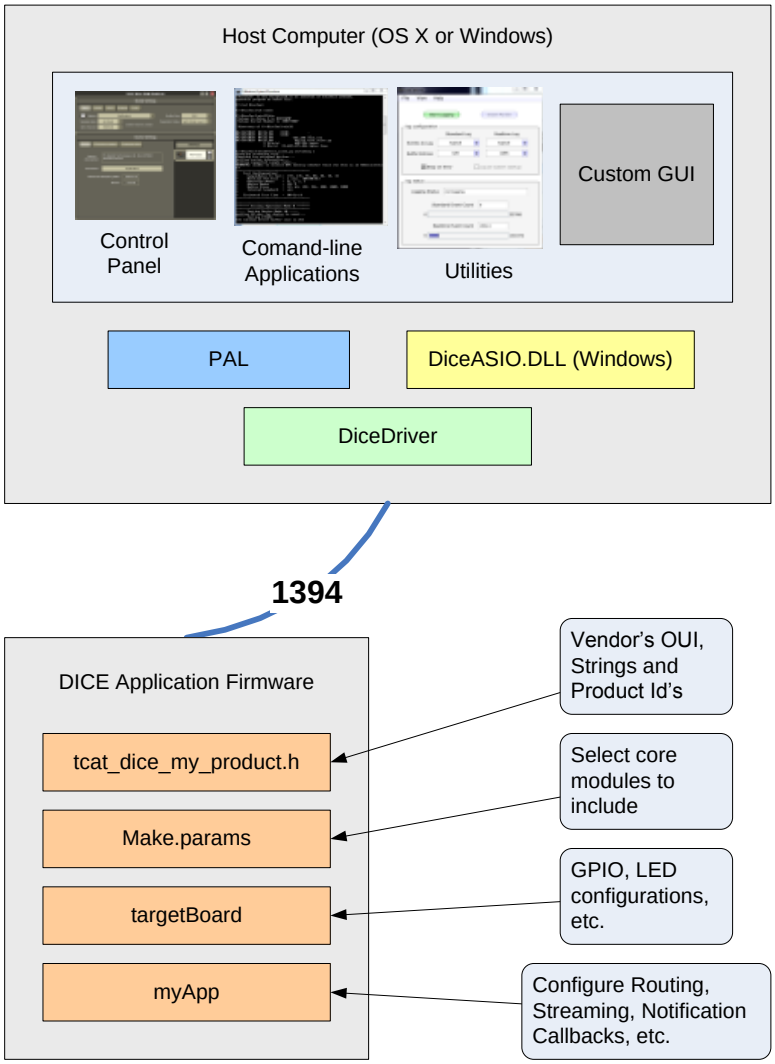
Windows Host Interfacing Example (DiceDriver)

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 DiceDriver implementation is a complete, low-latency, cross-platform method and is available today. Contact TCAT for Host Development Kits.
The figure below illustrates the components used to make a Windows DICE device using the DiceDriver target.

Host Computer



Device Discovery

When a DICE device is connected to the computer, the computer enumerates it by reading its Configuration ROM, which contains information that help the host computer operating system install the appropriate drivers.

Customization and Branding

In each software release, TCAT builds the drivers for you and includes them in your Subversion tag, along with installers that install the drivers and Control Panel for all operating systems. Your tcat_dice_myproduct.h file contains the info needed to build your drivers for your vendor information. Vendors can also customize the Control Panel and can provide TCAT with logos, icons, etc. for branding. The installer also creates an operating system setting that tells audio applications such as DAW's where the Control Panel executable is when it is opened from within the application.

Platform Abstraction Layer

The PAL is a cross-platform C++ API that developers can use to create Custom GUI applications, Command-line tools and other utilities. It provides control and monitoring, various notifications, firmware loading and other services.

When the device is enumerated, the Host driver queries the device firmware for various information such as clocking, and streaming capabilities, channel names, etc. This information is exposed through the PAL for applications to use.

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

The *Control Panel* is an application that is run stand-alone, or is invoked using the device configuration API in most ASIO enabled applications. 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 other User Interfaces on the host for control and monitoring of device functions.

DICE Firmware

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

Device Discovery

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

`/interface/tcat_dice_my_product.h`

See the comments in the file for more information. When you are ready to test production firmware that has your own vendor information, you will send us your modified version of this file (along with other custom graphics and settings files for Host installers and Control panel) and TCAT will create a custom Subversion tag for you.

The firmware uses this information to build the configuration ROM at run time, and the drivers are built using this information as well.

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 to configure for the audio and MIDI streaming configuration to work with the device.

Stream Configuration

Stream configuration and DICE Audio Interface configurations are handled in

`/firmware/project/myProject/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/myProject/module/dicedriver`

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/myProject/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/myProject/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 sources, 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.

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.

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/myProject/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 Projects

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

The 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. Production firmware will not typically use the myMode method.

New users will benefit from using an EVM Project along with a DICE EVM for experimenting and prototyping firmware before your custom hardware design is complete.

14. Software Updates

Version control

Firmware updates will be made available directly from a revision control server using Subversion. Each customer will use a tag that is prepared for them depending on their needs.

Once you have access to your tag, you can *export* it and do your work from within that checkout. Your firmware work will generally take place within a project directory that you create from a template using one of the project creation scripts.

You should also keep a separate *checkout* version of the tree, which is useful to use to see what changes have been made since the last release. You can then merge the changes in the checkout tree into your export tree.

New releases usually contain changes throughout your svn tag, so it's recommended that you check out the root level of the tag, rather than just the /firmware and /interface directories.

Since TCAT occasionally enhances the template projects, it's also useful to check for any changes to the project templates in your svn checkout and to merge any relevant changes into your custom project directories.

Firmware Updates

Occasionally TCAT will make announcements about new releases. You should make sure you are on the TCAT Developer Newsletter distribution list so you can be notified when a new release has been made and you can update your checkout.

Use the **svn status -u** command, or the Tortoise SVN 'Check for Modifications' command with the 'Check Repository' option, to check if there are new updates on the server.

