Extended Basic Tools (EBT)

User Manual

Version 1.2 27 Aug 2020

(Record of Revisions located in Appendix 1)

Introduction to EBT

Extended Basic Tools (EBT) is a set of extensions for the Coridium BASICtools IDE which Coridium Corporation provides for free when used with any of the Coridium SBC Boards. EBT is provided for free and relies on the existence of BASICtools. Without BASICtools, EBT is, by design, materially non-functional. The scope of this document is related to EBT and it is not intended to be a User Manual for Coridium's BASICtools IDE (BT). This work assumes that one has a basic to good understanding of BT, ARMbasic, and how to use each on SBCs that have ARMbasic Firmware thereon.

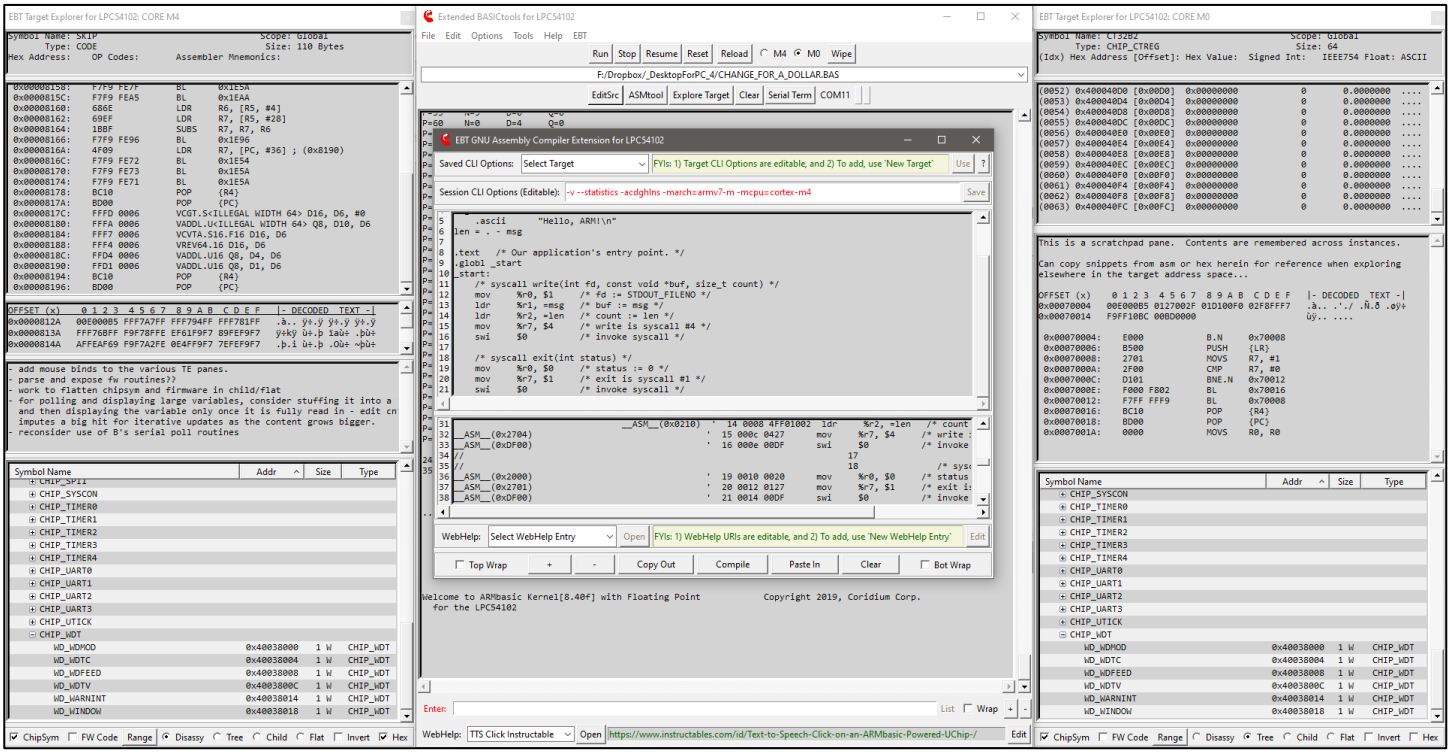


Figure 1: Extended BASICtools with Dual Target Explorers (one instance per core) and ASMtool Modules activated

Background/History/Inspiration/Dev Intro

(a single-page wall of text – feel free to skip it…)

Good day. My name is Tod Wulff, an aerospace & defense professional that is also a bit of a geek at heart.

Hailing from the era of dial-up BBS, 8-bit Microcontrollers, Kaypro/Commodore/Tandy/TI-994A personal computers, when Radio Shack stores were plentiful (the good ole days), one of my first hobby embedded projects was working with a MEK6800D2 Motorola Microprocessor Training Kit, which I had purchased while working as a co-op at the MSU EE labs (after completing my High School Electronics VoTech training in Southern Lower MI). That project involved my prototyping the Radio Shack SP0256 NARRATOR™ SPEECH PROCESSOR onto the MEK6800D2, wiring it up and programming the 6800 to get it to emit pseudo-speech (those who've worked with the SP0256 based HW know exactly what I am alluding to). It worked wonderfully and I progressed down the path of cutting my teeth in embedded microcontrollers and Assembly. After High School, life got in the way, Military, War, Spouses, Children, entering the civil sector, starting a career, etc. all added up to my shelving my hobby in favor of pursuing the endeavors of life in a Western culture (here in the US).

Skip forward 20 years, coming to the inevitable time where the kids are maturing to the point that the Bride and I are severely distracting to their social status (and life in general), the mortgage, vehicles, college bills are slowly being paid down, earnings getting better with advancements, and my having enough spare time to the point where I could start to refocus on some selfish interests, I picked back up on the hobby electronics gig. Anyways, given my lineage and history, I sought out and found a dev environ that I quickly bonded with - ARMbasic - BASIC was my first love and this fit the bill of not only reacclimating myself to programming, but working with hardware that was vastly more powerful than what I had started with decades earlier, and thus the journey began. Having done some windows scripting for a couple years prior, using AutoHotKey (AHK), I decided to try to integrate a home-brewed AHK-based IDE, intended for use with early Coridium goods, into Notepad++, my editor of choice back then (remaining so thus far). That endeavor was only partially successful.

This was circa 2006-2009. Then, for reasons well beyond our control, life changed (as it had for many during those years – housing crisis). Hobbies shelved - focus on a new career, recovering from financial struggles (was heavily vested in the real-estate domain and we took it in the shorts and the youngins were just getting to the point where College funding was an imperative). Basically, Life and First-World problems (we are really blessed, considering the challenges and toils that folks in other parts of the world struggle with on a daily basis) manifested themselves and ... the hobby got shelved. I picked back up briefly on it in 2011-2012-ish then was met with another career change - hobby shelved yet again.

Fast forward another decade and ... I am back and, Good Lord Willing, hopefully for the duration (until I take that proverbial dirt nap and start pushing daisies up from below). So, here we are. Wow - Arduino (what is that weird word?) had stormed the market. Makers?? What the heck are they?! ... :) My friends at Coridium Corp (proprietors of ARMbasic and ARM-based microcontroller dev boards) had remained steadfast and true. Now, instead of the LPC2xxx series of controllers, there is this new (to me) entity of ARM, and Cortex M0/M3/M4, and Arduino, and ... WOW! The culture has morphed quite a bit, and in many a great way. Peeps are collaborating remotely and, indeed, globally. Hardware is getting amazingly fast and powerful, and ARMbasic, having matured and steadfastly hardened with employment across many different families of silicon, is a thing of beauty for me and many others.

EBT is, comparatively, somewhat recently birthed, with initial code being drafted in November 2018, and a mature alpha coming into existence by February 2019. Further development to EBT was sporadic until July 2019 when I was tasked professionally on a temporary duty assignment that ended up being 9 months long (so much for 'temporary'…). Returning home in April/May 2020, EBT dev was resumed slowly and continues into today, and the foreseeable future.

One initial intent for EBT (not being called such when conceived, but rather 'Tweaked BT' [sigh]) was to provide a means to have a workflow that somewhat automated the inclusion of ARM Assembly constructs into an ARMbasic user app. The next extension was to change the default color scheme from white to gray, to match how Notepad++ was set up, to provide visual continuity, if you will.

Very quickly, it took on a life of its own and has been updated over the course of the last year and a half, pseudo-continuously. The initial roots of EBT can be found in the ASMtool module within EBT. Being an Android junkie and somewhat spoiled with customization/theming, coupled with being picky about how the tools used on a regular basis are customizable to one's liking, to include workflow aspects that conform to what is perceived as one's needs or desires, caused EBT to morph into what is perceived as a substantive set of extensions to BT.

Note: Overt care to not confuse 'Extensions' with 'Enhancements' is employed. The former is an appropriate descriptor, while using the latter to describe EBT would be presumptively inappropriate as it is, admittedly, a quite subjective term...

Ok, 'nuf with the diatribe - the purpose of this document is to explain EBT and its use - let's get to it.

The best way to see and begin to understand EBT is to 'install' it and use it. Given it is an extension to an existing toolset, there are prerequisites needing to be fulfilled in order to make use of same.

EBT Prerequisites

The first requirement is to have BASICtools (BT) for ARM installed on an x86 32-/64-bit Windows dev box. BT can be download from here. An archive of the current (at time of drafting) complement of EBT files can be downloaded from here. This User Manual (online .pdf version) is also linked to in EBT's WebHelp MiniTool, detailed later in this document.

NOTE: At its root BT is, and by inclusion EBT is also, a set of TCL scripts that provide a UI framework and, when coupled with compilers, disassemblers, etc., comprise a complete and extended ARMbasic Integrated Development Environment. TCL is a powerful cross-platform interpreted scripting language. Sadly, not being a Linux or iOS type, no effort or assertions are made regarding EBT's functionality when installed on a machine that is not a Windows box. For those users who are on Windows boxes, assertions that it ***may*** work are made (see license/warranty on first page). For *nix-based OSes, EBT's author is not personally interested in progressing through the curve to ensure that EBT works with a BT installation on those machines. However, there is no opposition to someone taking the time to check and advise if EBT works thereon or not. If code changes/additions are needs to enable EBT to work on same, and if you, the reader of this document, are able and willing to tackle modifying EBT's set of scripts to work on Linux or iOS dev boxes, it is endorsed and will be supported, as long as the effort is reasonable and not too disruptive to other endeavors (Selfish? Maybe. | Practical? Absolutely.).

Once BT-proper is installed, the next step is to get the current build of EBT downloaded (link above) and extract same (preserving directory structure) into the Coridium BASICtools directory (often located at %ProgramFilesx86%/Coridium/ (the default offered by the installer)). Administrator approval will likely be queried for by the OS.

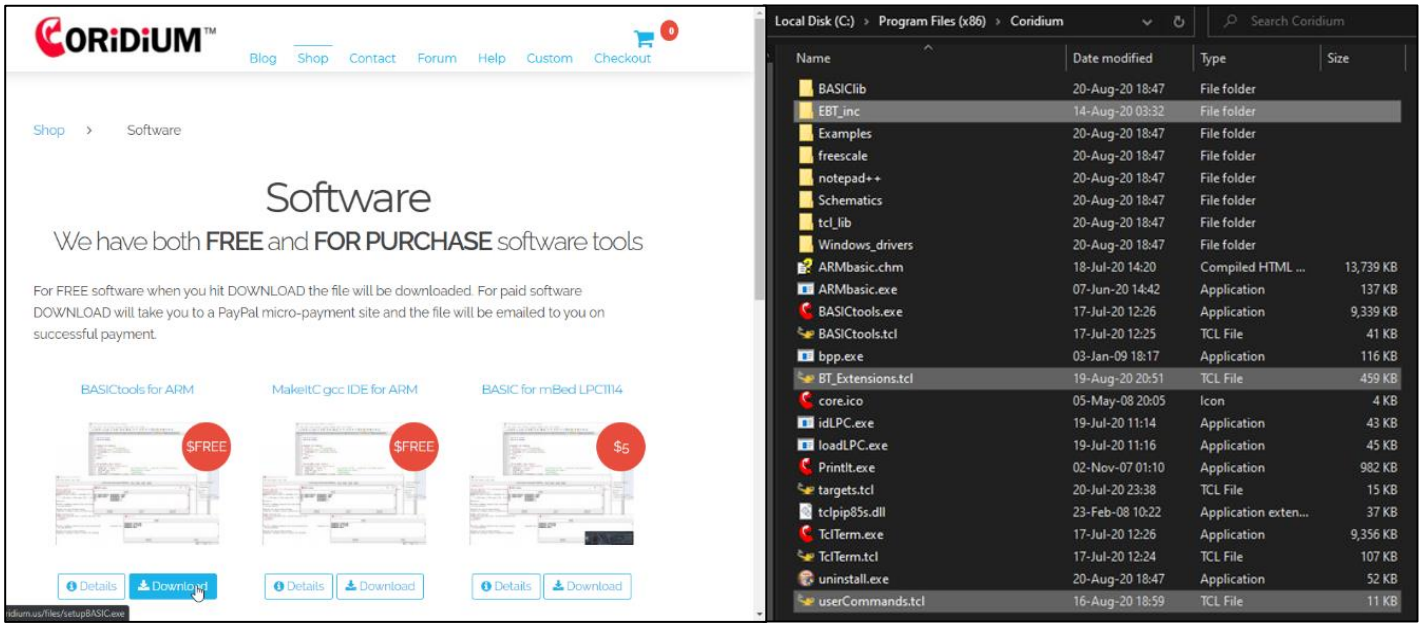


Figure 2: BASICtools Download Page - EBT files extracted into Coridium folder

EBT consists of two .tcl scripts in the root directory of the Coridium Install – BT_Extensions.tcl and userCommands.tcl. Additionally, there are a number of utility binaries, support scripts, and other support files of varying types included in the EBT archive. These other files, when extracted to the Coridium folder, should end up in the EBT_inc subdirectories.

The userCommands.tcl script is organically 'sourced' in by BT, when it is started. The original intent was to enable BT's user devs to have custom commands be intercepted from the embedded target and be able to have the IDE/host dev box programmatically react to same. An example is either clearing the BT console or responding to a BELL ASCII character being sent by the target to cause the host system to emit sounds through the PC speaker.

This capability of being able to run customized user drafted .tcl scripts at BT's startup was latched onto and heavily employed to Extend BT's core capabilities and UI. In a nutshell, userCommands.tcl checks if an BT .ini saved setting reflect that EBT is turned on and, if so, launches BT with the EBT extensions enabled. If the EBT .ini is missing (i.e. first run), or the enablement of EBT is set to 'off', then BT-proper is launched and EBT is not started, enabling a stock Coridium BASICtools IDE experience.

Speaking of the .ini settings file. BT and EBT each have their own .ini. This was done to help ensure that if EBT is disabled, that there is the smallest possibility of EBT impeding the operation of BT. This philosophy was carried forward in EBT in that EBT doesn't materially alter the BT-proper menus/functionality, but rather has its own menu entry/hierarchy and functionality. The intent is to have EBT extend BT's capabilities, never impeding and keeping material alteration of same to a minimum. Extension of, not enhancement to, is the theme here.

And while we're talking about other directories, as additional information, BT's configuration .ini file is stored in %AppData%/Roaming/Coridium/ (EBT's is in the EBTini subfolder therein). Temp files, used during various phases of the tool chain's endeavors, possibly being touched on later herein, are stored in %AppData%/Local/Temp/Coridium/.

Once BT is installed, and EBT's archive is extracted into the Coridium program folder, BT should be started. Once BT is started, to start EBT, one would click on the ***Tools|Debug|Extended Debug*** menu option that Coridium was kind enough to code into BT. If things are as they should be, BT will source EBT's scripts and one will be met with EBT's default UI.

