**Embedded Tool Set (EMTS)**

User Manual

and  
Software Engineering Manual

David T. Ashley ([dashley@gmail.com](mailto:dashley@gmail.com))

Table of Contents

[1 Introduction and Overview 4](#_Toc29499766)

[2 Licensing 4](#_Toc29499767)

[2.1 Licensing for Source Code, Binaries, and Installation Packages 4](#_Toc29499768)

[2.2 Licensing for Use of Web-Based Tools 4](#_Toc29499769)

[2.3 Licensing for Other Software Used Within the Embedded Tool Set 4](#_Toc29499770)

[3 Tool Set Architecture 5](#_Toc29499771)

[3.1 General Organization 5](#_Toc29499772)

[3.1.1 Bundling and Packaging 5](#_Toc29499773)

[3.1.2 Lack of DLL or Other Dependencies Outside the Individual Executable 7](#_Toc29499774)

[3.1.3 GUI Appearance 8](#_Toc29499775)

[4 Tool: Ill-Formatted Source File Scanning and Correction 12](#_Toc29499776)

[5 The Clike Programming Language 12](#_Toc29499777)

[5.1 Simple Variables, User Types, and Type Casting 12](#_Toc29499778)

[5.1.1 Language Features 12](#_Toc29499779)

[5.1.2 Implementation Implications 13](#_Toc29499780)

[5.1.3 Software Design 13](#_Toc29499781)

[5.2 Field Selectors and Array Elements 14](#_Toc29499782)

[5.2.1 Language Features 14](#_Toc29499783)

[5.2.2 Interpreter Design 14](#_Toc29499784)

[5.3 Full Bytecoding in Advance of Execution 14](#_Toc29499785)

[5.4 Support for Threads 14](#_Toc29499786)

[5.5 Credits 15](#_Toc29499787)

[6 Tool Set Requirements and Goals 15](#_Toc29499788)

[6.1 Overview of Requirements and Goals 15](#_Toc29499789)

[6.2 Explanation of Individual Requirements and Goals 15](#_Toc29499790)

[7 Supported Platforms and Build Variants 15](#_Toc29499791)

[8 Build Instructions 19](#_Toc29499792)

[9 Coding Standards 19](#_Toc29499793)

[10 Design Standards 19](#_Toc29499794)

[11 Testing Standards, and Testing 19](#_Toc29499795)

[12 Procedures and Checklists 20](#_Toc29499796)

[12.1 Creating Dave Ashley’s Home Server from Stock Linux Distribution 20](#_Toc29499797)

[12.2 Moving Directories Under SVN on \*nix Server 20](#_Toc29499798)

[12.3 Setting Attributes for Keyword Expansion and EOL on \*nix Server 20](#_Toc29499799)

[13 Topics to be Filed 20](#_Toc29499800)

[13.1 Use of Code Signing Key 20](#_Toc29499801)

[13.2 Mixing C and C++ 20](#_Toc29499802)

[14 Index 22](#_Toc29499803)

# Introduction and Overview

## Overview

The *Embedded Tool Set* (or *EMTS*) is a set of open-source software tool collections, open-source software tool aggregations, open-source software tools, and open-source software components designed to support embedded system development.

Each tool collection, tool aggregation, tool, or component is designed to run on some combination of:

1. Windows-based personal computers.
2. \*nix-based personal computers.
3. \*nix servers.
4. Android-based phones/tablets.
5. iOS-based phones/tablets.
6. FireOS-based phones/tablets.
7. Embedded systems.

The tool collections, tool aggregations, tools, and components don’t have names which are differentiated by platform, which is a potential source of confusion. The only naming conventions are:

1. Names for each individually distinguishable item begin with *emts\_*.
2. Tool aggregations for Windows-based personal computers and \*nix-based personal computers have names that begin with *emts\_a\_*.

For Windows, both console and GUI-based capability is used (depending on the tool collection, tool aggregation, tool, or component). For \*nix, only console-mode capability is used.

## List of EMTS Tool Set Components (Tool Collections, Tool Aggregations, Tools, and Libraries)

Table 1 provides a list and brief description of the EMTS Tool Set components.

