EEMBC Version 1.1 Release

Porting Guide
and
Release Notes

January 28, 2020

Alan R. Weiss

alan@ebenchmarks.com

Rick Foos

rickf@ebenchmarks.com

EEMBC Certification Laboratories, LLC (ECL)

TABLE OF CONTENTS

1 DISCLAIMER AND COPYRIGHT 5

2 REVISION HISTORY 5

3 ACKNOWLEDGEMENTS 6

4 INTRODUCTION 7

5 TESTED TOOLCHAINS 13

6 IMPROVEMENTS AND CHANGES DETAILS 14

7 THIS IS NOT VERSION 2 14

8 BACKGROUND / HISTORY FOR UNDERSTANDING THIS RELEASE 15

9 DIRECTORY STRUCTURE 16
9.1 Top Level Directory Structure 17
9.2 Benchmark Directory Structure 18

10 FILE NAMING 20

11 THE TCDEF 20

12 TCDEF USAGE 24
12.1 Standard for 8-16 Bit Benchmarks 25
12.2 Standard for TH Regular 25
12.3 Standard for TH Lite 26

13 TH LITE API 27

14 HEAP MANAGEMENT 28

15 TH LITE ADAPTATION LAYER API 30

16 HEADER COMMENTS 30
16.1 Utility Header 33

17 DISPLAY OUTPUT 34

18 MAKERULE USER GUIDE 35
18.1 NAME 35
18.2 SYNOPSIS 35
18.3 DESCRIPTION 39
18.3.1 Using MAKERULE 40
18.3.2 Important Notes 41
18.3.3 Command Files 42
18.4 Future Work Being Considered 44

19 RELEASE NOTES ADDENDA 44

20 INSTALLATION 45
20.1 Download 46
20.2 Install 46
20.3 Verify 47
20.3.1 Microsoft Visual C/C++ IDE 47
20.3.2 Cygwin and Visual C++ Makefile 48
20.3.3 Log Files 48

21 PORTING GUIDE 49
21.1 Build Procedure 49
21.2 Guidelines 50
21.3 Create Empty Toolchain and Platform 50
21.3.1 Platform, Toolchain, Targets, and Root 54
21.3.2 Unix Porting 55
21.4 Toolchain 55
21.5 Directories 57
21.6 Run57
21.7 Platform 57
21.7.1 Benchmark Compile and Execution 58
21.7.2 Adaptation Layer 58
21.7.3 Adaptation Layer Application Overrides 59
21.8 Results 59
21.8.1 Iterations Override 60
21.8.2 Generating Benchmark Results 60
21.8.3 Generating Application Summary Results 61
21.9 Automation 61

21.9.1 Certification Results Summary 62
21.10 Benchmark CRC Options 62
21.11 The Test Harness Code Layout 65
21.11.1 Adaptation Layer Files 65
21.11.2 Functional Layer Files 65
21.12 Benchmark Files 67
21.12.1 The Adaptation Layer 68
21.12.1.1 Company and Processor Definition 68
21.12.1.2 The Command Line Length 70
21.12.1.3 Target Timer 70
21.12.1.4 Debug 70
21.12.1.5 Endian-ness 70
21.12.1.6 Floating Point Support 71
21.12.1.7 Printf Engine Control 71
21.12.1.8 Heap Debugging 71
21.12.1.9 Malloc and Free mapping 71
21.12.1.10 TH Size Control 72
21.12.1.11 EEMBC Data Types 72

22 SUPPORT 72

1 Disclaimer and Copyright

This document is EEMBC Confidential and Proprietary.

2 Revision History

Date	Revised By	Reason for Revision
July 26, 2001	A. R. Weiss (ECL)	8/16 Bit Porting Guide (as a reference)
April 10, 2002	A. R. Weiss, R. Foos (ECL)	EEMBC Version 1.1 Beta 1
April 30, 2002	A. R. Weiss, R. Foos (ECL)	EEMBC Version 1.1 Beta 1c Addition of Porting Guide and deltas between 1.1 Beta 1 and this version
June 28, 2002	A. R. Weiss, R. Foos (ECL)	EEMBC Version 1.1 Release Candidate / Beta 2 Restructured build environment to better support multiple compiler/platform targets based on input from Chuck Corley.
July 17, 2002	R. Foos (ECL)	EEMBC Version 1.1 Release Candidate / Beta 2a Applied changes based on feedback on Beta 2,
July 30, 2002	R. Foos (ECL), A. R. Weiss	Verify Beta 2a Fixes, editing updates
August 9, 2002	R. Foos (ECL)	Add Non Intrusive Checksum to TH Regular Benchmarks.
September 30, 2002	R. Foos (ECL)	EEMBC Version 1.1 Release Fixes to diffmeasure/verify, and make environment fixes. Added size extraction to allsize log file.

3 Acknowledgements

The following people provided valuable input that went into this 1.1 Release Candidate:

Alan R. Weiss (ECL)
Amanuel Belay (Motorola)
Brian Grayson (Motorola)
Chris Dorman (ARM)
Chuck Corley (Motorola)
Dan Temple (MIPS)
Elena Stohr (ARM)
Hamish Fallside (Xilinx)
Lawrence Spracklen (Sun Microsystems)
Michele Christie (Green Hills)
Patrick Webster (ARM)
Paul Messier (ECL)
Rick Foos (ECL)
Robert Whitton (Siroyan)
Ron Olson (IBM)
Sergei Larin
Shay Gal-on (Improv Systems)
Stig Linander (MIPS)
Tom Karzes (Equator)
Wilco Dijkstra (ARM)
All the people who reported defects/bugs in Release 1.0

4 Introduction

This is the ***release*** of **EEMBC Version 1.1** that incorporates a small number of improvements over EEMBC Version 1.1 Beta 3. Here is what has been changed and improved with respect to 1.1 Beta 3:

- Closed Bugs#: 195, 198 regarding diffmeasure, and iterations settings in one telecom benchmark.
- Added Bugs# 199, 200 regarding a Linux run problem in Microcontroller Bit Manipulation, and Networking Packet Flow CRC failures on multiple TH Regular runs.
- Added Toolchain dependent extraction of size logs.
- Fixed $(RUN) command to use $(RUN_FLAGS)
- Set packet flow checksums for NW_DGM_ALIGN=4
- Enhanced extract time Awk script to strip \r's from th regular th_report_results print of benchmark description.
- Cleaned up section names in Toolchain files.

This is the ***Release Candidate / Beta 3 release*** of **EEMBC Version 1.1** that incorporates a number of improvements over EEMBC Version 1.1 Beta 2. Here is what has been changed and improved with respect to 1.1 Beta 2:

- Corrected Diffmeasure math algorithm
- Added Trap code for MAP_MALLOC_TO_TH where malloc called before C program loaded. Occurred in routelookup where Windows Kernel called malloc before crt0 was finished.
- Silence echo's during results generation of make.
- Reset iterations to modulo 11 of requested value because of periodicity in timing/workload
- Matched Heap control options between Regular and Lite for regression testing of timing and size. Added option in th regular to use compiler malloc.
- Fixed iterations reported in log file to actual program iterations taken.
- Matched log file reporting in regular and lite.
- Added failure in th_report_results for actual vs. expected iterations.
- Added NI crc checking and verification (telecom) to all th regular kernels.
- Matched math routines in diffmeasure and verify. Moved math to single location in verify.c. Diffmeasure is a wrapper performing file io for verify.c.
- Added FLOAT_SUPPORT trap in regular benchmarks using diffmeasure. Floating point support will calculate S/N internally, no uuencode output. No Floating point will uuencode output as before.
- Do not send buffer back to host if CRC checking enabled.
- Report CRC option (Intrusive, Non-Intrusive, No CRC checking) in log file.
- Add NDEBUG define to gcc.mak.

- Change fgetc return variable to int to pick up -1 EOF vs. 0xff char.
- Compile out command line -i (iterations) when CRC_CHECK enabled.
- Display Required Iterations when CRC_CHECK enabled.
- Added trap in bezier for USE_FPU && !FLOAT_SUPPORT
- Modified CRC routines to check individual points, and convert e_f64 to e_u32.
- Added intrusive CRC to Bezier.
- Added top level makefile with 'forall makeall' functionality.

*** Beta 2b

- Fixed MS Visual C projects that stopped building because of CRC_CHECK in the settings.
- Removed techtag/empty projects from makeworld MS Visual C workspace.
- Added CRC column to util/awk/extracttime.awk.
- Added -a "iterations(VER)(PLATFORM).mak" to makerule and benchmark depgen files to force benchmark compile on iterations file changes.
- Added -Ixxx command line override for iterations in th lite benchmarks.
- Fixed iteration dependencies in automotive NI CRC's.
- Rollback cjpeg/djpeg checksum fix, change checksum to crc8 with e_u8 cast
- Rollback pktflow_init.c cast. Change NW_ALIGN -(x) to (0-x) like roundup.
- Fixed th/src/thfl.c to pass command line options beginning with '-' to the benchmark.
- Stripped remaining eembc_id check code from th regular.
- Added ID field in report_results from harnesses.
- Added <subcommittee> <benchmark> key string in benchmark inits of eembc_id field.
- Added <subcommittee> <benchmark> and CRC fields to log files in util/awk/extracttime.awk
- Added Bezier float to automated benchmark build. (Ron O)
- Added error for !COMPILE_OUT_HEAP && HAVE_MALLOC_H in th lite configuration.
- Add certification results summary. Unique keys for all benchmarks, and generate tab delimited log files with column headers.
- Document CRC options available in each benchmark
- Add BMDEP, Benchmark Dependencies, for al headers and iterations to trigger benchmark rebuild.
- Fix Segmentation Fault in Bitmnp816 on Linux 7.3, random error location that could not be isolated in gdb, documented with ifdef LINUX.
- Fix Diffmeasure fault on Linux 7.3, could not duplicate, restored diffmeasure to build using libraries.
- Add Timing, ERROR, and CRC fields to summary log files.
- Fix bit allocation with rpent.dat Cannot allocate 500 bits over 20 carriers, could not duplicate
- Fix Results.mak files for V2 support.

- Added timing results, Expected CRC, and ERROR fields to certification summary.
- Add harness.h in each application to all overrides of TH settings.

*** Beta 2a

- th_lite: CRC_CHECK enabled in benchmarks that do not have CRC_CHECK code can result in checksum failure.
- th_lite: HAVE_MALLOC_H and COMPILE_OUT_HEAP both false. Fails in benchmarks that do not call mem_heap_initialize before th_malloc. Trap code depended on heap_pointer=0 set by compiler.
- Lines in script files have to be delimited by LF's - not CRLF's on Unix. Will recommend unzip –a in release notes.
- forall: 's1 == s2' is illegal in sh, use 's1 = s2' instead
- forall: While '! -n s1' is legal, I suggest using '-z s1' instead.
- Added ability to pass parameter to make with forall, makeall, makereg, and makelite (i.e. forall makeall clean)
- Fixed make clean to remove thobjs.a library.
- Release notes updated
- Unix porting: In makerule.pl one may need to change the first line (shebang) to #!/usr/local/bin/perl'
- Unix porting: In gcc.mak One may need to set 'TOOLS' to '/usr/local'
- networking/pktflow/pktflow_init.c", line 495: warning: integer conversion resulted in a change of sign
  dg_dsc_size = NW_ALIGN(sizeof(struct dgmdsc)+desc_padd);
  dg_dsc_size = (size_t)NW_ALIGN(sizeof(struct dgmdsc)+desc_padd);
- consumer/filters/datasets hardgsmall.c, hardcsmall.c
  < static Char RawRGB[16 + 320*240*3] = {
  > static const char RawRGB[16 + 320*240*3] = {
  Note: (GHS 3.5) use the '-signedchar' option to reduce warnings.
- heapport.h: remove warnings from heap.c, use HEAP_ALIGN instead of ROUNDUP4 in th_lite.
  < #define HEAP_ALIGN(x) (((x) +((HEAP_ALIGN_V)-1)) & -(HEAP_ALIGN_V))

  > #define HEAP_ALIGN(x) (((x) + ((HEAP_ALIGN_V)-1)) & ((HEAP_ALIGN_V)-1))
  *
  < #define ROUNDUP4(x) (((x) + 3) & -4)

  > #define ROUNDUP4(x) (((x) + 3) & ~3)
- Consumer cjpeg/djpeg checksum fix: The problem lies in the use of char in the CRC checking code. The variable img_start in {c,d}jpeg/bmark_lite.c should be defined as unsigned char* so that the CRC is correctly calculated. Fixing this gives a CRC 1689 for djpeg and 7177 for djpeg as Elena reported.

- if CRC_CHECK && NON_INTRUSIVE_CRC_CHECK are true, a compile error is generated.
- cjpeg/djpeg output file under WINDOWS_EXAMPLE_CODE fixed.
- Telecom outputdata default file names now key off the input file name <input file>Output.dat
- Bug 186: ctype.h should be included in telecom/viterb00/bmark.c (bmark_lite.c updated)
- Bug 189: log file target could be improved in makefile
- Bug 188: Corrections to the Release notes for Version 1.1 Beta 2
- Bug 190: Small improvements to the Version 1.1 Porting Guide
- Bug 191: Green Hills compiler needs a space between -o and object file (toolchain comments)
- Bug 192: Suggested Fix to 191 is in error (toolchain comments)
- Bug 193: TH Lite heap not initialized. HAVE_MALLOC_H and COMPILE_OUT_HEAP are both false. Fails in benchmarks that do not call mem_heap_initialize before th_malloc.
- Added TARGETS variable that allows dependency files generated by makerule.pl to be re-used on multiple tool chains.

This is the ***Release Candidate / Beta 2 release*** of **EEMBC Version 1.1** that incorporates improvements over EEMBC Version 1.1 Beta 1c. Here is what has been changed with respect to 1.1 Beta 1:

- Moved Test Harness and Utilities to be a peer of applications in the source tree. There is a single location for the harness and utilities.