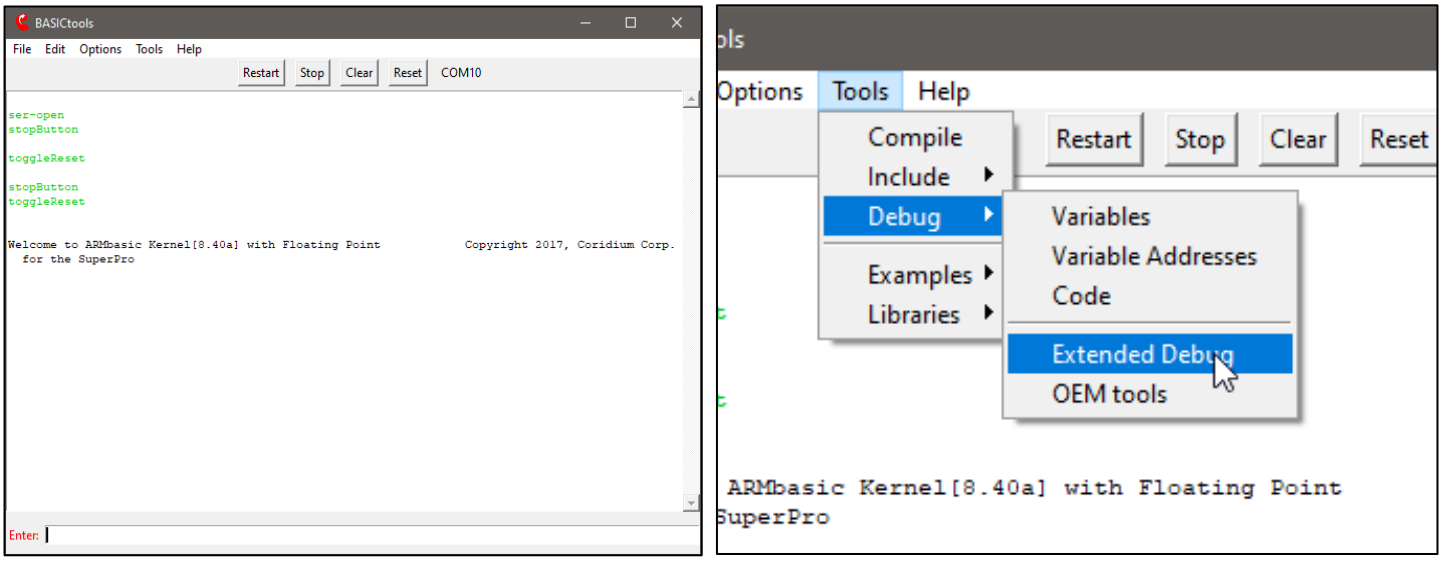


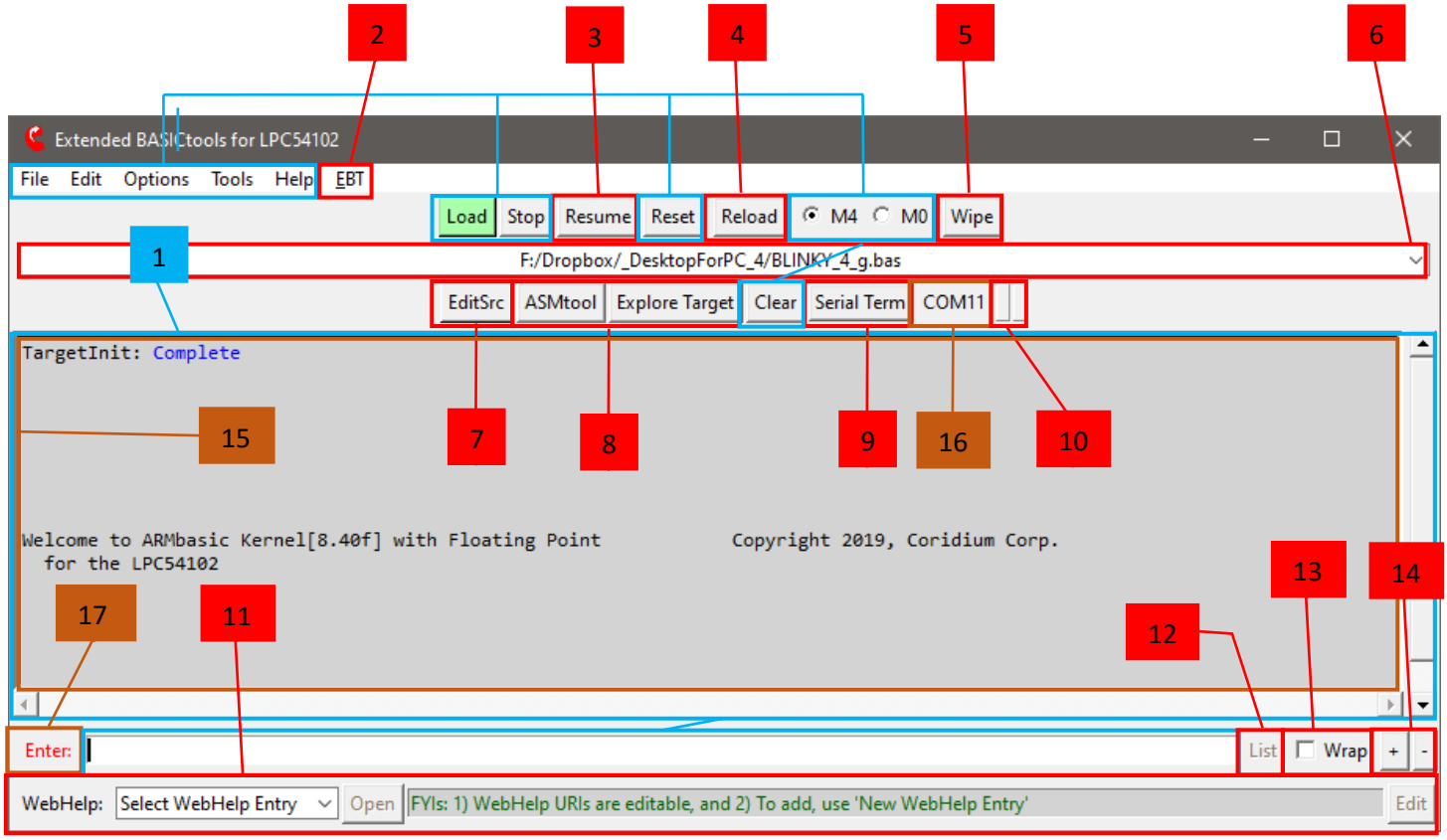
Figure 3: BASICtools Default UI – Starting EBT from BT proper

To further explain EBT functionality herein, screen shots and descriptive text will be used to illustrate the various aspects of using EBT. Also, be advised that there are two 'submodules' that EBT offers for dev's use – ASMtool and TargetExplorer (TE). Both of those tools offer substantive functionality which warrant their own addendums to this EBT User Manual. Those will be drafted and released as supplements hereto as quick as practical.

With that, let's get down to capturing tool images and explaining as much as practical about the feature sets and functionality of EBT's extensions to BASICtools.

EBT UI ELEMENTS – Existing, New, and Extended

The following image/table depict the elements that comprise the main EBT UI and provide details regarding the controls, their functions, and if functional differences which may exist between themselves and the BT-proper counterparts (if any).



EXISTING BT UI W/ MINOR OR NO CHANGES | EBT ELEMENT W/ NEW FUNCTIONALITY | BT ELEMENT W/ EXTENDED FEATURES

ITEM	DESCRIPTION
1	Existing **Stock BT UI elements** with functions that are largely unfettered (color/font/label/content excepted :-)
2	**EBT** Menu – will be detailed on dedicated pages herein
3	**Resume** Button – simulates the user typing a carrot (^) – useful after a 'STOP' debug command
4	**Reload** Button – mirrors the BT Reload menu function, with the User App in the DDL (#6) being used
5	Core **Wipe** Button – to overtly erase ARMbasic user app from the MCU – on Core M0, codes an endless loop
6	Historized **Recent** User App Drop Down List – similar to Recent Files in BT File menu
7	**Edit** Source File Button – opens the DDL (#6) selection in the editor configured during BT setup
8	**ASMtool** and **TargetExplorer** Modules – each of which to be detailed in dedicated manual addendums hereto
9	**Serial Term**inal Button – opens a TclTerm stand-alone serial terminal (for debug/monitor)
10	Serial Traffic 'LEDs' – **Red** = Host Tx (lt), **Green** = Host Rx (rt), are also buttons to unlock UI elements if needed
11	**WebHelp** MiniTool – will be detailed on dedicated pages herein
12	**List** Button – becomes enabled with iterative compilation activities, used to list the WIP User App code
13	**Wrap** Toggle – to turn on and off wrapping of text in the console
14	Increase**(+)**/Decrease**(-)** Font Size – for real-time adjustment of BT/EBT UI element text
15	BT/EBT Target **Console** – this is the drag and drop target for user AB apps (*see wrapped scripts dialog @ end*)
16	BT/EBT **COMxx** Port Indicator – in EBT is also a button that will re-enumerate the current serial instance
17	BT/EBT **Enter** Label – in EBT this is also a button to transmit a carriage return out the serial port

Table 1: Extended BASICtools UI Elements Index

EBT Menu

EBT's menu allow the user to access and/or control various aspects of EBT's functionality:

- Preprocessor Selection and various Preprocessor Settings for the selected EBT Preprocessor - FilePP.
- Menu selectable short-cuts to relevant explorer folders on the host dev box.
- Menu selectable short-cuts to various intermediate files generated by the tool chain during compilation.
- Control of various options that control EBT's UI, 'Windowing' on host OS, and functional behavior.
- Cessation of EBT with Restoration of stock BT functionality (imputes an automatic restart of the toolchain).
- On-demand Instantiation of TclTerm as an additional serial console (useful at times for debug/monitoring).
- Access to a composite pre-processed ARMbasic (AB) Source File – the toolchain's input to the AB compiler.

Each of these will be touched on, some in granular detail where appropriate, in the following dialog and images.

EBT Menu | EBT Source Code Preprocessors

BT-proper makes use of a customized version of CPP, which functions in a manner similar to CPP, and is called BPP, presumably an acronym for BASICpreprocessor (a reasonable assumption never thought to have affirmed). BPP is, by design and lineage, pretty focused at one thing, preprocessing source-code files with a rigid structure of functionality and capability. It is not intended to be expandable and is pretty limited in a myriad of other ways, as it was designed to do just one thing. Don't mistake these comments as non-appreciation. It does perform its designed functions pretty darn well. However, when one starts to request/expect CPP/BPP to do more, the wheels can depart the bus pretty quickly.

These limitations (was and still is perceive as such) was the initiating factor that caused the hunt for other preprocessors to begin. It didn't take too long to find FilePP. After testing FilePP, and eventually realizing just how extremely powerful such an extendible preprocessor solution was, it became evident that FilePP needed to become an EBT team member, and was integrated as one of the key extensions that EBT brought to the table for a dev to leverage in any AB dev effort.

It is acknowledged that, for the vast majority of AB dev work, BT's Stock BPP is more than adequate at preprocessing AB source file, doing so really well. However, it would be appropriate to acknowledge that there are some specific feature sets that FilePP empowers an AB dev with that simply cannot be accomplished with the BPP tool as it exists today.

While in-depth details about what FilePP does, how it does it, and how to get it to do it, is well beyond the scope of this manual, the following listed items are some of FilePP's features that see regular employment by EBT's dev:

- Compile-time Maths
- Multi-Level Preprocessing Debug
- C comment Removal
- AB Comment Removal
- For-Next Loop Preprocessing
- Multi-Line Macros
- Selectable Boundary Macro Expansion
- Inner Literal Macro Expansion
- Regex-based Macro Definition & Expansion
- Multi-file Macro Definition, Build-up, Expansion

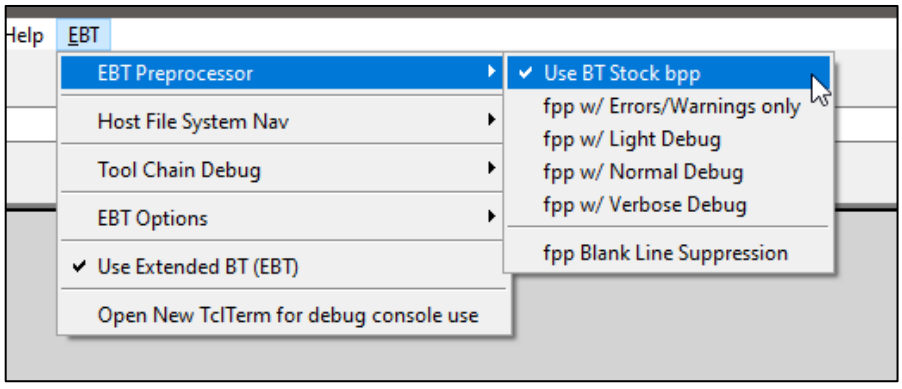


Figure 5: EBT Preprocessor Selection and Control

EBT Menu | Selectable Windows Explorer folders short-cuts

It should be pretty self-evident what these various menu selections do. If one doesn't grasp this, one may be well served to consider taking up knitting or some other hobby (kidding – if you get wrapped around the axle on these, just ask in the forums and someone should be able to help).

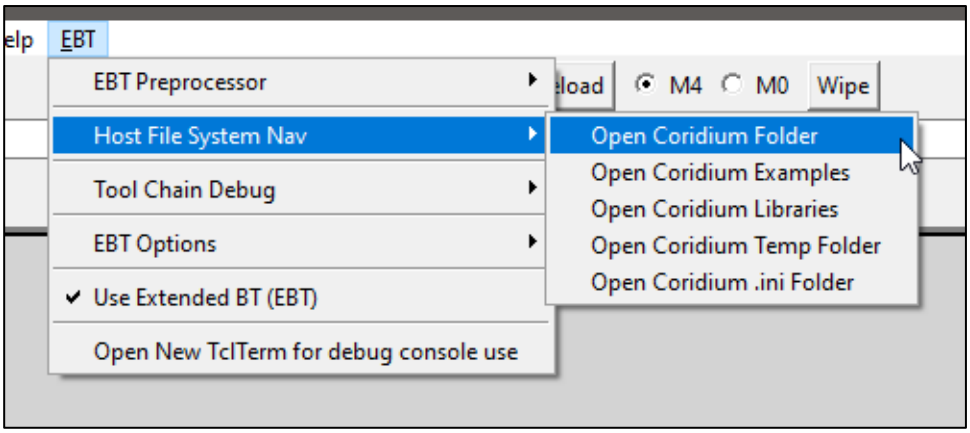


Figure 6: EBT Host File System Folder Short-cuts.

EBT Menu | Tool-chain Intermediate File short-cuts

As with the previous, and for anyone familiar with a typical tool-chain process, extrapolating what these menu-selectable options do should be pretty self-evident. Again, if one gets wrapped around the axle on these, just ask in the forums and someone should be able to help.

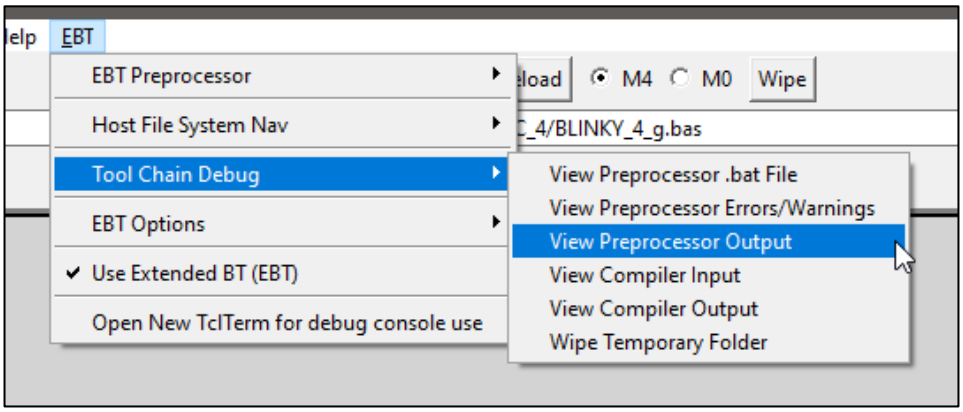


Figure 7: EBT Tool-chain Intermediate File Short-cuts.

EBT Menu | EBT's UI Options, Host-OS 'Windowing', and Optional Functionality

BT and EBT have 'normal' windows on a Windows OS Host box. By normal, one is inferring that the window size is adjustable, has a title, has an icon on the task bar, and can be manipulated to be on top or not, as the case may be.

Windows supports having Modal windows or Tool Windows, where the windows have title bars, but lack some of the dressing and control that 'normal' windows have. One of these differences are that tool windows don't normally have icons on the Windows Task Bar (WTB). Turning on tool mode for the Target Explorer (TE) removes the WTB TE Icons. If one elects to have their WTB configured to always combine the icons thereon, if TE windows were normal windows, and one of them were minimized, when one clicked on the WTB icon, one would then be offered thumbnails of the windows to click on to select which window is desired to be brought to the top and focused. It can be frustrating to have to do so.

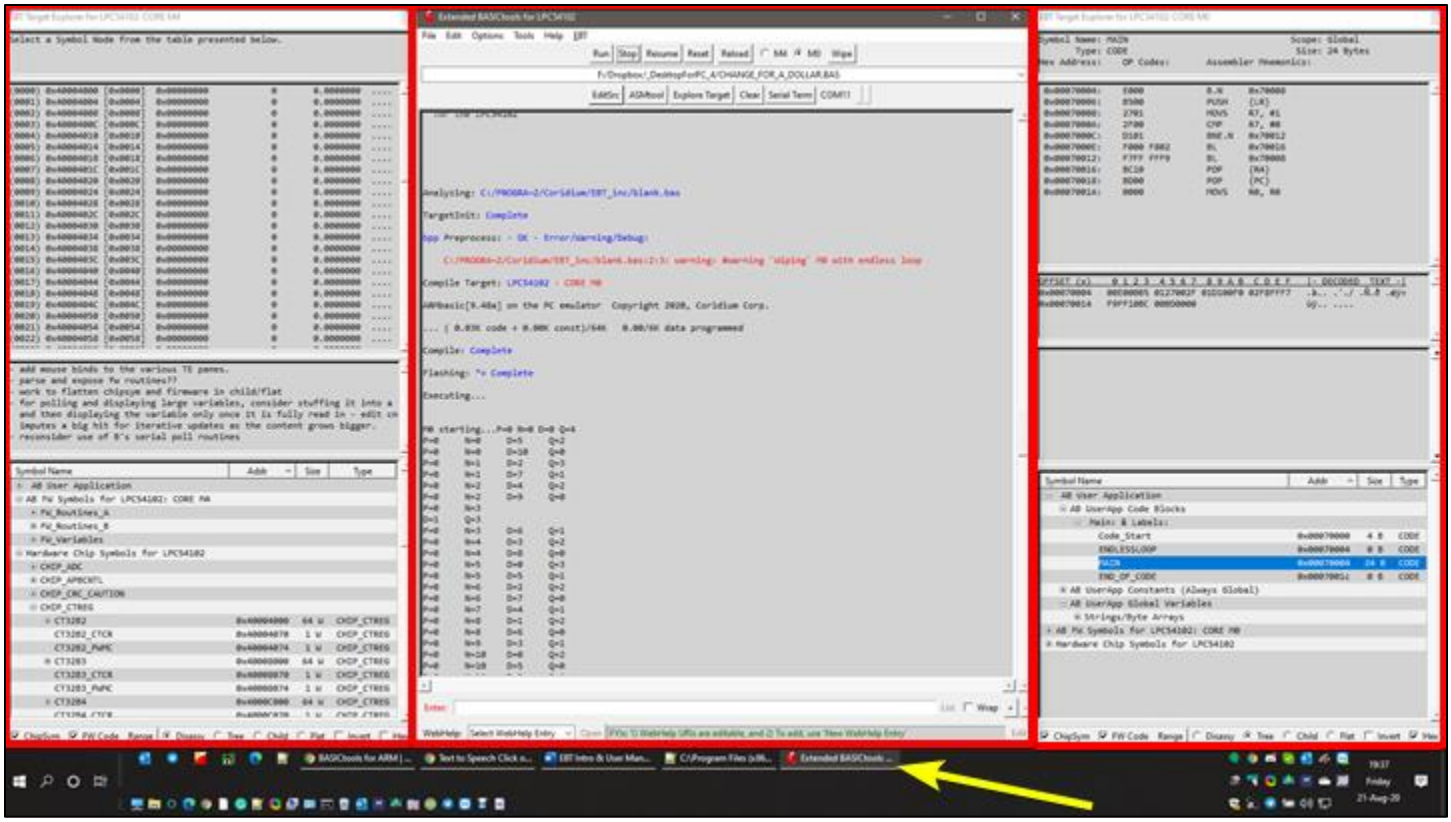


Figure 8: EBT Main UI Window with two TE 'Tool Windows' (note the single Task Bar Icon)

Thus, the option to make TE windows 'Tool' windows was coded into EBT. Additionally, it was determined that in some use cases, it might be desirable to have an option to force windows to keep either EBT's main window on top, or to be able to do so with the TE windows, regardless if they are 'normal' windows or tools windows. The options for indication and control of these window modes exists on the EBT Options menu.

With tool windows, typical observed behavior is to have tool windows minimize with the host-app's main window. The 'Minimize TE when EBT Minimized' option enables this behavior.

