**BMPTK**

**a cross-target minimalist**

**Bare Metal Programming Toolkit**



**Manual**

|  |  |
| --- | --- |
|  | Wouter van Ooijen  wouter@voti.nl  Version 1.0, work-in-progress, last modified 2016-02-19 |

Summary

Bmptk is a free minimalist make-based cross-target toolkit for developing bare-metal micro-controller applications on Windows or Linux, in assembler, C, or C++, using GCC toolchains.

**Table of contents**

[1. Introduction 4](#_Toc439672983)

[2. Getting started 5](#_Toc439672984)

[2.1. Bmptk 5](#_Toc439672985)

[2.2. Tool locations 5](#_Toc439672986)

[2.1. Make 5](#_Toc439672987)

[2.2. Python 6](#_Toc439672988)

[2.3. Toolchains and download tools 6](#_Toc439672989)

[2.3.1. Native 6](#_Toc439672990)

[2.3.2. ATmega 6](#_Toc439672991)

[2.3.3. MSP430 7](#_Toc439672992)

[2.3.4. ARM & Cortex 7](#_Toc439672993)

[2.3.5. ESP8266 9](#_Toc439672994)

[2.4. Serial port configuration 9](#_Toc439672995)

[2.5. Editors 10](#_Toc439672996)

[2.5.1. CodeLite 10](#_Toc439672997)

[2.5.2. PSPad 12](#_Toc439672998)

[2.5.3. Sublime Text 14](#_Toc439672999)

[2.5.4. Geany 14](#_Toc439673000)

[2.5.5. Notepad++ 15](#_Toc439673001)

[2.5.6. Programmer's Notepad 17](#_Toc439673002)

[2.5.7. Eclipse: sorry, not supported 19](#_Toc439673003)

[3. Use 20](#_Toc439673004)

[3.1. Overview 20](#_Toc439673005)

[3.2. Quick test of your setup 20](#_Toc439673006)

[3.3. Makefiles 21](#_Toc439673007)

[3.4. Bmptk services 24](#_Toc439673008)

[3.4.1. Header 24](#_Toc439673009)

[3.4.2. Functions 24](#_Toc439673010)

[3.4.3. Configuration macros 24](#_Toc439673011)

[4. Targets 26](#_Toc439673012)

[4.1. Chips 26](#_Toc439673013)

[4.2. Boards 26](#_Toc439673014)

[4.3. Adding new targets 27](#_Toc439673015)

[4.4. Unsuccessful attempts 27](#_Toc439673016)

[5. Internals 29](#_Toc439673017)

[6. Bugs, limitations, issues, etc. 30](#_Toc439673018)

# Introduction

Bmptk (Bare Metal Programming Toolkit) is a make-based tool for developing and downloading bare-metal[[1]](#footnote-1) micro-controller applications on Windows or Linux in assembler, C or C++, using GCC toolchains. It is a thin ‘compatibility layer’ on top of the target-specific tools. The intended users are micro-controller developers who want to be able to switch easily between different micro-controller targets without remembering all the toolchain details.

Bmptk relies on external GCC toolchains (compiler, linker etc.) for the various targets, which are invoked by the bmptk makefile. Different targets require different variations of the GCC toolchain to be installed.

The bmptk makefile can be used from the command line, or it can be called from an editor or IDE. It supports building and downloading an application, and (when possible) communication with the application over a serial line. Debugging is not supported. [[2]](#footnote-2)

Bmptk is a first step towards target-agnostic development: for the user, the build-download-run process is the same for all targets, but the application code still needs to be target-aware: the programming interface offered by the various target chips in the form of vendor-supplied header files varies a lot. The hwcpp library (distributed with bmptk, but mostly independent from it) offers more target-independence by offering a portable interface on which an almost target-agnostic C++ application can be built.

The bmptk files are provided under the Boost license, which basically means that you can do everything you want with it, except that when you re-distribute the source, it must be under that same license. Bmptk contains some third-party files that might be under a different (but still free) license; check the license.txt file for the details.

Bmptk makes a number of assumptions that are appropriate for the kind of targets it is intended for. In particular (by default):

* For bare-metal targets, memory allocation is supported, but memory de-allocation is not.
* The -Wall -Werror flags are used, so (with a few exceptions) all warnings are enabled and all warnings are treated as errors.
* For C++: RTTI, exceptions, and global objects that require initialization, are disabled.
* There is no support for interrupts. This doesn’t rule out that an application uses interrupts, but it must do all the work itself.

Bmptk is 'work in progress'. If you somehow found and used bmptk I am interested to hear your experiences and comments.

Wouter van Ooijen

wouter@voti.nl

# Getting started

Bmptk can be used on Windows and Linux (tested on Fedora and Raspbian Jessie). To use bmptk you will need bmptk itself, a toolchain (compiler etc.) for your target(s), a downloader for your target(s), and (unless you want to use bmptk from the command line) an editor. If you want to do cross-compilation on a Raspberry Pi get (at least) a 64 Gb flash card (32 Gb is OK for native work).

## Bmptk

To start using bmptk you must first get bmptk itself and place it somewhere on your computer. Bmptk doesn’t care where, but don’t make the path too long and (on Windows) don’t use a funny location (Desktop, virtual drive, etc.) If you don’t care, I suggest you put it in C:\bmptk (Windows) or in ~/bmptk (Linux). ~~You can download the latest ‘stable’ version as a zip from~~ [~~www.voti.nl/bmptk~~](http://www.voti.nl/bmptk)~~, or the bleeding edge from~~ [~~http://code.google.com/p/bmptk~~](http://code.google.com/p/bmptk)~~, as zip or as a git repository:~~

|  |
| --- |
| ~~git clone https://code.google.com/p/bmptk~~ |

## Tool locations

Unless instructed otherwise bmptk assumes that you use the default tools, installed in their default locations on drive C. If this is the case for you, you can skip this. When you have installed one or more tools in a different location you must:

* In the bmptk root directory, make a copy of the Makefile.local file and call it Makefile.custom
* In that Makefile.custom, comment/uncomment the appropriate tool lines, and/or edit the appropriate lines to reflect where you placed the tools.

The purpose of this copying is that when you install a new version of bmptk over the old one, your Makefile.custom (which is not present in bmptk) will be preserved. If you think you will never do this (and you like a surprise when you inadvertently do) you can choose to edit the Makefile.local directly.

Depending on the Windows variety, 32-bit applications will be put either in C:\Program Files or in C:\Program Files (x86). The Makefile.custom uses the WPF32 macro (defined by the Makefile.inc). which points to the windows directory for 32-bit applications, so you don’t have to spell this out.

## Make

Bmptk uses the make tool to run make scripts. Unfortunately, the script language use by various Windows make versions is not 100% compatible. Various applications tend to install their own make executables, and add their location to the PATH. The result is that execution a ‘make’ command might invoke a make executable that is not compatible with the bmptk makescripts. To prevent this bmptk has its own make executable, called bmptk-make. It is in the bmptk/tools directory. On Windows, you must add the bmptk/tools directory to your PATH so the bmptk-make executable can be found. How this must be done might vary with your Windows version. I use

Start Button 🡺 Configuration 🡺 System 🡺 Advance System Configuration 🡺 Environment Variables

or

Start 🡺 right click Computer 🡺 properties 🡺 Advanced System Settings 🡺 Environment Variables

On Linux there is no need to do this when you use bmptk from the command line, because the make executable of the distribution can be used. But the Code Lite and PSPad editor files that in the bmptk examples directories call bmptk-make. If you want to use these files on Linux, you must make a dummy bmptk-make executable. Assuming make is in /usr/bin (check with ‘whereis –b make’), the command is

|  |
| --- |
| sudo ln /usr/bin/make /usr/bin/bmptk-make |

## Python

Some bmptk tools are Python scripts, hence you need Python. On windows, install a Python if you don’t have one, and modify the Makefile.custom file accordingly. The scripts are compatible with Python version 2 and version 3. Linuxes generally have a Python installed by default, and bmptk assumes that it is in the path.