- Enhanced and reorganized Toolchain definition file into sections for: System Environment, Compiler, Assembler, Linker, Librarian, and Build Control.

- Changed primary build control point from the standard makefile to the Toolchain/Platform definition file.

- Added Iterations control to build and all bmark.c files to allow iterations to be set at compile time for each benchmark.

- Multiple Toolchains and simulators tested. MIPS, ARM, LSI, Intrinsity, x86 Visual C/C++, and others. Enhancements added as a result of these ports.

- More test coverage demonstrating timing stability on V1.0 to V1.1 comparisons.

- More compiler warnings cleared from wider tool chain exposure.

- `Makrule.pl` (perl script) tool enhanced to generate a compile only make dependencies, and support both Test Harness link as objects or from a Test Harness Library (thobjs.a).

- A standard ifdef structure for TH Lite and TH Regular is used in all benchmarks. Both structures support compilation of DEFAULT and user defined recommended iterations. The Lite Structure additionally supports CRC_CHECK with required iterations, and NON_INTRUSIVE_CRC_CHECK with recommended iterations.

Note: This is a Release Candidate, and requires EEMBC member testing before it can be declared the Final Version 1.1 Release. Please send feedback and bug/defect reports via the EEMBC website (http://www.eembc.org/membership/bug.asp).

From our last Release Notes, here are the changes and improvements compared to EEMBC Version 1.0:

- Includes Automotive/Industrial, Consumer, Networking, Office Automation, Telecom, and 8/16 Bit Microcontroller Suites
- Version 1.0 did not include 8/16 Bit Suite. We only included TH Lite versions of the 8/16 Bit Microcontroller Suite in this Version 1.1 Beta because those processors cannot generally support TH Regular, and has never been the design goal of this particular suite.
- Architecturally consistent and consistent directory and file naming
- This will form the basis for Version 2
- Includes Test Harness Regular and Test Harness Lite for all of the benchmarks, including self-checking for all TH Lite versions
- All TH Regulars have the same test harness version -- no more "TH3.2E1 vs. TH 3.3"
- All TH Lites have the same test harness versions
- Top-down MS-Visual C Workspace File in each application space and individual Project Files
- Unix-style makefiles (just type "make" or "make LITE=_lite")
- Can work using Cygnus Cygwin gcc() environment, Linux, AIX, BSD Unix, or Solaris
- Can also build Visual C/C++ from standard makefiles
- Makefiles are all the same
- Dependencies in the makefiles are generated using makerule.pl (requires Perl 5.0 or above)

- Dramatically cleaned up -- much fewer warnings during build
- Log file time and duration extraction using gawk() (requires Cygwin (PC) or gnu tools (Unix))
- All lower-case filenames
- Empty Benchmarks were added to all Applications for code size calculations

5 Tested Toolchains

Version 1.1 has been built for using:

- This version is built for Win32 and embedded targets using:
    - Microsoft Visual C/C++ Version 6.0 SP5
        - Intel Pentium III and AMD Athalon
    - Cygnus Cygwin gcc() (gnu C compiler) Version 2.95.3-5
        - Intel Pentium III and AMD Athalon
    - GCC version 2.96, Red Hat Linux 7.3
        - AMD Athalon
    - GCC version 3.2, Solaris 8
        - Ultra Sparc 30
    - ARM ADS 1.2 (with execution on the ARMulator Simulator)
        - ARM 1020E 32-bit
    - Green Hills Software MIPS C Compiler 3.5
        - MIPS 20KC
        - Toshiba RBTX4927 32-bit
    - Intrinsity C Compiler and Simulator 2.96
        - Intrinsity FastMATH[tm] 16-bit DSP
    - LSI Logic 16-bit DSP
        - ZSP 16-bit Instruction Simulator
        - ZSP 16-bit Architecture Simulator
        - ZSP 16-bit EB402 Evaluation Board
    - PowerPC 32-bit Big Endian
        - Green Hills C/C++ Version 3.0

- Metaware High C/C++ Version 4.5
- WindRiver (DiabData) Version 5.0a
- Metrowerks (CodeWarrior) Version 6.0

6 Improvements and Changes Details

Release 1.1 Beta 1 was partly a joint development and testing project between ECL and ARM, Ltd. ARM's contribution was make sure that the TH Lite versions would work under a Simulator.

Release 1.1 Release Candidate / Beta 2 (this release) was based on defect reports submitted by various EEMBC members, as well as ECL's desire to correct very old defects/bugs and to make porting to Unix and Unix-like environments easier.

Release 1.1 Release Candidate / Beta 3 was based on feedback reports submitted by various EEMBC members.

The benchmark suite has been changed to support a build process using gnu make for certification. The source base builds both TH Lite, and TH regular versions of the benchmarks within the same directory structure.

The benchmark suite has been changed to support a consistent multiple IDE builds using uniquely named benchmark specific project files. These project files are located at sub-directories of each benchmark. IDE workspaces at both top level and application level support global build from the IDE environment.

This release comprises all benchmarks with modifications for consistency in directory structure, file naming, and cross-subcommittee harness changes intended to increase the efficiency of certification, while preserving the Subcommittee ownership of updates and packaging.

All subcommittee benchmarks now have the same TH Lite code. Existing Version 1.0 TH modifications made by subcommittees were carried forward in this release.

For each sub-committee, the TH Lite harness resides in the TH_Lite folder, and the TH Regular harness resides in the TH folder.

7 This is not Version 2

EEMBC/TechTAG (Technical Advisory Group) and ECL are working on EEMBC Version 2, which will include the Test Harness technology and general architectural

improvements found in this release. However, EEMBC Version 2 will contain all new kernels and bigger applications.

The one exception to this is that the 8/16 Bit Microcontroller Benchmark Suite is included. Technically, this is the first release of Version 2, but we felt it important to convert all benchmarks previously released.

One goal of this Release is to verify that this technology is suitable for EEMBC Version 2 deployment.

A key goal was to make sure that this version did not alter performance compared to the existing (or "old") EEMBC Version 1 by more than 3% for any individual kernel. Testing results indicate that we have met this target, and in fact in nearly all cases we are within about 1% of EEMBC Version 1.0. Moreover, the differences between Test Harness Regular and Test Harness Lite are also within 1-3%.

8 Background / History for Understanding this Release

The EEMBC benchmark suites are divided into Subcommittee Application Areas with separate "owners." Originally, the benchmarks themselves were developed by the Subcommittees, which had a Chairperson and Committee members. The Test Harness (what we now call TH Regular) was archictected by Alan R. Weiss when he worked for Motorola, and was developed by Richard G. Russell of Advanced Micro Devices (AMD). It is his implementation that has formed the basis for Version 1, and has proven to be some of the most portable embedded code around. [Test Harness Lite was first developed by EEMBC TechTAG, specifically Sergei Larin and Alan R. Weiss].

In all cases, the Subcommittees hired and managed contractors experienced in the separate application spaces to do the work. This resulted in fairly good algorithms, but almost a complete lack of architectural or code consistency. Furthermore, schedules were slipping dramatically for completing this work in 1998, so ECL volunteered to take on the task of "bringing the project home."

ECL applied successive waves of code cleanup and code improvements to the mass of software C code, and stabilized it enough to get it ready for shipment. The EEMBC Board of Directors decided that the exit criteria was:

- Successfully tested on big-endian and little-endian processors
- Successfully ported to 16 and 32 bit platforms
- No Severity 1 Defects

Note that an exit criterion was not "zero defects." In fact, ECL and EEMBC/TechTAG found over 30 defects that were not "Severity 1" defects, but the EEMBC Board of Directors at the Phoenix, Arizona meeting specifically prohibited TechTAG from fixing those defects – by that time, benchmark scores were coming out, and it was thought that it would alter the benchmark scores.

ECL has been supporting the existing code base for a few years now, and it has become apparent that with Simulators, VLIW DSP's, and other interesting real and virtual architectures the time was right, within the context of working on EEMBC Version 2, to radically overhaul Version 1 but minimize the performance differences between:

- EEMBC Version 1 and EEMBC Version 1.1 Test Harness Regular
- EEMBC Version 1.1 Test Harness Regular vs. EEMBC Version 1.1 Test Harness Lite

The file and directory standardization was done to reflect the dual nature of the EEMBC benchmarks for distribution, update, and now including certification. The current benchmarks reflect the Subcommittee ownership by packaging sets of applications, and test harnesses into a release package. Each subcommittee then makes changes to this package as needed. Over time the Subcommittee packages have evolved in different directions making it increasingly difficult to find a common theme, build process, or execution method that applies to the entire benchmark suite.

9 Directory Structure

There are 9 Visual C/C++ workspaces, and 98 Visual C/C++ projects included with this release comprising the entire Version 1 set of applications, datasets, and benchmarks. Many of the file and directory structures will be familiar from Version 1. A consistent naming and hierarchy convention was derived from reviewing all benchmarks in order to come up with a global build capability.

These conventions are described in this section.

The Top Level files of the tree, Test Harness Regular, Test Harness Lite, and Utilities are packaged separately from the Applications, and are downloaded from the Test Harness section of the EEMBC Web site (http://www.eembc.org).

Each Application contains a Visual C++ Workspace, and Project files to build both TH Regular and TH Lite executables within each benchmark. Empty Benchmarks were added to all Applications.

Combining one or more Applications with the Test Harness package is all that is needed to build and execute the benchmarks.

9.1 Top Level Directory Structure

In the following screen snapshot, the top-level directories of the EEMBC Benchmark Suite are shown. The Test Harness and Utility directories are expanded for discussion.

Test Harness Regular and Lite have the same directory structure. The directories and contents are:

The Functional Layer source code is contained in the src directory;

Each TOOLCHAIN defines a directory containing the makefile dependencies of the Harness (harness.mak). Two tool chains are defined, VC and Gcc.

Each PLATFORM defines a directory. One platform is defined, x86. The Platform directory contains the test harness application layer, and can optionally be use for script files or other platform specific files from porting.

The Application Layer source code is contained in the al directory of the Platform.

Standard Utilities for building and executing the benchmarks in the makefile environment are contained in util. The directories and contents are:

The awk directory contains scripts for processing benchmark results;

The doc directory contains documentation for the scripts;

The make directory contains the Tool chain definition files (vc.mak, Gcc.mak), the standard makefile, and the dependency file to generate dependencies with makerule (makerule.mak)

The Perl directory contains the makerule script for generating dependency files.

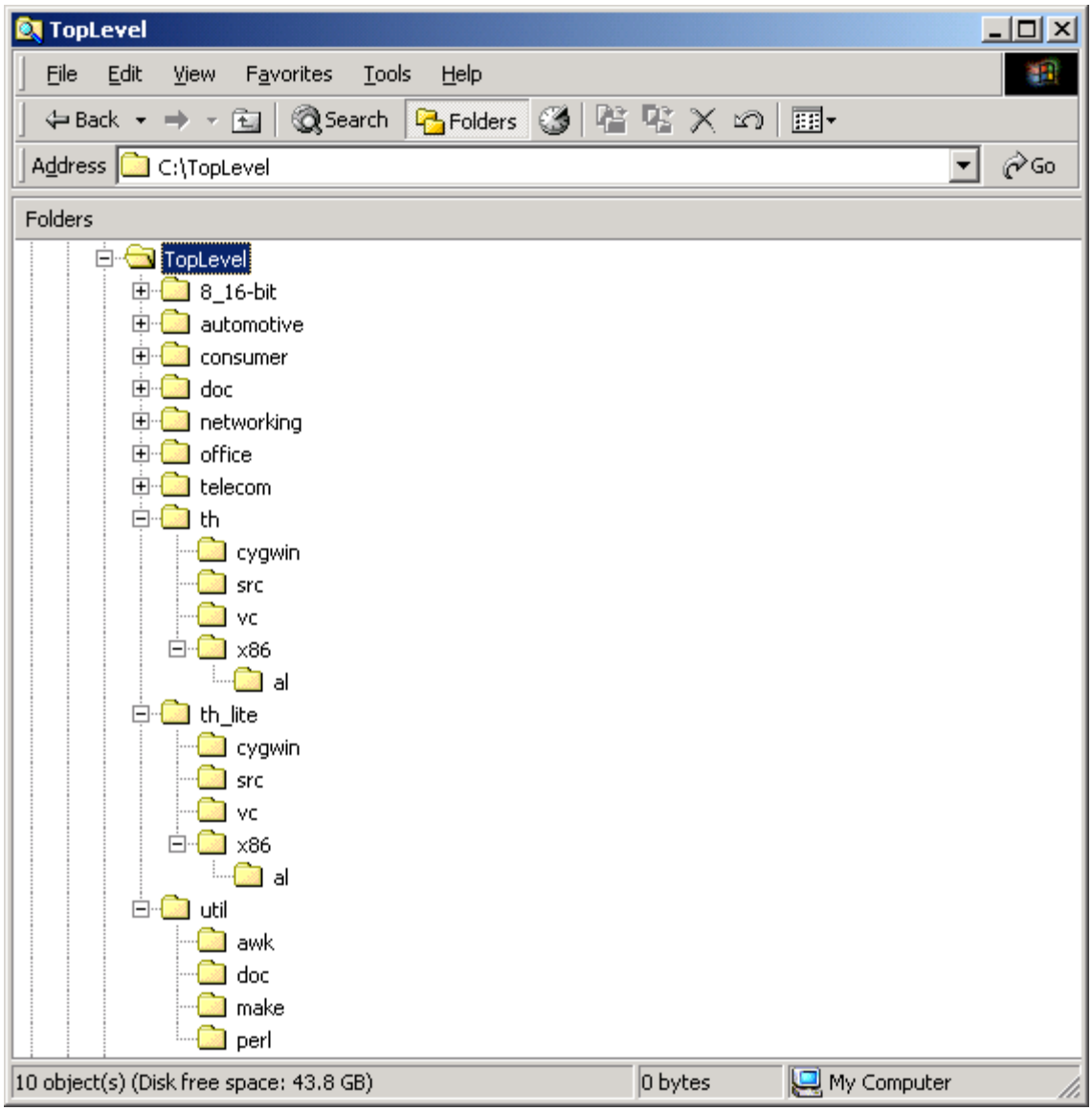

Figure 1 Top Level Directory Structure

9.2 Benchmark Directory Structure

In the following screen snapshot, the 8/16 Bit Microcontroller Benchmark Suite is shown containing the basic directory structure. The gcc Toolchain generates the `Gcc` directory and the Microsoft Visual C/C++ Toolchain generates the `vc` directory. Within each Toolchain root directory:

The obj and obj_lite directories contain object files from the compile step. There is a sub-directory for each benchmark kernel such as `bitmnp816`. The test harness objects common to all benchmarks reside at the top level;

The `bin` and `bin_lite` directories contain binaries produced from the link step;

The `results` and `results_lite` directories contain log files of each benchmark run, and the size output for each benchmark.

