**BMPTK: a minimalist**

**Bare Metal Programming Toolkit**



**Manual**

|  |  |
| --- | --- |
|  | Wouter van Ooijen  wouter@voti.nl  2015  Version 1.0, work-in-progress, last modified 2015-04-15 |

Summary

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

# Table of Contents

[Table of Contents 2](#_Toc416681685)

[1. Introduction 3](#_Toc416681686)

[2. Getting started 4](#_Toc416681687)

[2.1. Bmptk 4](#_Toc416681688)

[2.2. Toolchains 4](#_Toc416681689)

[2.3. Download tools 5](#_Toc416681690)

[2.4. Serial port configuration 6](#_Toc416681691)

[2.5. Tool locations 7](#_Toc416681692)

[2.6. Make 7](#_Toc416681693)

[2.7. Editors 7](#_Toc416681694)

[2.7.1. PSPad 7](#_Toc416681695)

[2.7.2. Sublime Text 9](#_Toc416681696)

[2.7.3. Geany 10](#_Toc416681697)

[2.7.4. Notepad++ 10](#_Toc416681698)

[2.7.5. Programmer's Notepad 12](#_Toc416681699)

[3. Use 16](#_Toc416681700)

[3.1. Functionality 16](#_Toc416681701)

[3.2. Test your setup 16](#_Toc416681702)

[3.3. Your own project 16](#_Toc416681703)

[3.4. Bmptk services 18](#_Toc416681704)

[3.5. Command-line macros 18](#_Toc416681705)

[3.6. Linker information 18](#_Toc416681706)

[3.7. Command line use 18](#_Toc416681707)

[4. Internals 20](#_Toc416681708)

[5. Targets 21](#_Toc416681709)

[5.1. Chips 21](#_Toc416681710)

[5.2. Boards 21](#_Toc416681711)

[5.3. Adding new targets 22](#_Toc416681712)

# Introduction

Bmptk (Bare Metal Programming Toolkit) is a make-based tool for developing bare-metal micro-controller applications on Windows or Linux using assembler, C or C++, using GCC toolchains. The intended audience is micro-controller developers that want to be able to switch easily between different micro-controller targets.

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.

Bmptk does not include a toolchain (compiler, linker etc.). It relies on external GCC toolchains 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 appropriate) communication with the application over a serial line. Debugging is not supported. [[1]](#footnote-1)

Bmptk attempts to enable target-agnostic development: for the user, the build-and-download process is the same for all targets. Of course the code of an application 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) attempts to offer a target-independent interface on which a truly 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 somewhat different (but still free) license; check the license.txt file for the details. What you do with your application that was built by and with bmptk is completely up to you, without any strings attached.

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.

An additional limitation is that 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

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.

## Bmptk

To start using bmptk you must first get bmptk itself and place it somewhere on your PC. It does not 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>, as zip or as a git repository:

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

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 varies with your Windows version. I use Start Button 🡺 Configuration 🡺 System 🡺 Advance System Configuration 🡺 Environment Variables. On Linux there is no need to do this, because the make executable of the distribution is used.

## Toolchains

Bmptk can build for a number of targets. The table below shows the target families. The list of chips and boards is show in the chapter Target (p 21). You need to install (only) the toolchains and download tools for the targets that you are interested in. For Windows the table lists several options, the first one is the default. For Linux the packages to be installed for Fedora are shown. For other Linuxes you will have to find out for yourself which packages are needed.