## Toolchains and download tools

Bmptk can be used on Windows or Linux, and can build for a number of targets. The following sections show the details for each target family. The full list of supported chips and boards is show in the chapter Target (p 26). You need to install (only) the toolchains and download tools for the targets that you are interested in. For Linux the commands for Fedora and Raspbian Jessie are show, for other Linuxes you will have to find things out for yourself.

### Native

The native targets are meant for test-running your code on your native (Windows or Linux) platform, not for developing full-featured native applications. On Windows a MinGW toolchain is used, on Fedora the standard GCC/G++ must be installed. Raspbian Jessie has a GCC installed by default.

|  |  |  |
| --- | --- | --- |
| Host | Toolchain | Source, or install command |
| Windows | MingGW | <http://mingw-w64.org/doku.php/download/mingw-builds> |
| Fedora | GCC/G++ | sudo yum install gcc gcc-c++ |
| Raspbian Jessie | GCC/G++ | installed by default |

### ATmega

For ATmega chips on Windows the toolchain provided by the manufacturer as part of the AVR studio development environment is used. Be prepared: AVR Studio is a big install, with some (trivial) user interaction. On Fedora and Raspbian Jessie the indicated AVR packages must be installed.

|  |  |  |
| --- | --- | --- |
| Host | Toolchain | Source, or install command |
| Windows | AVR Studio | Easier:  <http://www.atmel.com/tools/ATMELAVRTOOLCHAINFORWINDOWS.aspx>  <http://www.atmel.com/microsite/atmel_studio6/>  <http://www.fischl.de/usbasp/>  <http://zadig.akeo.ie> |
| Fedora | avr-gcc, avr-libc, avrdude | sudo yum install uisp avr-libc avr-gcc-c++ rxtx avrdude |
| Raspbian Jessie | GCC/G++ | sudo apt-get install gcc-avr binutils-avr avr-libc avrdude |

AVRDUDE is used as download tool, either

* with the Arduino boot-loader (if it is already present on the chip), or
* with the AVRISP mkII programming hardware, with the required USB driver (I found the zadig tool to be the only way to install the libusb driver on my PC).

AVRDUDE itself for Windows is provided with bmptk. For the Linux the install commands show how to get avrdude, and the UBS driver is part of the Linux kernel.

### MSP430

For MSP430 chips on Windows the toolchain of CodeComposer is used. Don’t be intimidated by the rather aggressive compliance statement that you must fill in to download this packages. On Fedora the mps430 packages must be installed.

|  |  |  |
| --- | --- | --- |
| Host | Toolchain | Source, or install command |
| Windows | CodeComposer | * <http://www.ti.com/tool/ccstudio-msp430> * http://www.ti.com/tool/msp430-gcc-opensource |
| Fedora | GCC/G++ | sudo yum install msp430-gcc msp430-libc |
| Raspbian Jessie | GCC/G++ |  |

For downloading TBW

|  |  |
| --- | --- |
| Launchpad driver +  MSP430Flasher | Source, or download command |
| Windows | <http://www.ti.com/tool/MSP-EXP430G2> + <http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer> |
| Linuxes | <http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430Flasher/latest/index_FDS.html>  <http://www.ti.com/tool/msp430-flasher>  ☹ MSP430 Flasher cannot be built in 64bit Linux environments  install to /bin/MSP430Flasher\_1.3.3 |

### ARM & Cortex

For ARM7 and Cortex chips on Windows a pre-built version of the GCC toolchain is used. If the installation does not take care of this, you must make sure that the directory that contains the executables is added to your PATH, because it contains .dll files that would otherwise not be found. On Fedora the arm-none-eabi packages must be installed, on Jessie the arminarm package is used.

|  |  |  |
| --- | --- | --- |
| Host | Toolchain | Source, or download command |
| Windows | GCC ARM Embedded | <https://launchpad.net/gcc-arm-embedded> |
| Fedora | GCC/G++ | sudo [yum](https://apps.fedoraproject.org/packages/gcc-c++-arm-linux-gnu) install arm-none-eabi-gcc-cs  sudo [yum](https://apps.fedoraproject.org/packages/gcc-c++-arm-linux-gnu) install arm-none-eabi-gcc-c++-cs  sudo yum install arm-none-eabi-newlib |
| Raspbian Jessie | ARMinArm  (= GCC/G++) | git clone https://github.com/ARMinARM/arminarm  cd arminarm  ./setup  (select options 0, 1, 2, reboot) |

The chip header files (as provided by the manufacturers) for Cortex chips are part of bmptk. For the LPC2148 (ARM7) bmptk contains a custom-made header file, because there doesn’t seem to be a free set of header files for pre-Cortex ARM chips.

For NXP chips (LPC…) the lpc21isp serial-port downloader is used. It is slightly modified from the standard lpc21isp, which on a Cortex does not correctly start an application. Lpc21isp uses the serial port handshake lines to force the target into bootload mode. Bmptk assumes that your target hardware supports this. The Windows executable is provided with bmptk, for the Linuxes the executable is build (using GCC) by a ‘make’ command in the bmptk/tools/lpc21isp\_197 directory.

|  |  |
| --- | --- |
| lpc21isp for | Source, or download command |
| Windows | (included in bmptk) |
| Linuxes | cd bmptk/tools/lpc21isp\_197  make |

For an Arduino Due with the ATSAM3X8E chip the bossac tool is used, via the download interface provided by the USB-to-serial converter on the Due board. The port is first opened at 1200 baud to force the ATSAM3X8E into bootload mode (using the DOS mode command). Bossac (for Windows) is provided with bmptk. For Linuxes the bossa-cli package must be installed.

|  |  |
| --- | --- |
| bossac for | Source, or download command |
| Windows | (included in bmptk) |
| Fedora |  |
| Raspbian Jessie | sudo apt-get bossa-cli |

For downloading over USB to one of the STM development boards the ST-LINK Utility is used, which must be installed separately.

|  |  |
| --- | --- |
| ST-LINK for | Source, or download command |
| Windows | <http://www.st.com/web/en/catalog/tools/PF258168> |
| Fedora |  |
| Raspbian Jessie | sudo apt-get install libusb-1.0-0-dev git dh-autoreconf  (subdir)  git clone https://github.com/texane/stlink stlink.git  cd stlink.git  ./autogen.sh  ./configure make  cd flash  make sudo cp st-flash /usr/bin  cd .. sudo cp \*.rules /etc/udev/rules.d  sudo restart udev (or reboot) |

MLK25 chips have an USB bootloader that emulates an USB storage device.

|  |  |  |
| --- | --- | --- |
| **Host** | **Toolchain** | **Source, or download command** |
| **Windows** | **copy\_to\_drive** | **bmptk built-in** |
| **Fedora** |  | **Not supported yet** |
| **Raspbian Jessie** |  |  |

For XMC chips

|  |  |  |
| --- | --- | --- |
| Host | Toolchain | Source, or download command |
| Windows | JLinkExe | <http://www.segger.com/jlink-software.html> |
| Fedora |  | Not supported yet |
| Raspbian Jessie |  |  |

### ESP8266

For the ESP8266 development on Windows Boseji has conveniently bundled a number of packages: GCC, Espressif toolkit, examples, downloader, etc. ESP8266 development is a bit different because you write a user\_init(), which is called by the Espressif code. There seems to be no way (yet) to make a really bare-metal application (without the Espressif code taking control first). TODO: extra for C++.

|  |  |  |
| --- | --- | --- |
| Host | Toolchain | Source, or download command |
| Windows |  | Unofficial Espressif Windows SDK bundle from  <https://github.com/boseji/ESP8266-uof-windows-sdk-portable> |
| Fedora |  |  |
| Raspbian Jessie |  |  |

For downloading the esptool (the Python application, not the converter that is part of the GCC toolchain) is used. This esptool is part of the Windows SDK bundle.

## Serial port configuration

When downloading to or communication with the target uses a serial interface bmptk assumes that it is COM4 (Windows) or /dev/ttyUSB0 (Linux), at 38400 baud. These settings can be changed installation-wide in the Makefile.custom, or for a particular project in the projects Makefile.

When you use an USB-to-serial converter you must likely configfure the port that Windows created for it to COM4. This is done in the device manager (StartButton 🡺 Configuration 🡺 DeviceManager). Right-click on the PORT, select Properties 🡺 Port Configuration 🡺 Advanced. Now you can change the assigned port to COM4. If Windows complains that this port is in use you can ignore this (except of course when it is in use by a hardware port or another converter that you currently have connected.)