The 0x prefix option enables TE to display hex numbers with the non-standard 0x prefix notated onto a hex number. BASIC standard is often either $ or &H (ARMbasic is &H). For some users, the 0x prefix either is visually easier to employ, or is a habit of muscle memory borne from working with other dev languages. This is why the 0x option exists.

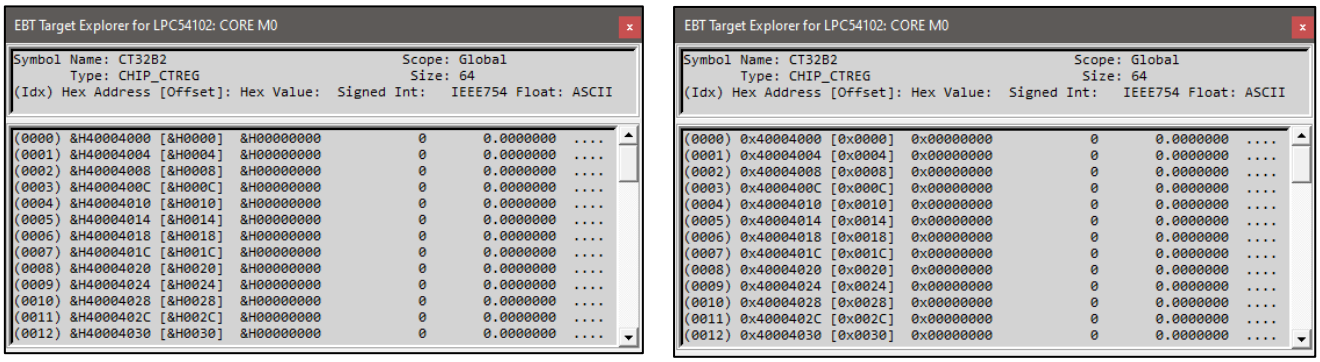


Figure 9: Stock AB &H hex prefix (left) vs. EBT's Optional 0x hex prefix (right)

Anther BT UI default behavior is to wait until the first time a target is programed before it updates the UI to reflect said target, not only in the title bar, where it depicts the controller with an expressive verbose name, but also on the UI, if the unit is a multi-core device (i.e. LPC4330, LPC54102, etc.), where the MCU Core selection radio buttons come into existence.

EBT's approach can be made to be a bit more aggressive in where it actively queries the target controller at startup and updates the UI accordingly. The reasoning for having EBT's optionally behave be this way is two-fold:

1) It enables the UI elements to be updated at startup
    a. admitted eye candy, but for those who have a bit of CDO (OCD in alphabetical order, as it should be…) in them, this can make for a lower-stress user experience, at the price of a bit of startup latency to EBT.
2) It enables EBT to perform some background operations related to the firmware options in TE.
    a. Note: These FW options within TE are still being worked on and are NOT mature yet (hence the 'placebo').

Another way in which to customize EBT is to modify the fonts used on the UIs. This is accessed by the Choose EBT Font option in the EBT Options Menu. Selecting same pops up a font-picker UI. It is relevant to note that the Monospaced fonts are highlighted in Red text on the font name selection control. Additionally, in the sample control, one can observe what is often salient to developers – Visual presentation of specific characters (Slashed Zero (Ø) vs. letter o/O, Upper-/Lower-case iI vs lL vs 1, 2 vs Z, 5 vs S, etc.). The author prefers Consolas due to its commonality and these visual elements. The font selection is for all of the UI elements that are generated by EBT. Menus and control text is all stock BT font.

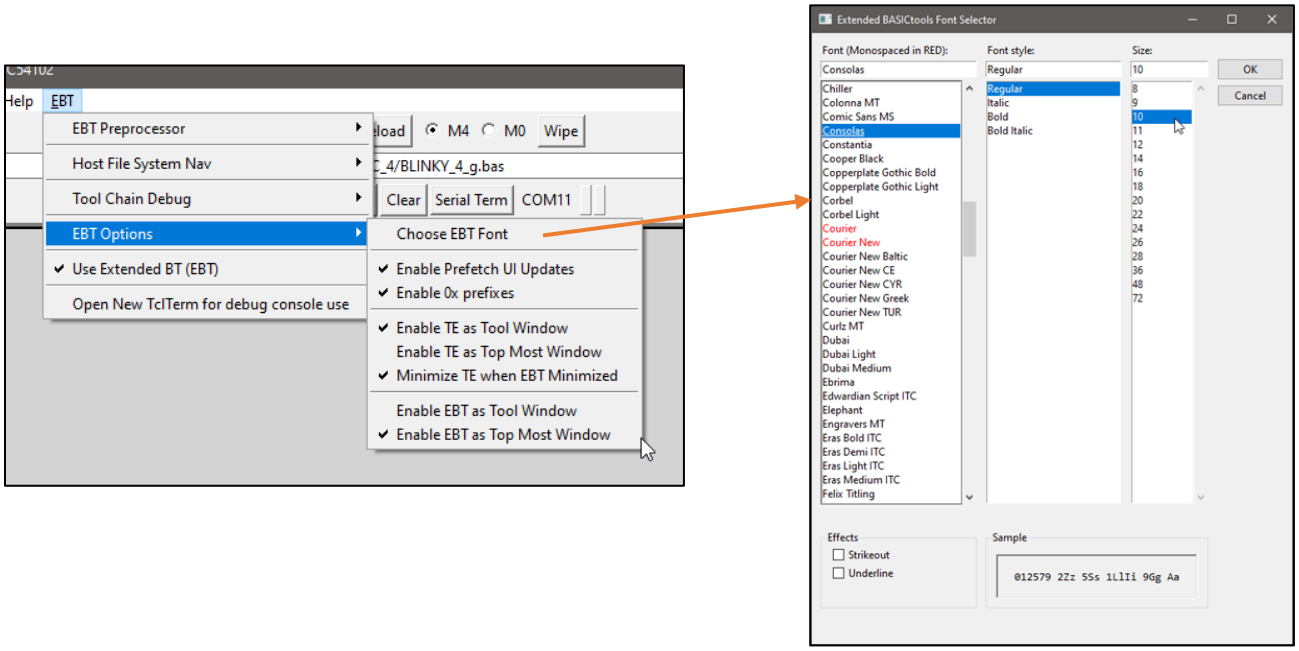


Figure 10: EBT Options - Window Modes, Prefetch, and Font Selection

EBT Menu | Use Extended BT (EBT)

When EBT is active, this option is enabled. It is provided on the EBT menu to enable the dev user to deselect and restore the Stock BT functionality. Deselecting this menu option will cause a restart of the toolchain, in which the stock BT UI will be restored upon restart.

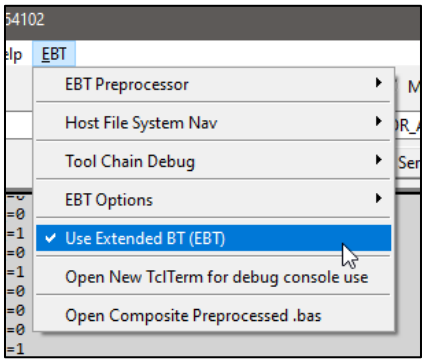


Figure 11: Use Extended BASICtools – Deselect to restore Stock BT-proper functionality

EBT Menu | Open New TclTerm Instance

This menu option opens up a separate instance of BT's TclTerm Serial Terminal Program. A dev may find this useful to have the target stream debug out of a separate serial port (possibly at uber high rates – author has had 1.2mbps debug streaming from a Coridium SBC Target). Same could also be used to enable serial IO comms with a 2nd core on a multi-core target (4330/54102/…) via a separate serial port (peripheral or bitbanged via GPIO).

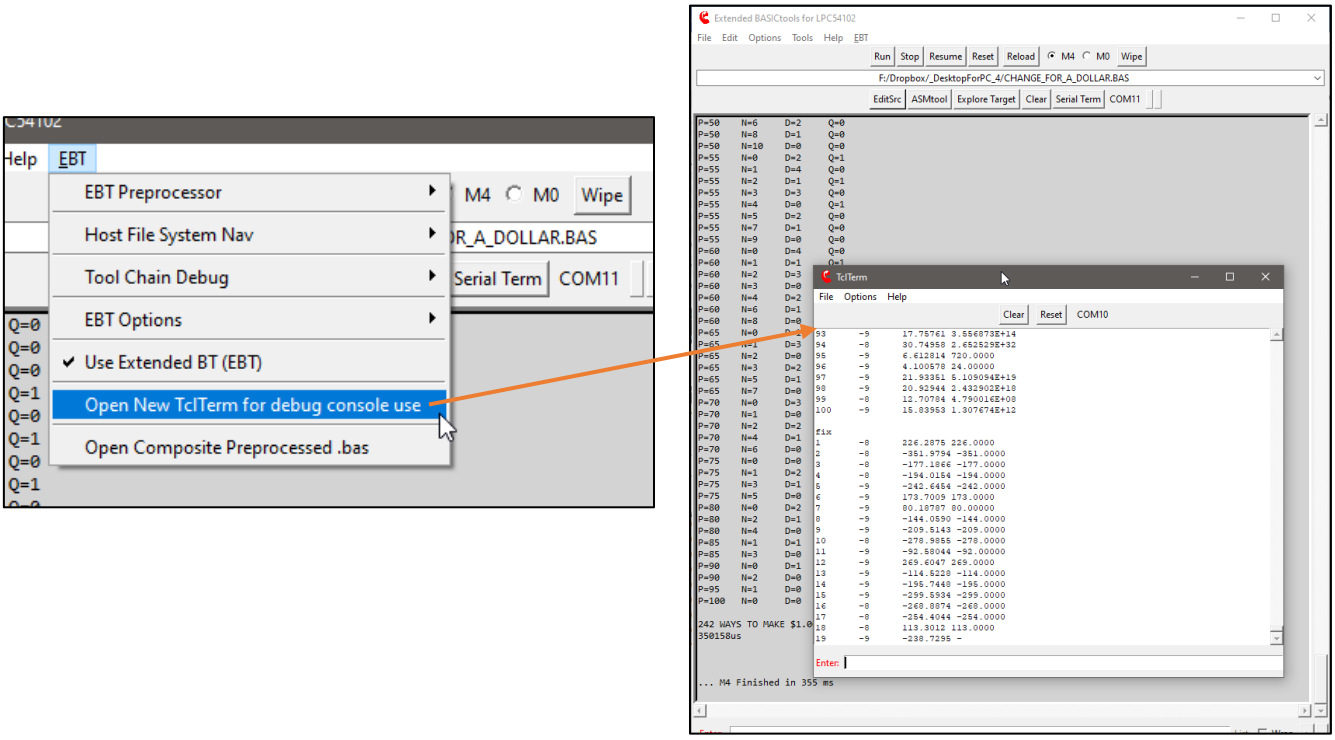


Figure 12: Open New TclTerm Instance (useful for Debug Console (or for 2nd-core comms) via other serial port)

EBT Menu | Open Composite Preprocessed.bas

This menu option, when available (disabled prior to compilation), opens up an instance of the configured editor to allow the dev to review the composite .bas (after all BPP/FilePP preprocessing) to validate preprocessor impacts, etc.

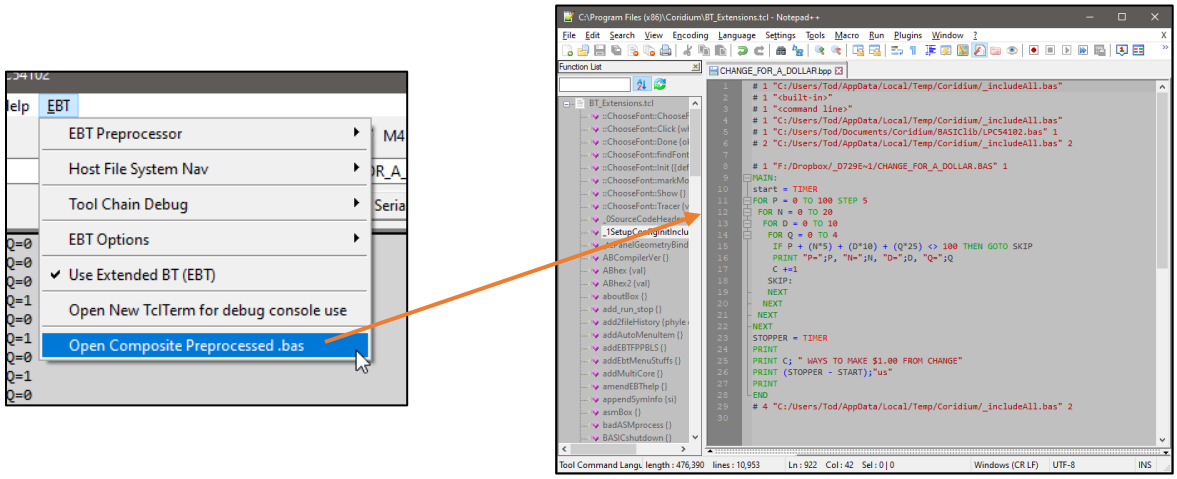


Figure 13: Open Composite Preprocessed.bas – available after sessions first compilation/flash load

EBT WebHelp MiniTool

EBT's main UI contains what has been termed as the WebHelp MiniTool at the very bottom of the window.

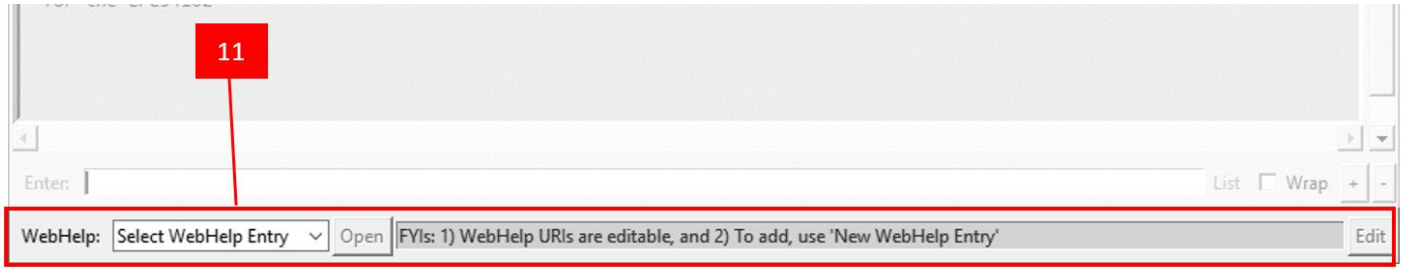


Figure 13: Open Composite Preprocessed.bas – available after sessions first compilation/flash load

This tool is intended to be a place in which links relevant to EBT, AB, MCUs, or whatever else a dev might find useful to have at their fingertips when working within EBT. As one can see in the following image, the author has a number of web resources that can be quickly opened by merely selecting same and then clicking the Open button alongside the DDL.

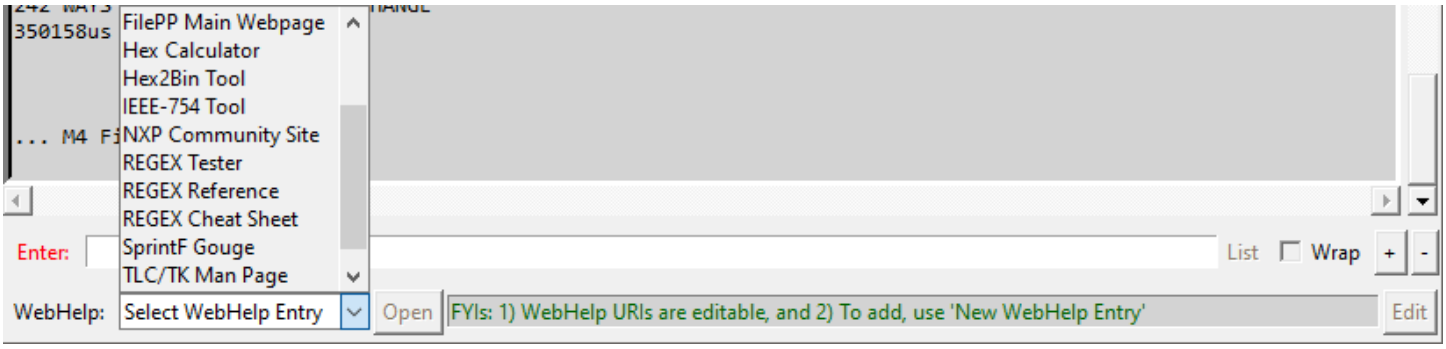


Figure 14: WebHelp MiniTool Drop Down List depicting the current entries available for selection

Figure 15: REGEX Tester selected and Open being clicked – will open a browser to regex101.com, as depicted

The author has programmed EBT's WebHelp MiniTool to allow new WebHelp entries to be added or edited in an easy and intuitive manner. The steps needing to be followed are detailed in the following images:

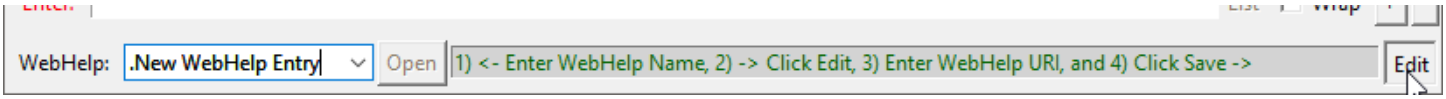


Figure 16: Step 1: Select New WebHelp Entry

Figure 17: Step 2: Clear the entry name and click Edit (Edit button changes to a Save but)

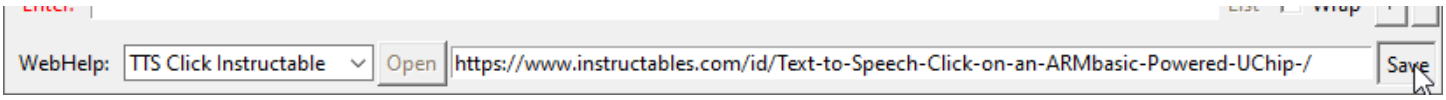


Figure 18: Step 3: Enter WebHelp Entry Name, Enter URL for the resource, and then click Save button

That is all there is to it. Anytime it is desired to visit that resource, just select it and then click the Open button alongside.

Figure 19: Selecting one of the entries in the WebHelp MiniTool

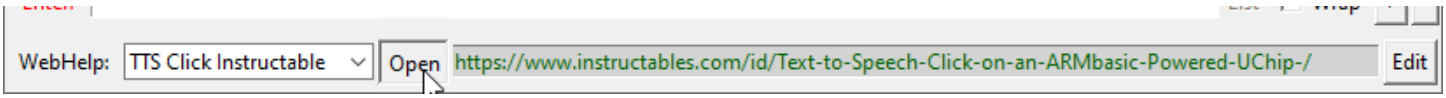


Figure 20: Clicking Open to launch a browser to open the selected resource

All EBT WebHelp MiniTool entries are backed up to the .ini file.