|  |  |  |
| --- | --- | --- |
| **Target family** | **Toolchain Windows/Linux** | **Can be downloaded from** |
| native | MinGW | * <http://gnutoolchains.com/mingw64/> * <http://nuwen.net/mingw.html> |
| GCC / G++ | Fedora:  sudo yum install gcc gcc-c++ |
| ATmega | AVR Studio, Arduino | * <http://www.atmel.com/microsite/atmel_studio6/> * <http://arduino.cc/> |
| avr-gcc, avr-libc, avrdude | Fedora:  sudo yum install uisp avr-libc avr-gcc-c++ rxtx avrdude |
| MSP430 | CodeComposer | * <http://www.ti.com/tool/ccstudio-msp430> |
| GCC | Fedora:  sudo yum install msp430-gcc msp430-libc |
| ARM & Cortex: LPC, STM32, MKL25Z, XMC,  ATSAM3 | GCC-ARM | * <https://launchpad.net/gcc-arm-embedded> * <http://gnutoolchains.com/arm-eabi/> * <http://www.freddiechopin.info/en/download/category/11-bleeding-edge-toolchain> |
| GCC | Fedora:  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 |
| ESP8266 | GCC (and more) | Unofficial Espressif Windows SDK bundle from  https://github.com/boseji/ESP8266-uof-windows-sdk-portable |
|  | GCC | Not supported yet |

The native targets are 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. On Fedora this is mandatory, because some bmptk tools are provided as source, which must be compiled with GCC.

For ATmega chips on Windows either the toolchain provided by the manufacturer as part of the AVR studio development environment can be used, or the toolchain that is part of the Arduino environment. AVR Studio is a big install, with some (trivial) user interaction, but currently (January 2015) it provides a much more recent GCC than the Arduino installation. (There is also WinAVR project, but the last update was in 2010, so I did not try its.) On Fedora the avr packages must be installed.

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.

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. 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. There doesn’t seem to be a free set of header files for pre-Cortex ARM chips. On Fedora the arm-none-eabi packages must be installed.

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).

## Download tools

A download tool is used to downloads an application to your target chip. The download tool you use will depend on the target chip, and often on the target board (which can include supporting hardware for downloading). You need to install only the download tool(s) that you need for your situation.

|  |  |  |
| --- | --- | --- |
| **Target family** | **Download tool**  **Windows/Linux** | **Can be downloaded from** |
| Native | - | - |
| ATmega | AVRDUDE with AVRISP mk11 or Arduino bootloader | bmptk built-in |
|  | Not supported yet |
| MSP430 | Launchpad driver +  MSP430Flasher | <http://www.ti.com/tool/MSP-EXP430G2> + <http://processors.wiki.ti.com/index.php/MSP430_Flasher_-_Command_Line_Programmer> |
|  | <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 |
| LPC | lpc21isp | the executable is in bmptk/tools/lpc21isp\_197 |
| lpc21isp | the source is in bmptk/tools/lpc21isp\_197 |
| STM32 | ST-LINK Utility | <http://www.st.com/web/en/catalog/tools/PF258168> |
|  | Not supported yet |
| MKL25Z | copy\_to\_drive | bmptk built-in |
|  | Not supported yet |
| XMC | JLinkExe | <http://www.segger.com/jlink-software.html> |
|  | Not supported yet |
| ATSAM3X8E | bossac, using the loader interface of the Arduino Due | bmptk built-in |
|  | Not supported yet |
| ESP8266 | esptool | part of the SDK bundle |
|  |  | Not supported yet |

One obvious advantage of the native target is that you don’t need a separate download tool ☺.

The bmptk distribution contains the following download tools:

* lpc21isp (source, and windows executable)
* copy\_to\_volume (source, and windows executable)
* bossac (windows executable only)
* avrdude (windows executable only)

Lpc21isp is a serial-port downloader, used for lpc target chips. It is slightly modified from the normal 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.

Copy\_to\_volume copies the application to a windows drive that is identified by its drive letter or (preferably) by its volume name. This is used for targets that present itself as a USB mass storage device. (Most boards that support OpenSDA can do this, and some LPC chips have a built-in bootloader that supports this.)

For ATmega chips AVRDUDE is used, either with the Arduino boot-loader (if it is already present on the chip), or with AVRDUDE and the AVRISP mkII programming hardware. VRDUDE is part of bmptk.

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.

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

For downloading to an ESP8266 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 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 installatioin-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.)