Table : EMTS Tool Set Components (Collections, Aggregations, Tools, and Libraries)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Legend:  ‘Y’: Collection / aggregation / tool / component designed to run on platform.  ‘-‘: Collection / aggregation / tool / component not intended to run on platform, but will run on platform.  ‘ ‘: Collection / aggregation / tool / component will not run on platform. | | | | | | | | |
| Collection / Aggregation / Tool / Component Name and Description | Type | Windows PC | \*nix PC | \*nix Server | An-droid Phones / Tablets | iOS Phones / Tablets | FireOS Phones / Tablets | Embedded Systems |
| EMTS Cloud | Collection |  |  | Y |  |  |  |  |
| emts\_ifsfscan | Tool | Y | Y | - |  |  |  |  |
| EMTS Embedded Library | Library |  |  |  |  |  |  | Y |

## Languages, Availability of Binary Distributions, Required Development Tools, and Required Software Packages

The following languages are used:

Table : EMTS Tool Component Availability of Binary Distributions, Required Development Tools, and Required Software Packages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Collection / Aggregation / Tool / Component Name and Description | Languages | Availability of Binary Distributions | Required Development Tools | Required Software Packages |
| EMTS Cloud | PHP |  |  | Apache  MySQL  PHP |
| emts\_ifsfscan | C++, C |  | Windows: Visual Studio 2019 Community Edition or better.  \*nix: g++, gcc. | None. |
| EMTS Embedded Library | C |  | C compiler. |  |

Both Windows (graphical and console) and \*nix (console only) are supported. Most of the tool set is geared towards embedded software development; but it is an eclectic collection.

The EMTS consists of the following components:

* Mixed C++ and C[[1]](#footnote-1) source code that can be compiled on both Windows and \*nix systems.[[2]](#footnote-2)
* Windows binaries.[[3]](#footnote-3)

This document is designed to accommodate both *users* of the tool set (those that only desire to use the tool set), and *developers* (those that desire to modify or extend the tool set).

The user manual (normally intended for users) and the software engineering manual (normally intended for developers) are combined in order to eliminate the redundancies that would occur if they were separate documents.

# Licensing

## Licensing for Source Code, Binaries, and Installation Packages

The *Embedded Tool Set* (source code, binaries, documents, images, and all other files) is released under *The MIT License*. The license text is reproduced below.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## Licensing for Use of Web-Based Tools

The use of web-based tools will be under a license/disclaimer similar to the MIT License (no liability). The license/disclaimer needs to be written, and the web-based tools need to be browse-wrapped.

## Licensing for Other Software Used Within the Embedded Tool Set

No software or software components with a separate license are contained within or packaged with the EMTS.

# Tool Set Architecture

The general architecture of the tool set is described before the requirements (§6). This presentation order allows the architecture to be traced to the requirements in §6.

Each subsection is divided into the architectural features and the implementation implications.

## General Organization

### Bundling and Packaging

#### Description

The tool set is hierarchically decomposed into:

* Tool Set (TS). (The entire EMTS tool set.)
* Tool Collection (TC). (A set of logical tools that are packaged together in one executable.)
* Tool (TL). (A set of functions that have related functionality: for example, analyzing a first-order filter and designing a first-order filter. The *tool* is the minimum unit that can be bundled within an executable—tools, by design rule, cannot be split up.)
* Tool Executable (TLE). (An executable containing only a single tool.)
* Tool Script (TSC). (Script components implementing tool functionality; implemented in PHP, Python, Perl, etc.)
* Tool Web Interface (TWI). (Script components that generate the web interface to tool functionality; implemented in PHP, Python, Perl, etc.)
* Tool Function (TF). (An individual function within a tool.)
* Tool Function Executable (TFE). (An executable containing only an individual tool function.)
* Tool Function Script (TFSC). (Script components implementing only an individual tool function; implemented in PHP, Python, Perl, etc.)
* Tool Function Web Interface (TFWI). (Script components that generate the web interface to a tool function; implemented in PHP, Python, Perl, etc.)

Tools and tool functions are packaged within the tool set in 4 ways:

* As built-in functions for a script interpreter.[[4]](#footnote-4) (The script interpreter can process built-in functions both interactively and as part of a script.)
* As GUI interfaces within executables.
* As stand-alone console-mode executables that implement specific sets of tools or tool functions.
* As functionality implemented by a web page.

#### Implementation Implications

##### Packaging

Each tool should probably be packaged into source files as depicted in Figure 1 (a minimum of 9 source files). That way, each type of application only incorporates the modules it needs.