Rarely, TCAT will change the URL of your tag (and let you know beforehand of course). In this case you can use the **svn switch --relocate** command or the TortoiseSVN->Relocate… command to point your local checkout to the new URL.

See the documents in the public area of the repository for more info.

https://dev.tctechnologies.tc/tcat/public

15. Shell

The Cygwin component's setup program uses a 'DOS' window for the **bash** shell by default in the original distribution. This environment 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`.

1) The shell is launched using the Cygwin icon in the Start Menu. 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

16. Editors

Emacs

GNU Windows Emacs is no longer included in the DICE Firmware Development Envronment, however GNU Windows Emacs can be found online, and the environment will be configured to work with it.

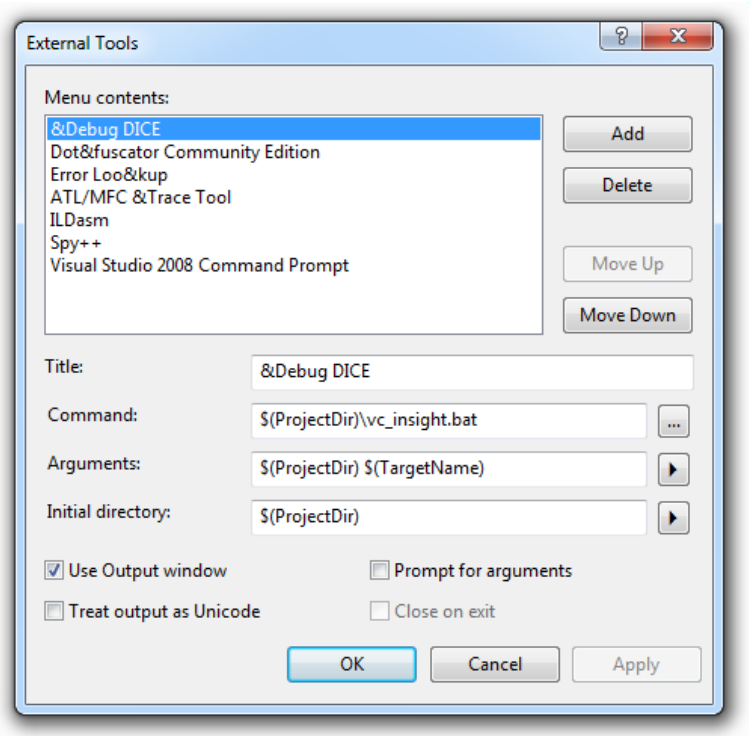
Visual Studio

For developers who are used to using IDE's like Visual Studio for their editor, a solution file and workspace project files are provided in the projects make directory.

This is useful for browsing the dice code, and also adds commands that build the dice code, redirecting output to the output window.

For convenience you can also launch the debugger from within Visual Studio. Use the Tools->External Tools… dialog to add the debug tool as shown below. The command is $(ProjectDir)\vc_insight.bat. Arguments are $(ProjectDir) $(TargetName). The initial directory is $(ProjectDir). This external tool will work regardless of which firmware solution is currently loaded.

You can then add a toolbar button for the debugger for quick access.



Inactive Code

This version of Visual Studio looks at #defines in the sources for inactive code. However, it does not see the #defines as specified in the firmware build system, so most of the #defines are added to the included Visual Studio Project preprocessor definitions property. You can add any new #defines to the project's environment and it will then see all of the code as reachable, and you can then view and browse the sources as usual.

17. 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.' The debugger can be used in the command-line mode in a bash shell, or within 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.

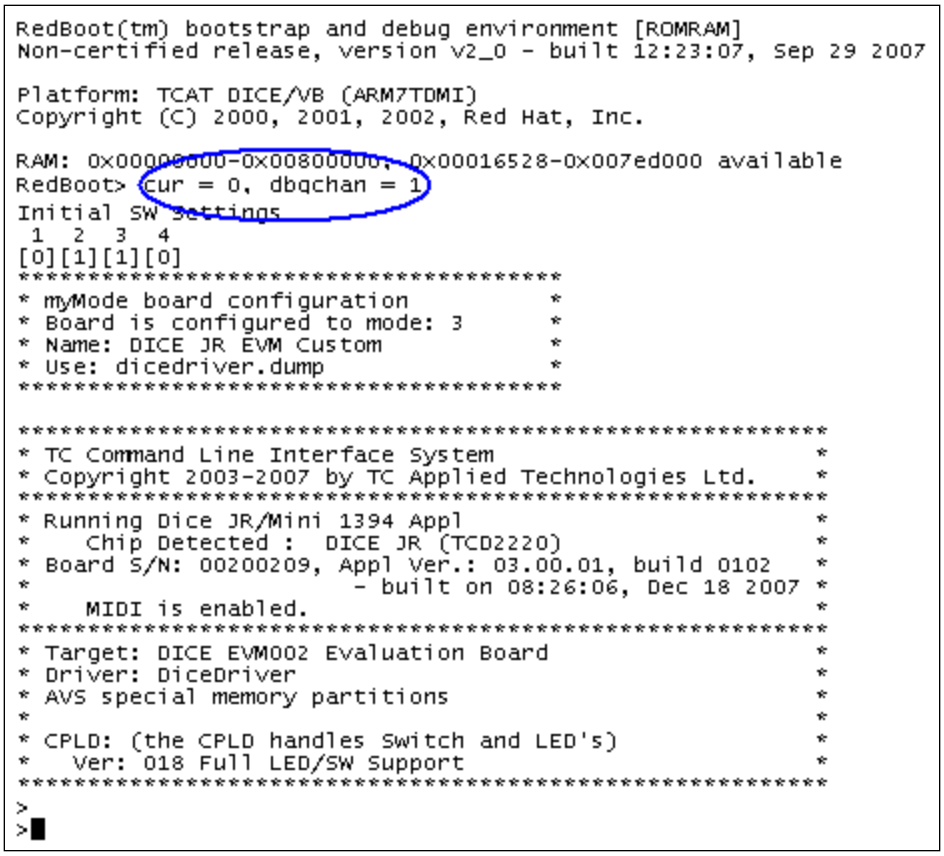
Note

The default debug serial port on the classic 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.