Also, the ASMtool, to be detailed in an EBT UM addendum (when completed), also has a WebHelp MiniTool. While the UI and controls are the same, the WebHelp entries therein are a separate set (also backed up in the EBT .ini) given the context of using the ASMtool to help compile ARM Assembly is likely [much] different than when developing ARMbasic.

Conclusion

Thank you for taking the time to read about using the EBT toolset. If you have already tried EBT, or decide to do so, the author will be pleased. It is understood that it might not be to everyone's liking and that is OK. Trying to please everyone ensures a quick and painful trip to certain failure, which is not the intent here. If it does provide a few folks some usefulness, then it has achieved a goal.

If you experience trouble with EBT there are three options available to you to try to secure assistance:

1) Posting to thread on the Coridium Forums where EBT was announced,
2) Cracking open the tcl sources and see if you might glean a better understanding of what is transpiring and how it might be able to be resolved, or
3) Joining the Telegram ARMbasic Channel (https://t.me/ARMbasic) and seeing if anyone there might be someone who can offer input or assistance.

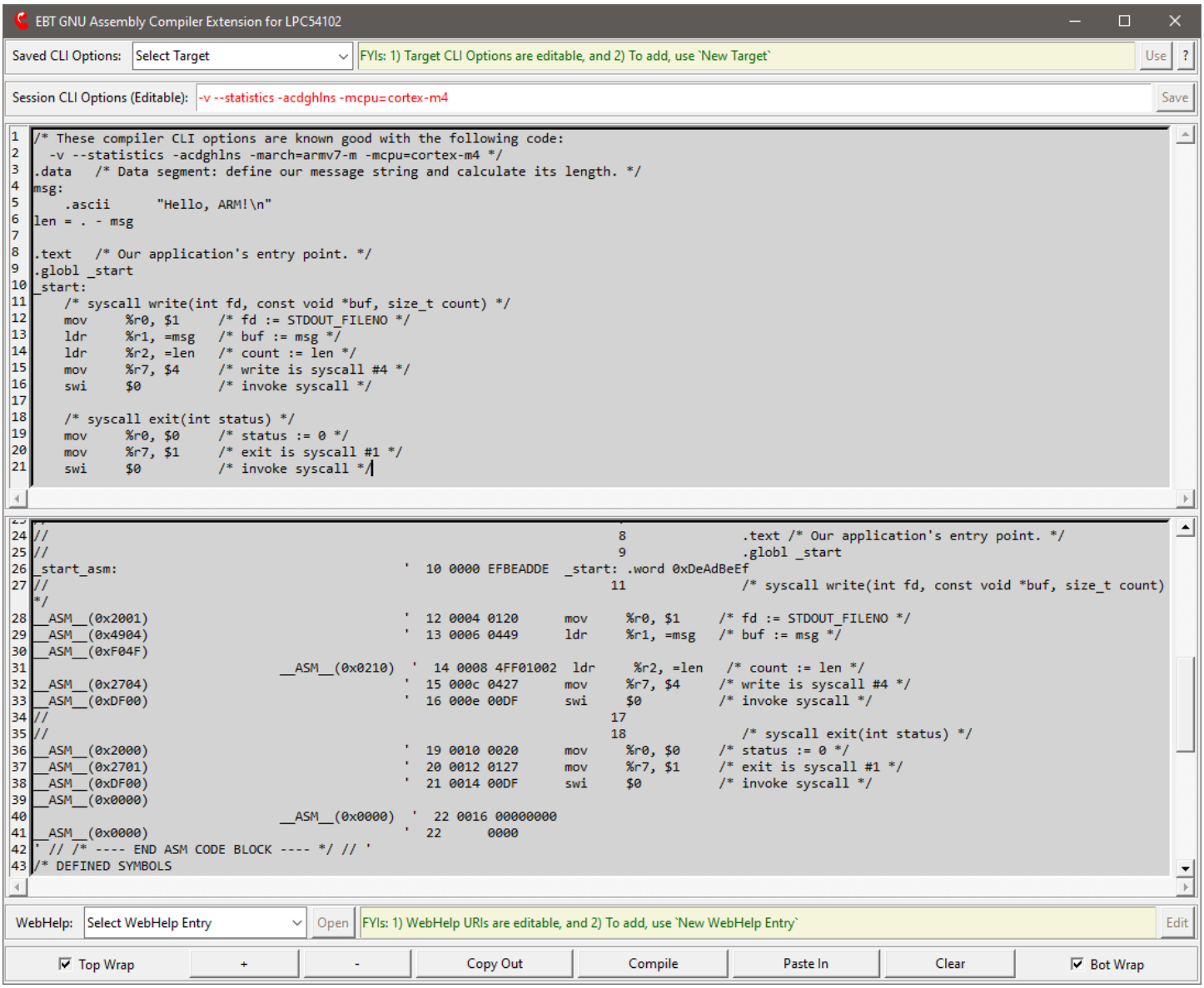
The author bids you well and hopes each and every one of you experience success. Take care and Happy Coding!

-t

Wrapped .tcl scripts … a note on borked Drag-N-Drop features of EBT

.tcl scripts can be ran directly, if a TCL/TK Wish Shell Interpreter is installed. Search for TCL Windows Binaries and you should come across something that will work. Author uses Iron TCL. Coridium enables BT to be used sans an installed interpreter via FreeWrap tcl script wrapper, providing an interpreted runtime wrapped around a .tcl script. BASICtools.exe is a 'wrapped' .tcl script. It has been noted that the more recent versions of FreeWrap creates a context that breaks the Drag-N-Drop EBT feature but is otherwise non-intrusive to EBT's ops. **Three options exist**: **1)** use EBT w/ BASICtools.exe accepting no DND support, **2)** Install a tcl interpreter, associate .tcl scripts w/ the tcl interpreter, & run BASICtools.tcl directly, or **3a)** Create a BASICtools.exe w/ an older version of FreeWrap or **3b)** request same via forums/telegram channel.

Addendum 1: EBT ASMtool Use … PLACE HOLDER for ASMtool Manual



Addendum 2: EBT TargetExplorer (TE)

User Manual

Version 1.0 26 Aug 2020

(Record of Revisions located in SubAppendix 1)

Introduction to EBT's TargetExplorer (TE) Module

TargetExplorer (TE) is an Extended Basic Tools (EBT) module that allows an ARMbasic (AB) Developer to interact with the Target microcontroller in a somewhat-intimate manner, to aid in debugging and AB User App Development. EBT is a set of extensions for the Coridium BASICtools IDE which Coridium Corporation provides for free when used with any of the Coridium SBC Boards. The scope of this document is related to the TE and it is not intended to be a User Manual for Coridium's BASICtools IDE BASICtools (or EBT). This work assumes that one has a basic to good understanding of BT/EBT, ARMbasic, and how to use each on SBCs that have ARMbasic Firmware thereon.

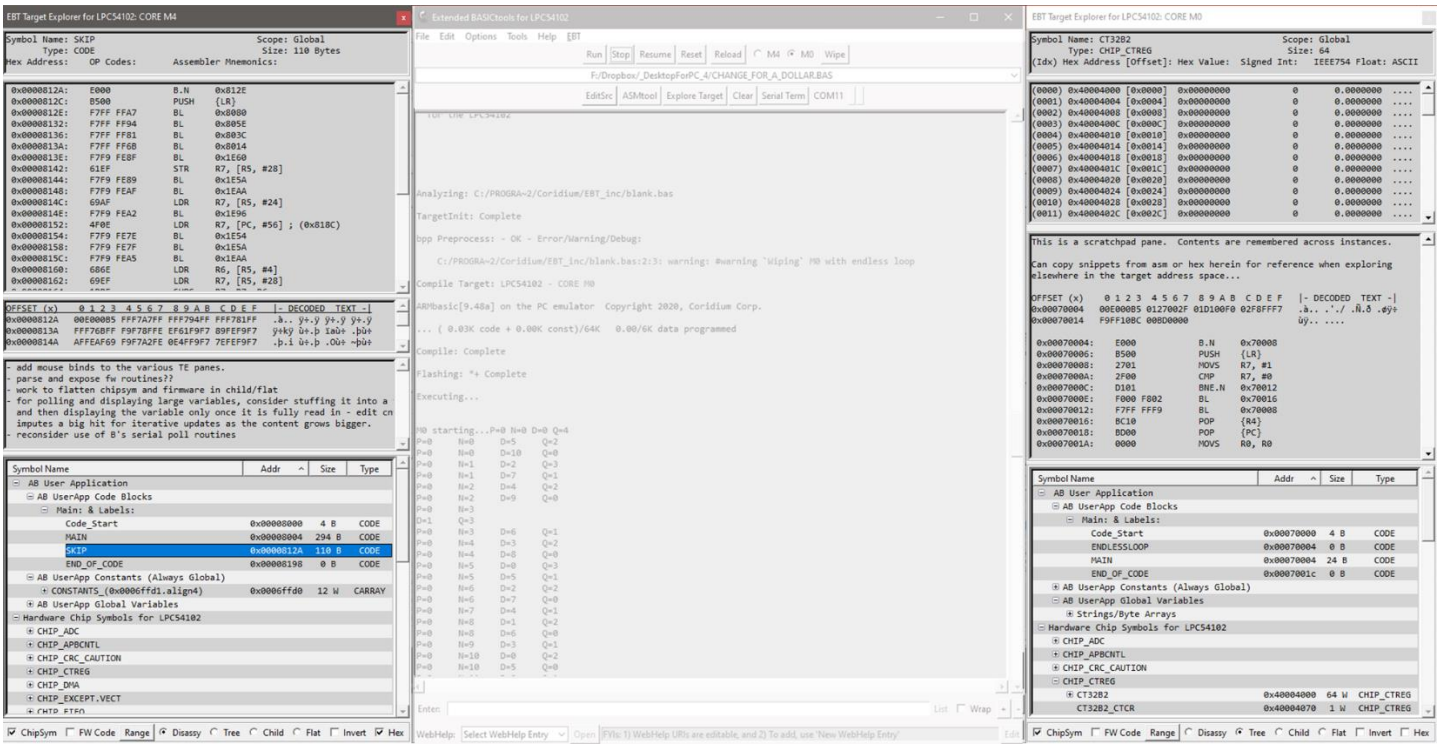


Figure 1: EBT (grayed out) with Dual Target Explorer Windows activated

TE UI Activation and Window Modes

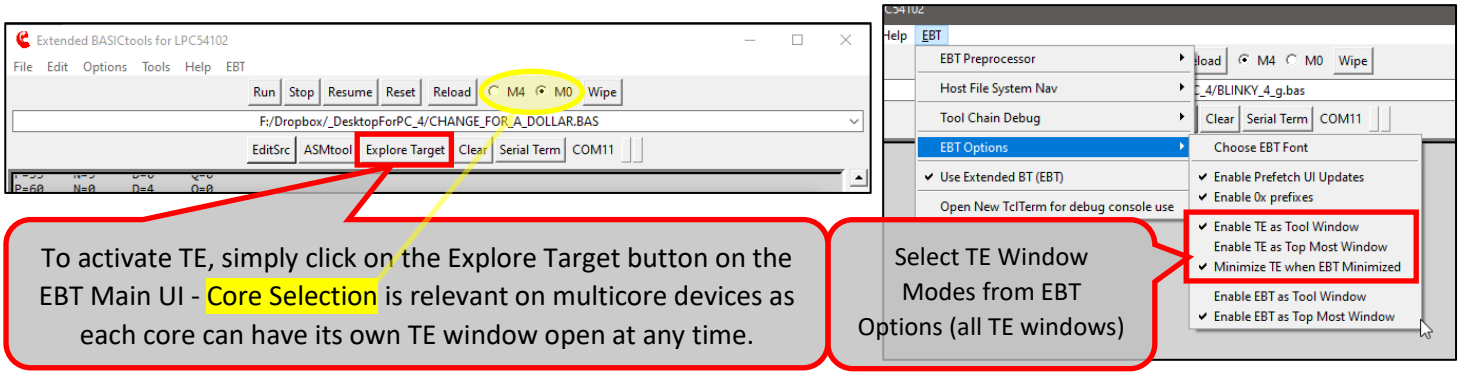


Figure 2: Activating Target Explorer – Selecting TE Windowing Mode in Host OS

EBT defaults to all modules (EBT, ASMtool, TargetExplorer) having 'normal' windows on a Windows OS Host box. By 'normal', the author is inferring that the window size is adjustable, has a title, has an icon on the task bar, has minimize and restore title bar buttons, and can be manipulated to be on top, full screen, or not, as the dev's workflow may drive.

Additionally, Windows OS supports having Modal or Tool Windows, where the windows may have title bars, but lack some or all of the dressing and controls that 'normal' windows have. One of these differences are that tool windows don't normally have icons on the Windows Task Bar (WTB). Turning on tool mode for the Target Explorer (TE) removes the WTB TE Icons. When one elects to have their WTB configured to always combine the icons thereon, if TE windows were normal windows, and EBT was minimized, when one clicked on the EBT WTB icon, one would then be offered thumbnails of the windows to click on to select which window is desired to be brought to the top and focused. It can be frustrating to have to do so, when one might reasonably just expect the EBT window to restored or be brought to the top and focused. Selecting Enable TE as a Tool Window on the EBT Options menu serves to impute that operating mode for TE windows.

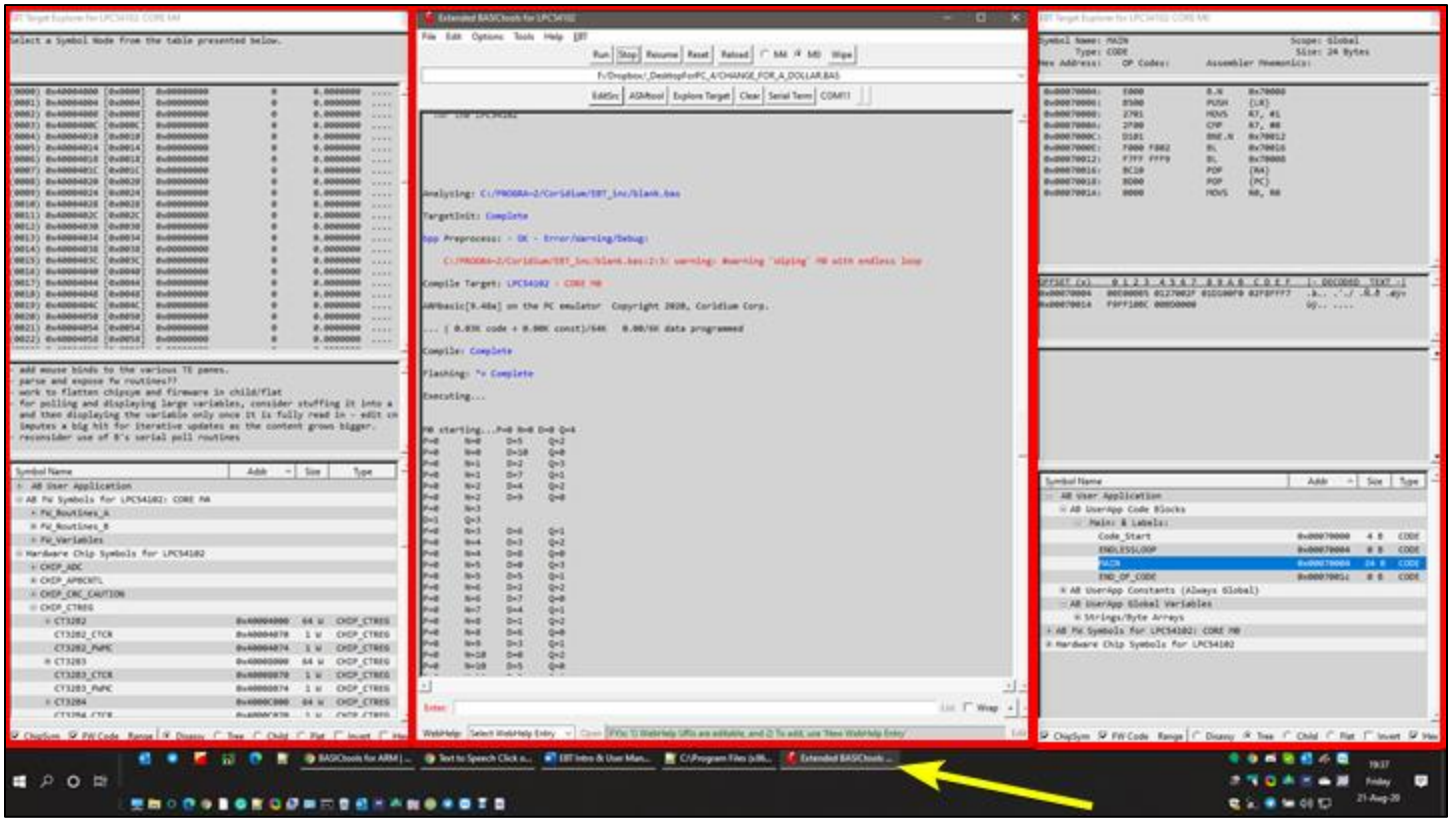


Figure 3: EBT Main UI Window with two TE 'Tool Windows' (note the single Task Bar Icon)

Additionally, it was determined that in some use cases, it might be desirable to have an option to force windows to keep either EBT's main window on top, or to be able to do so with the TE windows, regardless if they are 'normal' windows or tools windows. The options for indication and control of these window modes exists on the EBT Options menu.

With tool windows, typical observed behavior is to have tool windows minimize with the host-app's main window. The 'Minimize TE when EBT Minimized' option enables this behavior. Restoring EBT's TE windows with restoration of EBT is on the menu of UI behavior to enable and have menu selectable. Not that the TE UI is activated and controlled how a dev user desires, the next item on the agenda is to discuss the TE UI.

TE UI Element Overview

The following image depicts the elements that comprise the Target Explorer UI and enumerates the name of the controls, as referenced in the TE UI Element and Control Details section immediately following.