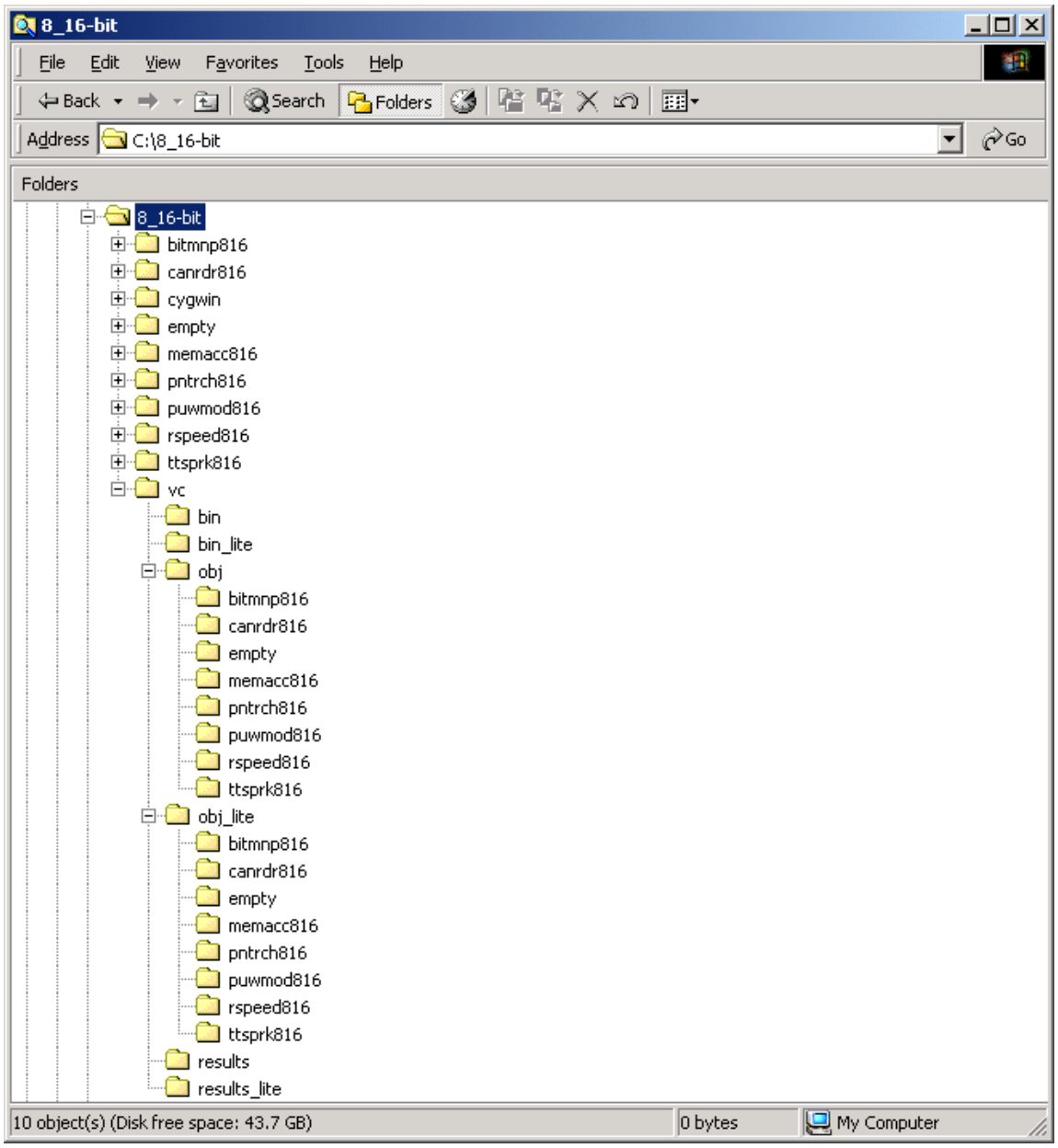

Figure 2 Benchmark Directory Structure

Finally, there is a new `util` directory containing such useful things as the awk and Perl scripts used by the new makefiles and log file extractor utilities.

There are still Win32 directories used by Microsoft Visual C/C++ (ignore the CVS directories – that is an artifact of our own machines. All EEMBC source code is managed by ECL using Concurrent Version System (CVS) so that we can do professional-grade software development). `DEBUG_LITE` and `RELEASE_LITE` are indicated in this example. That is where Visual C/C++ will put the built binaries – this is in a separate location than the `makefiles` use, however.

In the Automotive/Industrial benchmark suite, the familiar Version 1 kernels are present, along with the new `bin, bin_lite, empty`, and `util` directories we discussed earlier.

The `results` and `results_lite` directories contain the log files from benchmark runs. *The benchmark run step emits log files that contain results and size information! This is post-processed by various tools, and is available for your own analysis.*

In each benchmark suite, there will be a `UTIL, TH`, and `TH_LITE` directory containing the appropriate files. In `TH`, for example, it will contain the SRC and AL directories as well as the `EMPTY` benchmark.

The original directory structures for some benchmarks were designed with expansion in mind. In the cases where expansion hasn't taken place, there are additional directory levels that increase the complexity of builds.

10 File Naming

All benchmarks directories have a main program for both TH Regular, and TH Lite. The source code specific to a TH Regular harness will reside in a file called `bmark.c`. The source code specific to a TH Lite harness will reside in a file called `bmark_lite.c`.

11 The TCDef

The TCDef data structure is the main user interface for returning benchmark test data to the user, and passes the benchmark entry points to the Test Harness. In TH Regular TCDef is combined with a second Results structure. This is then printed on the display by th_report_results. Both TH Regular and TH Lite support th_report_results to allow consistent results extraction. TH Lite has a single control point for console display that allows this to be disabled for platforms with no display capability.

TH Lite uses a slightly different TCDef essentially combining the results structure into the main TCDef. The primary usage is for boards without display capability. The test may be run from a debugger, and at the end the memory location of the TCDef is used to gather the results.

The original TCDef for TH Lite did not contain the results field, nor was there an analogous results data structure. A CRC field was added for verification results, as passing a file back to the host is not part of this harness.

Several of the benchmarks have validation code to check output results (diffmeasure). In fact, diffmeasure has been incorporated into the Telecom benchmarks directly (in the LITE versions). Errors from these routines were passed to the TH Regular user by printing to the display. The results structure was mostly unused.

This was addressed by a coordinated change of the bmark and bmark_lite results code to use the results structure for validation errors, or diffmeasure results.

Concurrently the TH Lite TCDef was extended to handle the v1 – v4 fields of the TH Regular results. This allows the TH Lite user to view validation results from a debugger in a similar fashion to the CRC validation data.

The TCDef is defined in thlib.h as follows:

```
typedef struct TCDef{
   /*-----------------------------------
    * This section is the same
    * for all versions of this structure
    */
   char                     eembc_bm_id[ 16 ];/* id flag
*/
   char                     member[ 16 ];     /* the
member id */
   char                     processor[ 16 ];  /* the
processor id */
   char                     platform[ 16 ];   /* the
platform id */
   char                     desc[ 64 ];       /*
benchmark description */
   e_u16                revision;         /* The revision
of this structure. */
   /*-----------------------------------*/
   version_number       th_vnum_required;  /* TH Version
Required        */
   version_number       target_vnum_required;  /* Target
Hardware Version Required */
   version_number       bm_vnum;  /* the version of this
bench mark */
   e_u32                rec_iterations;
   e_u32                iterations;
   e_u32                duration;
   e_u16                CRC;
   size_t               v1;         /* Verification Data,
can be double via union */
   size_t               v2;
   size_t               v3;
   size_t               v4;
} TCDef;
```

The only change from the original Lite TCDef is the addition of v1-v4. These were done as `size_t` types to be compatible with the TH Regular usage of these fields.

An enhancement to Telecom was the incorporation of diffmeasure as a validation method. This is done within the bmark and bmark_lite routines, and so can be compiled out if code size is a concern on the target. The result of diffmeasure is a 64-bit floating-point number, or double. An option was added for loading floating-point values by paring v1-v2, and v3-v4. This is done with a union supplied in Telecom/diffmeasure/verify.h. The TH Lite harness display in VERIFY_FLOAT mode supports this as well by loading

two consecutive size_t locations with a double. This capability can be completely disabled for target platforms and Toolchains lacking floating point support.

```
/* Used to convert output into THResults */
      typedef union {
            double      d;
            size_t      v[2];
      } d_union;
```

In order to load the TCDef structure from bmark, a variable of type d_union is declared and used as follows:

```
        d_union         dunion;
    dunion.d = diffmeasure (golden_result, NumPoints, COMPLEX, OutData,
NumPoints, COMPLEX);
    tcdef->v1       = dunion.v[0];
    tcdef->v2       = dunion.v[1];
```

The display options are set by the following defines in thcfg.h.

```
/*-------------------------------------------------------------
--------------------
 * Display verification
 * VERIFY_INT - v1, v2, v3, v4 as size_t
 * VERIFY_FLOAT - union to double v1,v2 -> dv1, v3,v4 -> dv2
 *-----------------------------------------------------------------------*/
#define    VERIFY_FLOAT
#define VERIFY_INT
```

These are only applicable for display or log file output. The code is written so that any combination of these options may be used. This avoids benchmark specific thcfg.h files for a display only change.

> **NOTE:** When both VERIFY_FLOAT and VERIFY_INT are used, and a floating point number is loaded, the integer versions will be displayed, but they will not be valid numbers. The opposite case is true when valid integers are displayed as floating point.

12 TCDef Usage

> ***In the current code base, the TCDef for TH Lite, and TH Regular are different. This is preserved in this release, and the issue of unifying this structure is left for a Version 2.***

The 8-16 Bit benchmarks all have an internal CRC_CHECK defined. No other benchmarks use CRC checking. This type of CRC Checking has a required number of ITERATIONS for the test to succeed.

All Benchmarks have recommended iterations. This field is inconsistently defined. Most benchmarks define this within the TCDef structure. Some define this with an ITERATIONS ifdef.

A NON_INTRUSIVE_CRC_CHECK was developed as part of this. It is an Iteration independent measure of the output results. Unique algorithms to calculate this were required for each benchmark data type. Many only required a simple buffer scan. Others like Text required a tree walk to collect data values, and avoid pointers that can change in value from system to system.

Since all benchmarks have recommended Iterations, and some have a need to override this when CRC_CHECK is enabled, a standard IFDEF structure for ITERATIONS was developed, and now sets the TCDef value.

All TCDef's have been modified to support an ITERATIONS define for recommended iterations. Iterations may now be defined at compile time using the ITERATIONS flag, and required iterations for CRC_CHECKS are enforced by the ifdef structure defined below.

For TH Lite, CRC_CHECK, and NON_INTRUSIVE_CRC_CHECK are defined. CRC_CHECK is an iteration dependant CRC performed within the benchmark algorithm. NON_INTRUSIVE_CRC_CHECK is performed outside the benchmark algorithm.

The standard define chosen for the Expected CRC value is: EXPECTED_CRC. Another name can be chosen, this one was the most common, and the goal in this release is a single name for Expected CRC value.

Each TCDef has an appropriate set of defaults that allow compiler overrides. The following standards handle the overrides by setting recommended or required ITERATIONS, and the correct Expected CRC value BITMAN_CRC.

12.1 Standard for 8-16 Bit Benchmarks

8-16 Bit benchmarks are the only ones implemented with an internal CRC Check that is intrusive (the other TH LITE versions of the benchmarks have non-intrusive checks). This check is dependant upon the number of iterations, which is different than the recommended benchmark iterations.

In Beta 2, consistent form of setting iterations, and pre-calculated CRC values is provided for all benchmarks.

12.2 Standard for TH Regular

All `bmark.c` files contain this definition supporting iterations. An individual benchmark may be set by using the file `iterations_$(PLATFORM).mak`. The `CRC_CHECK` option is not used in Regular, but is included for consistency and future integration. In all cases, the required and recommended iterations are the same.

```
/* Define iterations */
#if !defined(ITERATIONS) || CRC_CHECK ||
```

```
ITERATIONS==DEFAULT
#undef ITERATIONS
#if CRC_CHECK
#define ITERATIONS 5000 /* required iterations for crc */
#else
#define ITERATIONS 5000 /* recommended iterations for
benchmark */
#endif
#endif
```

The TCDef definition then uses ITERATIONS to place the resulting value in the recommended iterations field.

12.3 Standard for TH Lite

All `bmark_lite.c` files contain this definition supporting iterations and crc checking. An individual benchmark may be set by using the file `iterations_$(PLATFORM).mak`.