On Linux, when you use a built-in serial port, you might need to disable the login process that automatically runs on the built-in ports. Fort usb-to-serial ports this is not needed.

## Editors

As far as bmptk is concerned, it is used by invoking its Makefile. So if you want, you can use bmptk directly the command-line. Most people will prefer using an editor to get the 'IDE' feeling, and to jump to the source line where the compiler found an error. The editor must be able to start the 'bmptk-make build', ‘bmptk-make clean' and ‘bmptk-make run' commands, to capture the output of ‘bmptk-make build', and preferably jump to the correct source line when you click on a build error.

For quick work I prefer PSPad, not because it is the world's best editor, but because the integration with bmptk is simple and reliable. The bmptk example directories have an appropriate \_PSPad\_project.ppr file so they can be used 'out of the box' with PSPad.

For more extensive work (involving multiple projects) I prefer CodeLite.The individual example directories have an \_CodeLite\_workspace.workspace file for just that example. Higher directories often have an \_CodeLite.workspace file too, which starts the editor with all the projects in the subdirectories. This takes a little longer to start, but is easier when you work as set of projects.

But don’t let that frighten you away from using your own favourite editor! The next sections describe how some popular editors can be used with bmpt. Choose your favorite, or use it as inspiration to configure your favorite editor (and let me know how you did it so I can add it).

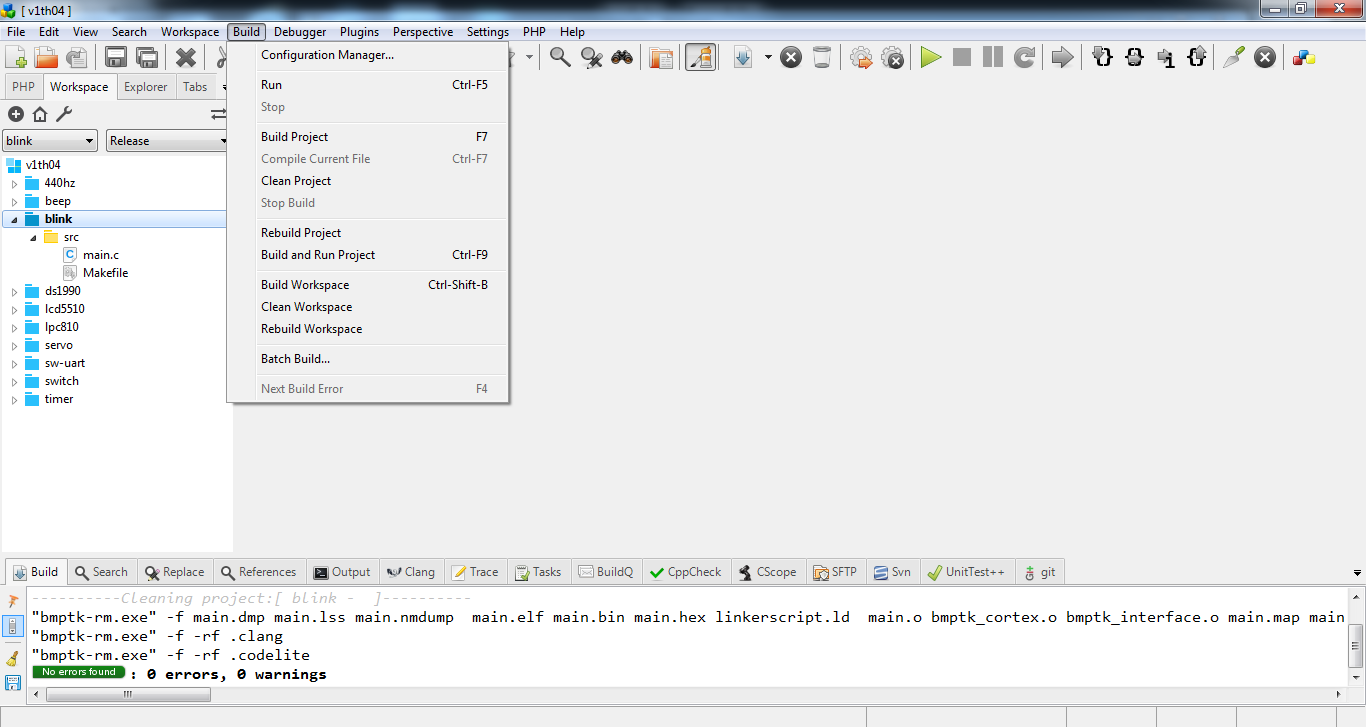
### CodeLite



CodeLite (http://codelite.org/) is an open-source IDE, available for Windows, Mac, and a few Linuxes.

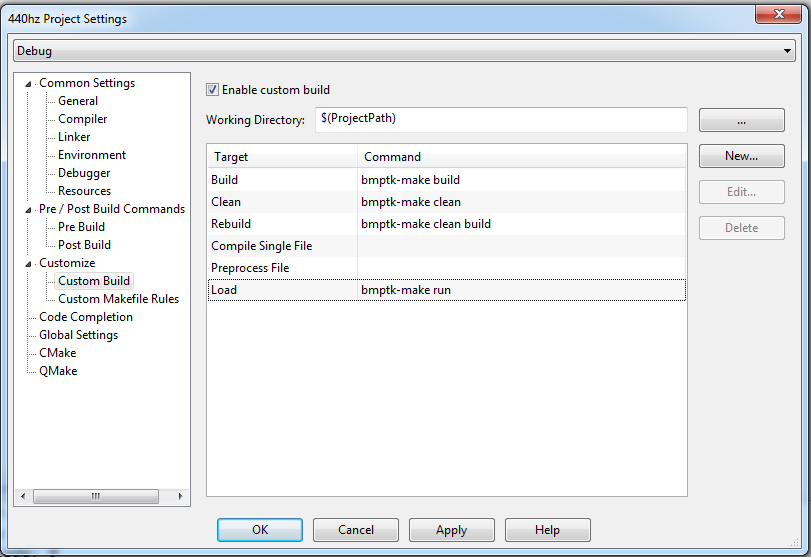
CodeLite is organized around Workspaces, that each contain a number of projects. A \*.workspace file contains pointers to the \*.project files that are part of the workspace. A project file doesn’t contain a pointer back to its workspace, hence a project can be part of more than one workspace, but you can’t start CodeLite with just a project (it needs some workspace to be a part of).

On windows, you can start CodeLite with a specific workspace by double-clicking on a \*.workspace file (maybe you must associate .workspace with the CodeLite application). Under the build tab you can choose Run, Build Project, Clean Project and Rebuild Project. The button and the CTRL-F5 can also be used to run (which implies a build).



Each bmptk example directory has an appropriate \_CodeLite.workspace and an \_CodeLite.project file. One level higher the directory will often contain a \_CodeLite.workspace file that contains all the projects. When you start a new bmptk project using CodeLite, the easy way is to copy an existing directory and adapt it. The rest of this section explains how CodeLite can be configured to be used with bmptk.

By default CodeLite generates its own makefiles, but it can work with an external makefile by creating a project (Workspace -> New Project) using the the custom template (Others -> Custom Makefile) . Now you can configure the project’s build calls (right click on the project, select settings -> Customize -> Custom Build). As shown, edit the Build, Clean and Rebuild targets to issue the appropriate bmptk-make commands. The ‘Enable custom build’ should be checked.