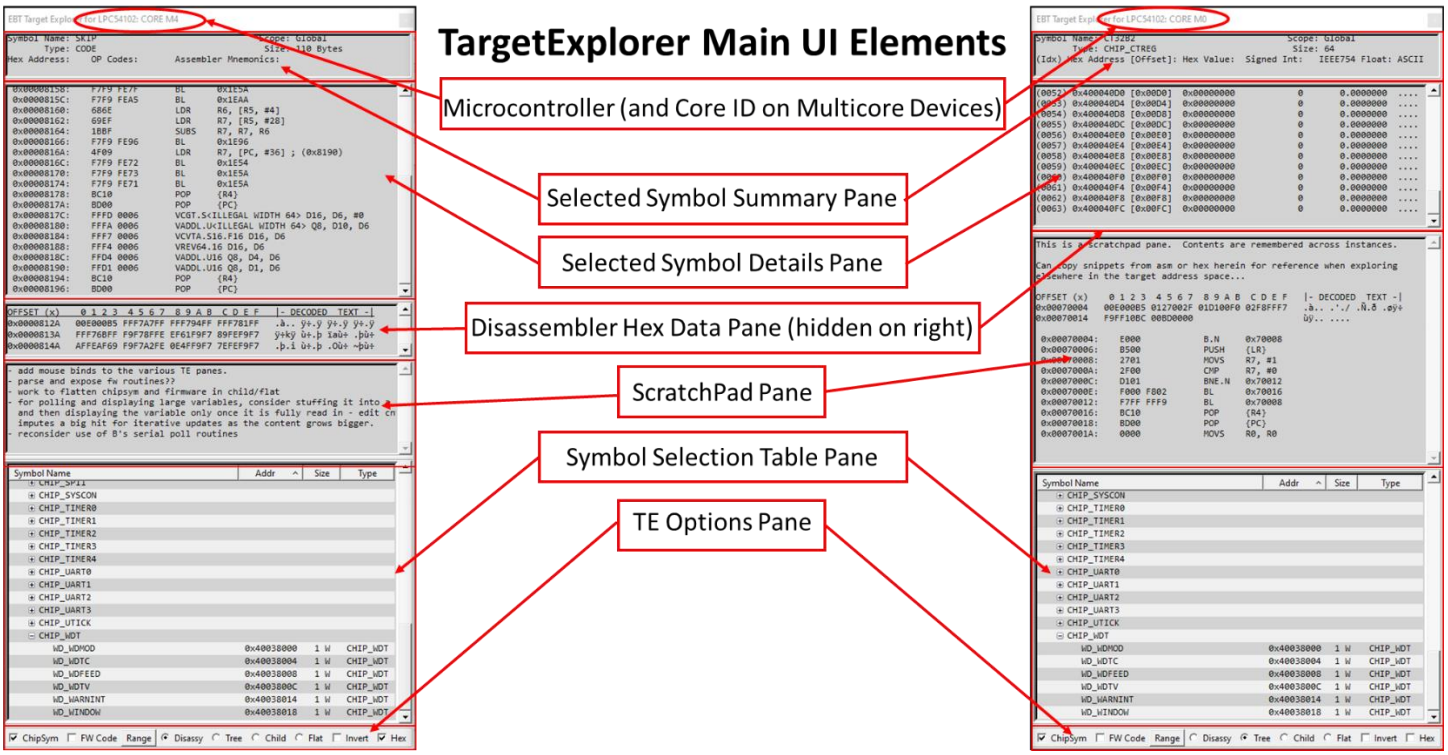


Figure 4: TargetExplorer UI Elements (Dual Core LPC54102 depicted, with a TE instance for each core (M4 & M0))

TE UI Element and Control Details

TE is, at its roots, a multi-pane control. In its current design, there are 6 panes, in addition to the title bar:

- Selected Symbol Summary Pane
- Selected Symbol Details Pane
- Disassembler Hex Data Pane (hidden on right)
- ScratchPad Pane
- Symbol Selection Table Pane
- TE Options Pane

Each are explained in detail in the following. The panes positioning is sticky across instances, being an .ini file saved item. It is prudent to note that the top and bottom panes are automatically positioned. These are event driven adjustments. In some cases, it is not always as one might expect - just adjust another pane and these should snap into place as designed.

Title Bar - Microcontroller (and Core ID on Multicore Devices)

TargetExplorer's title bar reflects the Connected Target Name and, in case of a multicore device, details the core that a particular TE is 'connected' to – i.e. Core M4 or Core M0. The author has found that this is most useful when there is more than one instance of EBT running, supporting concurrent connectivity to multiple targets, and when multiple instances of TE are running.

Selected Symbol Summary Pane

This pane displays high-level header details for the symbol selected in the Symbol Table Pane. The title bar depicted in the following images are what it looks like when TE is configured as a Tool Window.

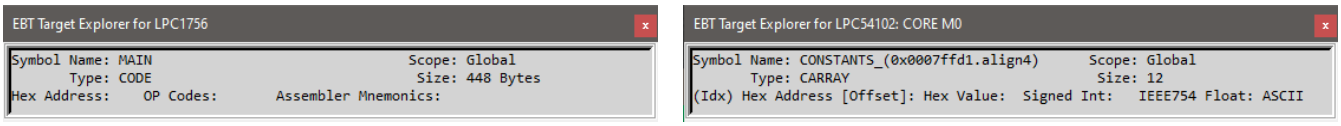


Figure 5: TE Summary Panes – Code Header on Left, Variable Header on Right (note Core M0 indication)

Selected Symbol Details Pane

This pane displays details for the symbol selected in the Symbol Table Pane. Data is read from the target, based on the symbol selected in the Symbol Selection Table Pane – a code symbol on the left and a 12-word Constant Array on the right.

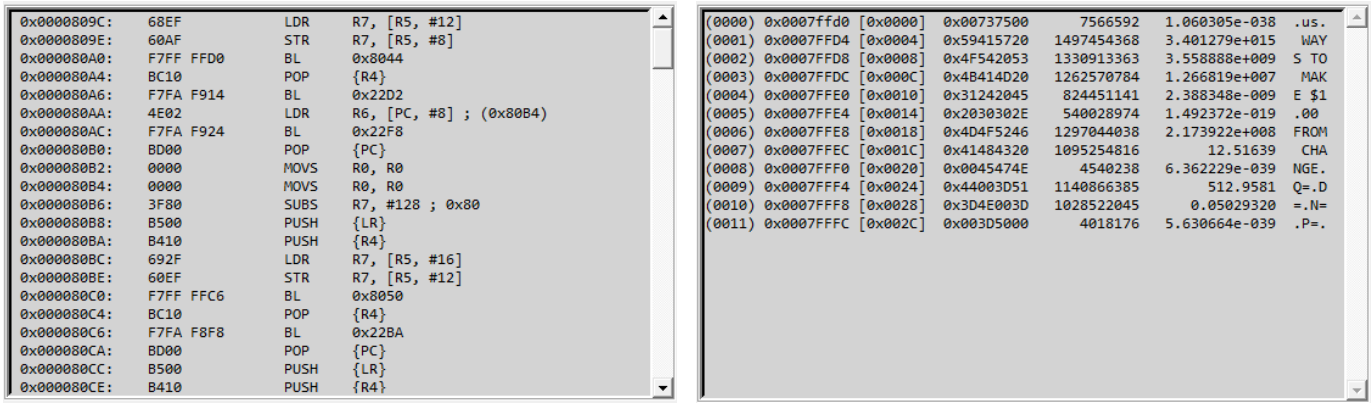


Figure 6: TE Symbol Detail Pane – Code Disassembly on Left, Variable details on Right

Currently there are not any context menu mouse binds on this pane, so to copy content from therein, one needs to use the shortcut Control-C after highlighting with the mouse. A context menu will be added in the not-too-distant future.

Disassembler Hex Data Pane

This pane is opened when a code block is disassembled. It depicts raw hex retrieved from the target when it was polled in response to a Code Symbol being selected in the Symbol Selection Table. It closes automatically when a non-Code symbol is selected.

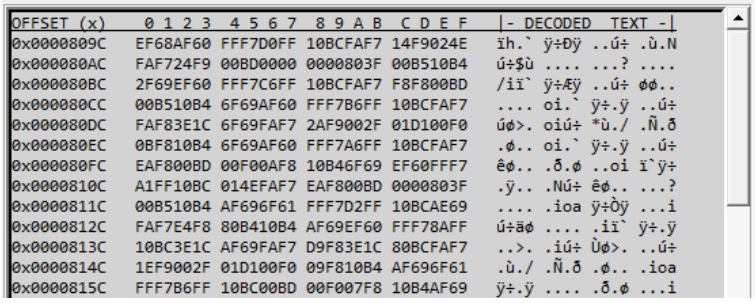


Figure 7: TE Raw Hex Data depicts the data received when target was polled for Code Disassembly

Note the Hex Pane is normally hidden when a non-code symbol is selected in the Symbol Table. If there is a need to look at the latest disassembly hex, when viewing a variable, one can click the Hex Checkbox at the bottom right of a TE window.

ScratchPad Pane

This is a pane that was originally intended at the Hex Data holder, but during TE Dev, it was determined that it might be useful to have a scratchpad pane that would be able to receive text snippets (typed or pasted in), and to keep the content of same saved in the .ini file, so that the same data is available across restarts of EBT or TE. It survived as such… ;)

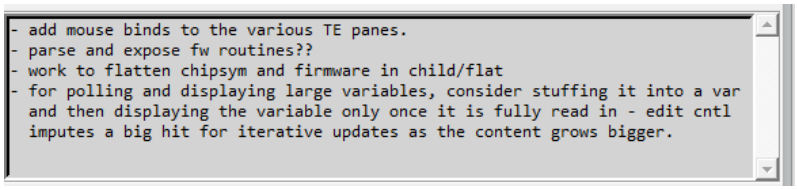


Figure 8: ScratchPad Pane in TE – for simple note taking that is .ini backed up

The intent here is merely as a simple down & dirty scratchpad, not that of an editor. If one wishes to save or manipulate the data in the manner offered by an editor, just copy the data for use in an editor, as an editor the scratchpad is not nor will it ever be. Braces are special characters in tcl and are transcoded during saving of the ini file – { & } become « & ».

Symbol Selection Table Pane and the various 'modes' thereof

The heart of TE is the Symbol Table Pane. Clicking on an entry herein will cause TE to poll the target and display related data that is received and processed for display.

TE Options Pane

The bottom pane of TE is the TE Options Pane. This is where the Symbol Table display mode is controlled, whether Chip Symbols or Firmware address monuments are displayed, if multi-element symbols are inverted in sequence when displayed, and also contains a [Manual] Range button for ad-hoc polling and disassembly of any valid address range. Each of these are detailed in the following:

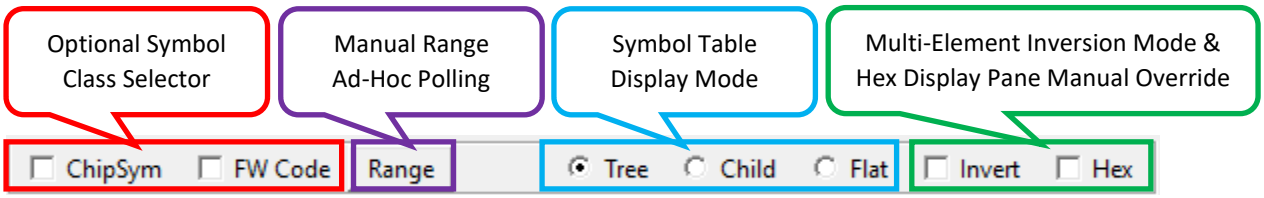


Figure 9: TE Options Pane Controls

ChipSym[bols]

If a chip's symbol library exists for a target that has a TE session open, the ChipSym checkbox is enabled and, if the dev user so desires, s/he can select to have the Chip Symbols displayed in the Symbol Selection Table Pane by clicking to place a check in this box.

It is prudent for the author to point out that these libraries are guaranteed to NOT be free of errors and are very likely incomplete. They were built on an ad-hoc basis from either .bas AB chip libraries that Coridium includes with BASICtools, or from chip header libraries secured from an OEM's library (likely original source of the former).

These are gladly subject to community review and revision where appropriate. Please do not hesitate to offer edits for the existing Chip Library .tcl files, or for new AB controllers in which a Chip Library doesn't yet exist. Example Symbol table display, with the Chip Library selected, and some parent entries expanded, is depicted on the next page.

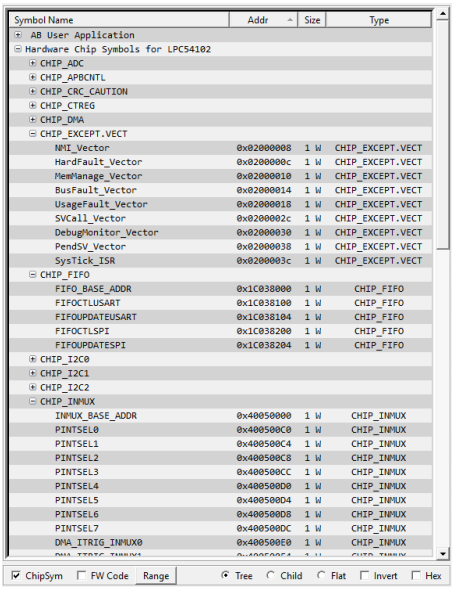


Figure 10: TE Symbol Table with Chip Symbols enabled on a LPC54102-based Coridium SBC

It is appropriately noteworthy to mention that the mere act of reading some registers can have unintended consequences. In those cases where the author has identified same a '_Caution' suffix is appended onto the potentially offending Symbol table entry/groups of entries.

F[irm]W[are] Code (likely to be changed to FW Addys soon)

Currently this is a Work In Progress (WIP) and only displays bogus placebo data. The author hasn't come to a final determination on what this will actually do once this option's implementation matures. Standby … More to Follow, eventually…

Invert

This checkbox serves to invert the display of multi-element symbols – i.e. arrays.

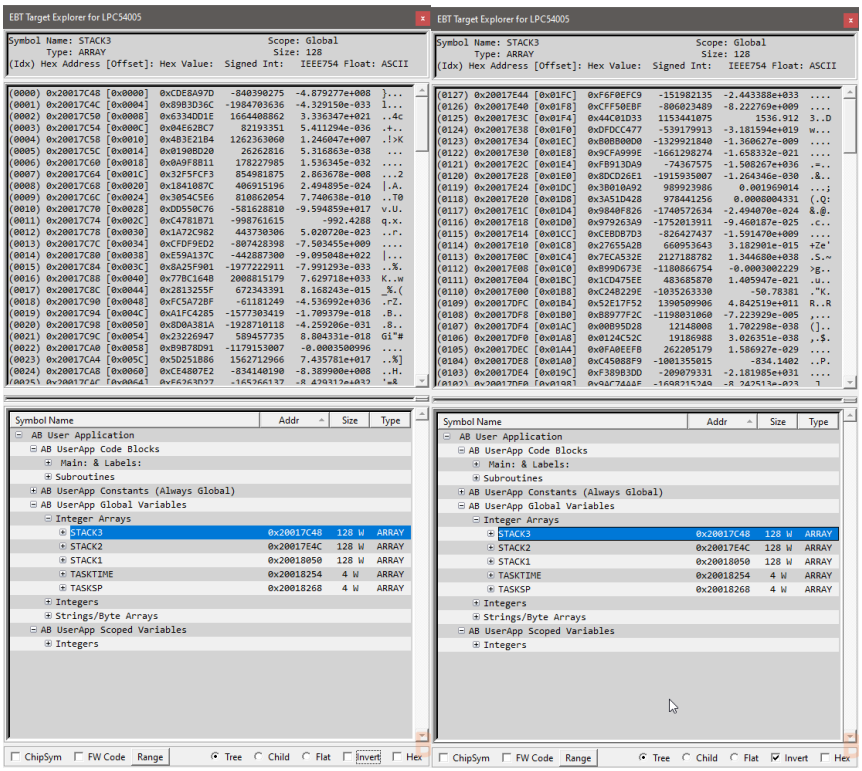


Figure 11: Non-Inverted (Left) vs. Inverted (Right) Array Display

The author elected to implement this functionality primarily for the purposes of reviewing fully descending stacks that are variable based (i.e. in a multi-task or RTOS application's context).

Hex

This checkbox is primarily automated, becoming enabled when a code block is disassembled. When checked, opens the raw hex pane to display the raw hex that is read in from the target and piped to the disassembler when a code block is disassembled for display in TE. This control can be clicked when reviewing a variable to see the latest disassembly in the event same might be useful when reviewing code/variable aspects of an AB User App. Reference Figure 7 above.

[Manual] Range Button

The Manual Range button pops up a dialog where the dev user can enter two addresses for disassembly. See Figure 12 below. Once entered and Go is clicked, TE polls the target, generates the hex/disassembly and displays the results in TE.

Symbol Selection Table Pane and the various 'modes' thereof

The symbol selection table pane is an enhanced grid control that enables a depiction of data/information in a hierarchical manner. The table support parent rows. Parent rows can have children that are also parent rows, or actual data rows.

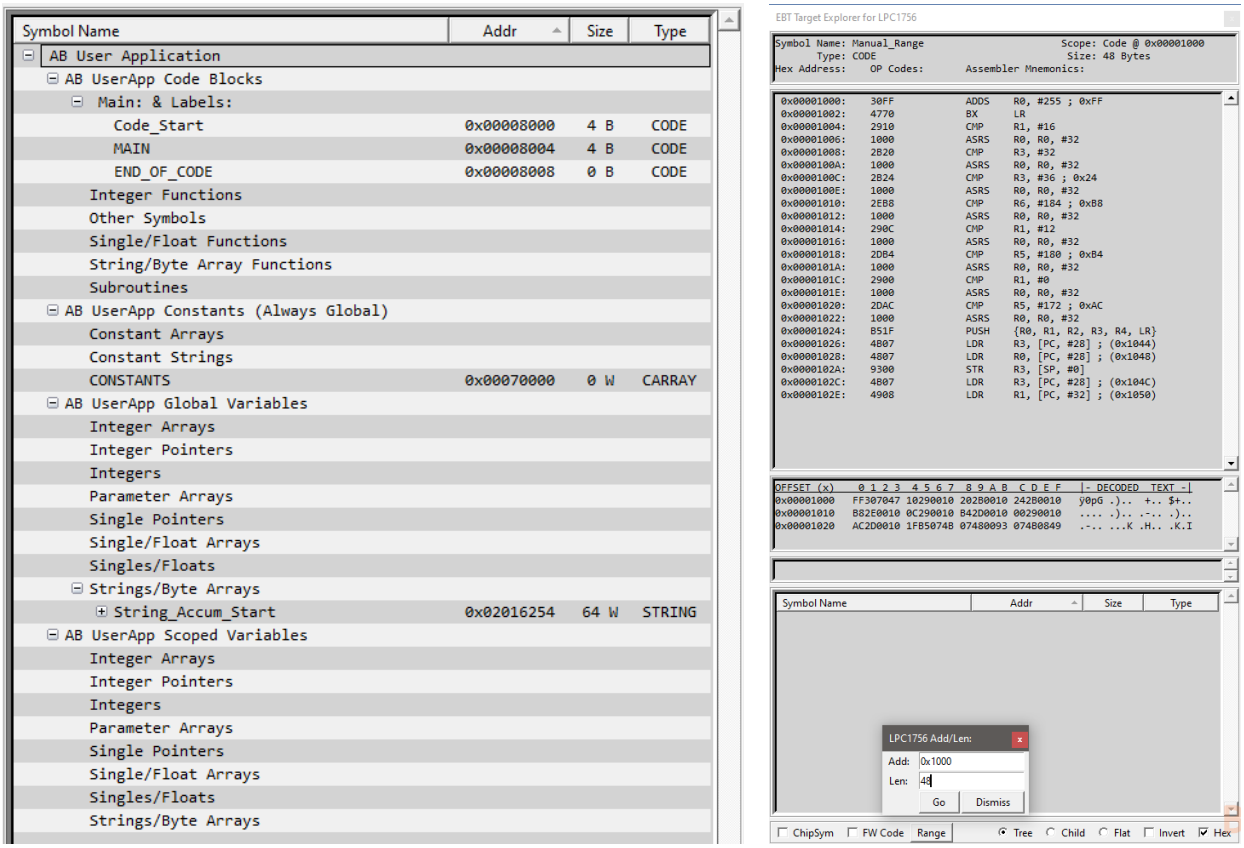


Figure 12: Hierarchical Structure of an AB App (empty Parents Retained) and an Ad-Hoc Manual Range Example

With Tree View, sorting by column headers is supported, while retaining parent/child relationships. When a column is the sort column an arrow is presented thereon, point up for ascending sort, or down for descending sort.