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 the figure 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 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 use the debug **–g switch** *at the same time as the optimize* **–O2** *switch*. This means, of course, that your debug firmware application 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 change the –O2 optimization to –O0 in the compiler flags and rebuild.

Compiler flags are found in the file **Make.params** in the project's make directory, for example **/firmware/project/ evm003_tcd22x0/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/evm003_tcd22x0/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 will place your debug image in, for example, the **/firmware/project/evm003_tcd22x0/bin** directory, where the **evm003_tcd22x0Debug** image contains debug symbols for gdb (and the **evm003_tcd22x0Debug.bin** image is a binary that can be downloaded to flash memory and run from there).

user@computername /firmware/project/myProject/bin
$ arm-elf-gdb.exe –nx evm003_tcd22x0Debug

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/evm003_tcd22x0/bin
$ arm-elf-gdb.exe -nx evm003_tcd22x0Debug
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)
```

Running gdb from a bash shell

JTAG Debugging

OpenOCD

The Open On-Chip Debugger is the recommended interface for JTAG debugging, and it is included in the Firmware Development Environment. It provides a debug server which gdb can attach to for debuggin, and can be used to program flash memories on new boards. It is compatible with a growing number of low-cost and easy to find JTAG hardware interfaces. Your subversion tag includes documentation and examples for using this software for debugging and board bring-up.

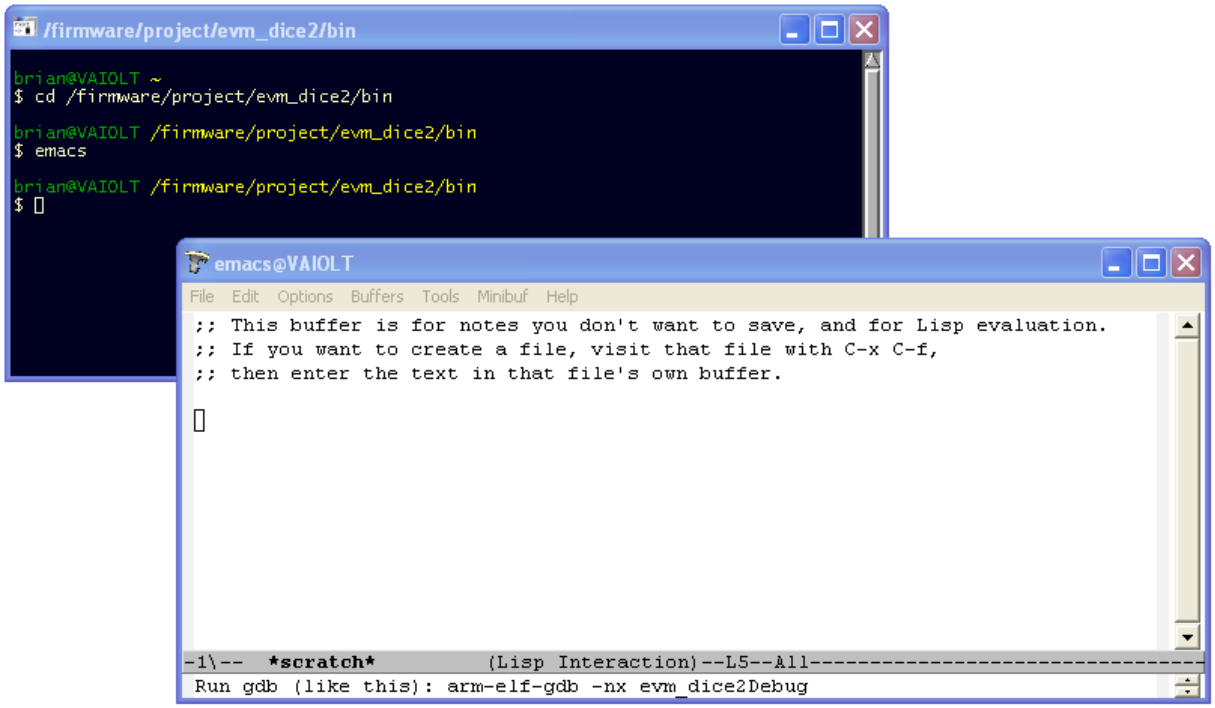
You can find this in your Subversion tag in **/docs/public/firmware/OpenOCD**

Other