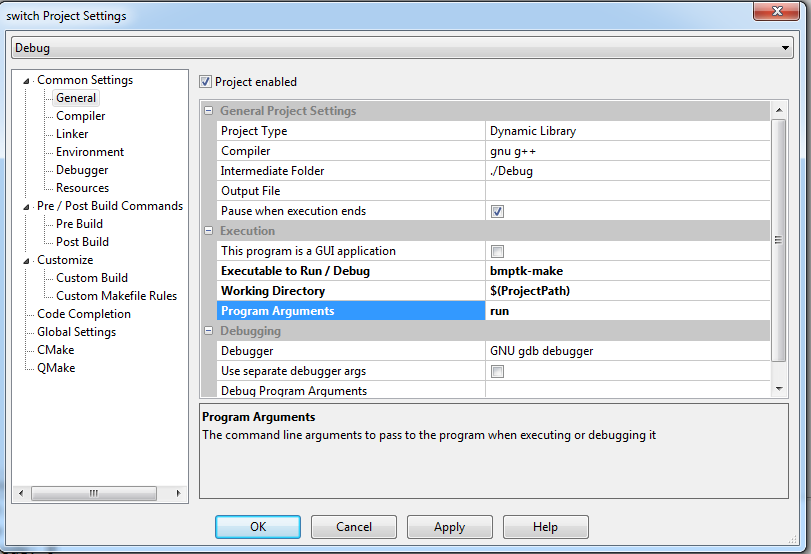


CodeLite creates two subdirectories (.clang and .codelite). The ‘bmptk-make clean’ command removes the .clang subdirectory, and attempts to remove the .codelite subdirectory, but when run from inside CodeLite this will fail because it contains files that are opened by CodeLite.

You could add Run target to the Custom Builds, here, but that would be executed in the build window, which does not support user interaction. Better edit the settings for Execution to invoke bmptk-make with the ‘run’ argument.

On Raspbian, installing Code Lite doen’t associate it with the .workspace extension. In the File Manager you can do this manually: right-click a .workspace file, select ‘open with’, select the ‘Custom Command Line’ tab, and enter

* codelite %f
* Code Lite
* And check ‘Set selected application as default action for this file type’.



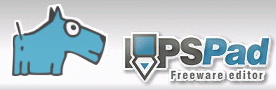
Note that by default CodeLite maintains two different sets of settings, one for Debug and one for Release. I use only the Release setting, because bmptk doesn’t support debugging.

You can check “This program is GUI application” to suppress the “press any key to continue” after downloading and running the terminal emulator.

A fresh Custom Makefile project has virtual folder tab, so you must create one by right-clicking the project and selecting New Virtual Folder. You can make virtual folders as you like, but note that they have no meaning for bmptk: the content of the Makefile defines which files are compiled to build your project.

|  |  |
| --- | --- |
| Platform | Source, or download command |
| Windows |  |
| Fedora |  |
| Raspbian Jessie | sudo apt-get install codelite |

### PSPad



PSPad (http://www.pspad.com) is a free code editor for Windows. It can be configured to work with bmptk by including a few lines in the .ppr file to attach external commands to two buttons. The example directories each contain such an \_PSPad\_project.ppr file.

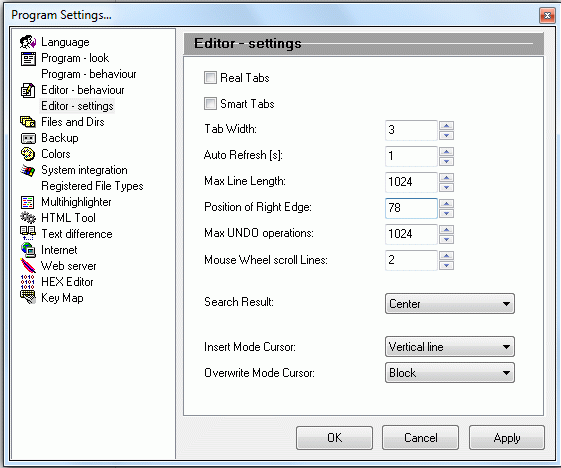
With this setup in place the "run external compiler" button (or CTRL-F9) can be used to run "bmptk-make build". The result of this command is captured in the result window, and you can click on an error message to jump to the corresponding source location.

When this has been successful the "open active file in external program" button (the one with the lightning bolt, no keypad shortcut) can be used to run "bmptk-make run" or "bmptk-make clean".

This lightning bolt button can also be used for building the application, but because the command output is not captured by PSPad this is less useful. For these buttons and commands to work it is required that the active file is one of the source files (.h or .cpp) in the working directory. If another type of file is open the buttons will be inactive (grayed out).

If a source file in another directory is active PSPad will run the bmptk-make commands in **that** directory. Watch out when you open a new file in PSPad. The default directory that appears in the file selection menu is the last directory that you used in this way, NOT the directory PSPad was started in. Hence when you switch projects it is very easy to open the wrong source or Makefile and be flabbergasted that your changes seem to have no effect at all.

To open the Makefile you must right-click it and choose "open as text file". Don't forget to switch to a source file when you subsequently want to build or run the application. If things seem to go very wrong it might be an idea to open the .ppr file as text file to check which source files you really are editing.

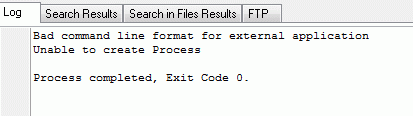


PSPad will save all changed files before it runs an external command. By default PSPad will use tab characters to create indentation, which IMHO is very bad because no two programs agree on how much indentation each tab creates.

Use Settings 🡺 Program Settings 🡺 Editor Settings to correct this. I prefer 3 spaces per indentation level.

Some more tips:

* view => Line Numbers : toggle showing line numbers
* view => Special Chars : toggle showing tabs, newlines, etc
* Tools => EyeDropper : get the 16-bit color value of a pixel on our screen
* Setings => Program Settings => Editor - behaviour : uncheck "completion of chars ({[<"'"
* On some versions of windows, opening a second copy of PSPad by clicking on a .ppr file will give a number of very weird error messages. Close these messages, and eventually the new project will be opened.
* Running bmptk make completely ignores the project file list as maintained by PSPad.
* By default PSPad replaces TABs with spaces. This is OK, except when editing makefiles. You can insert a TAB in a makefile by cut-n-pasting an existing TAB.
* You can use the run command (ALT-F9) to build and run your application, but remember that unlike the build command (CTRL-F9) it won't save your changes first! This is PSPad behavior, I don't know how to change it.
* When you run the application a (great) number of times from PSPad some resource seem to get depleted and further bmptk calls (build, run, clean) will fail. Restarting PSPad will solve this, but it is rather annoying when you don't know why everything fails.
* On my laptop (windows XP) PSPad show the names of the files in the edit tabs, but somehow this does not work on my 64-bit Windows 7 PC.
* When freshly started with a project directory copied from another PC PSPad will sometimes produce this error:



I am not sure what causes this problem but it can be fixed by quitting PSPad and restarting it.

### Sublime Text



Sublime text (http://www.sublimetext.com/) is a text editor for Windows, Linux and a few other OSes. Evaluation is free, but after the evaluation period it will nag you to pay ($70). It can invoke external commands via ‘Build Systems’. To add a Build System for bmptk select Tools 🡺Build System 🡺 New Build System. Insert the text show below, and save it as bmptk.sublime-build.

|  |
| --- |
| {  "cmd": ["bmptk-make", "build"],  "variants": [  {  "name": "Clean",  "cmd": ["make", "clean"]  },  {  "name": "Run",  "cmd": ["make", "run"]  }  ]  } |

Now you can build the project via Tools 🡺 Build (Ctrl-B). The other commands (run and clean) must be invoked via the Tools 🡺 Command Pallete. Note that all bmptk commands are executed in the directory of the file you are currently editing.

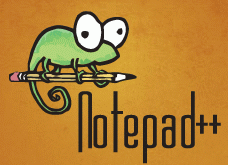
### Geany

Geany (http://www.geany.org) is a free code editor that is available for a number of platforms. It can call an external makefile. You can use Build => Make (Shift-F9) to build the project in the directory of the currently active file. To run or clean you can use Build => Make Custom Target (Shift-Control-F9) and type the target you want to 'build'. While the make is running the editor is totally blocked, and no progress is shown until the make has finished. When you called fora ‘bmptk-make run' this means: until you stop the executable or emulator, which is not very convenient when you want to see the output of your application or you want to interact with it.

You can use Build => Set Build Commands to add new commands, for instance 'run' to call ‘bmptk-make run' directly. Such options appear in the Build menu item, but not in the Build dropdown list. These customizations seem to be stored globally, so you have to customize Geany only once.