`If t`he `CRC_CHECK` option is enabled, the required iterations is always used. If a benchmark does not support a particular crc check, the `EXPECTED_CRC` is defined to be 0.

```
/* Define iterations */
#if !defined(ITERATIONS) || CRC_CHECK ||
ITERATIONS==DEFAULT
#undef ITERATIONS
#if CRC_CHECK
#define ITERATIONS 10000    /* required iterations for crc
*/
#else
#define ITERATIONS 10000    /* recommended iterations for
benchmark */
#endif
#endif

#if CRC_CHECK
#define EXPECTED_CRC 0x0000
#elif NON_INTRUSIVE_CRC_CHECK
#define EXPECTED_CRC 0xbeef
#else
#define EXPECTED_CRC 0x0000
#endif
```

The TCDef definition then uses ITERATIONS to place the resulting value in the recommended iterations field.

13 TH Lite API

The original TH Lite API was expanded to avoid code changes from the original bmark routines, and to avoid static allocation of data previously allocated via th_malloc.

> ***There were problems with several benchmarks when static allocation was used. Fixing these dependencies would have altered the benchmark code, and therefore the harness was modified.***

The HEAP operation was also modified so that the heap is initialized on the first malloc() call. The previous call to mem_heap_initialize() is still supported, but is not needed in the TH Lite bmark code. This better reflects the operation of compiler supplied malloc routines.

There are also TH routines called within the benchmark code that would have needed deletion for TH Lite. This would also alter benchmark code, so stub routines were added to avoid these changes and more closely duplicate TH Regular performance.

Including thlib.h allows access to the API. Additionally eembc_dt.h, and thcfg.h are included from thlib.h allowing access to platform specific data types and porting defines from a single header file include. The API is defined as follows:

```
/*-------------------------------------------------------
--------------------
 * The Test Harness Lite API
 */
/* th_malloc automatically calls mem_heap_initialize if it
hasn't been called yet */
#define th_malloc( size ) th_malloc_x( size, __FILE__,
__LINE__ )
void *th_malloc_x( size_t size, const char *file, int line
);
#define th_free( blk ) th_free_x( blk, __FILE__, __LINE__ )
void    th_free_x( void *blk, const char *file, int line );
void    th_heap_reset( void );
void mem_heap_initialize(void);

/* Timer Routines */
void   th_signal_start( void );
e_u32 th_signal_finished( void );
int th_timer_available( void );
int th_timer_is_intrusive( void );
size_t th_ticks_per_sec( void );
size_t th_tick_granularity( void );

/* System Routines */
```

```
void   th_exit( int exit_code, const char *fmt, ... );
int th_report_results(TCDef *tcdef, e_u16 Expected_CRC );

/* CRC Utilities */
#if  CRC_CHECK || NON_INTRUSIVE_CRC_CHECK
e_u16 Calc_crc8(e_u8 data, e_u16 crc );
e_u16 Calc_crc16( e_u16 data, e_u16 crc );
e_u16 Calc_crc32( e_u32 data, e_u16 crc );
#endif

/* Display control */
int th_printf( const char *fmt, ... );
int th_sprintf( char *str, const char *fmt, ... );

/* Benchmark source code compatibility stubs */
int th_harness_poll( void );
```

New timer routines were added in Beta 2 for compatibility with TH Regular.

14 Heap Management

The original TH Lite included heap management and allocated 4Mb statically for the heap. The interface to th_malloc() and th_free()were not implemented so this code was not usable. This was fixed in the release.

Adding a statically allocated “heap” in this fashion dramatically increases the code size. To use the same TH Lite harness across all applications requires the ability to adapt this heap manager. Including the heap does not impact reported code size, as it is subtracted with the empty benchmark, but it can impact the ability to run the benchmarks on some targets.

The allocated heap used to require the compiler or loader to initialize the area to 0. This initialization has been removed as it can cause 4Mb executables to be created under some compilers, gcc for example. This does not impair the operation of any current benchmark.

Two examples are given. The first example eliminates the heap. The second example expands the heap. This covers the range of heap management required by the benchmark code.

Networking pktflow requires more than 4Mb of heap storage. An AL directory was added to pktflow to raise this to 5Mb for that benchmark only using heapport.h in the TH Lite AL directory. TH Regular does not require this override.

An additional option was added to use compiler supported malloc if available. This option combined with the COMPILE_OUT_HEAP option removes the 4Mb array allocation of the heap manager, as well as allows th_malloc to be used if needed.

```
/*-------------------------------------------------------
------------------
 * Set this define to TRUE to completely 'compile out' the
memory
 * allocation routines.  This saves code and data space.
 * You may accomplish the same in code that doesn't call
malloc by not linking
 * heap.c,
 * BOTH TRUE OR BOTH FALSE, to support malloc
 * COMPILE_OUT_HEAP TRUE, HAVE_MALLOC_H FALSE, minimize
code size.
 *-------------------------------------------------------
------------------*/
#if !defined ( COMPILE_OUT_HEAP )
#define COMPILE_OUT_HEAP (TRUE)
#endif
#if !defined ( HAVE_MALLOC_H )
#define HAVE_MALLOC_H (TRUE)
#endif
```

The code added by compiler generated malloc() is small, and can be subtracted out in the Empty_Benchmark.

15 TH Lite Adaptation Layer API

The original TH Lite included `thlib.c` as the adaptation layer, removing all references to `thal.c`. Many of the routines in `thlib.c` should be common for all benchmarks, and it was unclear from the interface that could be changed by the user. Thlib was moved back to the Functional layer, and a subset of `Thal.c` routines necessary for porting were restored.

TH Lite does not implement an external command interpreter, so it is possible to use a simpler exit method. The exit routine was changed from TH Regular setjmp to the use of the ANSI C exit routine. The exit code is returned to the system.

```
   /*-------------------------------------------------
------------------------
    * The Test Harness Lite Application Layer API
    */
   size_t al_ticks( void );
   size_t al_ticks_per_sec( void );
   size_t al_tick_granularity( void );
   void   al_signal_start( void );
   size_t al_signal_finished( void );
   void   al_exit( int exit_code);
   int      al_printf(const char *fmt,  va_list args);
   void     al_report_results( void );
```

The standard TH Regular exit codes are implemented by the addition of `therror.h` to the functional layer.

16 Header Comments

There are two similar header blocks used throughout the code base. The first was a long copyright message, and scattered use of file descriptions or other information about the source code. The second included file descriptions generated by MKS, which is no longer used. A third variation in a few files includes lengthy descriptions of the algorithm and reference material.

The header blocks in general have not been kept up to date. The original author material was valid, however the change history was not.

1) A composite of the header block themes was put together and incorporated into a new header block.

2) The original comment block fields were preserved, unless replaced by one of the following changes.

3) The long copyright message was requested over the short one.

4) The EEMBC field was added to denote the entity within EEMBC that owns the code.

5) TechTAG was listed in a few places as the Harness owner, so this was applied to the TH Lite code.

6) CVS generated fields are used in the comment to generate filename, last change author, date, and history.

7) Original authors, when available, were added as an AUTHOR field, outside of CVS generation.

8) DESC was preserved but limited to a single line, with other text moved to the NOTES field.

9) Documentation can also participate if it is stored as text. RTF is portable, and you can see the results of CVS variable updates.

```
/*=================================================================
=========
 *$RCSfile: releasenotes.rtf,v $
 *
 *   DESC : Test Harness Library Interface
 *
 *  EEMBC : EEMBC Technical Advisory Group (TechTAG)
 *
 * AUTHOR : ARM Ltd., ECL, LLC
 *
 *    CVS : $Revision: 1.18 $
 *          $Date: 2002/10/02 18:19:33 $
 *          $Author: rick $
 *          $Source: d:/cvs/eembc2/doc/releasenotes.rtf,v $
 *
 * NOTE   :
 *          This header file contains the interface functions and Data
 *          Structures for the Test Harness Library, which implements
 *          The API.
 *
 *-----------------------------------------------------------------
---------
 *
 * HISTORY :
 *
 * $Log: releasenotes.rtf,v $* Revision 1.18  2002/10/02 18:19:33
rick* Comment POSIX CLOCKS_PER_SEC effect on duration** Revision 1.17
2002/10/01 19:10:07  rick* Updates to build procedure for final
release** Revision 1.16  2002/09/25 21:38:13  rick* Changes from Adrian
Wise (siroyan)** Revision 1.15  2002/08/16 18:58:52  rick* Add unix
notes** Revision 1.13  2002/08/09 22:40:28  rick* Updated by Alan for
content/formatting** Revision 1.12  2002/08/09 22:20:48  rick* Beta 3
updates** Revision 1.11  2002/07/30 20:34:09  rick* Beta 2b final*
```

```
 *
 * Revision 1.9  2002/07/19 23:11:37  rick
 * Add notes for failure modes
 *
 * Revision 1.8  2002/07/18 18:59:43  rick
 * Make iterations.mak a dependency to each benchmark
 *
 * Revision 1.7  2002/07/17 22:14:03  rick
 * Beta 3 consolidated fixes
 *
 * Revision 1.6  2002/07/12 19:43:02  rick
 * Add support for Null Platform
 *
 * Revision 1.5  2002/07/11 14:26:26  rick
 * Bug 190
 *r the Test Harness Library, which implements
 *          The API.
 *
 *-------------------------------------------------------------------------
---------
* Copyright (c) 1998-2002 by the EDN Embedded Microprocessor
 * Benchmark Consortium (EEMBC), Inc.
 *
 * All Rights Reserved. This is licensed program product and
 * is owned by EEMBC. The Licensee understands and agrees that the
 * Benchmarks licensed by EEMBC hereunder (including methods or
concepts
 * utilized therein) contain certain information that is confidential
 * and proprietary which the Licensee expressly agrees to retain in the
 * strictest confidence and to use only in conjunction with the
Benchmarks
 * pursuant to the terms of this Agreement. The Licensee further agrees
 * to keep the source code and all related documentation confidential
and
 * not to disclose such source code and/or related documentation to any
 * third party. The Licensee and any READER of this code is subject to
 * either the EEMBC Member License Agreement and/or the EEMBC Licensee
 * Agreement.
 * TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, EEMBC DISCLAIMS
ALL
 * WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED
TO,
 * IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR
 * PURPOSE, WITH REGARD TO THE BENCHMARKS AND THE ACCOMPANYING
 * DOCUMENTATION. LICENSEE ACKNOWLEDGES AND AGREES THAT THERE ARE NO
 * WARRANTIES, GUARANTIES, CONDITIONS, COVENANTS, OR REPRESENTATIONS BY
 * EEMBC AS TO MARKETABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR
OTHER
 * ATTRIBUTES, WHETHER EXPRESS OR IMPLIED (IN LAW OR IN FACT), ORAL OR
 * WRITTEN.
 *
 * Licensee hereby agrees by accessing this source code that all
benchmark
 * scores related to this code must be certified by ECL prior to
publication
 * in any media, form, distribution, or other means of conveyance of
 * information subject to the terms of the EEMBC Member License
```

```
Agreement
 * and/or EEMBC Licensee Agreement.
 *
 * Other Copyright Notice (if any):
 *
 * For conditions of distribution and use, see the accompanying README
file.
 *
=========================================================================
====*/
```

Comment Style

The commenting style used both ANSI (/* */), and the non-ANSI line comment (//). In most cases the line comment was used for pretty printing within the ANSI comment. The result is a nested comment. For example:

```
/*************************
// Local Variables
*************************/
```

In Office Automation, a makefile in the build directory was previously used to build under gcc with ANSI warning messages. As part of porting, this makefile was fixed, and run on the benchmarks. The benchmarks, and TH Lite generated compiler **errors** under this because of the nested comments.

Global changes comment style replacements were made to over 30 files which involved replacing "//" with " *". In almost every case, an average of 60 changes occurred, with no compiler errors from eliminating correctly used line comments.

The nested comments that included special characters (i.e. "$") caused gcc errors. Line comments under gcc ANSI are just warnings.

The use of nested comments should be eliminated. The recommended approach is to do a global replace of "//" with " *".

```
/*******************
 * Local Variables
 *******************/
```

16.1 Utility Header

All source code uses the full header. A number of small make utility level files are now involved where the size of the standard header is larger than the original files. Since these files are used in porting, and the development history is not relevant, a shorter utility header was developed to simplify editing. This header is as follows:

```
#=========================================================================
========
```

```
#$RCSfile: releasenotes.rtf,v $
#
#   DESC : Build Script
#
# AUTHOR : Rick Foos, ECL, LLC
#
#  EEMBC : Technical Advisory Group (TechTAG)
#
#    CVS : $Revision: 1.18 $
#-------------------------------------------------------------------------
--------
# Copyright (c) 1998-2002 by the EDN Embedded Microprocessor
# Benchmark Consortium (EEMBC), Inc. All Rights Reserved.
#=========================================================================
========
```

17 Display Output

Traditionally, TH Lite was run from a debugger, and produced no output to the display. This mechanism is supported, and enhanced by a new Adaptation Layer routine al_printf(), and a th_report_results() function. The parameters to th_report_results() have been changed for TH Lite.

The output used in TH Regular is standard out via printf statements. A local printf engine is used rather than the compiler supplied printf. In TH Lite the local printf engine is not included. The compiler supplied vprintf or vfprintf may be used by supplying adding a call to al_printf(). Note that the arguments are passed from the harness in vprintf format, **not** printf.

An example that prints all th_printf() output to stderr follows:

```
/*-------------------------------------------------------
---------------------
 * FUNC   : al_exit
 *
 * DESC   : The traditional vprintf, control output from
here.
 *
 *
 * PORTING: Standard C in stdio.h
 * -------------------------------------------------------
-------------------*/
int         al_printf(const char *fmt, va_list  args)
{
      return      vfprintf(stderr,fmt,args);
}
```

18 Makerule User Guide

A new utility called makerule is used to generate dependency files for all benchmarks, and the test harness. This is an integrated part of the make procedure. It is optional, and can be disabled by editing the makefile, and commenting out the line: "include $(TH)/$(TARGETS)/harness.mak"

Makerule can be used to generate new dependency files when files are added to benchmarks for optimized certifications, or new harness files as part of the porting process. It supports the options and formats of the build process, and generates dependencies for both C and assembly files which use the #include directive. Other source files with different include directives are supported, but the dependencies for header files may not be generated.