Other JTAG probes have been reported to work successfully with gdb and the DICE ARM7TDU CPU core. If your probe is compatible with gdb and ARM, there is a good chance that it will work with DICE.

Emacs

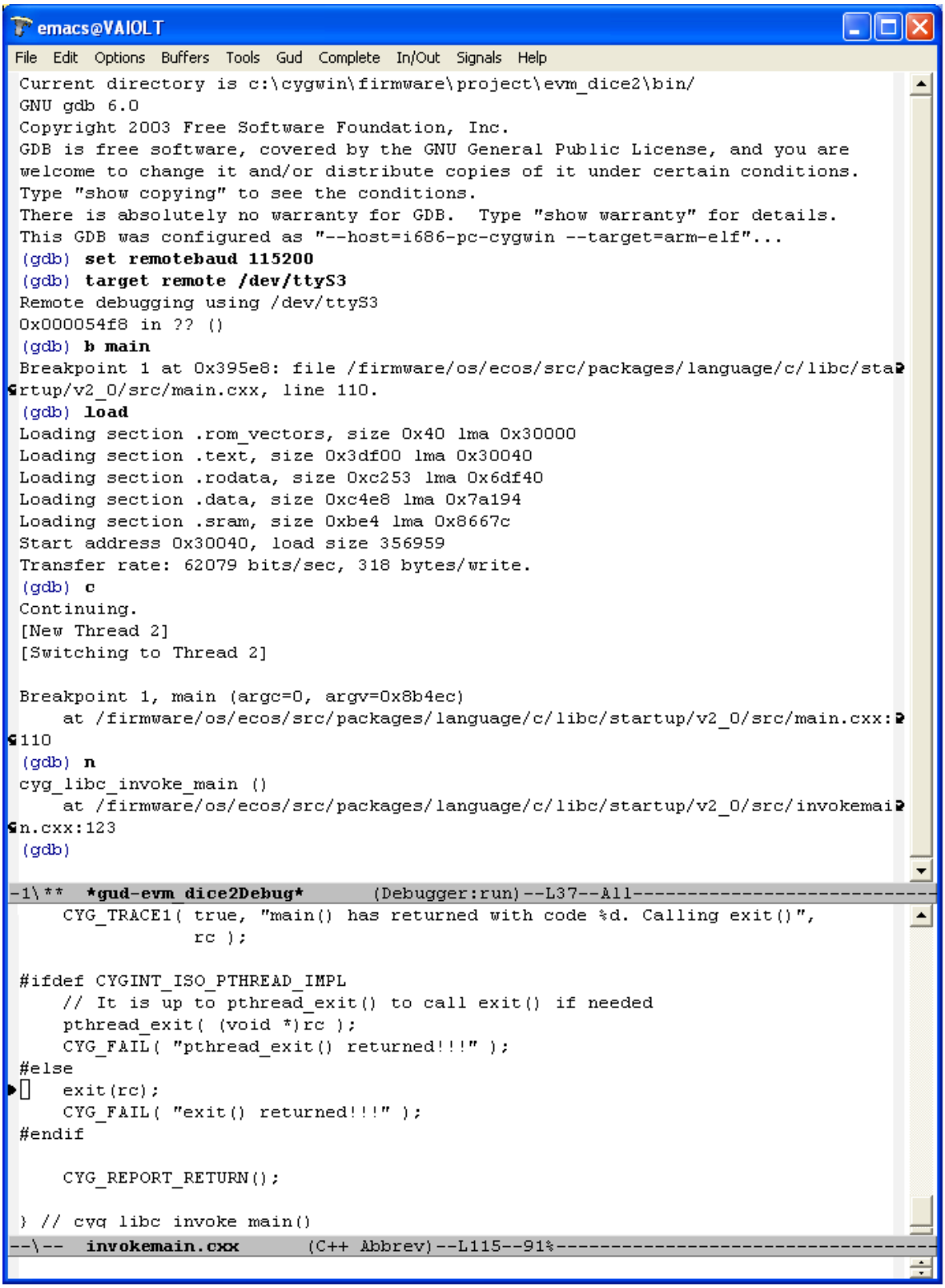
You can invoke gdb using the emacs menu option: 'Tools->Debugger (GUD)…' and then typing for example "**arm-elf-gdb –nx evm_dice2Debug**" into the scratch buffer. If you haven't evoked emacs from the directory containing the debug image, then enter the full path to the file as well. Note that tab-completion works in the emacs scratch buffer.



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.



Loading, running and stepping though dice code with Emacs

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.

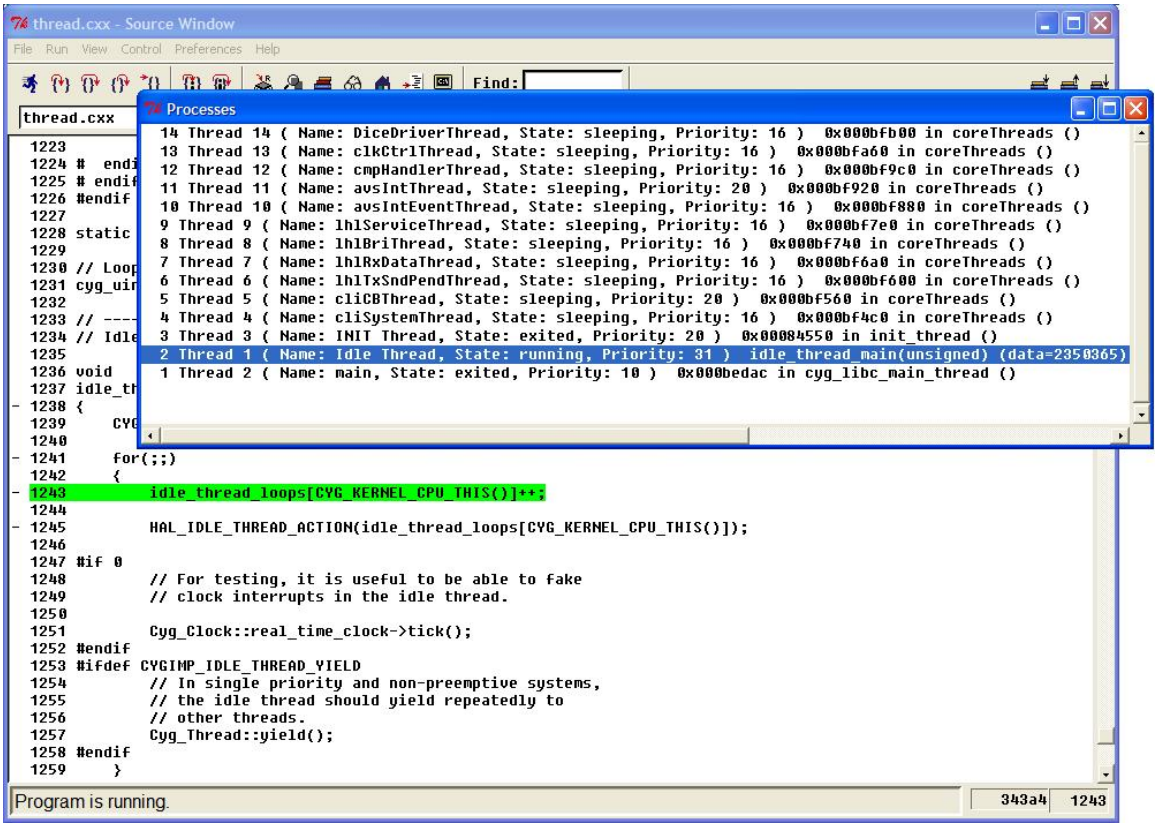
As in the figure above, when you "continue" in gdb, you'll see that the application splash screen has appeared, indicating that you are now running your code.

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.

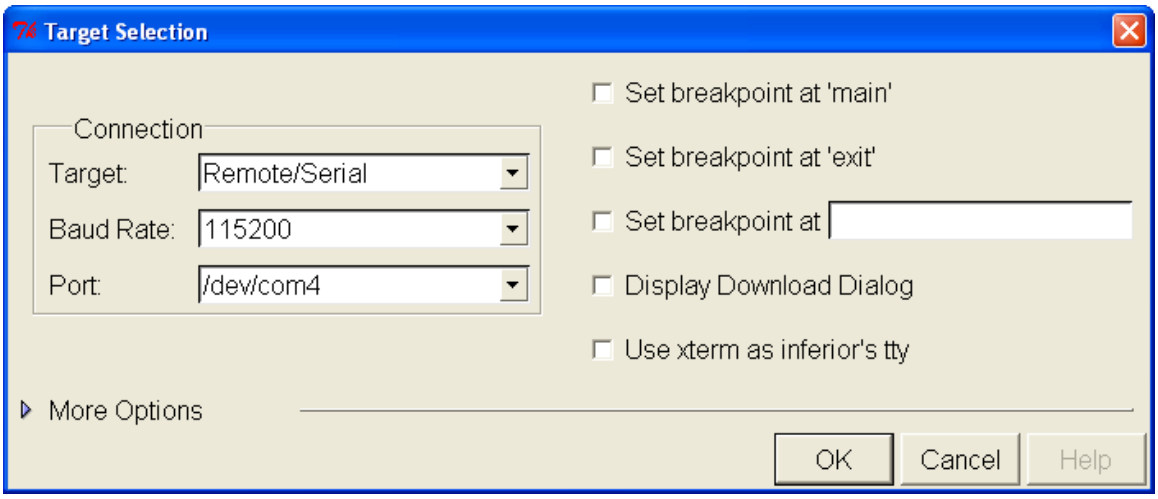


Insight debugger

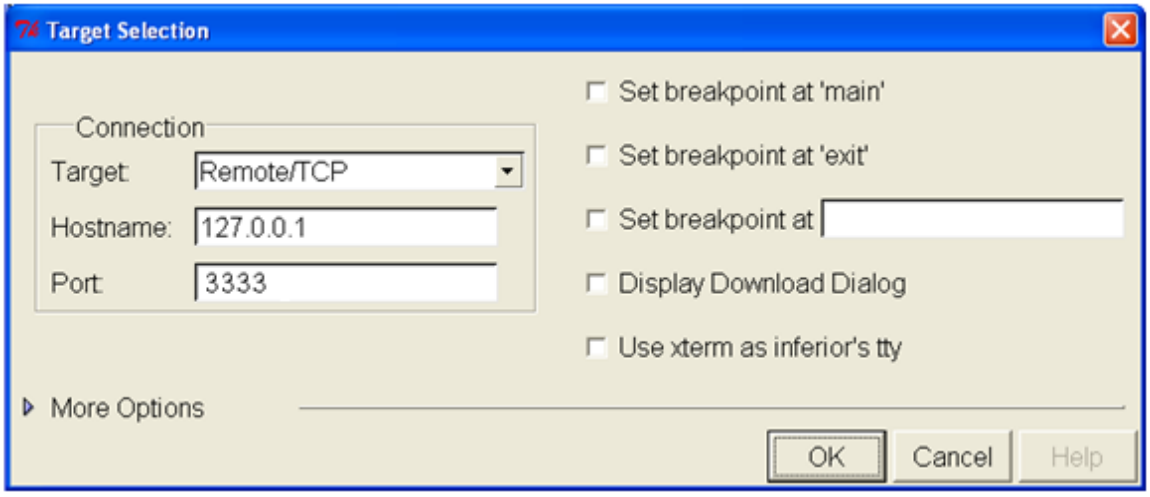
The Target Settings Dialog

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.



Setting up Target Settings in Insight for serial debugging



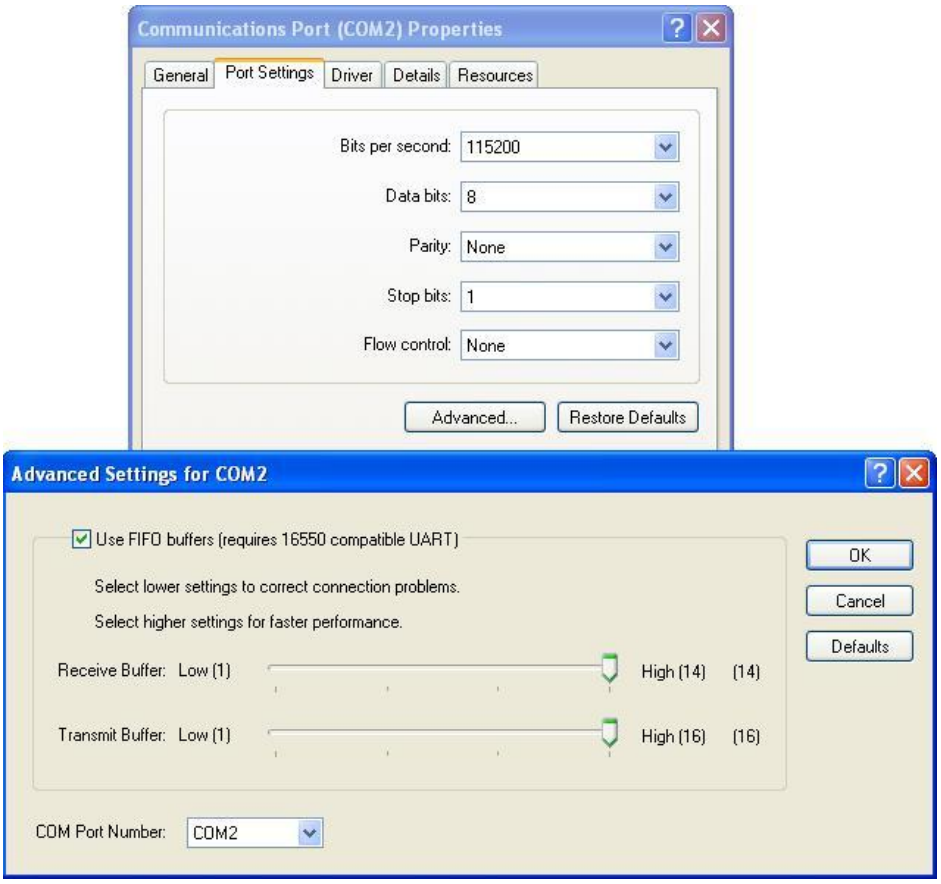
Setting up Target Settings in Insight for JTAG debugging with OpenOCD

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 the figure below, or use the utilities that came with your serial adapter hardware where appropriate.



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.

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.

18. Command Line Interface

The built-in CLI in the dice code is a powerful tool for experimenting and prototyping with the dice API's and getting the current status of the various firmware modules and hardware registers. 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.

19. Bringing up Hardware and Debugging with JTAG

Under normal circumstances, you will develop application firmware with DICE target hardware using JTAG connection, especially if your application uses one of the DICE UARTs for MIDI.