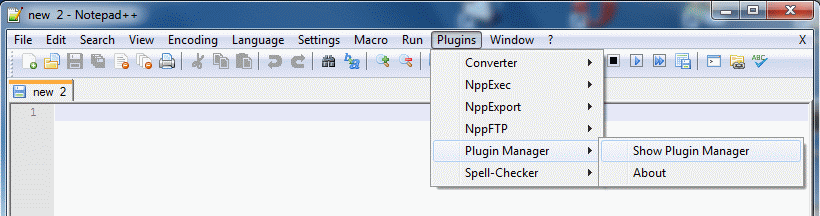
By default Geany does NOT save the edited files before starting an external command, which is a bit counter-intuitive. I suppose this behavior can be changed, but I have not yet found how.

### Notepad++

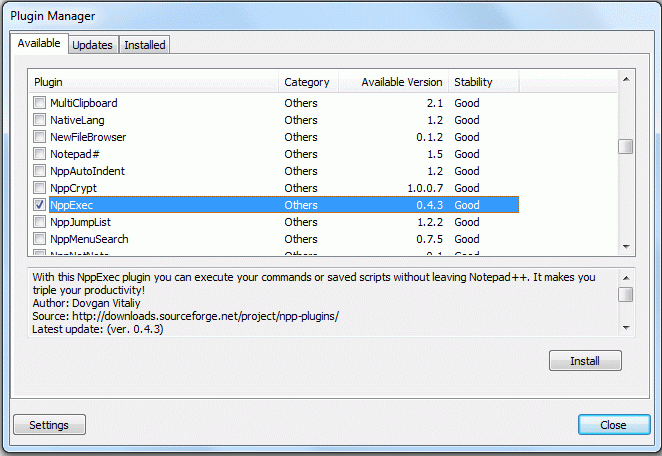


Notepad++ (<http://notepad-plus-plus.org>) is a free lightweight text editor for Windows. With the NppExec plugin Notepad++ can call do the basic chores of an IDE: call an external tool, capture the output, and jump to the source line that corresponds to an error message.

To install the NppExec choose Plugins => Plugin Manager => Show Plugin Manager.



Check the box for NppExec and click Install, and let Notepad++ restart itself.

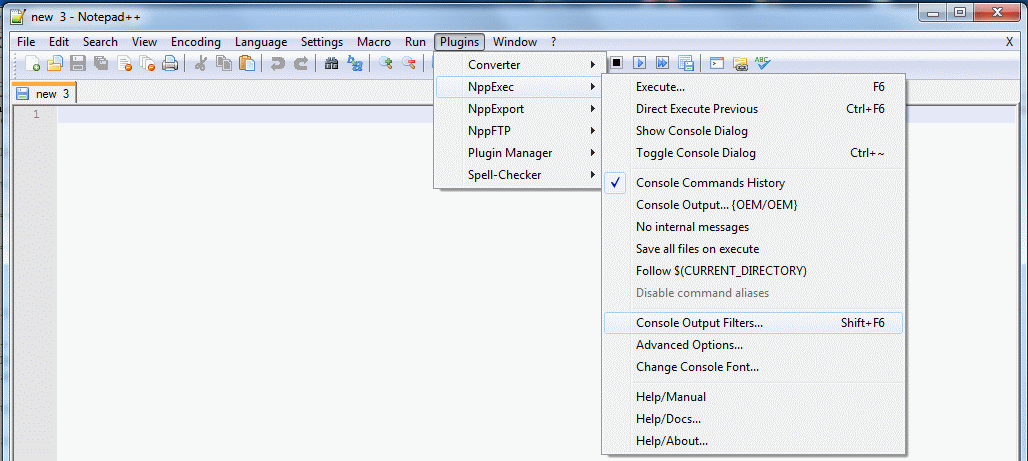


Now F6 will open a window in which you can type the command you want to be executed:

|  |
| --- |
| npp\_saveall  CD $(CURRENT\_DIRECTORY)  bmptk-make build |

You can save it, I suggest the name ‘bmptk-make build'. This script will save all files, CD to the directory of the active file, and run ‘bmptk-make build'. The output will be captured in an output window at the bottom of the screen. Likewise you can create ‘bmptk-make run' and ‘bmptk-make clean'.

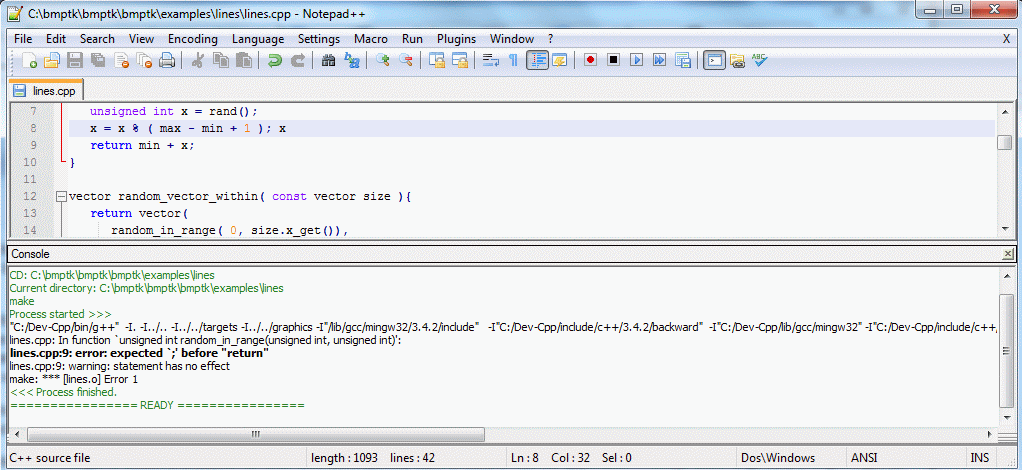
The next step is to get the NppExec to recognize the source references in the output. Open Plugins => NppExec => Console Output Filters.



In the filters window you must specify the pattern for errors, and I suggest the pattern for warnings too. Select B for the errors to make them stand out.

|  |
| --- |
| %FILE%:%LINE%: error:\*  %FILE%:%LINE%: warning:\* |

Now you can use F6 to open the command window, change the command, and run it, or CTRL-F6 to run the previous command without getting the command window. The result will appear in the command result window at the bottom, with the errors in bold. Click on an error to jump to the corresponding source line.



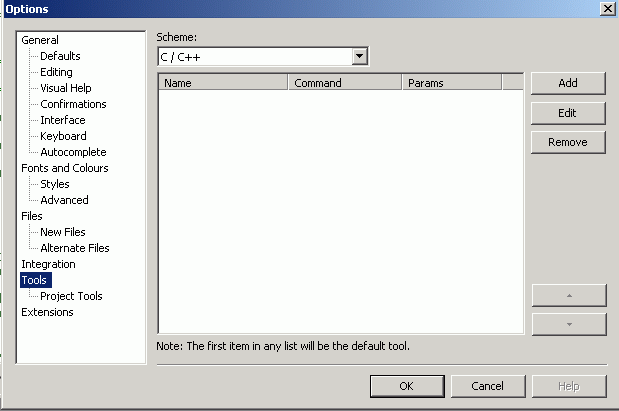
When you have a successful build you can use run (F5) to run ‘bmptk-make run'.

More information about the command language can be found at <http://sourceforge.net/apps/mediawiki/notepad-plus/index.php?title=Compiling_Source_Code>

### Programmer's Notepad



Programmer's Notepad (http://www.pnotepad.org) is another free lightweight text editor for Windows, specifically designed for editing source code. It is available as source, but I haven’t seen any ports to other OSes. With a little customization PN can call do the basic chores of an IDE: call an external tool, capture the output, and jump to the source line that corresponds to an error message.



Open a C++ file, and select Tools => Options => Tools:

Now click Add. Create an entry for ‘bmptk-make build'. You can leave the Console I/O tab unchanged. The shortcut key can be configured by selected the field and pressing the key (or combination) you prefer. I choose F6 for building, F8 for running, no shortcut for cleaning. Likewise make the entry for cleaning.

|  |  |
| --- | --- |
|  |  |

Finally, make an entry for running. Now you can activate the bmptk-make commands either by their shortcut or from the Tools menu.

|  |  |
| --- | --- |
|  |  |

As shown the build and run commands will first save all modified files.