A possible 10th source file would be one containing some commonality between two or more interface modules.

##### Unicode Support

At present, EMTS supports only the ASCII character set. It is anticipated that the tools will be migrated towards full Unicode support, but there isn’t a timetable or sequencing plan.

##### Threadedness

The factors to balance are:

* Modern processors—even sometimes for laptops—are typically quad-core or better. This much of a speedup opportunity should not be discarded by not planning for it.
* A user may desire a single-threaded application. A program that uses all cores and brings the PC to a standstill might not be appreciated.

The approach taken is:

* Stand-alone programs:
  + All programs are single-threaded by default (or dual-threaded for GUI programs).
  + A command-line option and also a menu pick are provided to allow multi-threaded operation, and to specify how aggressively threaded the program is.
* Web auxiliary CGI program, cmdline app that feeds data back so script.
  + Single-threaded by default.
  + Command-line options can be used to allow multi-threaded operation, and to specify how aggressively threaded the program is.
  + To keep it simple:
    - All programs should share the same threadedness options.
    - If a program cannot be multithreaded and threadedness options are used, the option should just be ignored (rather than being an error).
* Clike interpreter command (both built into GUI app and cmdline app).
  + In an executable with both Gui apps and the Clike interpreter, the two may be running concurrently. This implies that the tool core (and, for consistency, things above it), should be thread-safe.
  + The plan is also that Clike support multi-threading (similar to Unix *fork*). Multiple threads within Clike may use the same tool at the same time. It follows that the Clike interface and the tool core have to be re-entrant.

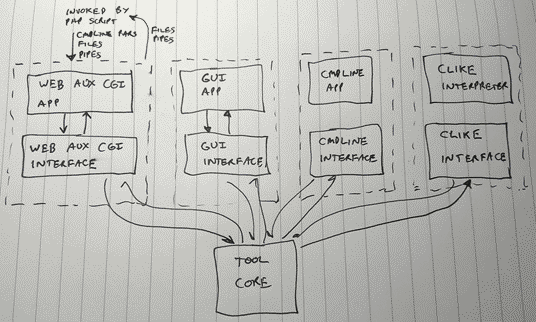


Figure 1: Modularization of Tools for Bundling and Packaging

### Lack of DLL or Other Dependencies Outside the Individual Executable

#### Description

These use cases and needs are anticipated:

* The tool set would need to be used on different computers and different operating systems with minimal chance of different behavior.
* Individual executables might be copied from machine to machine with no formal installation process.
* It would be advantageous to use a cryptographic hash to identify the executable(s) of the tool(s) being used (and desirable that this single hash give *all* information about sameness of an executable and its behaviors).
* It might be advantageous to archive tool executables with source code for a reproducible build.

#### Implementation Implications

These requirements imply that:

* The executables would need to not rely on DLLs (as much as this is possible). (This is the static linking option in Visual Studio, and results in larger executables.)
* The minimum required components are brought into any given executable. (If this isn’t done, the executables would be larger than necessary.)
* The software should self-signature in four ways:
  + All tool executables should have a self-hash option where the executable calculates a cryptographic hash of itself and reports it. (It isn’t possible for it to compare the value for validity, because embedding the hash in the executable for comparison would change the hash, and it would be impossible to get a match, so it can only *report* the value, but not check it.)
  + All tool executables should be able to report on demand:
    - The version control strings in each source file used for the build (or perhaps even for the whole tool set’s source at the time the build was done).
    - The hash of each source file used for the build (or perhaps even for the whole tool set’s source at the time the build was done).
  + Clike should have a built-in function to calculate the cryptographic hash of files. (This could be used for checking tools or for other purposes.)

Regarding the requirement for minimally-sized executables:

* Use of libraries: I have two libraries planned (will also speed up builds).
  + Visual Studio has an option to extract only the required functions from each module, and this is turned on by default for release builds.
    - If the option can be turned on for debug builds, if it works as expected, and if there is a similar option for *gcc*, then this should do the trick.
    - If no to any of the above, the libraries can be built in the standard way (compiling one function per module). (Annoying, but standard practice for libraries.)
* The modularization of tool functionality (Figure 1) ensures that no unnecessary functionality will be linked in.
* Tools have to be easily movable between executables so that as an executable grows too large, its components can be shifted.
  + The Clike interpreter might best be viewed as a “tool” in that framework.
  + It can be present or not present in a GUI build.
  + In a Cmdline build:
    - Either it won’t be present at all, because the tool does something else, OR
    - It will be the only tool, and any toolish functionality will be implemented via Clike built-in commands.
    - The Clike interpreter and a tool are mutually exclusive ideas. A Cmdline build is one or the other, but not both.