## Tool locations

Unless instructed otherwise bmptk assumes that you use the default tools, and they are 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 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 WPF macro (defined by the Makefile.inc) to point to the windows directory for 32-bit applications, so you don’t have to spell this out.

## Make

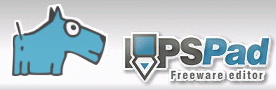
Bmptk uses the make tool to run make scripts. Unfortunately, the script language use by various make versions is not 100% compatible. Various applications tend to install their own make executables, and add their location to the PATH. The result is that execution a ‘make’ command might invoke a make 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. For this reason, you must add this directory to your PATH.

## 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. 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 \_project.ppr file so they can be used 'out of the box' with PSPad.

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).

### PSPad



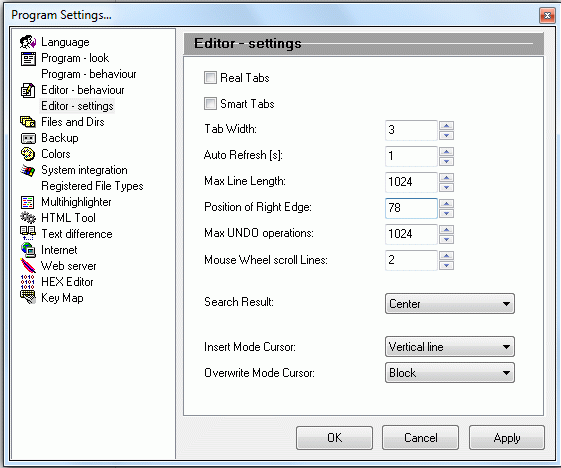
PSPad (http://www.pspad.com) is a free code editor for Windows. It can be configured to work with bmptk by including a few lines in the .ppr file to attach external commands to two buttons. The example directories each contain such a \_Project.ppr file.

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".

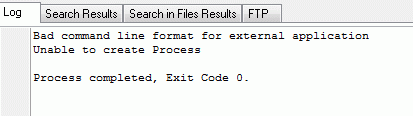
This button is less useful for building the application because the command output is not captured by PSPad. For these buttons and commands to work it is required that the active file is one of the source files (.h or .cpp) in the working directory. If another type of file is open the buttons will be 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 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 does not.
* 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

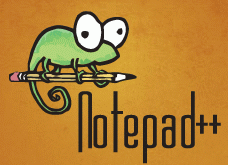


Geany (http://www.geany.org) is a free code editor that is available for a number of platforms. It can call an external makefile. You can use Build => Make (Shift-F9) to build the project in the directory of the currently active file. To run or clean you can use Build => Make Custom Target (Shift-Control-F9) 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.

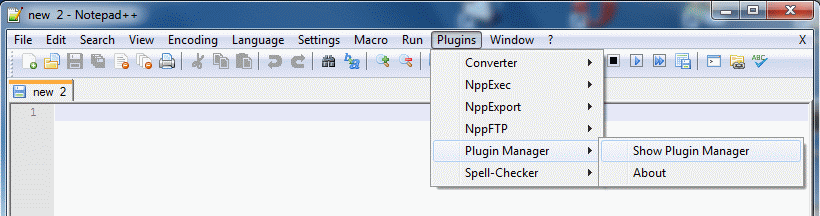
By default Geany does NOT save the edited files before starting an external command, which is a bit counter-intuitive. I suppose this behavior can be changed, but I have not yet found how.