Tips:

* Enable View 🡺 Line Numbers
* The make commands will be executed in the directory of the selected file, hence you must select a file in your project directory!

### Eclipse: sorry, not supported

I spend some time trying to get bmptk tow work from Eclipse. Running the make commands to build and clean the project is no problem, but downloading failed because both the run command and the external tool facility insist that the external executable can’t do I/O redirection, and you can’t give it input from the keyboard. Both are needed for bmptk, so until someone shows me how to solve both problems I can’t support Eclipse.

# Use

## Overview

The bmptk makescript accepts three commands:

* **make[[3]](#footnote-3) build**  
  The build (compiles and links) your project. All produced files are put in the current (project) directory.
* **make run**  
  The run command downloads your application to the target, starts it, and (when possible) starts a terminal to communicate with it.
* **make clean**  
  The clean command removes all files created by build files from your directory.

Bmptk builds your project using the gcc toolchain and the libraries that are provided with the toolchain, but (in most cases) it provides its own startup code and linkerscript. It is expected that bmptk applications will make very limited use of the C and C++ libraries, because most standard library functionality uses far too much ROM and/or RAM, or has run-time properties that are not compatible with a small real-time embedded application.

For C++, the bmptk linkerscript generates an error when your application has global ctors (global objects that require run-time initialization). When appropriate, the startup code provides a very simple (and fast) malloc / new implementation, and a free / delete that generates a linker error when it is used. The idea is that a bmptk application can use malloc (or new) in its initialization, but it should never free memory.

For C++, the (default) compiler settings exclude exceptions and the use of RTTI (Run Time Type Information). Dummy implementations for a few exception-related functions are provided to avoid dragging in large amounts of library code.

## Quick test of your setup

To test your setup, go to an example for your target, for instance bmptk/examples/boards/db103/blink, or bmptk/examples/boards/windows/hello-c. You can start the PSPad editor (assuming you have installed it) by double-clicking on the .ppr file. If that does not work immediately, associate the .ppr extension with the editor's executable.

In the editor, press CTRL-F9 or click the file-to-10101 icon to build the application. Next you can press SHIFT-F9 or click the file-with-lightning-bolt icon and select run to run the application. For the native target this will run the application. For a micro-controller target this will start the download tool to download the application to the microcontroller and run it. If things don't work as expected, the window in which the action takes place might close before you can read what has happened. In that case you should use the command line to do a “make build” or “make run” in the choosen directory.

## Makefiles

Each bmptk project directory has a project-specific Makefile. This file specifies the properties of the project, specifies where the bmptk installation is located, and finally invokes the central bmptk Makefile.inc (which is in the bmptk root directory).

For the bmptk examples the last two steps are done by assuming that the bmptk main directory (and hence the location of the Makefile.inc) is the next higher directory.

|  |
| --- |
| # Defer to the next higher Makefile.inc to do the work  BMPTK := $(BMPTK)../  include $(BMPTK)Makefile.inc |

Each intermediate directory contains a Makefile.inc with the same lines, so the include calls ripple up through the directory tree to the bmptk root, where the real Makefile.inc is invoked.

When you put a bmptk project outside the bmptk tree, you must specify the bmptk location using an absolute path. An example can be found in bmptk/examples/boards/db103/blink-absolute-path.

|  |
| --- |
| # specify where the bmptk files can be found  BMPTK := F:/\_\_\_\_bmptk/bmptk/  # defer to the bmptk Makefile.inc to do the work  include $(BMPTK)Makefile.inc |

As a minimum, a project Makefile must specify the target for which the application is to be built. The target can be either a board or a chip. The lists of valid targets are in the chapter Targets (p. 26) in the first column of the Chips and Boards tables.

|  |
| --- |
| TARGET := db103 |

By default the Makefile assumes that main.cpp (or main.c, main.cc, or main.asm, whichever is present) is the one and only source file to be compiled, and that this is to be the (base) name of the project files (executable, map file, etc.) that are to be generated. If your project has a different name, you must mention it in the Makefile, (without extension), and a .cpp, .cc, .c or .asm file with that name will be included in the project:

|  |
| --- |
| PROJECT := alarmclock |

If your project has more source or header files than just the main file, you must add them in the makefile. Using the += assignment operator so you can have multiple lines that add sources and headers.

|  |
| --- |
| # Specify project-specific files (other than the main project file, if any)  SOURCES += display.c timer.cpp  HEADERS += display.h timer.hpp |

The sources are the files that will be compiled. When any of the headers are changed, the makefile will re-compile all your source files. This is a bit conservative, but at least it is on the safe side, and for the type of small-system projects that bmptk is meant for recompiling everything should not take long.

If some of the source files are not in the current directory you must specify in which directories the system must look for missing files. Again, using the += operator you can have multiple lines that add search directories.

|  |
| --- |
| SEARCH += ../uart ../timer |

By default a build creates the files that are needed to download and run your application. You can request extra things to be created, like an assembler listing (.lst) for a specific source file.[[4]](#footnote-4)

|  |
| --- |
| # specify (extra) results you want to get , for instance a specific .lst file  RESULTS += main.lst |

The default amount of RAM reserved for the stack depends on the target. The values are shown in the Targets table (p. 26), in most cases it is75% of the total RAM. You can change this by setting the STACK\_SIZE parameter.

|  |
| --- |
| # explicitly specify the stack size  STACK\_SIZE := 4096 |

Alternatively, bmptk can (attempt to) determine the required stack space for you by analyzing the .lss (assembler output) for the application. This imposes some restrictions on your application: the call tree must be visible from the assembler output (no recursions, no calls through function pointers or through virtual methods) and the stack usage for local variables must be fixed at compile time (no run-time sized local variables). On the bright side, those are restrictions you would want to impose on your embedded project anyway, and with a calculated stack size they are enforced. The patterns that the stack size analyzer recognizes are limited to what I found in the compiler output, so you might run into trouble with it. Let me know and I might fix it. For large source files, the stack size calculation (a Python script) can take a long time.

|  |
| --- |
| # let bmptk determine the stack size (not supported for all targets)  STACK\_SIZE := AUTOMATIC |

You can put all settings required for your project in the project’s Makefile, but when you have a set of related projects in subdirectories, you can have each project’s Makefile defer to the higher directory, and put a Makefile.inc there that contains the common settings. In that case you should use weak assignments (?=)[[5]](#footnote-5) in that common Makefile.inc, so the individual Makefiles can still override the common Makefile if they need to. This is used in the examples directories,

The bmptk Makefile uses a number of settings for which the default value will in most cases be OK. You can preempt a default value by assigning a value in your projects Makefile, or (when you want to affect all projects) in the Makefile.custom.

When a serial port is needed to download the application to the target (and to communicate with the target) the system by default uses COM4 at 38400 baud. When this does not suit you can specify the port and/or the baudrate.

|  |
| --- |
| # Specify a port and baudrate for serial downloading and communication  SERIAL\_PORT := COM1  SERIAL\_BAUDRATE := 115200 |

If you want to change some settings globally (for all your projects) you can do so in the bmptk/Makefile.custom file. Create it by copying bmptk/Makefile.local and add the declarations you want to have effect for all your projects. Note that this file is included after your local Makefile, so assignments should be made with ?= so they do not overrule assignments made in the individual project's Makefile.

|  |
| --- |
| # in bmptk/Makefile.custom:  # The serial port I use on this PC  SERIAL\_PORT ?= COM19 |

The following table summarizes the variables that can be used to modify the bmptk make process to your needs. Unless stated otherwise, the default value is empty.