Serial debugging is also supported as described previously, but it requires a preloaded RedBoot ROM monitor to be running on the target hardware. The Rom monitor can be programmed to flash with a JTAG probe. If you have a DICE EVM, it will come preprogrammed with a ROM monitor.

If you are using OpenOCD, then you can load RedBoot using the supplied batch files in your svn tag in the /docs/firmware/OpenOCD directory.

If you are using a different JTAG interface, consult your documents and configure for use with gdb. Your JTAG tools may include a flash programming utility as well. If not, follow the steps below.

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 from RAM 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 debug application code using serial port debugging.

Below is the detailed description of the steps:

1) Install the Environment and configure your bash environment as described in the *DICE Firmware Development Environment Installation Guide*.
2) Find the RedBoot images in your subversion tag in /firmware/extras/RedBoot.

For the EVM project, do this as follows:

$ cd /cygdrive/c/myDevTree/firmware/extras/RedBoot

3) Modify the .gdbinit file in that directory for your JTAG software. It will usually connect via a localhost TCP connection on some port. Change the target remote command to use the correct port, or make any other changes as required.
4) Connect your JTAG interface to the board and start your JTAG debug server program
5) Connect a serial terminal program from the PC to the target hardware, primary serial port. Use 115200 baud, 8, N, 1, no handshaking
6) Run gdb

$ arm-elf-gdb.exe

7) The .gdbinit file in that directory will connect and load redboot.elf and then execute it.

You should now see a RedBoot prompt in the serial terminal. You can see that you are running the correct [RAM] version of RedBoot by looking in the splash screen.