The files generated by makerule can be hand edited without triggering makerule generation for final customization. The following documentation is in POD format, and also found in util/doc/makerule.html.

18.1 NAME

makerule.pl - generates make file rules (dependency info) for C,C++, and assembly language files with # comments.

18.2 SYNOPSIS

makerule <switches> <file specs>

Switch	Option	Description
-?		Dump usage help then exit (-help also works)
-i	<directory>	Adds a directory to MAKERULE's include path.
-ix		Clears (empties) MAKERULE's include path.
-p	<env variable>	Adds the contents of an environment variable to MAKERULE's include path. For example, when using Microsoft compilers, the INCLUDE environment variable contains the include path used by the compiler. Just use the -p switch to use it.
-v		Turns on verbose mode messages.
-v-		Turns off verbose mode messages.

-q		Enables quiet mode. When being quiet, MAKERULE only prints errors to STDERR.
-q-		Turns the quiet mode off.
-s		Enables the processing of '#include <stdlib.h>' include statements. Normally, MAKERULE ignores 'angle bracket' include statements.
-s-		Turns off processing angle bracket include statements.
-cq		Place double quotes around obj or exe output file options to the compiler.
-cq-		Disables quoting.
-co	<obj output>	Enables generation of object file output statements in the rule generation. Used for compilers that do not support $* directives. The obj output string is used to identify the output. It is usually -o, but for VC++ it is -Fo. If using VC++ from gnu make, the quoted string option should also be enabled -q.
-cox		Disables generation of object file output statements.
-ce	<exe output>	Enables generation of executable file output statements in the rule generation. Used for compilers that do not support $* directives. The obj output string is used to identify the output. It is usually -o. If using VC++ from gnu make, the quoted string option should also be enabled -q.
-cex		Disables generation of executable file output statements.
-r	<recipe>	Sets the recipe to be placed after each object rule. **NOTE:** If -ce is enabled, the recipe will have the dependant file appended, so $* extensions cannot be used.
-rx		Clears the current recipe.
-tr	<recipe>	Sets the recipe to be placed after each target rule. NOTE: the recipe will have the dependant file appended, so $^ extensions are not needed.
-trx		Clears the current target recipe.

-ta	<dependency>	Adds additional dependency to target. For example, -ta $(THOBJS) causes each benchmark to be rebuilt when the harness is changed.
-tax		Clears the additional dependency targets.
-te	<library>	Adds additional dependency library to the end of the target rule. For example, -te $(THLIB) so that harness can be linked as a library, with $(THOBJS) as a dependency list for each benchmark.
-tex		Clears the additional dependency library.
-b	<extension>	Sets the file extension of the target file in the generated rules. For example '-b .obj' or '-b .o'. The default is '.obj'.
-tb	<extension>	Sets the file extension of the target executable file in the generated rules. For example '-tb .exe' for Windows executables, and '-tb .lib' for Windows Libraries. The default is ''.
-o	<directory>	Sets the target directory used to generate each rule. For example, on NT systems it is common to put objects in a 'debug' or 'release' subdirectory. Using '-o debug' cause targets such as 'debug/foo.obj : foo.c' to be generated. The default is an empty string.
-ox		Clears the current target directory.
-to	<directory>	Sets the target executable directory used to generate each rule. For example, it is common to put executables in a bin directory. Using '-to bin' cause targets such as 'bin/foos.exe : foos.o' to be generated. The default is an empty string.
-tox		Clears the current target executable directory.
-g	<file name>	Ignore the specified include, or source file. If MAKERULE finds this file, it will be ignored. You can have as many -g switches as you like. MAKERULE keeps a list of files to ignore.
-g-	<file name>	Remove the file name from the ignore list.
-gx		Clear the entire ignore file list.

-j	<expression>	Ignore the files defined by a regular expression. If MAKERULE finds this file, it will be ignored. You can have as many -j switches as you like. MAKERULE keeps a list of regular expressions used to ignore files.
-j-	<expression>	Remove the regular expression from the ignore list.
-jx		Clear the entire ignore regular expression file list.
-w	<directory>	Ignore the specified include directory. MAKERULE will NOT search this directory for include files. You can have as many -w switches as you like. MAKERULE keeps a list of directories to ignore.
-w-	<directory>	Removes the directory from the ignore list.
-wx		Clears the ignore directories list.
-m	<number>	Sets the right margin. This defaults to 72 but you can set it is small as 60 (which is the minimum).
-s		Turns the STRICT flag on. When STRICT is set, then MAKERULE exits with an error if it can't find an include file, or if a directory on the include path does not exist.
-s-		Turns strict mode OFF.
-t	<dependency>	Sets the current target. If there is a current target, then all object file's are associated with only that target.
-t-		Clears the current target.
-tx		Clears the target list.
-td		Dumps the object file list for the current target and clears the current target.
-tdx		Dumps the object file list for the current target and clears the ALL targets. This is used to handle duplicate target names in a build.
-tf	<file name>	Dumps the current target to the specified file.
-tfx		Resets the output file name to STDOUT.

-cmd	<file name>	Gets more input arguments from a file. Command files can be nested.
-debug		Turns on DEBUG mode. This also turns on verbose mode. If you have problems figuring out what MAKERULE is doing, then turn on the debug mode. You may want to redirect STDERR to a file so you can read all of it. Under the korn shell, or Windows, this is done like this: 'makerule 2> err >makerule.dep'. The rules go to the file 'makerule.dep' and all the output to STDERR goes to the file 'err'.

18.3 DESCRIPTION

MAKERULE automagically generates dependency rules for C, C++ and assembly language files using #include. These rules can then be included in a make file.

MAKERULE needs to basic pieces of information:

```
   #1 A list of source files to scan

   #2 A list of include directories in which to look for

      include files.
```

Given this simple information, MAKERULE will generate rules like this:

```
obj/app.obj : ../xvutil/assert.h ../platform/am186/DEF186.H \

              ../platform/am186/in186.h ../platform/am186/tmr186.h \

              ../platform/am186/am186leds.h
../platform/am186/am186ser.h \

              ../platform/pio.h ../common/compiler.h \

              ../platform/am186/printf.h ../common/stdinc.h\

              C:/MSVC/INCLUDE/stddef.h ../platform/am186/ptypes.h \

              ../common/serialx.h ../common/serial.H
C:/MSVC/INCLUDE/stdarg.h  \
```

```
                ../common/printfe.h

obj/app.obj : ../app/app.c

        $(COM) ../app/app.c


APP = \

        app.obj


app:

app.exe : app.obj

        $(CC) -o $(APP) app.exe


targets:: app
```

MAKERULE generates human readable output which can be directly included in a make file.

18.3.1 Using MAKERULE

Makerule generates make rules for object files and executable targets. Multiple targets are supported. The normal operation is to define a target, and a set of source files. At the end of each target, the object files processed will be placed in a make variable, and an executable line built.

At the end of processing, a list of the generated targets will be placed in a target rule (target:: t1...tn).

MAKERULE must be run from the same directory that you run your make file. This is because MAKERULE puts full or relative path names in its output. If you don't run them from the same directory, then make won't look in the right places.

Command line arguments and switches control #2 Makerule. All 'unswitched' arguments are taken to be either directories or file names. They can contain wild cards. For example '*.c' or '*.cpp'.

Switches that require arguments may be followed by a space such as

```
    -I c:\msvc\include
```

or, you can stick the together like this:

```
-Ic:\msvc\include.
```

Both the above examples add the 'c:\msvc\include' directory to MAKERULE's include path.

Note that all command line arguments are *NOT* case sensitive. I hate case sensitive command line switches. So '-I' and '-i' are handled the same way.

A simple makerule execution might look like this:

```
makerule -I c:\msvc\include *.c > deps.mak
```

This simple command will generate a set of rules for all the C files in the current directory. Note that the output goes to STDOUT which, in this case, is redirected to the file 'deps.mak'.

The key thing to remember about MAKERULE is that it processes the command line in order from left to right. So, put -I switches before file specs.

18.3.2 Important Notes

By default MAKERULE makes a best effort at findign all the include file dependencies. However, there are a few things you should know.

#1 By default, MAKERULE ignores #include <....> statements. MAKERULE assumes that angle-bracket includes are only used for libraries such as the standard library, or other libraries. e.g. they are used for code that you will NOT be changing during development.

```
    However, not all folks use angle-bracket includes this way.  So, if
you

    want to have MAKERULE search for include files specified using

    angle-bracket includes, then use the '-a' switch.  Note that you
can

    turn this back off using the '-a-' switch.
```

#2 By default, MAKERULE keeps going if it can't find an include file. So, if you have an #include ``foo.h'' and 'foo.h' cannot be found in the source file's directory or in the

include path, then MAKERULE just skips this file and goes on. E.g. it is not included as a dependency in the rule for the source file that tried to include it.

```
    This could happen for several reasons, most likely it will be an

    legitimate include path issue.  For example, you might want to use
the

    '-a' switch to process angle-bracket include statements but you
don't

    want to include stuff from your compiler's standard library.  So
you

    will leave the compilers include directory (like 'C:\MSVC\INCLUDE'
)

    out of the include path MAKERULE uses to generate its rules.  This
is

    a perfectly fine thing to do.


    BUT! If you want to makes sure MAKERULE doesn't miss anything then

    turn on STRICT mode by using the '-s' switch.  When the strict mode

    is set, then MAKERULE will exit with an error if it can't find the

    specified directory, source file or include file.
```

#3 Sometimes you *want* MAKERULE to ignore header files, even if you have strict mode enabled. The -g switch does this. '-g <fn>' adds a file name to a list of files to ignore. '-g- <fn>' removes a file name from the ignore list. '-gx' clears the entire ignore file list.

#3 Sometimes you *want* MAKERULE to ignore certain directories in the include path. For example, a setup shell script (or batch file) may setup an INCLUDE environment variable. You would like to use it, but there is one directory in the compiler's path that you don't want to search. The -w switch does this. This switch can also be used to ignore directories that do not exist, thus by-passing the strict mode check for that include directory. '-w <dn>' adds a directory to a list of include directories to ignore. '-w- <dn>' removes directory name from the ignore list. '-wx' clears the entire ignore directory list.

18.3.3 Command Files

What would a utility be if it couldn't be run from a command file??? Of course MAKERULE can read command line arguments from a command file. Just put a

@<filename> on the makerule command line. There can be more than one and command files can be nested. There is no limit to the nesting level.

By convention, I name command files using a '.cml' extension. MAKERULE doesn't care though.

Command files can also contain comments. These are comments just like in a makfile. Everything between a '#' and the end of the line is a comment.

Here is an example MAKERULE command file:

```
# MAKERULE Command File
#
# This entire file is tokenized and logically inserted into the command
# line where the @ character is

-a              # include #include <...> statements
-o obj/         # targets go in the obj directory

# use this include directories.
#
-I C:/rogue/workspaces/WINNT4/MSVC50/8s
-I c:/coding/am186tpl/common
-I c:/coding/am186tpl/xvkernel

-P INCLUDE       # also include the INCLUDE path in the environment

-r "$(COM)"      # C files are built with this recipe

source\*.c       # build all the files in the source directory
```

```
-ix              # clear the include path

-i source        # asm files don't need a fancy include path

-r "$(ASM) "     # use this recipe to build ASM files

source/*.asm     # build all the ASM files MAKERULE can find
```

18.4 Future Work Being Considered

- Use File::Find to locate files.
- Use GetLongOptions to parse. Will need to replace streaming the cml file in through ARGV to do it.
- Use a better C comment parser, or -M option in modern compilers.
- allow multi-line recipes to be placed in MAKERULE command files. I'll probably use something like 'here' documents.

19 Release Notes Addenda

- All TH_Lite benchmarks include CRC checking. There are two types of CRC checking. The CRC_CHECK mode performs intrusive CRC checking within the benchmark, and effects timing. This is used in 8-16 Bit benchmarks. The NON_INTRUSIVE_CRC_CHECK performs the CRC checking outside the benchmark code. All benchmarks other than 8-16 bit have new Non-Intrusive CRC Checking code.

- The benchmark algorithms are unchanged in this release. Some TH Regular benchmarks were modified outside the start/stop time for consistent error reporting.

- The Networking TH, enhanced version 1, was modified to fix bugs in the win32 version. It now correctly runs under Visual C++. All Networking benchmarks build and execute under both TH Regular and TH Lite on a Windows platform.

- The Bezier benchmark was repackaged and modified to correct the compiler optimization problems that have made this invalid under a growing number of tool chains. This requires testing on several tool chains that previously exposed the problem.

- Many compiler warnings were removed. Those that required code changes within the benchmark algorithm were avoided. A change in the algorithm code would alter the benchmark scores, especially in optimized code.

- The Networking TH, enhanced version 1, was modified to fix bugs in the win32 version. It now correctly runs under Visual C++. All Networking benchmarks build and execute under both TH Regular and TH Lite on a Windows platform.

- fprintf() removed from benchmarks. Replaced by th_printf(), and new thal call to al_printf giving user control of display output.

- Version 0.4 is an interim release of ITERATIONS, CRC_CHECK, and NON_INTRUSIVE_CRC_CHECK consistency, and directory renaming.

- A number of #if, #endif, directives did not begin in column one. This caused some problems with a compiler, and the harness and all benchmark code has #if structures starting at column one.

- All source code is in CVS at ECL.

20 Installation

The final version of EEMBC Version 1.1 will include InstallShield programs for each Subcommittee for use in installations on Windows platforms. **This is not included in this Release Candidate.**

This Release set consists of zip files for each subcommittee, and a top-level zip file stored in the test harness area. All benchmarks can be built and executed on the Microsoft Visual C/C++ IDE using the makeworld workspace, with manual execution. All benchmarks can be built and executed with Cygwin, and Visual C++ using the makefiles.