|  |  |
| --- | --- |
| **makefile variable** | **effect** |
| BMPTK | This variable must (in the end) point to the root of the bmptk tree. Its value can be absolute or relative (to the project directory). |
| DEPENDENCIES | This is a list of extra files (beyond the sources and headers) that the application depends on. |
| DOWNLOAD\_DRIVE | For targets that download by copying to a (virtual) drive: the drive to copy the image to. The default is ‘mbed:’. |
| TARGET | The value identifies the target board or chip for which the project will be built. |
| HEADERS | This is the list of the files that are included by the sources files. Bmptk (conservatively) assumes that when one of these header files has changed, all sources must be re-compiled. |
| OBJECTS | This is the list of extra object files that must be linked into the application. |
| PROJECT | This is the name of the project. It is used as the root file name for the generated files (.exe, .hex, etc.). The source file with this name and one of the supported extensions (.c, .cc, .cpp, .asm) is implicitly one of the SOURCES file. Default: ”main”. |
| RESULTS | Additional results (beyond the ones bmptk generates automatically) that must be created, for instance an .lst file for a particular source file. |
| SEARCH | This is the list of directories (other than the project directory, the compiler-implicit directories and the bmptk-implicit directories) that contain source files for the project. Each directory can be specified with an absolute path, or a path relative to the project directory. |
| SERIAL\_BAUDRATE | For targets that downloads or communicate using a serial port: the baudrate to use. The default is 34800 baud, unless the target requires a different baudrate. |
| SERIAL\_PORT | For targets that download or communicate using a serial port: the serial port to use. The default is COM4 (Windows) or /dev/ttyUSB0 (Linux). |
| STACK\_SIZE | The size of the target stack. Can be a numeric value, or AUTOMATIC (in which case bmptk will try to determine the required stack size from the generated .lss project (dis) assembly file). The default is a fixed size that depends on the target, generally about ¼ the size of the RAM. |
| SOURCES | This is the list of the files that must be compiled and linked to form the application. Don’t include paths in the specifications: use SEARCH to specify directories that must be searched. |

The next table shows the compiler and linker options that are by default used by bmptk, but can be overruled in a project makefile.

|  |  |  |
| --- | --- | --- |
| **makefile variable** | **default** | **effect** |
| CPP\_LANGUAGE | -std=c++11 | For C++, use the 2011 standard |
| CC\_LANGUAGE | -std=c99 | For C, use the 99 standard |
| AS\_LANGUAGE | -x assembler-with-cpp | For assembler, allow preprocessor directives |
| ALL\_ERRORS | -Wall –Werror | Enable all warnings (except those ignored by ALL\_IGNORE) and treat them as errors |
| ALL\_IGNORE | -Wno-maybe-uninitialized  -Wno-unused-local-typedefs | Ignore these warnings because they sometimes can’t be avoided |
| ALL\_OPTIMIZATION | -Os | Optimize for size |
| CPP\_RTTI | -fno-rtti | Don’t use RTTI |
| CPP\_EXCEPTIONS | -fno-exceptions | Don’t use exceptions |
| CPP\_THREADSAFE | -fno-threadsafe-statics | Don’t use threadsafe statistics |
| CPP\_CXA | -fno-use-cxa-get-exception-ptr | Don’t use a CXA get-exception pointer |
| ALL\_SECTIONS | -fdata-sections  -ffunction-sections | Use function and data sections |
| LD\_STARTFILES | -nostartfiles | Don’t use the standard startupfile |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Bmptk services

Bmptk is by design a very spartan environment: it gives you a minimal set of tools and handles to realize your application. This section documents what you get.

### Header

The bmptk.h header makes the bmptk services visible, and includes the vendors chip definition files (cmsis or the equivalent). Alternatively, if you prefer to use the target-specific header files directly, you can do so.

### Functions

For the bare-metal targets bmptk provides a very simple implementation of malloc() that allocates and returns the requested amount of memory from the free store, or returns NULL when it is out of memory. The bmptk free() implementation causes a linker error. The idea is that a small micro-controller application might use malloc to allocate memory in its initialization, but is should never free any memory.

On targets that provide their own heap management (native, esp8266) their malloc() and free() implementations are used.

### Configuration macros

Bmptk defines a number of macros that it needs for its own use. These macros might also be useful for the application. These macros are defined either on the command line (-D directives) or in bmptk.h.

|  |  |
| --- | --- |
| **Define** | **Meaning** |
| BMPTK\_ROM\_START=<address> | The (byte) start address of the ROM (Flash). |
| BMPTK\_ROM\_SIZEW=<size> | The ROM (Flash) size in bytes. |
| BMPTK\_RAM\_START=<address> | The (byte) start address of the RAM. |
| BMPTK\_RAM\_SIZE=<size> | The RAM size in bytes. |
| BMPTK\_STACK\_SIZE=<size> | The STACK size in bytes, or AUTOMATIC |
| BMPTK\_TARGET=<name>  BMPTK\_TARGET\_<name> | The TARGET name as specified by the project makefile. |
| BMPTK\_BOARD=<name>  BMPTK\_BOARD\_<name> | When a board is specified as TARGET in the project Makefile, it is passed in these two defines. When the target is a chip, these two defines are not present. |
| BMPTK\_CHIP=<name>  BMPTK\_CHIP\_<name> | The target chip name. It can be explicitly specified as TARGET in the Makefile, or implicitly by specifying a target board. |
| BMPTK\_XTAL=<value> | When applicable, the frequency of the main crystal in Hz. |
| BMPTK\_BAUDRATE=<value> | When applicable, the baudrate used for the serial download. |
| BMPTK\_VERSION=<value> | The BMPTK version. |
| BMPTK\_HOSTED  BMPTK\_EMBEDDED | Hosted is defined for the native target, embedded is defined for all other targets. |
| BMPTK\_HEAP | Defined for all bare-metal targets. This will cause bmptk to provide a minimal functional malloc(), and a free() that causes a linker error. |

# Targets

## Chips

The next table shows the chips that are supported. These chip names can be used in the TARGET specification in a project Makefile. Chip and board names are in lowercase, start with a letter, and contain only letters, digits, and underscores.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Chip** | **CPU** | **ROM**1 | **RAM**1 | **Stack**1 | **Fmax**  **MHz** | **Package** |
| native | native | - | - | - | - | - |
| atmega328p | AVR 1 | 32k | 2k | 1.5k | 20 | DIP28 (S) |
| msp430g2553 \* | mps430 | 16k | 512 | 384 | 16 | DIP20 |
| msp430fr4133 | msp430 | 15k | 2k | 1.5k | 16 | LQFP64 |
| lpc2148fbd64 | arm7tdmi | 128k | 32k 3 | 24k | 48 | TQFP64 |
| lpc810m021fn8 | Cortex-M0+ | 8k | 1k | 768 | 30 | DIP8 |
| lpc812m101jdh16 | Cortex-M0+ | 16k | 4k | 3k | 30 | TSSOP16 |
| lpc1114fn28 | Cortex-M0 | 32k | 4k | 3k | 50 | DIP28 (W) |
| lpc11c14fbd48 | Cortex-M0 | 32k | 8k | 6k | 50 | LQFP48 |
| stm32f051r8 | Cortex-M0 | 64k | 8k | 6k | 48 | LQFP48 |
| stm32l152rc | Cortex-M3 | 256k | 32k | 24k | 32 | LQFP48 |
| stm32f411re | Cortex-M4F | 512k | 128k | 96k | 100 | LQFP48 |
| mkl25z128vlk4 | Cortex-M0 | 128k | 8k | 6k | 48 | LQFP80 |
| xmc1100 | Cortex-M0 | 64k | 16k | 12k | 32 | VQFN24 |
| atsam3x8e | Cortex-M3 | 512k | 96k | 72k | 84 | LQFP144 |
| esf8266 4 | LX106 | 64k | 16k | ? | 80 | LQFP32 (?) |
| 1 The ROM, RAM and Stack sizes can be overruled in the project Makefile by defining ROM\_SIZE=<size>, RAM\_SIZE=<size> and/or STACK\_SIZE=<size>. The default stack size is ¼ of the RAM. Be aware that defining a ROM or RAM size larger than the available memory can have ‘interesting’ effects. On the targets that support this, STACK\_SIZE can also be defined as AUTOMATIC.  2 The AVR is a strict Harvard architecture, so in deviation from normal C/C++, a pointer to a constant (in ROM) has to be treated differently from a ‘normal’ pointer.  3 This chip has some more RAM, but this is not contiguous with the main RAM.  4 The esf8266 uses an external Flash ROM. There is a limit on the amount of code space from this ROM that it can use, but I am not sure what the limit is. The stack (size) is managed by the resident firmware.  \* Not tested with a real chip. | | | | | | |