#### Software Design

Need to cover:

* How Clike is treated as an application.
* The Clike interpreter in Cmdline and Gui builds.
* How windowing functionality is packaged.
* Any required tool support for the packaging selection, assembly, or build process.

### GUI Appearance

#### Description

When multiple tool GUIs are combined into the same executable, a scheme such as shown in Figure 2.

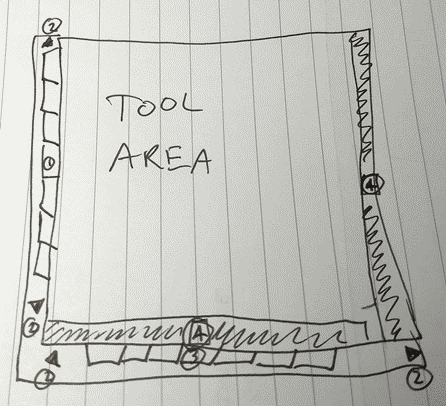


Figure 2: Multiple Tool GUIs Combined Into Same Executable

In Figure 2, the essential features are:

* The tabs along the left (#1 in the figure) are used to select between applications.
* Once an application is selected, if the application has multiple screens of content, a set of horizontal tabs will be presented at the bottom (#3 in the figure), similar to what Microsoft Excel does with multiple worksheets in the same spreadsheet.
* With both the tabs on the left and the tabs on the bottom, scrolling may be necessary (#2 in the figure).
* Within the area dedicated to the tool, vertical and horizontal scroll bars may be necessary (#4 in the figure).

One or more of the tabs along the left edge may be devoted to Clike.

There may be some provision to duplicate or clone the tabs along the left (to have more than one instance of a tool selectable by tabs).

No decision has been made about multi-document display.

When only one tool exists within an executable, the tabs along the left would either:

* Be absent, OR
* There would be only one tab.

#### Implementation Implications

Every effort has been made to keep naming and functionality consistent among the 4 packaging methods. Consistency helps to:

* Facilitate ease of use. (The same names and input and output formats are shared among all packaging styles.)

Minimize source code bulk. (The same source code is used to build all 4 packaging styles.)

Conceptually,[[5]](#footnote-5) each tool function accepts a regular language as input(s) and produces a regular language as output(s).