Getting Started

The following steps take you from download to executing benchmarks.

1. If you want to use the makefile system (rather than the Visual C/C++ Project Files and IDE), install cygwin from http://www.cygwin.com. ECL will not support you if you use MKS Tools, although we suspect that the awk, perl, etc. stuff will work fine. Note that we use gawk(), not awk, and gcc if you want to do this. Basically, if you have a Linux or other Unix environment, you will probably need to modify the makefiles for your world – but you already knew that, we're sure. <smile>

2. Unzip your application benchmark suite (e.g. "Consumer"). For Unix systems, use `unzip -a` to extract the correct end of line character.

3. Change all permissions so that read-only is not set on each file. You may or may not have to do this, depending upon whether or not your EEMBC Representative burned you a CD-ROM, or you go the code from the EEMBC website directly. But its good to check this common source of frustration.

4. If you have cygwin installed (GNU tools), try a top down build: type **`make`** <RETURN> You may have to edit the makefile to support your compiler name, linker name, location of files, etc. Review the `util/make/gcc.mak` file. This will describe how to install a new tool chain.

5. If you have Visual C/C++, go to the top Workspace File within an application, double-click on it to open it up, and use BATCH BUILD.

6. If you are porting to a real embedded platform, as you should, modify the AL directories (follow the comments in the code itself). ECL and TechTAG is looking for specific feedback to help improve our porting guide in this regard.

20.1 Download

To download the release:

Go to http://eembc.org/membership and log in.

Create a new directory to hold the zip files.

Download the EEMBC Version 1.1 Beta 2 zip files to a new directory.

If your company membership doesn't have access to one of the applications, it won't let you download the file. That's OK. All or part will work, instructions follow for this case.

On a Windows machine, you will need Cygwin installed (http://www.cygwin.com), and Microsoft Visual C++.

20.2 Install

To install Version 1.1, make sure all the zip files are in the new directory, and unzip all release files.

To do this, open a DOS windoow, or on unix a shell. Navigate to your zip file directory and type:

```
ls *.zip | xargs -n 1 unzip
```

This will unpack all the zip files, and for windows users verify that Cygwin is installed. Cygwin users may need to add the \cygwin\bin directory to your default path (i.e. set path=c:\cygwin\bin;%path%).

All DOS batch files, have Unix equivalents.

Edit the forall.bat (unix forall) file and delete the applications that you do not have from the "for" strings:

for example:

```
@for %%i in (8_16-bit automotive consumer networking office
telecom) do pushd %...
```

to

```
@for %%i in (automotive consumer networking office telecom)
do pushd %...
```

The Unix `forall` script has a single variable to edit:

applications="8_16-bit automotive consumer networking office telecom"

20.3 Verify

You can verify your installation by building and executing the benchmarks.

20.3.1 Microsoft Visual C/C++ IDE

To verify your installation with an IDE, start Microsoft Visual C++, and open the workspace named `makeworld.dsw` at the root of the installation. Project files from applications that are not installed can be removed from the workspace when you open it the first time.

Select Build -> Batch Build. Press the build button and the entire release will be built.

> **Note**: If all the project checkboxes are empty, select build, and then select batch build again. The next time batch build is invoked all projects will be selected.

Next, set the active project to the benchmark of interest. Press the exclamation point menu button to execute the benchmark. A DOS window will be launched, and the normal benchmark operation is underway.

The modified TH Lite displays text output in `WINDOWS_EXAMPLE_CODE` builds. There is no longer a need to run the debugger to view TH Lite results. The display is routed to stderr, and may be captured by a script file.

All TH Lite executables can be run via batch or script file. TH Regular executables cannot accept the required keyboard input via a file, so they must be run manually.

All Visual C++ build files reside in a win32 subdirectory in each benchmark. All executables reside under win32 in the following folders:

- Debug – The TH Regular Debug Version

- Debug_Lite – The TH Lite Debug Version
- Release – The TH Regular Release Version
- Release_Lite – The TH Lite Release Version

20.3.2 Cygwin and Visual C++ Makefile

For Visual C++ from a makefile, you need to make sure your DOS environment variables are set. Within each application there is a batch file called:

```
util/make/vcvars32.bat
```

Make sure that this points to the vcvars32.bat from your Visual C++ installation.

If you don't want to use Visual C, edit the `makereg.bat` and `makelite.bat` files. Remove the lines with 'make TOOLCHAIN= vc'. When you add your Toolchain, edit these files to add the new option.

For Visual C++ IDE, open the `makeworld` workspace at the top directory, and "Build ->batch build" will compile the entire release.

On a Unix machine, you will need standard gnu packages (gcc, gawk, perl). No special installs are required.

Then type:

```
forall makeall
```

This will build and run the entire 1.1 release, and place log files in your top-level directory.

20.3.3 Log Files

```
gcc_time.log      - th regular gcc

gcc_time_lite.log - th lite gcc

gccx86.log       - Cygwin x86 TH Regular compiler options

gccx86_lite.log  - Cygwin x86 TH Lite compiler options

vc_time.log          - th regular MS VC++

vc_time_lite.log     - th lite MS VC++
```

```
vcx86.log           - Microsoft Visual C++ x86 TH Regular
compiler options

vcx86_lite.log      - Microsoft Visual C++ x86 TH Lite
compiler options
```

The timing summary files are tab-delimited files used for comparisons, and can be loaded into a spreadsheet for further analysis. The compiler options files provide information for the certification disclosure, and comparison between TH Lite and TH Regular porting options.

If your platform can produce printed results, as part of porting save TH Regular, and New TH Lite output into log files. The awk script in `util/make/awk/extracttime.awk` can be used to produce files formatted in the same fashion from `th_report_results` output produced by each benchmark.

21 Porting Guide

The following paragraphs are intended to serve as a developer's guide to porting the Test Harness and benchmark code.

Porting the TH is straightforward, especially if the target has an RS232 serial port that can connect to a PC, a timer, and a debugger like gdb; however the TH supports many different communication links to a host system, and software simulators. RS232 is the most common. If you have a target system that does not have an RS232 port, contact ECL for support and words of gentle encouragement.

21.1 Build Procedure

The makefile based build process of Release Candidate/Beta 2 is similar to Version 1.1 Beta 1c. The directory tree has changed, as well as all of the make dependency files. In other words, you will need to diff for changes rather than copy files.

NOTE: If you have a previous port of TH_Regular 1.0 most of your abstraction layer changes (the al directory) can be merged into the new thal.c, thcfg.h, heapport.h, and eembc_dt.h files. If you have a port of TH Regular or Lite 1.1, do NOT COPY, but `diff` the files. There were some small changes. These changes should not require additional porting work, but they are required to build the suite.

The Build procedure supports multiple Tool chains, and Platforms in a single build tree. The TOOLCHAIN definition isolates dependency files, and generated object/binary files. The PLATFORM definition isolates Benchmark Runs and Results into a standard set of included makefile scripts and variables.

The release package is an example of two Tool chains executing on one host Platform. This structure covers simulators, and targets that can run individual benchmarks from a debugger. Your porting problem might be simpler, or more complex.

The framework is much more flexible than before, and with the new isolation of files you should be able to build and run multiple configurations within the same tree structure.

Individual control is provided in the make environment to:

1. Create output directories (make mkdir);
2. Generate dependencies (make harness);
3. Compile and Link (make targets);
4. Run Benchmarks (make run, make rerun);
5. Process Results (make results)

21.2 Guidelines

Porting for an embedded platform involves cross-compilation, test harness option selection, executing a benchmark on the target, and returning results to the host.

The following set of guidelines will simplify the porting process:

- Modify the makefile as little as possible. Use the <xxx>.mak files to work through each step of the port.
- Submit bug reports when you find a solution something that made porting or portability difficult.
- Isolate compiler/tool chain differences (and a few embedded platform specifics) to the TOOLCHAIN macro-defined file.
- Consider using platform-specific versions of the `dirs`, `run`, and `results` files for multiple tool chains. Running the benchmarks and collecting results is platform dependent.
- For a single Toolchain, and execution environment, you may not need both Platform and Toolchain abstraction. Set `PLATFORM=TOOLCHAIN` in <Toolchain>.mak and place your al directory inside the Toolchain directory.
- Regression test to preserve the ability to run on the original host platform per the distribution.

21.3 Create Empty Toolchain and Platform

The following procedure creates files and directories for a port that includes both TH Regular and TH Lite. For certification, you only need to choose one harness as appropriate for your platform.

On Unix hosts, make sure that you have used unzip –a or the makefile, awk, and perl scripts will have DOS end of lines which may cause problems. You should also remove `util/make/vc.mak`. This will remove Microsoft Visual C from the make environment build.

The following decisions need to be made first, and the naming conventions chosen will propagate to all the components necessary for a new Toolchain and Platform.

- Determine if the existing Toolchain dependency files, and Platform build scripts are adequate for your environment.
- Select a simple name for your Toolchain, (i.e. Diab). This name will be used in directory and filenames related to this Toolchain.
- Select a simple name for your Platform (i.e. evb123, archsim). If you have a single Toolchain for a single Platform, you can use the Toolchain name for your platform.
- Select an application to port first. In terms of the build process, all of the applications are the same, and the same steps can be repeated later after the first application port is complete.

Let's assume your Toolchain is called **chain**, and your platform is called **plat**, and you will be porting the **Telecom** application.

The following steps use these names for illustration. The top-level directory will be used as a starting point for directories. Start from the root directory and proceed as follows:

1. Change directory to `util/make`, and copy `gcc.mak` to `chain.mak`.

a. Edit `chain.mak`. Go to the last line of the file, and change
`TARGETS = $(TOOLCHAIN)`
to
`TARGETS = gcc`

b. Edit the chain.mak System Environment Section for your tool chain.

c. Return to the top-level directory and type:
`make TOOLCHAIN=chain`.
This will begin compiling all of the benchmarks for your new tool chain.

d. By reviewing the make process, determine the additional porting work required for Toolchains, and Platform. You may re-use any of these

components for your port. The following steps describe how to develop these structures.

2. To create a new test harness port, change directory to `util/make`, and edit `chain.mak`.

a. Go to the end of the file and replace:
PLATFORM = x86
with
PLATFORM = chain

3. Change directory to `th/` and `xcopy /s (cp –R) Gcc` to **chain**, and x86 to **plat**.

4. Change directory to your new `th/`**`chain`** and edit the file `depgen.cml`.

a. Modify the –tf line to change the filename from Gcc to chain. (i.e.: -tf `../th/`**`chain`**`/harness.mak`)

b. Change both occurrences of x86 to **plat** in the file section.
`-I../th/`**`plat`**`/al`
…
`../th/`**`plat`**`/al/*.c`

c. After editing `th/`**`chain`**`/depgen.cml`, create the target dependencies file by executing the makerule perl script, as in:

```
perl ../../util/perl/makerule.pl -cmd depgen.cml
```

NOTE: If this step is not successful, a later build will likely have the message "no rule to build THOBJS". There is a new file in each subcommittee named harness.h. For testing, create a dummy harness.h file to avoid errors. The benchmark build will set include paths to pick up the correct one in each subcommittee.

d. Verify that `th/`**`chain`**`/harness.mak` was created.

5. Change directory to th_lite. Create Platform and Toolchain directories as before and xcopy /s (cp –R) gcc to **chain**, and x86 to **plat**.

6. Change directory to your new th_lite/**chain**, and edit depgen.cml.

a. Modify the –tf line to change the filename from Gcc to **chain** (i.e.: -tf `../th_lite/`**`chain`**`/harness.mak`).

b. Change both occurrences of x86 to **plat** in the file section:

```
-I../th_lite/plat/al
...
../th_lite/plat/al/*.c
```

c. After editing `th_lite/chain/depgen.cml`, create the target dependencies file by executing the makerule perl script, as in:

```
perl ../../util/perl/makerule.pl -cmd depgen.cml
```

d. Verify that th_lite/**chain**/harness.mak was created.

7. Change directory to telecom

a. Copy the following files

```
copy dirsx86.mak dirsplat.mak
copy iterationsx86.mak iterationsplat.mak
copy resultsx86.mak resultsplat.mak
copy runx86.mak runplat.mak
copy depgen_gcc.cml depgen_chain.cml
```

b. Edit `depgen_chain.cml`. Go to the –tf line, and replace gcc with **chain** (i.e.: -tf targets_**chain**.mak)

c. Verify that targets_`chain.mak` is created using the `makefile` by typing:

```
make harness
```

8. Test your new Toolchain and Platform.

```
make TOOLCHAIN=chain
```

9. **Repeat steps 6 and 7 for all applications.**

You should have a new chain directory, and two new log files with the Benchmark Results summary.

```
chainplat.log
chain_time.log
```

Review benchmark timings in `chain_time.log`, and make environment variables in `chain`**`plat.`**`log`.

If there were any problems, review the previous steps, and correct the problem.

You may modify this procedure to re-use the platform and makerule definition files. This greatly simplifies the procedure. In most cases the x86 platform, and the gcc Toolchain definitions are adequate.

Two variables are defined at the bottom of the Toolchain file. The variable PLATFORM can be set to x86 to re-use the directories, run, iterations and results files. The variable TARGETS can be set to gcc, or another Toolchain, to re-use the dependency files generated by makerule.pl.

If both platform and dependency files are being reused:

a) Create a copy of the Toolchain file you are re-using in util/make directory.

b) Edit for your Toolchain paths.

c) Change TARGETS to the re-used Toolchain (i.e. gcc)

Test your Toolchain port. You will still need to modify the test harness for any platform specifics you may have.

21.3.1 Platform, Toolchain, Targets, and Root

Four control variables are used to identify unique filenames, directory names, and harness/utility location. The first three are defined in the Toolchain file, the Root variable is defined in the makefile. Following are the default values, and some example usages.