### Notepad++

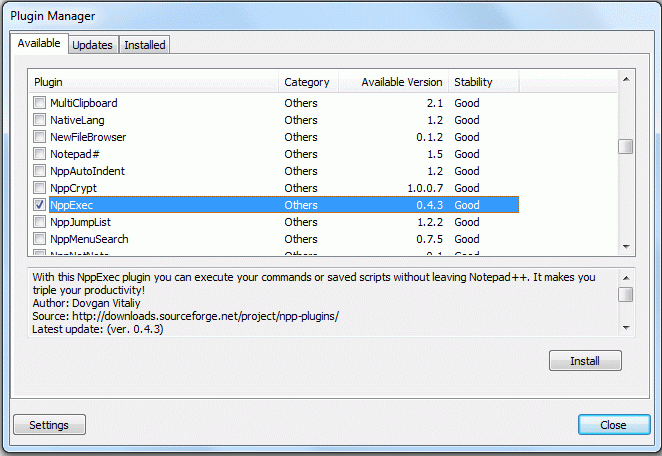


Notepad++ (<http://notepad-plus-plus.org>) is a free lightweight text editor for Windows. With the NppExec plugin Notepad++ can call do the basic chores of an IDE: call an external tool, capture the output, and jump to the source line that corresponds to an error message.

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



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

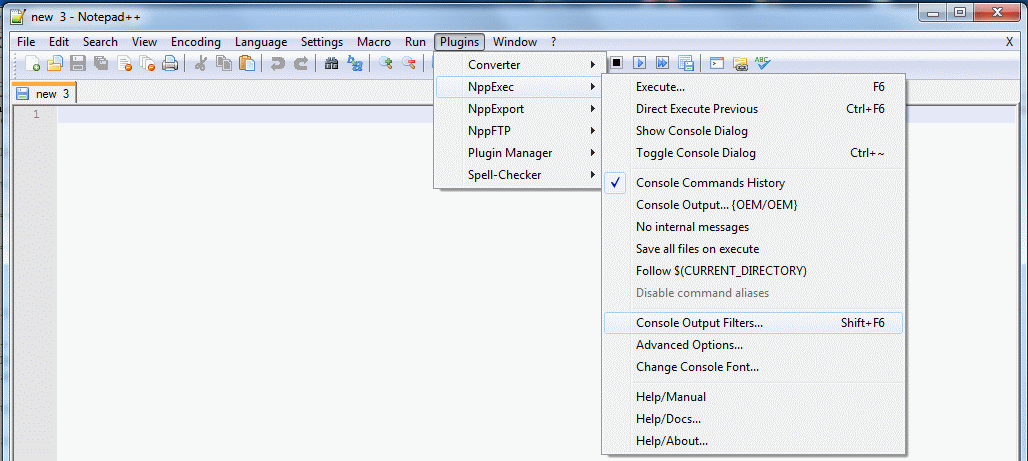


Now F6 will open a window in which you can type the command you want to be executed:

|  |
| --- |
| 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'.

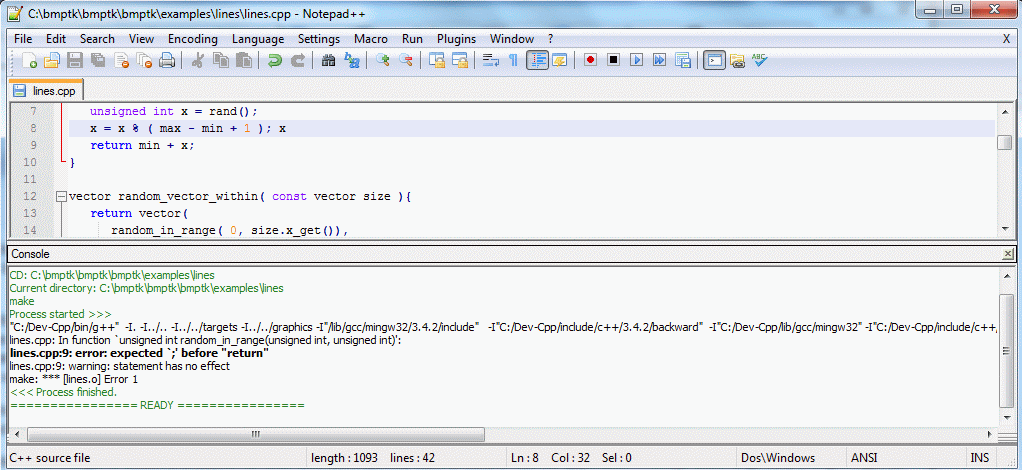
The next step is to get the NppExec to recognize the source references in the output. Open Plugins => NppExec => Console Output Filters.