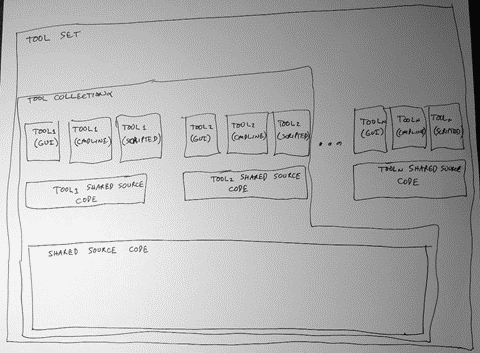


Figure 3: Tool Set Architecture

* Shared source code (Figure 3).
  + The tool set uses a great deal of shared source code. This source code is potentially shared between all tools.
  + The shared source code may take the following forms (TBD):
    - DLL’s are not permitted, for the reasons explained in §6. Shared source code must be statically linked into each executable.
    - Libraries (or possibly a single library), that is compiled separately from any tool and linked in to each tool.
    - Both approaches would hopefully be used in such a way as to minimize the bulk introduced in each executable. (Libraries may in some cases be better in this regard, as they can be arranged so that the minimum unit incorporated into the executable is the function rather than the module.)
* The tool set consists of a number of tools (in Figure 3, Tool1, Tool2, ToolN).
  + Each tool is logically cohesive and performs a fairly narrow function. Examples:
    - Analysis and reduction of Boolean functions.
    - Generation of code to implement state machines.
    - Analysis and generation of code to perform specialized approximations.
  + Each tool potentially exists in some combination of the following three forms:
    - A standalone text-based console-mode executable.
    - A GUI-based tool.
      * The GUI-based tool is possibly bundled as part of a tool-collection. If not, it might be considered a tool collection with one tool.
    - Command[s] built into the scripting language.
* Each tool may have source code shared between the three forms (GUI, Cmdline, Scripted).
  + Because the three forms are so close in functionality, there would be shared source code unique to the tool.
* Data storage for each tool.
  + When tools need to store files, they should be:
    - A regular text language, friendly to humans (not XML!).
    - Designed so that small changes in whatever is manipulated by the tool result in small changes to the file (so that a human can figure out what has changed using a diff tool).
    - Should version control well, with minimal false positive changes.
      * Minimizes delta size with many version control tools.
      * Makes it easy for a human to figure out what has changed.
      * Indentation and trivial changes need to be controlled by the definition of the language to prevent false positive changes.
    - The regular language should have a formal parser that can determine:
      * Membership in the language.
      * Errors or warnings.
      * This will facilitate hand edits.
* Data interchange between tools.
  + GUI:
  + Cmdline:
  + Scripted:
* A number of individual projects (i.e. programs):
  + Each project consists of:
    - The project files (Visual Studio project files, makefiles, etc.).
    - Source and graphics files that are unique to the program (the *main()* function, icons, etc.).
  + Each project may make reference to files in the shared source code (described below).
  + Each project parameterizes the build (by setting preprocessor directives) for the target platform.
* Shared source code:
  + Does not stand alone—it is included in a project.
  + Parameterized for the build platforms and variants.

# Tool: Ill-Formatted Source File Scanning and Correction

# The Clike Programming Language

The “$$” was inspired by Bash.

## Simple Variables, User Types, and Type Casting

### Language Features

Variable names are always prefixed with a dollar sign, i.e. “$x”.

Variables may be one of the following types:

* ASCII string (astring).
  + No further description needed.
  + Behavior the same on all platforms.
* Unicode string (ustring).
  + No further description needed.
  + Behavior the same on all platforms.
* Binary string (bstring).
  + Arbitrary ordered collection of bytes.
  + Can be an ordinary string like “Hello”, or buffer an entire object file (for example).
  + Automatically set as the type on assignment such as $x=”Hello”.
  + Limited only by available memory.
  + Behavior the same on all platforms.
* 64-bit signed integer (int).
  + Always the same size (8 bytes).
  + Behaves like C integer—will roll over and all that.
  + Automatically set as the type on assignment such as $x=34.
* IEEE double (double).
  + 64 bits, per the IEEE standard.
  + May be the most difficult to implement consistently across all platforms. This may be variable.
  + Automatically set on type assignment such as $x=3.14.
  + Overflow, underflow behavior as of yet undetermined.
  + May write the Clike standard to leave this up to the compiler, as there may be slight differences.
* Tool Defined
  + Each variable may be of a number of types understood only by a given tool.
  + When a tool registers its functions with the Clike interpreter, it also registers these types.
    - Within Clike, each such variable is a pointer to dynamically-allocated memory that Clike does not understand.
    - Along with the registration, the equivalent of copy constructor and destructor are defined. This is necessary if Clike does copies for function parameters, or has to dispose of a variable.
    - Default constructor is not necessary. This can just be a null pointer until an appropriate function needs to operate on it, then it can be constructed then.
    - Once a variable is of a tool-defined type, only tool operates can be performed on it.
      * Trying to apply operations to it (“++”, “>”, except for simple assignment) will fail.
    - This is the closest Clike comes to OO programming or user-defined types.
    - In general, binary strings are the preferred format unless there is some *very strong* efficiency advantage.

The binary string type allows writing arbitrary functions that use binary data (large integer functions, for example) that don’t need to do especially expensive type conversions.

Variables may be type cast.

Types can be cast as in *$j = (int)$k*. Conversion failures result in possibly lossy conversions or a default value. Casting a variable repeatedly (even between just two types) may cause its value to be altered substantially.

### Implementation Implications

Nothing noteworthy.

### Software Design

Nothing noteworthy.

## Field Selectors and Array Elements

### Language Features

Variables may have their names used in any of the following forms:

* $x
* $x[1000]
* $x.overflow
* $x[“1000”].

It isn’t necessary that an unqualified variable be set in order that an array element or structure member be set. It is acceptable if $x[1000] has a value but $x does not.

Structure member and array reference may be used interchangeably and to an arbitrary depth, i.e.