Platform defines the set of benchmark directories, associated with a Test Harness Adaptation Layer. This allows different Tool Chains to use the same platform.
The default setting is:
PLATFORM = x86
To simplify, you may leave this blank:
PLATFORM =
The result is that the TH al directory is a peer with src, and the dirs, iterations, run, and results files have the same names as in previous releases.
You could also use this to define multiple platforms. The directory structures remain unique for each Toolchain-platform combination.

- Toolchain defines the compiler/linker/debugger/simulator environment. It is required for each Tool Chain, and allows the user to choose a PLATFORM for that definition. The default setting is:
  TOOLCHAIN = Gcc
  The make scripts override this to additionally support the Microsoft Visual

C++ tool chain. Similar modifications can be made to add additional Toolchains within the same scripts.

- Targets is used to define the dependency files used by makerule, and make. These generated files are assumed to be different for each tool chain, but in some cases they can be reused. The default setting is:
  TARGETS=$(TOOLCHAIN)
  To use the gcc makefile rules with multiple tool chains, define the TARGETS variable in the <Toolchain>.mak file as follows:
  TARGETS=gcc
  This will allow other Toolchains to share the Gcc definitions.

- Root is used to define the location of the Test Harness, and Utilities with respect to the Subcommittee Application. The default value makes the Test Harness a peer to the Application. The default setting is:
  ROOT = ..
  If an application requires a specific test harness modification, the th, th_lite, and util directories could be moved into the application, and the makefile edited changing root as follows:
  ROOT = .

This can be complex at first, but each variable has been used for different situations in benchmark porting. The default settings cover the most common cases where a platform and Toolchain are added to the installed configuration.

21.3.2 Unix Porting

The GCC Toolchain name contains defaults for a windows platform using Cygwin. Any gcc compiler should work with this configuration. This configuration has been tested on both Linux and Solaris.

You may run into some of the following small issues:

- In util/perl/makerule.pl one may need to change the first line to '#!/usr/local/bin/perl'

- In util/make/gcc.mak One may need to set 'TOOLS' to '/usr/local'.

- The zip files release were built on a Windows platform, and have DOS eol's. To correctly unpack these on Unix, use **`unxip -a`** to unpack the files.

21.4 Toolchain

A tool chain is used to compile, link, and run each benchmark. The tool chain file util/make/<Toolchain>.mak contains the definitions needed to support a wide variety of tool chain environments. These definitions must be customized for your host and target environment.

The tool chain file is divided into the following sections that provide information used in the make environment.

1) **System Environment** contains definitions of paths to executables, output options for each tool, and file types of output files. The default tools are compiler, assembler, linker, librarian, and size utility.

2) **Compiler** contains include paths, option flags, warning level flags, and program defines.

3) **Assembler** contains include paths, option flags, warning level flags, and program defines.

4) **Linker** contains linker library include paths, and option flags..

5) **Librarian** contains option flags.

6) **Control** contains the supported makefile steps, benchmark execution program, the target platform, and dependency targets. The makefile steps consist of:

    a. mkdir: create, clean, and scrub directories;

    b. targets: build an executable using the target dependency files for the compiler, linker, and librarian;

    c. run: Post linker steps to execute the benchmark; and

    d. results: Post execution steps to summarize benchmark size and timing.

Edit the new Toolchain file `util/make/`**`chain.`**`mak`. This will include one or more of the following steps:

1. Edit the path to the tools directory and the file type suffixes.

2. Edit the compiler/assembler/linker command and option flags. In most cases compilation, assembly, and linking is all done from the compiler command line. In that case, the tools paths will all the same. For the rest of us, the options are provided for unique definitions.

3. Check that OBJOUT and INCLUDE are properly defined for your compiler/linker. While –o and –I are typical there are some compilers that are particular about spaces between these flags and the directory/filename. *NOTE: INCLUDE is an environment variable used by most compilers to point to the Toolchain include directory. When multiple Toolchains are resident on a system, defining it in the make environment ensures that the compiler will access the correct header files. It does not need to be defined directly to the compiler.*

4. Define COMPILER_INCLUDES to point to any compiler include files and LINKER_INCLUDES to any linker libraries as necessary.

5. Define LD for the linker, and LINKER_FLAGS. In most cases LD is the C compiler.
   NOTE: LIB is an environment variable used by linkers to point to the Toolchain lib directory. When multiple Toolchains are resident on a system, defining it in the make environment ensures that the linker will access the correct libraries. The Toolchains that have been ported so far have not needed this definition, deriving from the INCLUDE variable or using a separate environment variable. It was left out of the Toolchain file to avoid introducing unnecessary complexity that could have side effects.

6. Define the SIZE and LIBRARY commands and any applicable flags as appropriate for your Toolchain.

7. Define the RUN and RUN_FLAGS appropriate to your platform. You may be able to use these definitions to generate elf, s-records or binaries for downloading to the target, run a simulator, or execute a script that downloads and executes the benchmark on a target. Each target will likely be different. You may not be able to use RUN, if so see the ALL_TARGETS definition.

8. Define the ALL-TARGETS macro for as many of the targets – mkdir, targets, run, and results – as are applicable to your platform and as are commended in the associated .mak files. For example, the serial port or emulator and the results target could be used to move these files into directory structures expected by a terminal emulator script.

21.5 Directories

At this point return to the Telecom directory

Edit `dirs$(PLATFORM).mak` if you desire additions to the directory structure.

In most cases the default directories created will be adequate.

21.6 Run

Edit `run$(PLATFORM).mak` if your platform requires modifications to the run procedure. For example, the run target could be used to generate s-records or binaries using the RUN and RUN_FLAGS in the existing `run_$(PLATFORM).mak` but it may be easier to manually edit in your platform specifics.

21.7 Platform

The specifics for a platform are described in this section. This is a development process where the Application Layer is configured, and the additional files may need to be added to the standard build.

21.7.1 Benchmark Compile and Execution

Edit results$(PLATFORM).mak if your platform has modified the run procedure and requires a different method for recording results.

For example, the results target can be used to move s-record or binary files into a new directory structure expected by a terminal emulator script.

If your platform requires additional files in the abstraction layer, like crt0.s, or timer routines in assembly, then you must create the appropriate harness.mak files to include your new prerequisites and targets. You can edit th/$(TOOLCHAIN)/depgen.cml and run "perl util/perl/makerule.pl -cmd $(THPORT)/depgen.cml" to generate harness.mak or edit the one generated by running this utility on the existing th/$(TOOLCHAIN)/depgen.cml.

Most assemblers use a filetype of ".s", and include files with "#include". If this is the case, editing depgen.cml and adding a line to pick up the assembly files (..\th\plat\al*.s) will generate harness.mak with assembly files as part of the build.

21.7.2 Adaptation Layer

The following files in the application layer are edited for each platform:

al Directory Files
eembc_dt.h
heapport.h
thal.c
thcfg.h

In most cases only the Data Types and Timer need to be changed. The default timer uses the standard C clock function; this covers a wide range of Toolchains. The following steps are a simplified example applicable to both TH Regular and TH Lite.

Set the data types for your Platform by editing `eembc_dt.h`.

To set the system environment for your Toolchain and Platform by editing `thcfg.h`.. For example, if your Toolchain has a malloc function, you may use it instead of the Heap Manager. COMPILE_OUT_HEAP would be set to remove this from the Harness.

Set the timer, and other system function definitions by editing `thal.c`. The routines defined here control the display, timing, and reporting functions needed for the benchmarks.

21.7.3 Adaptation Layer Application Overrides

Each Application contains a header file (`harness.h`) in the top-level directory that can be used to override definitions in the Adaptation Layer. These overrides apply to both TH Regular and TH Lite from a single file. Default overrides are provided for the following applications:

- · 8-16 Bit Microcontrollers: Disable all heap management.
- · Telecom: Enable verification with floating point.
- · Networking: Set heap alignment, and increase heap size.

The other applications have empty `harness.h` files. This file is required in all test harness builds.

21.8 Results

The analysis of results from a benchmark run is described in this section. The `results<platform>.mak` file is used to generate timing and size summary log files for each application.

The raw log files for each benchmark execution are generated as part of the run step, and thus defined in `run<platform>.mak`. These log files are used by the steps defined in `results<platform>.mak` to create a standard summary file for all the benchmarks executions in an application.

For example, the results generated by building the benchmarks with two Toolchains, and executing using both th regular and th lite produce the following set of log files:

- Gcc_time.log – GCC tool chain using TH Regular harness.
- Gcc_time_lite.log – GCC tool chain using TH Lite harness.
- Vc_time.log – Microsoft Visual C tool chain using TH Regular harness.
- Vc_time_lite.log – Microsoft Visual C tool chain using TH Lite harness.

NOTE: The default gcc run on Unix systems will report large durations, where the last three or four digits are 0. This is NOT a problem. The correct wall clock time is calculated.

ANSI C, POSIX requires that CLOCKS_PER_SEC equals 1000000 independent of the actual resolution.

On Linux and Solaris hosts this results in durations to be large numbers which always end with three zeros. This is correct, because the clock resolution is less than the POSIX required resolution of 1000000. The resulting calculation to seconds is correct, and the actual resolution is measured to be 1000, or a millisecond timer.

Note that the time can wrap around. On a 32 bit system where CLOCKS_PER_SEC equals 1000000 this function will return the same value approximately every 72 minutes.

(Excerpt from GNU man page clock)

21.8.1 Iterations Override

Each benchmark includes a recommended iterations setting. This setting can be overridden at run time or compile time.

To change iterations at compile time for an individual benchmark, the `<iterations>.mak` may be edited. The distribution sets all benchmarks to DEFAULT. Each benchmark iteration can be replaced by an integer number, and this becomes the Recommended iterations when the benchmark is built.

To change iterations at run time, the run command line is used. Run the benchmark with "-ixxx" where xxx is an integer. This becomes the Recommended iterations in th lite, and the Programmed iterations in th regular.

21.8.2 Generating Benchmark Results

Timing results require that the `stdout` from each benchmark be saved to a log file. The `th_report_results` function is called by each benchmark. This function generates output to `stdout` using th_printf. The key fields used in benchmark timing results analysis are standard between both TH regular and TH Lite. The commands in run<platform>.mak generate these results as part of benchmark execution.

A log file is created for each benchmark run. The naming convention for timing files is: `<benchmark>.run.log`

NOTE: In some cases generation of timing results from benchmark execution on a target may not be possible in the run step. In this situation, use the ALL_TARGETS definition in the tool chain file to remove the ***results*** *step from make.*

Size results require that the tool chain provide a utility for extracting the code and data size needed by an executable. The utility is defined in the tool chain file, and executed in the run step.

A log file containing the code and data size is created for each benchmark executable file. The naming convention for size files is:
<benchmark>.size.log.

21.8.3 Generating Application Summary Results

Summary timing results are generated on the host from the benchmark run log files. A script using the AWK utility is provided that uses key words in the log file to extract the timing information for one benchmark. This allows results to be automatically extracted even if the tool chain produces additional output during the execution step (i.e. Firmware/Simulator version numbers, etc.).

The example awk script to extract timing results is `util/awk/extracttime.awk`. This script can be customized if needed as part of the porting process.

The `results<platform>.mak` file defines the steps needed to extract the summary timing results from the log files of each benchmark. The extract time script processes each benchmark log file, with the results appended to a tab delimited summary log file. A spreadsheet program for further analysis can then use this file.

The naming convention for the application summary log file is:
`<toolchain>_time<harness>.log`

The summary files can then be used to compare timing results between tool chains, and harness types.

Summary size results are generated on the host from the benchmark size log files. A Toolchain dependant script using the AWK utility is provided that uses key words in the log file to extract the size information for one benchmark.

The example awk script to extract size results for gcc is:
`util/awk/sizegcc.awk`.

21.9 Automation

This section describes tools for developing and certifying one or more applications in an automated fashion. These procedures can be used during development for timing comparisons of tool chains, as well as the generation of timing results for certification.

After the porting of one application is complete, repeat the appropriate steps for your other application ports. The same Harness port should work across all applications. The files defined in the Telecom directory are also found in each application, and will use the same editing done for your original port.

To automate the build and run for multiple applications scripts are provided in the top-level directory. For Windows systems, these are batch files, for Unix systems these files use `#!/bin/sh` to invoke the shell interpreter.

- Forall – executes a script in each application directory. If you are not porting all applications, edit this script and delete the missing application from the list. Forall is to be used with the following scripts to build all applications. An optional parameter is passed through to the make command in each script (i.e. forall makeall clean).

- Makeall – executes a make regular and make lite in an application directory. To execute type "`..\makeall`" from the application directory. Separate directories and summary results are built for regular and lite for comparison.

- Makereg – executes a TH Regular build for each Toolchain. Edit this file to add your Toolchain. To execute type "`..\makereg`" from the application directory.

- Makelite – executes a TH Lite build for each Toolchain. Edit this file to add your Toolchain. To execute type "`..\makelite`" from the application directory.

- Makefile – executes a make regular and make lite in all application directories. Automatically scans for new Toolchain files.

The set of files provided assume that all applications are available, and will be used. In many cases, only a few applications are of interest.

If all of the applications are not available, edit the `forall` scripts to remove the unneeded application directories.

21.9.1 Certification Results Summary

The makereg, and makelite scripts append the application summary logs into a single set of top-level log files. These log files are tab delimited with column titles suitable for a spreadsheet.

Each benchmark execution is uniquely identified using the first three fields. Existing results are not overwritten, and multiple runs can be appended.

21.10 Benchmark CRC Options

The benchmarks using TH Lite are listed below with the CRC options implemented in this release.

The CRC_CHECK option is iteration dependant, and may affect benchmark timing. It is used strictly for development checks.

The NON_INTRUSIVE_CRC_CHECK is independent of iterations, and does NOT affect benchmark timing. It may be used both in development and certification. Non-intrusive CRC checking is not required during certification runs.