```
RedBoot(tm) bootstrap and debug environment [RAM]
Non-certified release, version v2_0 - built 14:12:45, Mar  6
2007

Platform: TCAT DICE/VB (ARM7TDMI)
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.

RAM: 0x00000000-0x00800000, 0x000454c8-0x007ed000 available
RedBoot>
```

RedBoot RAM version

Note the [RAM] indication in the first line of the listing above.

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

   cygdrive/c/myDevTree/binary/firmware/RedBoot/redboot.bin

10) Write the image to flash

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

11) Quit gdb.
12) Power off the JTAG probe and remove it from the target hardware.
13) Reset the device to boot from flash, you should see a RedBoot prompt again.
14) 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.

20. 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, then any files that are used by the application for persistent settings storage, and any Application files after that. Storage files will have various names, such as evm_sps, eap_vol, am_sps, depending on the template that the project is derived from. You can find the name used by inspecting the code, or by running the application then checking the flash files to see what filename was created. When **setup**, storage, and application files (e.g. **dice**) have been loaded in the right order, 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  0x00030000  0x00010000  0x00030040
evm_sps           0x04030000  0x00000000  0x00020000  0x00000000
dice              0x04050000  0x00030000  0x00060000
0x00030040RedBoot>
```

A complete flash file system

This is a good time to check that your Application file (**dice** in this case) 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. If the Application file does not have room to grow in size with future updates without running into the setup file, this can result in fragmentation of the flash memory and the possibility of not having enough free contiguous space to write a new image. Making sure the application file is stored last prevents this.

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. For the latest pre-built binaries, check your subversion tag or check online at

https://dev.tctechnologies.tc/tcat/tags/release/public/latest/binary/firmware

Setup

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

Storage

Files such as evm_sps, eap_vol, am_sps are used by firmware applications to store persistent settings. These file should be created before the application image, 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 will be named something similar to
/firmware/project/evm003_tcd22x0/bin/evm003_tcd22x0Debug.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 RedBoot image is loaded via JTAG. Once the device is running RedBoot, you can load the rest of the files with a serial terminal.

The following instructions apply to the DICE EVM's, but will apply to most any DICE target hardware.

Loading RedBoot

You can find prebuilt RedBoot images in your subversion tag. If you are using OpenOCD, you can load RedBoot using the appropriate batch file in the docs/firmware/OpenOCD directory. See the section *Bringing up Hardware and Debugging with JTAG*.

Loading the Setup file

Loading the rest of the 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:

1) Create a text file containing the line below.
The file should look similar to this, *including a final carriage-return*:

```
SERIAL_NO=200209
```

SERIAL_NO is a 6 digit decimal number, which is typically derived from the serial number of your DICE microboard module's serial number, since this is where the flash memory resides. This is used to calculate the lower bytes of the device WWUID, and guarantees that your device will always have a unique WWUID on any 1394 bus used during development and test. See Section 16 *1394 WWUID and Device Serial Number* regarding this value.