$x[25].widget.lugnut.whatever[31].tire is fine.

Assigning a variable directly with no preparation (no dimensioning, etc.) is acceptable.

Intermediate values are not automatically assigned, i.e. if $x[25].widget.lugnut.whatever[31].tire is assigned, $x[25].widget will not have a value (it will only have children).

### Interpreter Design

Structure members and array elements are treated identically.

$x[“lugnut”] will have the same value as $x.lugnut.

All field selectors and array indices are strings.

## Full Bytecoding in Advance of Execution

## Support for Threads

The bytecoding of a script is a constant data structure, known before execution begins.

Variable fetches and assignments always have to be atomic (protected from concurrent thread access) due to the dynamic data structures.

Functions are always re-entrant. The standard expectations about automatic and static apply, except that a variable may not be corrupted “mid-stream”. The granularity would always be larger.

Threads are spawned by a CreateThread()-like mechanism. A function name and any number of parameters may be supplied. The function must accept those parameter.

After the CreateThread():

* The caller will receive a success code to indicate success or failure.
* Execution will begin at that function with the parameters.
* Any process may determine if it is the primary thread or a secondary, how many threads are running, etc.
* exit() will end a thread, or the program if it is the primary thread.
* Need mutual exclusion and IPC mechanisms.

## Credits

Many concepts are borrowed from Tcl, including interpreter and command extension design.

# Tool Set Requirements and Goals

## Overview of Requirements and Goals

All requirements and goals are listed immediately below, with further explanation and discussion provided in §6.2.

* License.
  + Availability of source code.
  + Lack of obligation to make source code changes publicly available.
* Source code.
  + Availability.
  + Documentation.
  + Lack of obligation to make source code changes publicly available.
* Verification of Computing Environment.
  + Ability to verify identity of all non-system components.
  + Ability to self-verify.
* Use of all available platform memory.
* Use of all available platform computing resources.
* Scriptability
  + Integration with a Turing-complete scripting language.
  + Similarity of scripting language to C.
* Extensibility.
  + Ability to add and integrate custom tools.
  + Ability to add and integrate built-in commands to scripting language.
* Cross-platform usability.

## Explanation of Individual Requirements and Goals

# Supported Platforms and Build Variants

The C/C++ code of the tool set is build is parameterized in a number of nearly orthogonal directions, as described in Table 1.

Within a build, every C/C++ source file is parameterized identically. In a product like Microsoft Visual Studio, the parameterization would be done via GUI options that affect the options provided to the C/C++ compiler. In a more traditional command-line build, the parameterization would typically be done via the “-D” compiler option.

Within each category, constants are mutually exclusive, and only one constant can be applied, for example, “-D EMTS\_PF=EMTS\_PF\_K\_MFC”. In the future, bit-masked constants (not mutually exclusive) may be added.