The following table shows the current CRC support in each benchmark routine.

Subcommittee	Benchmark	CRC_CHECK	NON_INTRUSIVE_CRC_CHECK
MIC	ttsprk816	Yes	
MIC	rspeed816	Yes	
MIC	puwmod816	Yes	
MIC	pntrch816	Yes	
MIC	memacc816	Yes	
MIC	canrdr816	Yes	
MIC	bitnmp816	Yes	
AUT	ttsprk01	Yes	Yes
AUT	tblook01	Yes	Yes
AUT	rspeed01	Yes	Yes
AUT	puwmod01	Yes	Yes
AUT	pntrch01	Yes	Yes
AUT	matrix01	Yes	Yes
AUT	iirflt01	Yes	Yes
AUT	idctrn01	Yes	Yes
AUT	canrdr01	Yes	Yes
AUT	cacheb01	Yes	Yes
AUT	bitmnp01	Yes	Yes
AUT	basefp01	Yes	Yes
AUT	aiifft01	Yes	Yes
AUT	aifirf01	Yes	Yes
AUT	aifftr01	Yes	Yes
AUT	a2time01	Yes	Yes
CON	rgbyiq01	Yes	Yes

CON	rgbhpg01	Yes	Yes
CON	rgbcmy01	Yes	Yes
CON	djpeg	Yes	Yes
CON	cjpeg	Yes	Yes
NTW	routelookup	Yes	Yes
NTW	pktflow	Yes	Yes
NTW	pktflow	Yes	Yes
NTW	pktflow	Yes	Yes
NTW	pktflow	Yes	Yes
NTW	ospf	Yes	Yes
OFW	bezier01	Yes	Yes
OFW	bezier01	Yes	Yes
OFW	rotate01	Yes	Yes
OFW	dither01	Yes	Yes
OFW	text01	Yes	Yes
TEL	autcor00	Yes	Yes
TEL	autcor00	Yes	Yes
TEL	autcor00	Yes	Yes
TEL	conven00	Yes	Yes
TEL	conven00	Yes	Yes
TEL	conven00	Yes	Yes
TEL	fbital00	Yes	Yes
TEL	fbital00	Yes	Yes
TEL	fbital00	Yes	Yes
TEL	fft00	Yes	Yes
TEL	fft00	Yes	Yes
TEL	fft00	Yes	Yes

TEL	viterb00	Yes	Yes
TEL	viterb00	Yes	Yes
TEL	viterb00	Yes	Yes
TEL	viterb00	Yes	Yes

21.11 The Test Harness Code Layout

The general directory structure for a particular benchmark application is shown in the diagram below:

The Test Harness code is made up of three distinct sections: the adaptation layer code, the functional layer code, and the benchmark code. The adaptation layer code is contained in the al directory. The functional layer code resides in the src directory. The source code for each benchmark resides in its own separate directory, but uses functions contained in the al and src directories. The doc directory contains this document, as well as other related notes and information.

For the developer, the greatest amount of porting work will be done in the adaptation layer and perhaps a bit in the benchmark directories. The functional layer code should need little or no porting. If the functional layer code is changed substantially, ECL cannot guarantee certification unless you contacted us first and cleared the changes to the code.

We will now examine the files in each directory in detail, starting with the adaptation layer.

21.11.1 Adaptation Layer Files

The table below contains a list of the files that make up the Test Harness adaptation layer.

al Directory Files	**File Description**
eembc_dt.h	EEMBC datatypes definitions. This file contains target-specific definitions and must be ported by the developer. Make appropriate changes for your architecture here.
heapport.h	Platform-dependent heap definitions. Use of the TH heap manager is optional. If the developer chooses not to use the TH heap manager, the standard "C" functions malloc() and **f**ree() will be used.
thal.c	Test Harness Adaptation Layer function definitions. These routines must be ported by the developer. Make appropriate changes for your architecture here.
thcfg.h	Test Harness configuration definitions. The developer must modify this file with target-specific definitions. Make appropriate changes for your architecture here.

21.11.2 Functional Layer Files

The TH Functional Layer files are contained in the `src` directory. The Functional Layer contains all of the portable system-independent source code. In general, the developer should need to make few changes to the TH functional layer. The table below contains a list of all files that make up the Test Harness functional layer.

src Directory Files	File Description
Anytoi.c	Conversion to integer routines
Anytoi.h	Conversion to integer function prototypes.
Ascii.h	ASCII control character definitions
assert.h	Error handling definitions
heap.c	Platform-independent heap management functions.
heap.h	Platform-independent heap management definitions and prototypes.
memmgr.c	Free system memory management functions
memmgr.h	Memory management prototypes and macros.
printfe.c	Platform-independent printf functions
printfe.h	Platform-independent printf function prototypes
ssubs.c	String-handling functions.
ssubs.h	String-handling definitions and prototypes.
stdinc.h	Common C definitions and constants.
thal.h	Adaptation layer header file. Although the body of this function (in the al directory) needs porting, this file should not need substantial porting.
therror.c	Test Harness error handling functions.
therror.h	Test Harness error handling prototypes.
thfl.c	Functional layer interface functions.
thfl.h	Functional layer interface definitions.
thfli.h	Additional functional layer interface prototypes.
thlib.c	Test Harness library functions. This file contains the API function bodies.
thlib.h	Test Harness API function prototypes.
Thvinfo.h	Test Harness version information.
uuencode.h	Function prototypes to support uuencoding.
uuencode.c	Implementation of uuencoding functions.

21.12 Benchmark Files

The contents of each benchmark directory differ by benchmark and application area, but there are similarities that are described here

This table describes each file in the benchmark directory.

Benchmark File	File Description
DataSets	This directory contains the data files used to verify the benchmark. The number of data files will vary by application area.
bmark.c	This file is the standard benchmark interface. Functions and data declarations that control the benchmark execution are contained here. This file will need to be ported by the developer, though not substantially. In general, only very minor changes are needed – sometimes none at all!.
Benchmark source file	The name of this file will vary, but should reflect the benchmark being executed, such as Viterb00.c. This function will contain the actual benchmark function(s).
Benchmark header file	Interface definitions for benchmark function(s).
Visual C++ project file	This project file is provided to allow the developer to initially build the benchmark executable in a native Windows environment. These are stored in a win32 sub-directory. Project files for other IDE's are inteded to be stored as sub-directories of the benchmark as well.

21.12.1 The Adaptation Layer

The adaptation layer definition resides in two files. THAL.H is kept in the SRC directory and defines the interface to the THAL.C file. THAL.H does not change for a port (the reason it is in the SRC directory). EEMBC_DT.H is in the AL directory.

Most of the work you need to do is in the THAL.C file, which is in the TEMPLATE\AL directory. Before this step, you should have copied this directory to your build directory.

Currently, these porting instructions assume that your target has an RS-232 port used for communicating with the host.

21.12.1.1 Company and Processor Definition

The first step in porting the adaptation layer is to modify the Company, Processor and Target definitions in the THCFG.H file in your adaptation layer directory. This is the directory where the file THAL.C resides.

THCFG.H is included in THAL.C. These values get plugged into the global THDef structure (named the_thdef), defined in THAL.C

Every benchmark should include this file to define the company specific fields in the TCDef structure (named the_tcdef).

This example is an implementation for an x86 PC board. The following table details what information you should include for your port.

```
#define EEMBC_MEMBER_COMPANY    "EEMBC"

#define EEMBC_PROCESSOR         "PC-32bit-x86"

#define EEMBC_TARGET            "PC-win32"



#define TARGET_MAJOR            0

#define TARGET_MINOR            0

#define TARGET_STEP             'R'

#define TARGET_REVISION         0
```

DEFINE	Description
EEMBC_MEMBER_COMPANY	Insert your company name. Use only printable ASCII characters. You may use spaces. Do not use the tab, newline, or other carriage control characters.
EEMBC_PROCESSOR	Insert a brief but descriptive name for the processor on the target board. Limit it to basic characters.
EEMBC_TARGET	Processor platform name. Similar to the company name, limit it to basic characters. Good examples are NET186, "Eval Board". Please avoid names that have the processor ID in them like "MC68020 Eval Board". (The

	processor name is in the 'processor' field.
TARGET_MAJOR TARGET_MINOR TARGET_STEP TARGET_REVISION	Specify the revision of the target board. This is a placeholder. If the Major, Minor and revision numbers are all zero, then the benchmark scripts ignore this field. Usually, boards have a single revision number like 1, 2, 3, and 4. These go in the revision field. In general HW will use the step 'R' (for revision) and increment the TARGET_REVISION for different versions of a board. For more info on the version structure, see the SRC\VNUM.H file.

21.12.1.2 The Command Line Length

The command line length and the maximum number of command line agreements can be programmed. Note the default values from the template should be fine for most systems. However, if your system has a small amount of RAM you may set these to smaller values to save a little static data.

CMD_LINE_SIZE	The size of the input command line defined in SRC\THFL.C.
MAX_ARGC	The maximum number of command line arguments parsed for the TH 'CL' command.

21.12.1.3 Target Timer

These defines tell the TH if the target timer is supported. If TARGET_TIMER_AVAIL is set to TRUE, then you must fill out the following functions in THAL.C

```
al_signal_start( void )

al_signal_finished( void )

al_ticks_per_sec( void )

al_tick_granularity( void )
```

TARGET_TIMER_AVAIL	If you are not going to support a target timer, leave this field FALSE. If you do support a target timer, set this field to TRUE. If you do not support target-based timing, you will have to use the HCP (Host Control Program) or similar to provide accurate timing. Simulation/IP based benchmarking will use its own tools to provide this; contact ECL for details.
TARGET_TIMER_INTRUSIVE	If you are supporting a target timer and your timer generates a periodic interrupt to keep time then set this field to TRUE. In all other cases leave this false. Make sure this is set correctly!

21.12.1.4 Debug

The BMDEBUG define can be enabled to turn on some TH debugging features. Normally, you should leave this set to (0).

You can also control BMDEBUG by setting the M_BM_DEBUG define from the compiler command line in the makefile. For example, -D M_BM_DEBUG=1.

21.12.1.5 Endian-ness

The BIG and LITTLE endian defines should be set properly for your target system and processor. Some processors can run in either BIG or LITTLE endian mode.

EE_BIG_ENDIAN	Set to TRUE if the processor runs in BIG endian mode. Set to FALSE otherwise.
EE_LITTLE_ENDIAN	Set to TRUE if the processor runs in LITTLE endian mode. Set to FALSE otherwise.

One of these must be TRUE and one must FALSE. Do not set both of these defines to the same value. The code is a bit convoluted, but works correctly for all cases.

21.12.1.6 Floating Point Support

This controls the floating point printf() support in the TH. The TH only supports floating point output if the USE_TH_PRINTF flag is set to FALSE.

FLOAT_SUPPORT	Set to TRUE if the processor and/or C library supports floating-point numbers (either in hardware or through software emulation). Setting this to TRUE enables floating point support in the printf() engine. Set this to FALSE if you do not need to print floating-point numbers.

21.12.1.7 Printf Engine Control

You have a choice of printf engines to use. You can use the one that your C library supports, or you can use the one built into the TH. Motorola's Chuck Corley has written a floating point implementation of printf() that is available on the EEMBC website. It is provided without warranties, of course.

USE_TH_PRINTF	Set this to (0) to use the printf engine that comes with your compiler's C Library. Set this to (1) to use the printf engine that is built into the TH.

21.12.1.8 Heap Debugging

These defines control the debugging features in the heap manger (SRC\HEAP.C)

TURN_ON_VERIFY_HEAP	Keep this set to FALSE unless you are debugging a heap problem.
TURN_ON_DEBUG_HEAP	Keep this set to FALSE. Setting this to TRUE enables several debugging messages from the heap manager in SRC\HEAP.C.

21.12.1.9 Malloc and Free mapping

Some C libraries call malloc() and free(). For example, some vsprintf() implementations do this to allocate intermediate buffers. Many systems do not support malloc() and free(), which is why the TH has its own heap manager. In such a case, you will need to define MAP_MALLOC_TO_TH to TRUE so that malloc() and free() are mapped to th_malloc() and th_free(). See the file THLIB.C to see how this is done.

MAP_MALLOC_TO_TH	Keep this set to FALSE unless you need to map malloc() and free() to th_malloc() and th_free().

21.12.1.10 TH Size Control

The TH has two configuration macros that can help you make it much smaller. How much smaller depends on the target architecture.

COMPILE_OUT_HEAP	Setting this to TRUE completely compiles out the heap management routines in HEAP.C and turns off some reporting stuff in MEMMGR.C. Note, with this set to TRUE, the TH cannot run benchmarks that need to call th_malloc(). For most ports, this define should be set to FALSE.
NDEBUG	This is not defined in THCFG.H. For release builds, you should define this macro. This is generally done from the compiler command line in a makefile. -dNDEBUG Defining this disables all the assert statements in the code which saves considerable code and data space. For debugging, this macro should be undefined. It is a good idea to keep this undefined for your development and debug builds. This is because the assert calls built into the TH can catch some bugs during porting.

21.12.1.11 EEMBC Data Types

Review the EEMBC_DT.H file in the AL directory. Verify that there are no hardware-specific definitions that conflict with your target processor.

The standard C data types have been remapped into a set of "e_" data types. Any changes to the standard C definitions require ECL approval.

22 Support

At this point, you have successfully installed the release, created a new Toolchain, and ported this to your platform.

Please send problems with solutions to the bug tracking system at:
http://eembc.org/MemberShip/BugMaint.asp.

If, after reading this document, you still have questions, you should:

1. Plan on attending the EEMBC / TechTAG Conference phone calls. Contact Alan R. Weiss, Chairman, EEMBC TechTAG for details on how to participate. Its a great forum for asking questions (and in TechTAG there are no dumb questions).

2. Send email to: mailto:support@eembc.org