2) **load -r –m ymodem –b 0x30000**
3) Transfer **config.txt** (using "1K Xmodem")
4) **RedBoot> fis create –b 0x30000 –l 0x10000 -e 0x0 –r 0x0 setup**
The **–l** length parameter should be at least the length returned by the load command (end – start +1). **0x10000** is more than sufficient here, but 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.

```
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
RedBoot>
```

Locating the start address of the setup file

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

This will show something similar to this:

```
RedBoot> x -b 0x4020000 -l 0x40 -1
04020000: 53 45 52 49 41 4C 5F 4E  4F 3D 30 30 32 30 30 32  |SERIAL_NO=002002|
04020010: 30 39 0D 0A 48 50 4C 4C  5F 43 4C 4B 3D 35 30 30  |09..HPLL_CLK=500|
04020020: 30 30 30 30 30 0D 0A 00  98 01 03 00 C4 01 03 00  |00000...........|
04020030: E0 01 03 00 00 00 00 00  A8 02 03 00 00 00 00 80  |................|
RedBoot>
```

Examining the contents of the setup file

Pre-loading the storage file

1) Examine your firmware source code. Look for the name argument to spsInitialize() in the code. For the EVM003 project, this is "am_sps"
2) You will need to determine the size of the file that's created and round up to the nearest sector boundary. The flash on the EVM has 0x1000 sector size. The EVM003 storage file requires two sectors, so create a file for this as follows:

   RedBoot> fis create –b 0x30000 –l 0x20000 –e 0x0 –r 0x0 am_sps

   This creates the file using whatever random bytes were in RAM at offset 0x30000. When the application boots, it will notice that the file is garbage and overwrite it.

Loading your Application file

1) **load -r –m ymodem –b 0x30000**
2) Transfer the application file, which is something like **/firmware/project/evm003_tcd22x0/bin/evm003_tcd22x0Debug.bin** (using "1K Xmodem")
3) **fis create –b 0x30000 –l 0x80000 –e 0x30040 –r 0x30000 dice**
   If your flash already has a file named "**dice**" confirm that you want to overwrite the existing file or choose a different name. 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

RedBoot's fconfig utility can be used to configure your board to run a script at boot time. Typically this is used to execute an application file. Additionally, the configuration area managed by RedBoot can contain aliases to be used in the script or in RedBoot's CLI. The benefit of aliases is that they do not get erased by the fis flash initialization command **fis init -f**. When the EVM is shipped it will have a **dice** application file and an alias for the **dice** file:

> alias IMAGE dice

IMAGE can now be used in scripts and CLI commands by referencing it as %{IMAGE}. The boot script is set as shown below

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 %{IMAGE}
   >> 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. The splash example below is from EVM002 firmware.

```
>splash

************************************************************
* TC Command Line Interface System                         *
* Copyright 2003-2010 by TC Applied Technologies Ltd.      *
************************************************************
* Running Dice JR/Mini 1394 Appl                           *
*    Chip Detected :  DICE JR (TCD2220)                    *
* Board S/N: 00200209, Appl Ver.: 03.00.01, build 0102     *
*                      - built on 08:26:06, Oct 18 2010    *
*    MIDI is enabled.                                      *
************************************************************
* Target: DICE EVM002 Evaluation Board                     *
* Driver: DiceDriver                                       *
* AVS special memory partitions                            *
*                                                          *
* CPLD: (the CPLD handles Switch and LED's)                *
*   Ver: 018 Full LED/SW Support                           *
************************************************************
>
```

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: 0x00016604 0x00830e11
>
```

Using Full Flash Images for Production

A common practice is to create a flash file system as described above, where the setup file has a serial number that will never be used in a shipped unit.

Once you have created a full flash image, you can download it from the device over 1394 to a Host computer using the Control Panel (with the Firmware Utils tab enabled) or by using the 'dice' command line tool. This image can be sent to your flash supplier to be preprogrammed before PCB assembly.

Often, the firmware application image is one that is used for factory test. Once the device passes tests, the application is programmed with the release firmware application and the setup fie is replaced with a unique serial number.

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.

You will find prebuilt RAM test executables in your svn tag in **/firmware/extras/in_ram_test**

In that directory, modify the load command at the bottom to load the appropriate test, i.e.

load in_ram_test_jr_mini.

To use the test, configure your JTAG interface software to be ready for a gdb session. Then run gdb

$ arm-elf-gdb.exe

RAM Footprint

This utility demonstrates the flexibility of eCos in that it can be configured to produce a very small kernel. The DICEII test uses the standard library printf() function for it's CLI, which uses approximately 7-8kB of memory. The DICE-JR/MINI version of the test uses a smaller alternative to printf() so it fits in the internal RAM on these chips.

This executable must be loaded and run via JTAG. When using the internal RAM test, the ARM Remap module must be configured to set the low portion of the address space to the internal RAM. This is done by writing a '1' to register 0xc0000008. Also, specify the SDRAM address as shown below. The .gdbinit file in the RAM test directory handles this for you by setting the SDRAM base address and setting the remap:

```
#SDRAM
p/x *((unsigned long *) 0x81000018) = 0x01000000
# Remap for the internal ram
p/x *((unsigned long *) 0xc0000008) = 0x00000001
```

Without 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.

The listing 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.
$
```

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.

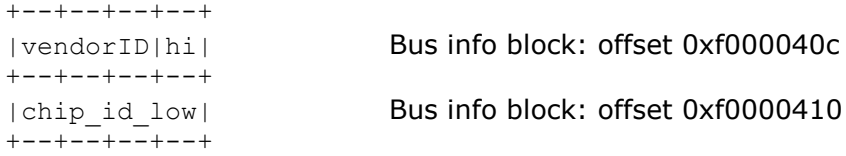
The DICE code uses a temporary OUI for your development. DICE EVM's are preloaded with the PCB production serial number for use as the rest of the WWUID. This serial number is written in the **setup** flash file on DICE EVM's.

Important 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.