## Boards

The next table shows the boards that can be specified as TARGET in a project Makefile. Defining a target board is preferred over defining the chip because it can enable board-specific settings for downloading.[[6]](#footnote-6)

|  |  |  |
| --- | --- | --- |
| **Board** | **Chip** 1 | **Download** |
| arduino\_uno | atmega328P | arduino bootloader, AVRISPmk2 |
| arduino\_nano | atmega328P | arduino bootloader, AVRISPmk2 |
| arduino\_due | atsam3x8e | bossac |
| msp\_exp430g2 \* | msp430g2553 | MSP430Flasher |
| msp\_exp430fr133 | msp430fr4133 | MSP430Flasher |
| hu\_arm\_v4 | lpc2148fbd64 | lpc21isp |
| db103 | lpc1114fn28 | lpc21isp |
| db104 | lpc1114fn28 | lpc21isp |
| db105 | lpc810m021fn8 | lpc21isp |
| lpc800\_mini\_kit | lpc810m021fn8 | lpc21isp |
| lpc800\_max \* | lpc812m101jdh16 | file copy |
| tiny\_11c14 | lpc11c14fbd48 | lpc21isp |
| stm32f0discovery | stm32f051r8 | ST-LINK |
| stm32l1discovery | stm32l152rc | ST-LINK |
| stm32nucleo | stm32nucleo | ST-LINK |
| frdm\_kl25z | mkl25z128vlk4 | file copy |
| xmc\_2go\_1100 | xmc1100 | Not working yet ☹ |
| esp\* | esp8266ex | esptool |

## Adding new targets

The range of targets supported by bmptk is limited by the hardware that I have available and time I can spend on it. Adding a new target that is similar to an existing one can be as simple as adding the appropriate section in the Makefile.inc, possibly a suitable chip header file, and a blink-a-led example project.

For a chip to be added to bmptk the following resources are needed:

1. A chip to test whether it works.
2. A (GCC) toolchain for compiling and building (using command-line calls).
3. The accompanying linkerscript and startup code. (As a second best, the manufacturers implicit linkerscript and startup can be use.)
4. Some means to download an application to the chip from a windows PC, preferably hands-off.
5. Sufficient documentation to write a blink-a-led application.

To support a new chip in bmptk:

1. Add it to Makefile.inc
2. Add the cmsis or equivalent files to a (probably new) targets subdirectory
3. Add or adapt or write the startup code
4. Add and test a blink-a-led example
5. Add it to the chips table
6. Extend the stack size calculator

To support a new board in bmptk:

1. When it is new, add the chip (above steps)
2. Add it to Makefile.inc
3. Add and test a blink-a-led example
4. Add it to the boards table

## Unsuccessful attempts

Boards/chips/toolchains that I considered or investigated but that were rejected or not tested succesfully:

* PSoC : apparently no cmsis files are available, and the CY8CKIT boards require button-push + unplug/plug cycle to start the bootloader.
* Maple (Arduino clone with an STM32F103RB): installing the USB driver on recent Windows versions requires disabling the device driver signature check.
* mbed LPC800 max: nice board, downloads via USB mass storage like the Freescale board. After one seemingly successful download each subsequent download gives a fail.txt containing “SWD ERROR”. Maybe because I re-purposed an SW pin? (And it seems to require a manual reset after downloading.)
* msp\_exp430g2: I could not get the downloader to recognize this board. It kept complaining that the “FRM is in use” or something similar. The other MSP board worked fine, so after a few hours I gave up on this one.
* For avr I tried the toochain that is part of the Arduino installation, but I could not get it to work with an external linkerscript (the linker kept crashing). The toochain provided with AVR studion doesn’t show those problems, so I used that one.

# Internals

This section briefly explains how bmptk works internally. Skip it if you just want to use bmptk.

Bmptk is make-driven, so everything happens because the make executable interprets the Makefile and takes the appropriate steps. A typical Windows developers PC is riddled with make.exe files that (from experience) might not be 100% compatible with the bmptk makefile, so in its tools directory bmptk provides a bmptk-make (and a few other windows executables). The other executables are invoked from the Makefile with their full path name, but that can’t be done for make itself. That’s why you must add this directory to your windows PATH.

The Makefile in your project directories defines a few make macros and then includes the bmptk ‘main’ makefile Makefile.inc, either directly or via a chain of Makefile.inc files. This main Makefile.inc file determines the Windows 32-bit application directory, and then includes Makefile.custom (if it exists), or else Makefile.local, to get the locations of the various external tools.

When your list of source file has one or more files with the .cpp or .cc extension, your project is assumed to be a C++ project and g++ is used for compilation and linking, otherwise gcc is used.

The TARGET value is checked against a list of known boards. If it matches the CHIP macro is set to the boards chip, otherwise it is set to the value of TARGET. In both cases it is not set when it already has a value. Next the CHIP value is checked against the known chips, and various settings are made, appropriate for the chip.

The settings for each chip are organized in a hierarchical way (like and inheritance tree), with the ‘higher’ levels (like embedded or atmel) inherited by the lower levels using $(eval $(…)).

The chip settings include a –I directive for the include directory for the chips header file, and a –D for the header file name. The bmptk.h header uses this –D to include the header file for the chip.

The build target will compile the source files that make up the application to .o files. For most targets, a generalized linkerscript is preprocessed with a few –D arguments (ROM, RAM) to get a linkerscript appropriate for the target chip. The .o files are linked using this linkerscript to an .elf file, which is in most cases further processed to a .hex file and an .srec file.

When the stack size is specified as AUTOMATIC, first intermediate application files (.elfx and .lssx) are generated. The lssx file contains the (dis) assembly of the full application. The bmptk-ssc tool reads this file and from it determines the call tree, the stack used by each function, and from that the total stack size needed. It generates a bmptk\_caculated\_size\_stack.c file that contains a stack of the exact size, which is linked into the real application files (.elf, .hex, etc). For targets with a descending stack, the stack is allocated at the start of RAM, which will cause a memory error when the stack overflows (instead of silently overwriting a part of RAM that is in use for globals or for the heap).

The run target will (when appropriate) check whether the (com) port is available, download the application, start a terminal, and wait for the user to hit the ESC key.

# Bugs, limitations, issues, etc.

1. Stack size calculation has not been tested extensively and works only for avr and cortex.
2. The C tools should be rewritten in python.
3. Most targets, especially msp430 and esp8266, haven’t been tested beyond ‘blink a LED’.
4. Document the ‘make editor files’ tool.
5. Document the make commands for non-application directories.
6. Use on Linux hasn’t been tested recenty.
7. Should use fixed-size stack by default
8. Segment sizes calculator does not work on avr
9. <http://www.hertaville.com/stm32f0discovery-part-1-linux.html> uses embedded arm toolchain
10. Tools on Linux are built with a .exe suffix, to avoid file mode problems on FAT formatted sticks.
11. Mention where the header files come from / what their license is.
12. MSP430 downloading

1. Bare-metal refers to the situation that there is only the application running: it has full control of the hardware, and is running without the support of a separate operating system. Operating system functionality can still be present, but in the form of library code, statically linked into the application. [↑](#footnote-ref-1)
2. It would of course be nice to support debugging, but it is quite complex to do this over a range of targets, and I rarely use a debugger, so I left it out. Feel free to add it and share the result! [↑](#footnote-ref-2)
3. On windows, it is advisable to use the command ‘bmptk-make’ to invoke a bmtk commands Using ‘make’ on Windows will call the first make.exe that appears in your PATH. If you are lucky this is a make that is compatible with bmptk. “Do you feel lucky, punk?” On Linux there is only the make that is installed, which is (on modern Linuxes) most likely compatible with bmptk. [↑](#footnote-ref-3)
4. You can also request an assembler listing for the application $(PROJECT).lss. This file is larger and less readable than individual .lst files, but it more accurately reflects your application (especially when whole-program optimization is used). For embedded targets the .lss file might be generated anyway, because it is used to determine the required stack size. [↑](#footnote-ref-4)
5. A weak (?=) assignment in a makescript assigns only when the target variable doesn’t have a value yet. [↑](#footnote-ref-5)
6. It can also be used by hwcpp to provide interfaces to board-specific hardware. [↑](#footnote-ref-6)