While parent/child hierarchy is very often useful, it is sometime more useful to have a completely flat table or, likely even more useful, a table where only multi-element constructs are identified and rolled up to a single line item (think word or string/byte arrays, both variables or constants, which are always sequential in memory and would possibly impute visual noise if expanded to the member/record level).

As such, the author has added radio buttons to the TE Option pane that enables the table to operate in three modes: Tree, Child, Flat, as enumerated in the previous paragraphs.

Tree Mode

Presents information in a manner that maintains a full hierarchical format. Chip symbols, AB Firmware Routines (once fully implemented), and AB User Apps are the three root table entries that can be populated, depending on use case.

Child Mode

All code/variable elements are flattened, with the exception of sequential arrays. This enables sorting by any column header as needed for a particular purpose. i.e. if one wants to see everything in an address sorted manner, irrespective of the type of data element types, one can click the address column header to achieve same.

Flat Mode

This mode enables a full flattening of the information in the symbol table, to include all individual elements of arrays. This might be useful for some reason.?. While the author can't come up with why a Flat table mode would trump a Child table mode, in terms of practical usefulness, it is acknowledged that not providing this as an option might be handcuffing someone else that might need exactly that which the author is too thick-skulled to conceive. So, for completeness' sake, Flat Mode exists and is supported. :)

TE Future Enhancements

The author has intentions to have firmware code monuments added to the symbol table from the address vectors provided to the compiler by the target's AB runtime code. These vectors are Code Entry Points that allows an AB user app to make use of the optimized and compiled FW routines. Effort will eventually be put into having these addresses that are called out in the disassembly be translated to these FW routines' names, so that there is better cognitive clarity regarding the AB Source Code after being processed by the Coridium AB compiler.

And, commensurate with the auto-translation of disassembled code addresses to FW Runtime Code Entry Point Names, it is desired that where there is a core/peripheral address-mapped-register encountered in the code to have TE parse that, perform the lookup(s) and depict in the code what peripheral address is being written to or read from. Tool tips might be a means for this, in addition to amending the ASM listing with salient comments.

The author has intentions to attempt to figure out how to get the core registers (R0-R12, LR, SP, PC) from the ARM controller when a STOP AB Breakpoint is encountered during runtime. It is suspected that these exist on the stack when an AB STOP is encountered in the AB User App after being processed by the Coridium AB compiler.

It is desired that there become a 'standard' way of documenting and generating ChipSymbols for the Chip Libraries that TE brings to the table. Right now, these were created in an ad-hoc manner when a specific need to look at peripherals on a specific target MCU. The immediate need outweighed the need to codify and implement a standard syntax and implementation of the Target Chip Lib's contents. This is one that is sorely felt each time the author opens up a chip lib's entries in TE.

The author also intends to have on-demand code/hex read in commensurate with mouse wheel scrolls up or down beyond the start or conclusion of a symbol's code.

Lastly, the author is amenable to receiving other's input regarding potential enhancements to TargetExplorer (or other elements of EBT). It is readily asserted that the AB context imputes a level of complexity and abstraction away from what a more sophisticated IDE would bring to the table with debug probes interfacing with the SWD fabric in the MCU's core. That is OK, as EBT and TE are not intended to become morphed into something so complicated. The author has been told by some old salty industry folks that what TE provides rivals what some multi-thousand-dollar packages bring to the table.

Those are very kind words. The author doesn't intend for these EBT/TE/ASMtool/FilePP extensions to BASICtools to become something that rivals same. It is BASIC after all… ;)

Conclusion

Thank you for taking the time to read about using the EBT TargetExplorer module. If you have already tried EBT, and the TargetExplorer module, or decide to do so, the author will be pleased. It is understood that it might not be to everyone's liking and that is OK. Trying to please everyone ensures a quick and painful trip to certain failure, which is not the intent here. If it does provide a few folks some usefulness, then it has achieved a goal.

If you experience trouble with EBT there are three options available to you to try to secure assistance:

1) Posting to thread on the Coridium Forums where EBT was announced,
2) Cracking open the tcl sources and see if you might glean a better understanding of what is transpiring and how it might be able to be resolved, or
3) Joining the Telegram ARMbasic Channel (https://t.me/ARMbasic) and seeing if anyone there might be someone who can offer input or assistance.

The author bids you well and hopes each and every one of you experience success. Take care and Happy Coding!

-t

SubAppendix 2.1: Document Control and Record of Revisions

Document Control	
Document Number	0235_1800_TE.um.001
Document Title	TargetExplorer (TE) User Manual
Authored By	Tod Wulff, Chief Hack @ The House of Wulff
Source Location	https://coridium.us/tod/EBT/TE_User_Manual.docx
Published Location	https://coridium.us/tod/EBT/CurrentRelease/EBT_User_Manual.pdf as an addendum thereto
Acknowledgements	Bruce Eisenhard, Founder/HMFIC @ Coridium Corp Gary Zwan, ARMbasic/Embedded Dev Compatriot

Record of Revisions			
Version Number	**Date issued**	**Revisor**	**Reason for Revision**
1.0	26 Aug 2020	Tod Wulff	Initial Revision
1.0a		Tod Wulff	- Added text about special characters in scratchpad - Added text regarding pane positioning automation - Minor grammar/grammatical cleansing

Addendum 3:

EBT File Preprocessor (FilePP)

User Manual

Version 1.0 27 Aug 2020
(Record of Revisions located in Appendix 1)

Introduction to FilePP

BT-proper makes use of a customized version of CPP, which functions in a manner similar to CPP, and is called BPP, presumably an acronym for BASICpreprocessor (a reasonable assumption never thought to have affirmed). BPP is, by design and lineage, pretty focused at one thing, preprocessing source-code files with a rigid structure of functionality and capability. It is not intended to be expandable and is pretty limited in a myriad of other ways, as it was designed to do just one thing. Don't mistake these comments as non-appreciation. It does perform its designed functions pretty darn well. However, when one starts to request/expect CPP/BPP to do more, the wheels can depart the bus pretty quickly.

These limitations (was and still is perceive as such) was the initiating factor that caused the hunt for other preprocessors to begin. It didn't take too long to find FilePP. After testing FilePP, and eventually realizing just how extremely powerful such an extendible preprocessor solution was, it became evident that FilePP needed to become an EBT team member, and was integrated as one of the key extensions that EBT brought to the table for a dev to leverage in any AB dev effort.

It is acknowledged that, for the vast majority of AB dev work, BT's Stock BPP is more than adequate at preprocessing AB source file, doing so really well. However, it would be appropriate to acknowledge that there are some specific feature sets that FilePP empowers an AB dev with that simply cannot be accomplished with the BPP tool as it exists today.

While in-depth details about what FilePP does, how it does it, and how to get it to do its thing, is detailed in Addendum 1 to this manual, the following listed items are some of FilePP's features that see regular employment by EBT's dev:

- Compile-time Maths
- Multi-Level Preprocessing Debug
- C comment Removal
- AB Comment Removal
- For-Next Loop Preprocessing
- Multi-Line Macros
- Selectable Boundary Macro Expansion
- Inner Literal Macro Expansion
- Regex-based Macro Definition & Expansion
- Multi-file Macro Definition, Build-up, Expansion

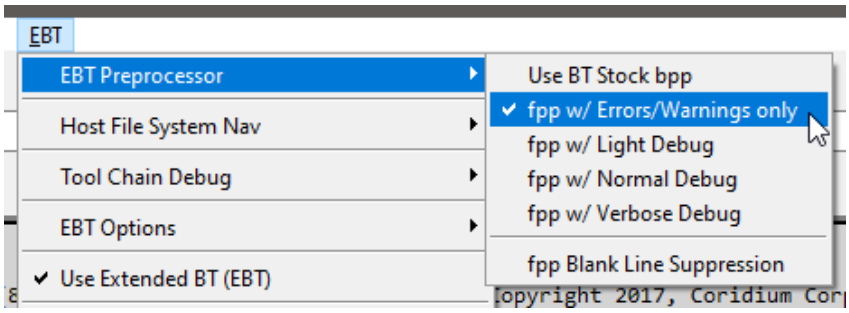


Figure 1: EBT Preprocessor Selection and Control

"Filepp was written by Darren Miller (darren at cabaret dot demon dot co dot uk). Many others have contributed patches, bug reports and suggestions, see the README for a list of some of the main contributors. Thanks to everyone who has helped make filepp what it is."

The above quote is from the current webpage of FilePP's author, Mr. Darren Miller. Darren has a nice succinct page about FilePP. Please visit it at: Filepp: The generic file pre-processor. The author has included the FilePP ManPage, virtually unfettered, as Addendum 1 hereto, in the interests of having all things EBT in a single manual (set).

Usage Examples

The author has a number of AB user apps that make use of the feature sets related to FilePP. Initial intent was to include them herein, but that seems a bit presumptuous, and likely mostly noise, as they are, objectively, quite convoluted hunks of esoteric code focused mainly on easing the dev workflow for apps using the forthcoming ABmt (and eventually ABrtos) through the use of regex, pre-compile maths, multi-line macros, preprocessor functions, compile time loops, and other 'magical preprocessor incantations'.

As such, the author respectfully suggests that the FilePP MapPage (Addendum 1 hereto) be consumed with vigor and then leveraged while employing the use of FilePP in one's workflow, as the best way to learn about using this powerful tool.

Additionally, if there is a need for EBT-/AB-specific FilePP guidance, or if you have related queries, it is likely best to engage for questions or other support, as detailed in the Conclusion below.

Conclusion

Thank you for taking the time to read about using FilePP, a component of the EBT toolset. If you have already tried EBT, or decide to do so, the author will be pleased. It is understood that it might not be to everyone's liking and that is OK. Trying to please everyone ensures a quick and painful trip to certain failure, which is not the intent here. If it does provide a few folks some usefulness, then it has achieved a goal.

If you experience trouble with FilePP or EBT there are three options available to you to try to secure assistance:

4) Posting to thread on the Coridium Forums where EBT was announced,
5) Cracking open the tcl sources and see if you might glean a better understanding of what is transpiring and how it might be able to be resolved, or
6) Joining the Telegram ARMbasic Channel (https://t.me/ARMbasic) and seeing if anyone there might be someone who can offer input or assistance.

The author bids you well and hopes each and every one of you experience success. Take care and Happy Coding!

-t

SubAddendum 3.1: FilePP ManPage

(w/ some minor corrections and redactions/additions (index, etc.))

NAME

filepp - A generic file preprocessor

DESCRIPTION

filepp is a generic file preprocessor designed to allow the functionality provided by the C preprocessor **cpp**(1) to be used with any file type. **filepp** is designed to be easily customized and extended.

OPTIONS

filepp accepts the following command line options:

-b

Suppress blank lines originating from include files (this has no effect on the top-level file).

-c

Read input from STDIN instead of a file. Note: if both **-c** and input files are specified, both are used as inputs in the order given.

-D*macro*

Predefine *macro* to have a definition of `**1**'.

-D*macro***=***defn*

Predefine *macro* to have a definition of *defn*.

-d

Output debugging information.

-dd

Output verbose debugging information. This option shows all normal debugging information, plus the full list of defined macros every time the list changes.

-dl

Output light debugging information. This option shows minimal debugging information.

-dpre*char*

Prefix all debugging information with *char* (can be character or string), can be used to make debugging easier to read.

-dpost*char*

Postfix all debugging information with *char* (can be character or string), this defaults to a newline. If *char* does not contain a newline, then no newline will be printed after debugging messages. (Newlines can be put in *char* using the __NEWLINE__ macro.)

-ds

Print debugging info on stdout rather than stderr.

-e

Define all environment variables as macros with prefix **envchar**.

-ec *char*

Set **envchar** (prefix of environment variables defined as macros) to *char*, defaults to **$**. (Note: this option only takes effect at the time the environment variables are converted to macros).

-ecn

Set **envchar** (prefix of environment variables defined as macros) to nothing (no prefix).

-h

Show summary of options.

-I*dir*

Append directory *dir* to the list of directories searched for include files.

-imacros *file*

Reads in macros from *file*, but discards everything else in the file.

-k

Turn off parsing of all keywords. This is useful if you just want to use the macro expansion facilities of **filepp**. With this option all keywords found will be ignored, **filepp** will just replace any macros specified with the **-D***macro*=*defn* option.

-kc *char*

Set keyword prefix character to *char* (can also be a string). All **filepp** keywords are prefixed with the character **#** by default. This option allows the prefix to be changed to something else.

-lc *char*

Set line continuation character to *char* (can also be a string). When the line continuation character is found with a newline following it, it and the newline are replaced by the line continuation replacement character. Default is **** (**cpp**(1) style).

-lec *char*

Set optional keyword line end character to *char* (can also be a string). This allows extra characters to be placed at the end of a line containing a keyword. The extra characters will be ignored. This is useful if keywords are to be embedded in HTML or C style comments. For example, to embed keywords in an HTML comment the keyword prefix character could be set to **<--!#** and the optional keyword line end character set to **-->**. An example keyword would then be:

<!--#include "header.h" -->

In the case the optional keyword line end characters --> would be ignored.

-lr *char*

Set line continuation replacement character to *char* (can also be a string). Default is a null string (cpp(1) style).

-lrn

Set line continuation replacement character to be a newline.

-m *module.pm*

Load module *module.pm*. *module.pm* is a perl(1) module which can be used to extend or modify the behavior of **filepp**. See section **FILEPP MODULES** for details of modules included with filepp and **FILEPP MODULE API** for details on how to write your own modules.

-M*dir*

Append directory *dir* to the list of directories searched for filepp modules. This list defaults to the directory the filepp modules are installed (if any) plus the default Perl module paths. (Note: this adds the directory to the Perl @INC list.)

-mp *char*

Prefix all macros with *char*. Macros are defined in the normal way, but will only be replaced when found prefixed with *char*. For example, filepp macros will behave similar to Bourne shell (sh(1)) variables if *char* is set to **$**.

-mpnk

Turns off macro prefixes within keywords. When using a macro prefix character this option allows macros to be used without the prefix in keyword processing. For example, if the macro prefix is **$** then and **#if** would be written as:

#if $MACRO == 1

Using the **mpnk** option allows the **#if** to be written as:

#if MACRO == 1

-o *name*

Write output to *name* instead of STDOUT. If there is only one input file and it has the same name as the output file, the original input file will be backed-up as *name*~.

-ov

Overwrite mode, causes the output file to overwrite the input file. Useful when modifying a large number of files at once, eg:

filepp -ov -DTHIS=THAT *

The original input file(s) will be backed-up as *name*~.

-ovc IN=OUT

Similar to overwrite mode, the difference is the output filename is input filename with **IN** part converted to **OUT**. For example, to process a set of files all ending with .in and have the output files all ending in .out do:

filepp -ovc .in=.out *.in

In this case a file called *test.in* will be processed and the output file will be *test.out*. Note: if the input file does not contain **IN** then the output file will have the same name as the input file and the original input file(s) will be backed-up as *name*~!

-pb

Preserve blank lines. Using this option attempts to keep as many lines in the output file as are in the input file, so all blank lines which normally would not get printed are printed. Useful when comparing input file with output.

-re

Treat keyword and macro prefix characters and line continuation character as Perl regular expressions instead of normal strings.

-s

Run **filepp** in safe mode. This turns off the **pragma** keyword.

-U*macro*

Undefine previously defined *macro*.

-u

Undefine all currently defined macros, including predefined ones.

-v

Show version of program.

-w

Turn on word boundaries when replacing macros. When word boundaries are on, macros will only be replaced if the macro appears in the text as a word. For example, by default *macro* would be replaced in both cases of the following text:

macro as word, macroNOTaword

but only the first occurrence would be replaced with the **-w** option.

With this option enabled **filepp** will only replace macros which contain alphanumeric characters. International (non-ASCII) character sets can be supported using Perl's locale handling.

KEYWORDS

filepp supports the following keywords:

#include <*FILE*>

Include a file in the file being processed. This variant is used for "system" include files. It searches for a file named *FILE* in a list of directories specified by you. Directories are specified with the command option `-I'. **filepp** does not predefine any system directories in which to search for files.

#include FILE

Include a file in the file being processed. This variant is used for include files of your own project. It searches for a file named *FILE* first in the current directory, then in the list of directories specified with the command option `-I'. The current directory is the directory the base input file is in.

#define *macro*

Define the macro *macro* to have a definition of `1'. *macro* can then be used with the keywords **#ifdef** and **#ifndef**.

#define *macro defn*

Define the macro *macro* to have the value *defn*. *macro* can then be used with the keywords **#ifdef** and #ifndef. Also, all instances of *macro* following the **#define** statement will be replaced with the string *defn*. The string *defn* is taken to be all the characters on the line following *macro*.

#define *macro(arg1, arg2, ...) defn*

Define the macro *macro* to have the value *defn* with arguments *(arg1, arg2, ...)*. *macro* can be used as follows:

#define *macro(foo)* defn with foo in

Now when replacing occurs:

macro(bar)

will become:

defn with bar in

Macros can have any number of comma separated arguments.