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'.

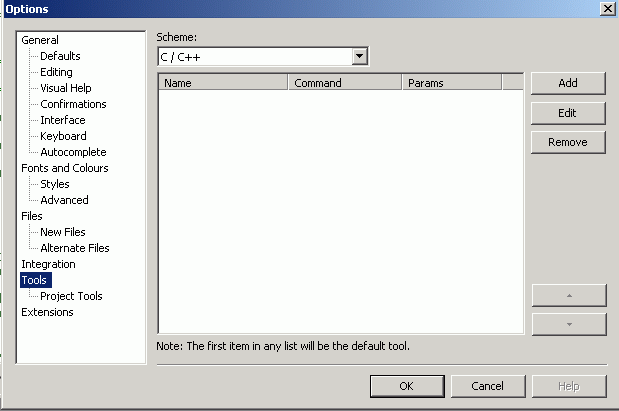
More information about the command language can be found at <http://sourceforge.net/apps/mediawiki/notepad-plus/index.php?title=Compiling_Source_Code>

### Programmer's Notepad

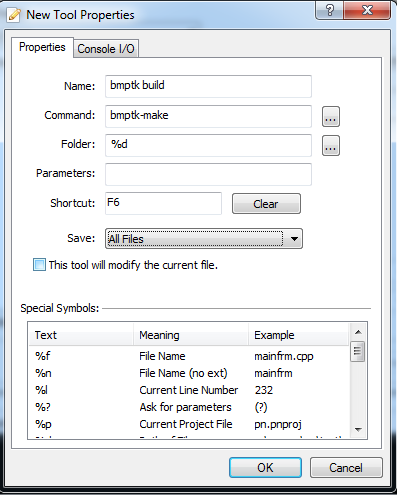


Programmer's Notepad (http://www.pnotepad.org) is another free lightweight text editor for Windows, specifically designed for editing source code. It is available as source, but I haven’t seen any ports to other OSes. With a little customization PN can call do the basic chores of an IDE: call an external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

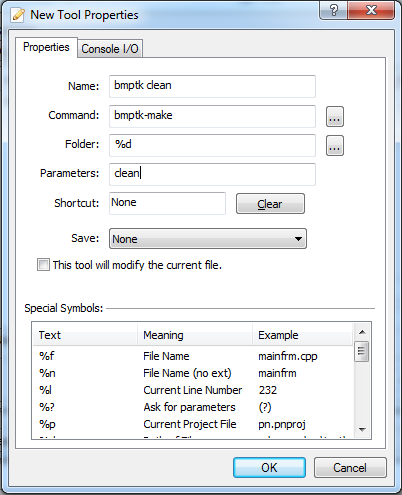
Open a C++ file, and select Tools => Options => Tools:



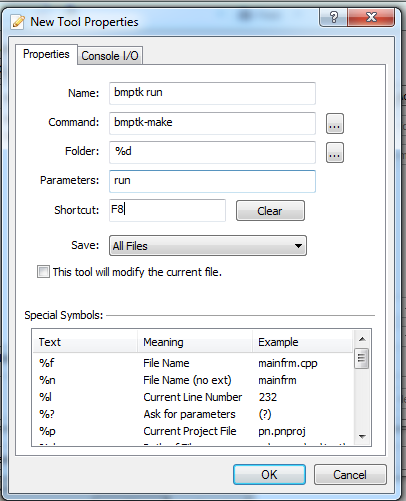
Click Add. Create an entry for ‘bmptk-make build':