Table 3: C/C++ Build Parameterization

|  |  |
| --- | --- |
| **PREPROCESSOR CONSTANT** | **DESCRIPTION** |
| **Platform (KEMTS\_PF) (All #define’s below mutually exclusive)** | |
|  | |
| **Operating System Word Size (K\_EMTS\_WSZ\_OS)** | |
| K\_EMTS\_WSZ\_OS | Should be set to the number of bits that characterize the operating system. In the case of Windows/x86, this differentiates 32-bit vs. 64-bit Windows. |
| **Processor for Assembly-Language (K\_EMTS\_PROC\_AL) (All #define’s below mutually exclusive)** | |
|  | |
| **Machine Word Size (K\_EMTS\_WSZ\_M)**  **(Note: Machine word size does not imply C or C++ default integer size.)** | |
| K\_EMTS\_WSZ\_M | Set to the number of bits in a native machine word. |
| **Integer Size (K\_EMTS\_SZ\_INT\_LL)** | |
| K\_EMTS\_SZ\_INT\_LL | Set to the number of bits in a C long long integer. |
| **Machine Word Size (K\_EMTS\_SZ\_INT\_L)** | |
| K\_EMTS\_SZ\_INT\_L | Set to the number of bits in a C long integer. |
| **Machine Word Size (K\_EMTS\_SZ\_INT)** | |
| K\_EMTS\_SZ\_INT | Set to the number of bits in a C integer. |
| **Machine Word Size (K\_EMTS\_SZ\_INT\_S)** | |
| K\_EMTS\_SZ\_INT\_S | Set to the number of bits in a C short integer. |
| **Machine Word Size (K\_EMTS\_MWS)**  **(Note: machine word size does not imply C or C++ default integer size.)** | |
| K\_EMTS\_MWS | Should be set to the number of bits in a native machine word. |
| K\_EMTS\_MWS | Should be set to the number of bits in a native machine word. |
| **Machine Integer Representation (K\_EMTS\_MIR) (All #define’s below mutually exclusive)** | |
|  | |
| **Machine Floating Point Unit (K\_EMTS\_MFPU) (All #define’s below mutually exclusive)** | |
|  | |
| **Project Type (K\_EMTS\_PROJTYPE) (All #define’s below mutually exclusive)** | |
|  | |
| **Project Link Type (K\_EMTS\_LINKTYPE) (All #define’s below mutually exclusive)** | |
|  | |
| **Project Debug Level Type (K\_EMTS\_DEBUGLVL) (All #define’s below mutually exclusive)** | |
|  | |
| **Project Debug Log Level (K\_EMTS\_DEBUGLOGLVL) (All #define’s below mutually exclusive)** | |
|  | |
| **Project Assertions (K\_EMTS\_ASSERTLVL) (All #define’s below mutually exclusive)** | |
|  | |
| **Program Type (K\_EMTS\_PROGTYPE) (All #define’s below mutually exclusive)** | |
|  | |
| **Screen Size (K\_EMTS\_SCREENSIZE) (All #define’s below mutually exclusive)** | |
|  | |
| **Threadedness (K\_EMTS\_THREADS) (All #define’s below mutually exclusive)** | |
|  | |
| **Character Mode (K\_EMTS\_CMODE) (All #define’s below mutually exclusive)** | |
|  | |
| **Memory Allocation Strategy (K\_EMTS\_MALLOCSTRAT) (All #define’s below mutually exclusive)** | |
|  | |

# Build Instructions

TBD.

# Coding Standards

TBD.

# Design Standards

TBD.

# Testing Standards, and Testing

TBD.

# Procedures and Checklists

## Creating Dave Ashley’s Home Server from Stock Linux Distribution

Need to include the step of modifying http.conf so that .svn contents not served.

[dashley\_wm@spock ~]$ diff /etc/httpd/conf/httpd.conf /etc/httpd/conf/httpd.conf~

225,231d224

< # The following lines prevent access to .svn directory internals. This is relevant

< # because some web content is served directly out of an SVN sandbox.

< <DirectoryMatch .\*\.svn/.\*>

< Deny From All

< </DirectoryMatch>

<

< #

## Moving Directories Under SVN on \*nix Server

## Setting Attributes for Keyword Expansion and EOL on \*nix Server

# Topics to be Filed

## Use of Code Signing Key

The use of a code signing key would require the expenditure of a few hundred dollars a year, which seems like an unnecessary expenditure.

Software is released on a website with a cryptographic hash. The assumption is that no attacker would be able to modify the cryptographic hash published on the website.

Code signing may be considered in the future.

## Mixing C and C++

C++ only. Use options for \*nix and MSVC++ to treat C files as CPP.

# Index

Embedded Tool Set, 4

EMTS, 4

TC, 5

TF, 5

TL, 5

TS, 5

1. C++ files have the extension .cpp, and C files have the extension .c. To simplify the build process, the C files are written to be compatible with both C++ and C compilers, but only a C++ compiler is used for the builds. C source files are retained to protect for the possibility of producing other programs for embedded targets that share the same source code. [↑](#footnote-ref-1)
2. For building for Windows, Microsoft Visual Studio 2019 Community Edition or better is required. For building for \*nix, the GNU C++ compiler is required. [↑](#footnote-ref-2)
3. \*nix binaries must be built from source code by the end user. [↑](#footnote-ref-3)
4. The script interpreter interprets a language called Clike, described fully in §TBD. The interpreter can be used both interactively (in this mode it might be described as a *very* powerful calculator) and to run scripts (which give programmatic access to all ETS functionality). [↑](#footnote-ref-4)
5. Within the CLIKE interpreter, for efficiency, conversions between string and other representations are performed only when necessary. For example, “*$y=2\*sqrt($x)*” will not result in a string representation of “*sqrt($y)*” being calculated before the multiplication by “*2*”. [↑](#footnote-ref-5)