Macros can also have variable numbers of arguments if the final macro ends in ..., for example:

#define *error(string, args...)* fprintf(stderr, string, args);

Here the first argument given becomes *string* and all other arguments will become *args*. If called as: *error("%d,%s", i, string)* it will give

fprintf(stderr, "%d,%s", i, string);

Also, if a macro with a variable number of arguments is passed no arguments for the variable argument, then commas can be optionally removed from the definition by preceding the definition with "##". For example:

#define *error(string, args...)* fprintf(stderr, string, ##args);

If this is called as: *error("empty")* then result will be:

fprintf(stderr, "empty");

The comma immediately before *##args* has been removed.

#if *expr*

A conditional statement, *expr* will be evaluated to true (1) or false (0). If *expr* evaluates to true, the text between the **#if** and the next **#else** or **#endif** will be included. If *expr* evaluates to false, the text between the **#if** and the next **#else** or **#endif** will be ignored. *expr* can use all the usual cpp style comparisons (==, !=, <, >, etc.). Multiple comparisons can be combined with and (&&) and or (||). The **defined** keyword can also be used to check if macros are defined. For example:

#if defined macro && macro == defn

Note: filepp's **#if** does not work in exactly the same way as **cpp**(1)'s **#if**. **cpp**(1)'s **#if** only does numerical style comparisons. Filepp's **#if** statement can also compare strings and regular expressions using **perl**(1)'s full range of comparison operations. For example, to test if two strings are exactly equal use:

#if "MACRO" eq "string"

To test if strings are not equal use *ne* instead of *eq*. Regular expressions can also be tested, for example to test if a macro has any whitespace in it use:

#if "MACRO" =~ /\s/

To test if a macro does not have any whitespace in it =~ can be replaced with *!~*.

Perl experts: **#if** works by first parsing *expr* for the **defined** keyword and checking if the macro it refers to is defined, replacing it with 1 if it is and 0 if it isn't. It then checks *expr* for any other macros and replaces them with their definition. Finally, it passes *expr* through Perl's **eval** function, which returns true or false.

#elif *expr*

#elif stands for "else if". Like **#else**, it goes in the middle of a **#if**[**n**][**def**]-**#endif** pair and subdivides it; it does not require a matching **#endif** of its own. Like **#if**, the **#elif** directive includes an expression to be tested.

#ifdef *macro*

A conditional statement, if *macro* has been defined the text between the **#ifdef** and the next **#else** or **#endif** will be included. If *macro* has not been defined the text between the **#ifdef** and the next **#else** or **#endif** will be ignored.

#ifndef *macro*

The reverse case of the **#ifdef** conditional.

#else

The **#else** directive can be added to a conditional to provide alternative text to be used if the condition is false.

#endif

Used to terminate a conditional statement. Normal processing resumes following the **#endif**.

#undef *macro*

Undefine a previously defined macro.

#error *mesg*

Causes **filepp** to exit with the error message *mesg*.

#warning *mesg*

Causes **filepp** to issue the warning message *mesg*.

#comment *mesg*

As **filepp** is supposed to be a generic file preprocessor, it cannot support any known comment styles, therefore it defines its own with this keyword. All lines starting with **#comment** are treated as comments and removed by **filepp**.

#pragma filepp *function arg1, arg2, ...*

The **#pragma** keyword immediately followed by the word **filepp** allows the user to execute a Perl function during parsing. The word immediately following **filepp** is taken as the name of the function and the remainder of the line is taken to be a comma separated list of arguments to the function. Any of the **filepp** internal functions (see section **FILEPP MODULE API**) can be called with the **#pragma** keyword.

Warning: There are obvious security risks with allowing arbitrary functions to be run, so the -s (safe mode) command line option has been added which turns the **#pragma** keyword off.

PREDEFINED MACROS

filepp supports a set of predefined macros. All the predefined macros are of the form **__MACRO__**, where **MACRO** is:

FILE

This macro expands to the name of the current input file.

LINE

This macro expands to the current input line number.

DATE

This macro expands to a string that describes the date on which the preprocessor is being run. The string contains eleven characters and looks like "Feb 27 2007".

ISO_DATE

This macro expands to a string that describes the date on which the preprocessor is being run. The string is in the format specified by ISO 8601 (YYYY-MM-DD) and looks like "2007-02-27".

TIME

This macro expands to a string that describes the time at which the preprocessor is being run. The string contains eight characters and looks like "20:02:16".

BASE_FILE

This macro expands to the name of the main input file.

INCLUDE_LEVEL

This macro expands to a decimal integer constant that represents the depth of nesting in include files. The value of this macro is incremented on every **#include** directive and decremented at every end of file.

NEWLINE

This macro expands to a newline.

TAB

This macro expands to a tab.

NULL

This macro expands to nothing. It is useful if you want to define something to be nothing.

VERSION

This macro expands to a string constant which describes the version number of **filepp**. The string is a sequence of decimal numbers separated by periods and looks like "1.8.0".

FILEPP_INPUT

This macro expands to a string constant which says the file was generated automatically from the current **BASE_FILE** and looks like "Generated automatically from ./filepp.1.in by filepp".

FILEPP MODULES

The following modules are included with the main filepp distribution:

FOR MODULE - for.pm

The for module implements a simple for loop. Its file name is **for.pm**.

The for loop is similar in functionality to that of other programming languages such as Perl or C. It has a single variable (a filepp macro) which is assigned a numerical value. This numerical value changes by a set increment on each iteration through the loop. The loop terminates when the value no longer passes a comparison test.

The for module implements the following keywords:

#for *macro start compare end increment*

The **#for** keyword is functionally equivalent to the following Perl or C style loop:

for(*macro*=*start*; *macro compare end*; *macro*+=*increment*)

The **#for** keyword requires the following space separated parameters:

macro : The name of the macro to which the for loop should assign its numerical value.

start : The value *macro* should be assigned at the start of the loop. *start* should be a numerical value.

compare : The comparison to make between the current value of *macro* and the value *end* to determine when the loop should terminate. Valid values for *compare* are <, >, >=, <=.

end : the for loop will terminate when the test

macro compare end

fails. *end* should be a numerical value.

increment : The value to increment *macro* on each iteration of the loop. At the end of each iteration the value of *increment* is added to the current value of *macro*. *increment* should be a numerical value.

#endfor

The **#endfor** keyword is used to signify the end of the loop. Everything within the opening **#for** and the closing **#endfor** will be processed on each iteration of the loop.

Example usage:

#for COUNTER 10 > 1 -2.5

COUNTER

#endfor

In the above example COUNTER will be defined to have values 10, 7.5, 5 and 2.5 for each successive iteration through the loop.

Nested loops are also possible, as is changing the value of the macro within the loop. *start*, *end* and *increment* should all be numerical values, however it is possible to use macros instead provided the macros are defined to have numerical values.

FOREACH MODULE - foreach.pm

The foreach module implements a simple foreach loop. Its file name is **foreach.pm**.

The foreach loop is similar in functionality to that of other programming languages such as Perl. It takes a list of values separated by a user definable delimiter (',' by default). It then iterates through all values in the list, defining a macro to be each individual value for each iteration of the loop. The loop terminates when all values have been used.

The foreach module implements the following keywords:

#foreach *macro list*

The **#foreach** keyword is functionally equivalent to the following Perl style loop:

foreach *macro* (split(/*delim*/, *list*))

The **#foreach** keyword requires the following space separated parameters:

macro : The name of the macro to which the foreach loop should assign the current list value.

list : The list of values, separated by *delim* (see **#foreachdelim** keyword for how to set *delim*). *list* can also be a macro or contain macros.

The loop will run from the **#foreach** keyword to the next **#endforeach** keyword.

#endforeach

The **#endforeach** keyword is used to signify the end of the loop. Everything within the opening **#foreach** and the closing **#endforeach** will be processed on each iteration of the loop.

Example usage:

#foreach VALUE one, two, three, four

VALUE

#endforeach

In the above example VALUE will be defined to have values one, two, three and four for each successive iteration through the loop.

Nested loops are also possible.

#foreachdelim /*delim*/

The **#foreachdelim** keyword is used to set the delimiter used in each list. The delimiter can be any character, string or regular expression. The delimiter should be enclosed in forward slashes, in the same style as Perl regular expressions. The default value for *delim* is ','. To set the delimiter to be a single space do:

#foreachdelim / /

To set *delim* to be any amount of white space do:

#foreachdelim /\s+/

See the Perl documentation on regular expressions for more advanced uses.

LITERAL MODULE - literal.pm

The literal module prevents macros appearing in literal strings from being replaced. A literal string is defined as having the form:

"literal string with **macro** *in"*

In the above example, **macro** will not be replaced.

The behavior of the literal module can be reversed by defining the macro **LITERAL_REVERSE** before loading the module, for example:

filepp -D**LITERAL_REVERSE** -m literal.pm <*files*>

This has the effect of only replacing macros which appear in strings.

SYNOPSIS

filepp [*options*] *filename(s)*

TOUPPER MODULE - toupper.pm

The toupper module converts all lowercase letters to uppercase.

TOLOWER MODULE - tolower.pm

The tolower module converts all uppercase letters to lowercase.

C/C++ COMMENT MODULE - c-comment.pm

The c-comment module removes all C style:

/ comment */*

and C++ style:

// comment

comments from a file. C and C++ comments are removed after keywords have been processed. If you wish to remove C and C++ comments before keywords are processed, define the macro **REMOVE_C_COMMENTS_FIRST** before loading the module, eg:

filepp -DREMOVE_C_COMMENTS_FIRST -m c-comment.pm

HASH COMMENT MODULE - hash-comment.pm

The hash-comment module removes all comments of the style:

 # comment

from a file. This is the commenting style used by Perl, Bourne Shell, C Shell and many other programs and configuration files. Hash comments are removed after keywords have been processed. If you wish to remove hash comments before keywords are processed, define the macro **REMOVE_HASH_COMMENTS_FIRST** before loading the module (Note: if you do this and also use # as the keyword character then the keywords will be removed BEFORE they are processed).

TIC COMMENT MODULE - tic-comment.pm

This module was integrated into the FilePP distributed with EBT by EBT's author.

The tic-comment module removes all comments of the style:

‘ comment

from a file. This is the commenting style used by ARMbasic. tic comments are removed after keywords have been processed. If you wish to remove hash comments before keywords are processed, define the macro **REMOVE_TIC_COMMENTS_FIRST** before loading the module (Note: if you do this and also use ‘ as the keyword character then the keywords will be removed BEFORE they are processed).

FUNCTION MODULE - function.pm

The function module allows the user write macros which call Perl functions. Its file name is **function.pm**.

The function module allows macros of the form:

macro(arg1, arg2, arg3, ...)

to be added to a file. When the macro is found, it will run a function from a Perl module, with arguments *arg1*, *arg2*, *arg3*, ... passed to the function. The function must return a string. The returned string will replace the call to the function in the output. The function can have any number of arguments. If the function has no arguments, it should be called with an empty argument list:

macro()

If the word *macro* is found in the input file without being followed by a *(* it will be ignored.

To use the function module, the user must provide a Perl function which optionally takes in arguments and returns a string. The function can either be one of filepp's internal functions or one of the user's own provided in a Perl module. The function can be added in two ways. The first way is through the **function** keyword:

#function *macro function*

> *macro* is the name of the macro which is used to signify a call to the function in the input file and *function* is the name of the function to be called.

The second method of adding a function is to call the Perl function:

Function::AddFunction(*$macro,$function*)

> which has the same inputs as the **function** keyword.

Functions can be removed either through the keyword:

#rmfunction *macro*

> or through the Perl function

Function::RemoveFunction(*$macro*)

MATHS MODULE - maths.pm

The module provides a set of macros which perform mathematical operations. When the macros are encountered in an input file, they are evaluated, and the result is returned in the output.

The maths module includes the following macros:

add(a, b, c, ...)

Takes in any number of arguments and returns their sum: (a + b + c + ...)

sub(a, b)

Returns a minus b: (a - b)

mul(a, b, c, ...)

Takes in any number of arguments and returns their product: (a * b * c * ...)

div(a, b)

Returns a over b: (a / b)

abs(a)

Returns the absolute value of a.

atan2(a, b)

Returns the arctangent of a/b in the range -pi to pi.

cos(a)

Returns the cosine of a in radians.

exp(a)

Returns the e to the power of a.

int(a)

Returns the integer portion of a.

log(a)

Returns the natural logarithm (base e) of a.

rand(a)

Returns a random fractional number between the range 0 and a. If a is omitted, returns a value between 0 and 1.

sin(a)

Returns the sine of a in radians.

sqrt(a)

Returns the square root of a.

srand(a)

Sets the random number seed for rand().

The maths module also defines pi as M_PI as e as M_E.

The maths macros are implemented using the **function.pm** module. Nested macros are allowed, as is passing other macros with numerical definitions as arguments.

FORMAT MODULE - format.pm

This module provides a set of macros for formatting strings and numbers.

The format module provides the following macros:

printf(format, arg1, arg2, ...)

The **printf** macro behaves in the same way as the Perl/C function printf. It takes in a format string followed by a list of arguments to print. See the **printf**(3) man page or Perl documentation for full details of the **printf** function.

toupper(string)

Converts input string to upper case.

toupperfirst(string)

Converts first character of input string to upper case.

tolower(string)

Converts input string to lower case.

tolowerfirst(string)

Converts first character of input string to lower case.

substr(string, offset, length)

Extracts a substring from input *string*. **substr** behaves in the same way as the Perl substr function. *offset* is used to specify the first character of the string to output (negative for offset from end of string), *length* is the length of the string to output. If length is omitted everything from the offset is returned. For further information on **substr** see the Perl documentation.

The format macros are implemented using the **function.pm** module.

BIGDEF MODULE - bigdef.pm

The bigdef module allows easy definition of multi-line macros. Its file name is **bigdef.pm**.

A multi-line macro is a macro which has a definition which spans more than one line. The normal way to define these is to place a line continuation character at the end of each line in the definition. However, this can be annoying and unreadable for large multi-line macros. The bigdef module tries to improve on this by providing two keywords:

#bigdef *macro definition...*

The **#bigdef** keyword has the same syntax as **#define**, the only difference being the macro definition is everything following the macro name including all following lines up to the next **#endbigdef** keyword.

#endbigdef

Ends a bigdef. Everything between this keyword and the last preceding **#bigdef** is included in the macro.

Any keywords found in the definition will be evaluated as normal AT THE TIME THE MACRO IS DEFINED and any output from these will be included in the definition.

Note: The difference between bigfunc and bigdef is the time keywords in the definition are evaluated. Bigdef evaluates them as the macro is DEFINED, bigfunc evaluates them whenever the macro is REPLACED.

BIGFUNC MODULE - bigfunc.pm

The bigfunc module allows easy definition of multi-line macros. Its file name is **bigfunc.pm**.

A multi-line macro is a macro which has a definition which spans more than one line. The normal way to define these is to place a line continuation character at the end of each line in the definition. However, this can be annoying and unreadable for large multi-line macros. The bigfunc module tries to improve on this by providing two keywords:

#bigfunc *macro definition...*

The **#bigfunc** keyword has the same syntax as **#define**, the only difference being the macro definition is everything following the macro name including all following lines up to the next **#endbigfunc** keyword.

#endbigfunc

Ends a bigfunc. Everything between this keyword and the last preceding **#bigfunc** is included in the macro.

Any keywords found in the definition will be evaluated as normal AT THE TIME THE MACRO IS REPLACED and any output from these will be included in the definition.

Note: The difference between bigfunc and bigdef is the time keywords in the definition are evaluated. Bigdef evaluates them as the macro is DEFINED, bigfunc evaluates them whenever the macro is REPLACED.

DEFPLUS MODULE - defplus.pm

The defplus module allows extra information to be appended to an existing macro. Its file name is **defplus.pm**.

The defplus module allows further things to be appended to existing macros. The module implements one keyword:

#defplus *macro definition...*

The **#defplus** keyword has the same syntax as **#define**, the only difference being if the macro is already defined then *definition* is appended to the existing definition of the macro. If the macro is undefined then **#defplus** behaves in exactly the same way as **#define**.

REGEXP MODULE - regexp.pm

The regexp module allows Perl regular expression replacement to be done with filepp. Its file name is **regexp.pm**.

Perl regular expression replacement allows a regular expression to be searched for and replaced with something else. Regular expressions are defined as follows:

#regexp /*regexp*/*replacement*/

It is very similar to the Perl syntax and the following Perl code will be executed on each line of the input file:

$line =~ s/*regexp*/*replacement*/g

For users who don't understand Perl, this means replace all occurrences of *regexp* in the current line with *replacement*.

A full description of regular expressions and possible replacements is beyond the scope of this man page. More information can be found in the Perl documentation using the command:

perldoc perlre

Any number of regular expressions can be defined. Each regular expression is evaluated once for each line of the input file. Regular expressions are evaluated in the order they are defined.

Regular expressions can be undefined in the following way:

#rmregexp /*regexp*/*replacement*/

This will remove the specified regular expression.

In debugging mode the current list of regular expressions can be viewed using the pragma keyword:

#pragma filepp *ShowRegexp*

When not in debugging mode, this will produce no output.

A single regular expression can also be defined on the command line using the *REGEXP* macro, for example:

filepp -D*REGEXP*=/*regexp*/*replacement*/ -m regexp.pm inputfile

Note: the *REGEXP* macro must be defined BEFORE the regexp module is loaded, putting -D*REGEXP*... after -m regexp.pm will not work. When using the command line approach, if the *REGEXP* macro is successfully parsed as a regular expression it will be undefined from the normal filepp macro list before processing starts. Care should obviously be taken when escaping special characters in the shell with command line regexps.

BLC MODULE - blc.pm

The Bracket Line Continuation module causes lines to be continued if they have more open brackets: "(" than close brackets: ")" on a line. The line will be continued until an equal number of open and close brackets are found.

Brackets can be prevented from being counted for line continuation by escaping them with a backslash: "(" and ")". Any brackets found with a preceding backslash will be ignored when deciding if line continuation should be done and then have the backslash removed once the full line has been found.

C MACROS MODULE - cmacros.pm

The cmacros module causes the definition of the following predefined macros to be quoted: **DATE, TIME, VERSION, BASE_FILE, FILE,** (note: predefined macros are written as __MACRO__).

This makes the macros more "C" like, as the C preprocessor also puts quotes around these macros.

C MACROS MODULE - cpp.pm

The cpp makes filepp behave in a similar manner to a C preprocessor **cpp**(1).