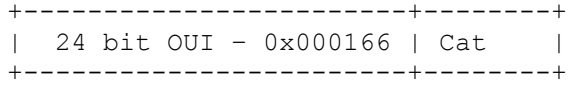
As mentioned before, the Vendor ID is a 3-byte value stored as the most significant 3 bytes of the appropriate CSR in the Bus Info Block of the Configuration ROM. Together with chip_id_hi and chip_id_low, this constitutes the entire WWUID for the node. Chip_id_hi and chip_id_low must be unique for all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

WWUID, as shown in Configuration ROM



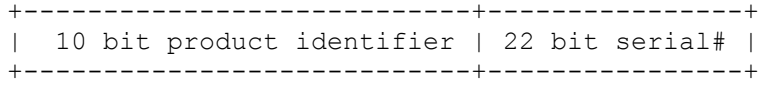
The manufacturer is free to use the lower 5 bytes of the WWUID in any way. The DICE firmware, by default, uses the chip ID as follows

Upper 32 bits of WWUID



Category 8 bits: TCAT uses **0x04** for this field.

Lower 32 bits of WWUID



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

TCAT uses the following for the 10-bit product identifier:

1 DICE-MINI device
2 DICE-JR device
3 DICE-II device

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/myProject/doxygen
$ doxygen

The resulting HTML start page is

/firmware/project/myProject/doxygen/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

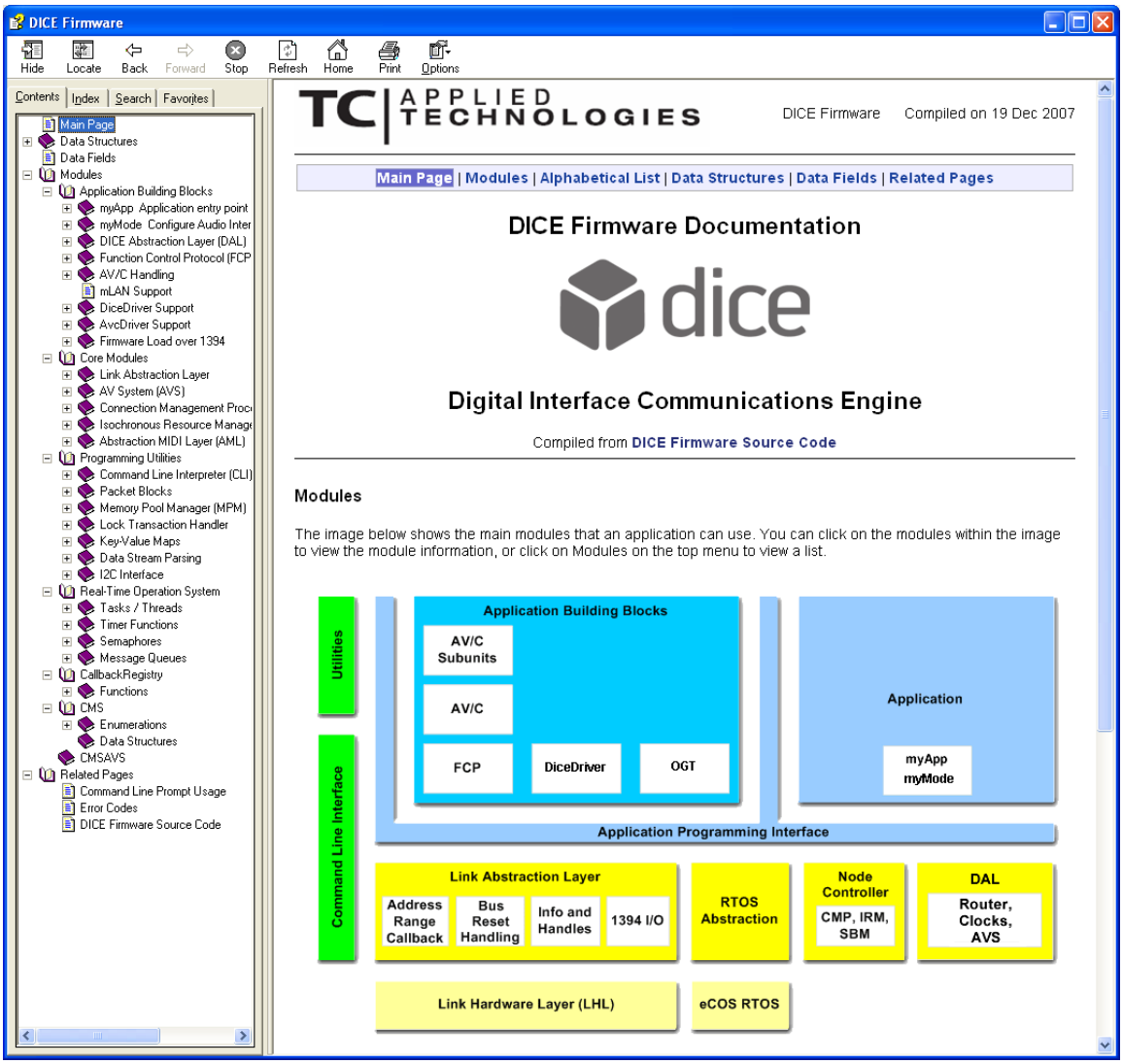
Note that if you download and install this program in its default directory, the doxygen command above will also compile the Windows Help file for you each time automatically. The resulting files are place in the

/firmware/project/myProject/doxygen/GeneratedDocumentation/help/

directory as ***FirmwareUserCompiledHelp.*** [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. 3.0.x release version of the DICE code is included, and a shortcut to it is provided in your DICE Firmware Development Envronment Program Group.

Converting HTML to Windows Help allows you to search the documentation as with any Help file.



Searchable Help generated from the Source Code
using HTML Help Workshop

24. Resources

Support Contact

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

References

DICEII/DICE-JR/DICE-MINI 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

OpenOCD Open On-Chip Debugger
Main site http://openocd.berlios.de/web/
Windows Installers http://www.freddiechopin.info/index.php/en/download

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

Version Control with Subversion

Client downloads
http://subversion.tigris.org/

Subversion book
http://svnbook.red-bean.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

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