DISCLAIMER: filepp is not meant to be a drop in replacement for a C preprocessor even with this module. I would not recommend using filepp as a C preprocessor unless you fully understand how it differs from a real C preprocessor. The output from filepp with the cpp module will not be the same as a real C preprocessor.

GRAB MODULE - grab.pm

The grab module is used to grab input before processing. Its file name is **grab.pm**.

The grab module is mainly for use in other modules, such as for.pm and bigfunc.pm. It grabs all input from a file before any processing is done on it. This allows other modules to do processing on the original input data before the main processing is done. For example, the for module will store the original input inside a loop and re-use it each time the loop is processed.

#grab *macro definition...*

The grab module will start grabbing of all input from the grab keyword, onwards.

#endgrab

Ends a grab. Everything between this keyword and the last preceding **#grab** will be grabbed and stored for use in other modules.

Grabs can be nested if required.

When calling grab from another module, use the following functions:

Grab::StartGrab(*$startkeyword*,*$endkeyword*)

$startkeyword is the keyword that StartGrab is called from. *$endkeyword* is the keyword that grabbing should stop at.

@List=**Grab::GetInput()**

Returns a Perl list containing all input grabbed from when grab was last run.

$line=**Grab::GetInputLine()**

Returns the line number of the input file where grabbing last started.

FILEPP MODULE API

The behavior of **filepp** can be modified or extended through the use of modules. **filepp** modules are in fact **perl**(1) modules, and the rest of this section assumes the reader has a knowledge of Perl.

filepp modules are **perl**(1) modules which extend or modify **filepp**'s behavior by either calling or replacing **filepp**'s internal functions. **filepp** has the Perl package name **Filepp** so its internal functions can be called within modules either as **Filepp::function()** or just **function()**. Any of **filepp**'s internal functions can be called or replaced from within a **filepp** module, the most useful ones are:

Debug(*$string*,*$number*)

Print *$string* as debugging information if debugging is enabled. *$number* is optional and can be used to set the debugging level at which *$string* should be printed, lower numbers being higher priority. Command line option **d** prints all debugging info for 2 and below, option **dd** prints all debugging information for 3 and below and option **dl** prints all debugging information for 1 and below. If *$number* is not provided, defaults to 1.

AddProcessor(*$function*,*$pos*,*$type*)

Allows the module to add a function named *$function* to **filepp**'s processing chain. The processing chain is a set of functions which are run on each line of a file as it is processed. The default functions in the processing chain are **ParseKeywords** which does keyword parsing and **ReplaceDefines** which does macro replacement. Further functions can be added to the chain, with each function taking a string (the current line) as input and returning the processed string as output.

By default, or if *$pos* is set to 0, the processor is added to the end of the processing chain. If *$pos* is set to 1 the processor is added to the start of the processing chain.

$type controls what the processor is run on. There are three options for this, 0 (default): the processor runs on everything passed to the processing chain; 1: the processor runs on full lines only; 2: the processor runs on part lines only (a part line is the text following a keyword such as **if** which needs to be parsed for macros).

Both *$pos* and *$type* are optional parameters.

AddProcessorAfter(*$function*,*$existing*,*$type*)

Adds function *$function* to the processing chain directly after existing processor *$existing*. If *$existing* is not found, then *$function* is added to the end of the processing chain. Regular expression matching is used to compare *$existing* with the names of the functions in the processing chain.

$type is optional.

AddProcessorBefore(*$function*,*$existing*,*$type*)

Adds function *$function* to the processing chain directly before existing processor *$existing*. If *$existing* is not found, then *$function* is added to the start of the processing chain. Regular expression matching is used to compare *$existing* with the names of the functions in the processing chain.

$type is optional.

RemoveProcessor(*$function*)

Removes the processor function *$function* from the processing chain.

$string*=**ReplaceDefines(***$string)

Replaces all macros in *$string* with their definitions and returns the processed string.

AddKeyword(*$string*,*$function*)

Add the keyword named *$string*. When the keyword is found in text processing the function named *$function* will be run with everything following the keyword passed as a single argument.

RemoveKeyword(*$string*)

Removes the keyword named *$string*.

RemoveAllKeywords()

Removes all the keywords currently defined for **filepp** (used for the -k command line option).

AddIfword(*$string*)

Adds keyword named *$string* to Ifword list. An Ifword takes in the string following the keyword and optionally parses it, returning a 1 if the string parses to true and 0 for false. The default Ifwords are **if**, **ifdef** and **ifndef**.

RemoveIfword(*$string*)

Removes keyword named *$string* from Ifword list (note: this does NOT remove the keyword, use **RemoveKeyword** for that).

AddElseword(*$string*)

Adds keyword named *$string* to Elseword list. An Elseword takes in the string following the keyword and optionally parses it, returning a 1 if the string parses to true and 0 for false. The default Elsewords are **else** and **elif**.

RemoveElseword(*$string*)

Removes keyword named *$string* from Elseword list.

AddEndifword(*$string*)

Adds keyword named *$string* to Endifword list. An Endifword should return a 1 to indicate successful termination of the if block. If the Endifword returns 0 the Endifword is ignored and filepp assumes the current if block carries on after the Endifword. The default Endifword is **endif**.

RemoveEndifword(*$string*)

Removes keyword named *$string* from Endifword list.

AddIncludePath(*$string*)

Adds the include path *$string* to the list of directories to search for include files (used for the -I command line option).

AddModulePath(*$string*)

Adds the path *$string* to the list of directories to search for filepp modules (used for the -M command line option).

AddOpenInputFunc(*$function*)

Adds a *$function* to a list of functions to be run each time a new base input file is opened.

AddCloseInputFunc(*$function*)

Adds a *$function* to a list of functions to be run each time a new base input file is closed.

AddOpenOutputFunc(*$function*)

Adds a *$function* to a list of functions to be run each time an output file is opened.

AddCloseOutputFunc(*$function*)

Adds a *$function* to a list of functions to be run each time an output file is closed.

AddInputFile(*$string*)

Adds another input file to the list of files to be processed (used for adding input files at the command line).

ChangeOutputFile(*$string*)

Closes the current output file and attempts to open a new one named *$string*.

SetKeywordchar(*$string*)

Set the initial keyword char to *$string* (used for the -kc command line option).

SetContchar(*$string*)

Set the line continuation char to *$string* (used for the -lc command line option).

SetContrepchar(*$string*)

Set the line continuation replacement char to *$string* (used for the -lr command line option).

SetOptLineEndchar(*$string*)

Set the optional keyword line end character to *$string* (used for the -lec command line option).

SetBlankSupp(1/0)

Turns blank-line suppression on/off (1 = suppress, 0 = don't suppress). When blank-line suppression is on, blank lines in input files will not be copied to the output. Unlike the corresponding command-line option (-b), this function can also have effect in the top-level file. The setting of blank-line suppression applies to the current file being processed and all files included in the current file.

ResetBlankSupp()

Resets blank-line suppression to the command-line specified value. This only affects the output of blank lines from the current file being processed and all files included in the current file. In the top-level file, this always turns blank-line suppression off.

SetEatTrail(*$string*)

If *$string* is a macro, whenever the macro is replaced all blank space between the macro's replacement and the next character on the line will be eaten. For example, if macro *foo* is defined to *bar* and *foo* has been set to have its trail eaten, the following:

```
eat my foo trail
```

is replaced with

eat my bartrail

CheckEatTrail(*$string*)

Returns 1 if macro *$string* will have its tail eaten, 0 otherwise.

SetEnvchar(*$string*)

Set the prefix of environment variables converted to macros (**envchar**) to *$string* (used for -ec and -ecn command line options).

DefineEnv()

Define all environment variables as macros with prefix **envchar** (used for -e command line option).

SetOutput(1/0)

Turns writing of parsed input file to output file on/off. This takes either 1 (output on) or 0 (output off) as input. When the output is turned off, the only output produced from **filepp** will be that generated by modules.

SetWordBoundaries(1/0)

Turns on(1) or off(0) word boundary checking when replacing macros (used for the -w command line option).

SetCharPerlre(1/0)

Turns on(1) or off(0) allowing of keyword prefix char and line continuation char to be Perl regular expressions (used for the -re command line option).

UndefAll()

Undefines all currently defined macros, including predefined ones (used for the -u command line option).

UseModule(*$string*)

Loads a **perl**(1) module named *$string* using the Perl command **require** (used for the -m command line option).

SetParseLineEnd(*$function*)

Sets the function to determine if line continuation should be done on current line to *$function*.

$string=**GetNextLine()**

Returns the next line (after line continuation has been dealt with) of the input file currently being processed. Returns NULL for end of file.

Write(*$string*)

Writes *$string* to the current output file.

Output(*$string*)

Conditionally writes *$string* to the current output file. If output is turned on, then writes *$string*. Output is toggled off/on using SetOutput function.

In addition, all the standard **filepp** keywords have equivalent functions which optionally take a single argument. The functions have the same name as the keyword, only with a capital first letter (eg: **#define** *string* calls the function **Define(***string***)**).

A full description of the **Parse** function and all the other **filepp** internal functions is beyond the scope of this man page. The **filepp** script is well commented and hopefully readable by a Perl programmer, so use the source Luke!

BUGS

filepp has no known bugs, only "features". If you find any "features", please report them to the author.

COPYING

SEE ALSO

cpp(1), **perl**(1)

AUTHOR

Darren Miller <darren@cabaret.demon.co.uk>.

FilePP Code Samples

```
163         // enable boundary-less macro expansion - needed for namespace manipulation in the following loop
164         #pragma filepp SetWordBoundaries 0
165         #for idxt 1 <= _ABmt_TaskCount 1
166             #if idxt eq 250
167                 wait(500)
168             #endif
169             ABmt_TaskEntryAddress(idxt) = addressof(ABT_idxt_MainLoop)
170             print "Task ";idxt," Indexed @ 0x";i2h(ABmt_TaskEntryAddress(idxt))," T: ";timer
171         #endfor
172         #pragma filepp SetWordBoundaries 1
173         #define Lo stored_define
174         #undef stored_define
175
176         print "User Tasks Indexed: "; ABmt_TaskCount," T: ";timer
```

```
#ifndef ABmt_TaskInclusionBegin         ' 'single Load' construct
    #define ABmt_TaskInclusionBegin

    #if !defined onEBT || !defined FilePP   ' danger will robinson, BT's BPP doesn't support non-boundary (intra-token) macro expansion
        #error ABmt is written to be used with the FilePP Preprocessor (integrated via EBT), to facilitate more robust compile-time functionality than what BT`s BPP can offer
    #else   ' with FilePP, intra-token macro expansion works, but path resolution behavior is different than BT's BPP
        // #warning ********** ABmt_TaskInclusionBegin

        #ifdef ABmt_SchedulerCompile
            #warning ========================'============================================
            #warning STARTING TASK INCLUSIONS
            #warning ========================'============================================

            // this starts the math construct that is used the the wrapper to do the math for each task that is being processed for inclusion
            #define _ABmt_TaskCount_buildup add(0

            // This depends on use of FilePP with word boundaries being turned off.!.
            #pragma filepp SetWordBoundaries 0

            // this are regex constructs that drives name unicity for each task by replacing any ABT_blah tokens with ABT_XX- where XX is a preprocessor derived counting index for each task included
            // And, as info, to turn off regex, each must be done individually and succinctly with a #rmregexp blah construct - see ABmt_TaskInclusionComplete.lib
            // this works in concert with the preprocessor math construct to enable counting which is updated by the TaskWrapper each time a task's sourcce is included
            // ABmt_TaskID will be expanded on each iteration of the taskwrapper #inc as ABmt_TaskID is defined in the wrapper

            // this macro expands and replaces ABT_ to ABT_ABmt_TaskID_ - to do so on each function, sub, label, and variable that is in the included task source
            #regexp /(\s*ABT)_/$1_ABmt_TaskID_/

            // this macro expands and replaces main to ABT_ABmt_TaskID_MainLoop
            #regexp /^\s*(?i)(main:).*/sub ABT_ABmt_TaskID_MainLoop/

            // this and the next macro were processing function and sub for macro expansion only, but needed to be more robust for labels and variables
            // #regexp /^\s*(?i)(function)\s*\b/function ABT_ABmt_TaskID_/
            // #regexp /^\s*(?i)(sub)\s*\b/sub ABT_ABmt_TaskID_/

            // this macro expands and replaces end with endsub, for concluding the each task's MainLoop
            #regexp /^\s*(?i)(end)\s\B/endsub/

        #else
            #error This lib is for use with ABmt and is intended to be included by the Scheduler

        #endif

    #endif

#endif
```

```
'   -   lots of other stuff... :)
'
' ============================================================================

'~~ // /* #ifndef ABmt_TaskWrapperLibVersion           ' 'single Load' construct intentionally disabled...*/
#define ABmt_TaskWrapperLibVersion  "0.02"

    #if !defined onEBT || !defined FilePP   ' danger will robinson, BT's BPP doesn't support non-boundary (intra-token) macro expansion
        #error ABmt is written to be used with the FilePP Preprocessor (integrated via EBT), to facilitate more robust compile-time functionality than what BT`s BPP can offer
    #else   ' with FilePP, intra-token macro expansion works, but path resolution behavior is different than BT's BPP
        // #warning ********** ABmt_TaskWrapper

    'the lack of a 'single-load' #ifndef construct here is intentional.
    'this #include is a wrapper for each task and is meant to be reloaded for every task that is #included for inclusion into the overall build.

    #ifdef ABmt_SchedulerCompile
        // This depends on use of FilePP with compile-time math features
        // this is where the math is bring done to increment a counter for each task that is being processed for inclusion,
        // to eventually yield a total count of the #indluded tasks
        #defplus _ABmt_TaskCount_buildup ,1
        // #defplus is not a type - see filepp docs

        // This depends on use of FilePP with word boundaries being turned off
        // This too depends on use of FilePP with compile-time math features
        // this is where the math is bring done for each task that is being processed for inclusion,
        // to yield the current task number fofr the task currently being processed
        #define ABmt_TaskID _ABmt_TaskCount_buildup )

        // too friggin bad that macros aren't expanded on #warning or #error constructs...
        // FIXME fail - REGEX??? ...  #define _ABmt_compiletime_warning_task_count  #warning #comment Building ABmt Task #ABmt_TaskID: ABmt_TaskCode
        #warning ========================
        // FIXME fail - REGEX??? ...  _ABmt_compiletime_warning_task_count
        #warning      BUILDING TASK
        #warning ========================

        // This is the actual inclusion of the task's source code
        #include ABmt_TaskCode

        #else
            #error This lib is for use with ABmt and is intended to be included by the Scheduler
    #endif

#endif

'~~ // /* #endif */
```

```
'
'v 0.02     24Nov18 Initial Release
'   •   ample comments/diatribe and code will eventually be descriptive of implementation
'
' TODO/FIXME:
'   -   Use preprocessor directives to expand this to handle both the 824 and 54102 (54102 currently)
'   -   add checks here to ensure version compatibility between this and the Scheduler
'   -   lots of other stuff... :)
'
' ============================================================================

#ifndef ABmt_TaskInclusionComplete          ' 'single Load' construct
    #define ABmt_TaskInclusionComplete

    #if !defined onEBT || !defined FilePP   ' danger will robinson, BT's BPP doesn't support non-boundary (intra-token) macro expansion
        #error ABmt is written to be used with the FilePP Preprocessor (integrated via EBT), to facilitate more robust compile-time functionality than what BT`s BPP can offer
    #else   ' with FilePP, intra-token macro expansion works, but path resolution behavior is different than BT's BPP
        // #warning ********** ABmt_TaskInclusionComplete

        #ifdef ABmt_SchedulerCompile

            // this removes prior defined regex preprocessor construsts - see ABmt_TaskInclusionBegin.lib
            #rmregexp /(\s*ABT)_/$1_ABmt_TaskID_/
            #rmregexp /^\s*(?i)(main:).*/sub ABT_ABmt_TaskID_MainLoop/
            #rmregexp /^\s*(?i)(end)\s\B/endsub/

            #defplus _ABmt_TaskCount_buildup )
            // #defplus is not a type - see filepp docs

            #define _ABmt_TaskCount _ABmt_TaskCount_buildup

            #pragma filepp SetWordBoundaries 1

            #warning ========================'============================================
            #warning TASK INCLUSIONS COMPLETE
            #warning ========================'============================================
        #else
            #error This lib is for use with ABmt and is intended to be included by the Scheduler

        #endif

    #endif

#endif
```

SubAppendix 3.1: Document Control and Record of Revisions

Document Control	
Document Number	0240_2330_FilePP.um.001
Document Title	EBT File Preprocessor (FilePP) User Manual
Authored By	Tod Wulff, Chief Hack @ The House of Wulff
Source Location	https://coridium.us/tod/EBT/FilePP_User_Manual.docx
Published Location	https://coridium.us/tod/EBT/CurrentRelease/EBT_User_Manual.pdf as an addendum thereto
Acknowledgements	Bruce Eisenhard, Founder/HMFIC @ Coridium Corp Gary Zwan, ARMbasic/Embedded Dev Compatriot Darren Miller, Developer of FilePP

Record of Revisions			
Version Number	**Date issued**	**Revisor**	**Reason for Revision**
1.0	27 Aug 2020	Tod Wulff	Initial Release

Appendix 1: Document Control and Record of Revisions

Document Control	
Document Number	0233_1824_EBT.um.001
Document Title	Extended Basic Tools (EBT) User Manual
Authored By	Tod Wulff, Chief Hack @ The House of Wulff
Source Location	https://coridium.us/tod/EBT/EBT_User_Manual.docx
Published Location	https://coridium.us/tod/EBT/CurrentRelease/EBT_User_Manual.pdf
Acknowledgements	Bruce Eisenhard, Founder/HMFIC @ Coridium Corp Gary Zwan, ARMbasic/Embedded Dev Compatriot

Record of Revisions

Version #	Date issued	Revisor	Reason for Revision
0.1	21 Aug 2020	Tod Wulff	- Initial draft document
1.0	22 Aug 2020	Tod Wulff	- Initial Release Version
1.0a	24 Aug 2020	Tod Wulff	- Various minor fixes: typos, missing words, extra text box removed, layout fixes, etc. - Added placeholder for FilePP Manual & Examples - Added note re Wrapped .tcl Scripts & borked DND
1.1	26 Aug 2020	Tod Wulff	- Integrated TargetExplorer User Manual as Addendum 2 herein
1.1a	n/a	Tod Wulff	- Fixed Manual Number in Document Control - Incorporated TE Manual Edits into Addendum 2
1.2	27 Aug 2020	Tod Wulff	- Integrated FilePP User Manual as Addendum 3 herein