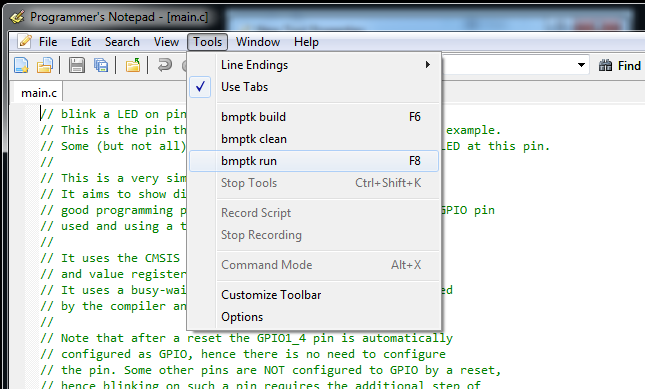
You can leave the Console I/O tab unchanged. The shortcut key can be configured by selected the field and pressing the key (or combination) you prefer. I choose F6 for building, F8 for running, no shortcut for cleaning. Likewise make the entry for cleaning:



And for running:



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!

# Use

## Functionality

On windows, use the command ‘bmptk-make’ to invoke a bmtk command.[[2]](#footnote-2) On Linux you can simply use ‘build’. The bmptk build command (Windows: ‘bmptk-make build’, Linux: ‘make build’) builds (compiles and links) your project. The run command downloads your application to the target, starts it, and (when possible) starts a terminal to communicate with it.

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 assumed 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 small a real-time embedded application.

For C++, the linkerscript generates an error when your application has global ctors (global objects that require run-time initialization). 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 it’s 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.

## Test 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 \_project.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 sometimes closes before you can read what has happened. In that case you should use the command line to do a “bmptk-make build” or “bmptk-make run” in the example directory.

## Your own project

To start a new bmptk project you can either copy an example, or copy the bmptk/Makefile.template to your directory. The Makefile contains the instructions for the make tool to build your project. The Makefile in your project directory contains only the project-specific aspects. As its last line it includes the bmptk/Makfile.inc, which does the real work.

An assignment you write in a Makefile can be overriding (:=) or appending (+=). In the main bmptk Makefile.inc most assignments use the ?= operator, which assigns only when the variable doesn’t yet have a value, so a user-assigned value is left in place.

The makefile must know where to find the bmptk files. The example directories are three directories deep in the bmptk tree, so the Makefiles in the examples have the line

|  |
| --- |
| # specify where the bmptk files can be found  BMPTK := ../../../.. |

If you place your new project somewhere within the bmptk tree you might have to change the number of ../'s. If you place a bmptk project outside the bmptk tree you should probably specify an absolute path, like ”C:/bmtk”.

The Makefile specifies 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. 21).

|  |
| --- |
| # Specify target chip or board or system, for instance  # native, db103, or lpc1114fn28  TARGET := |

|  |
| --- |
| BMPTK := C:/bmptk |

By default the Makefile assumes that main.cpp (or main.c, or main.asm) 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):

|  |
| --- |
| 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 += operator you can have multiple lines that add sources and headers.

|  |
| --- |
| # Specify project-specific files (other than the main, 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.

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, or an assembler listing for the application (.lss). The latter is larger and less readable, but it more accurately reflects your application (especially when whole-program optimization is used).

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

The default amount of RAM reserved for the stack depends on the target. The values are shown in the Targets table (p. 21), in most cases it is ¼ of the total RAM. You can change this by setting the STACK\_SIZE parameter.

|  |
| --- |
| # specify the stack size (default depends on the target)  # STACK\_SIZE := |

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

If you want to change 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 project's Makefile.

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

## Bmptk services

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

### Header

The bmptk.h header makes the bmptk services visible, and includes the vendors chip definition files (cmsis etc.).

### 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. 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.

### Macros

Bmptk defines a number of macros that it needs for its own use. These macros might alos be usefull 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).1 |
| BMPTK\_ROM\_SIZEW=<size> | The ROM (Flash) size in bytes. 1 |
| BMPTK\_RAM\_START=<address> | The (byte) start address of the RAM. 1 |
| BMPTK\_RAM\_SIZE=<size> | The RAM size in bytes. 1 |
| BMPTK\_STACK\_SIZE=<size> | The stack size in bytes. 1 |
| 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 passed. |
| 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. |
| 1 The addresses and sizes are in a format that is acceptable for the linker, which can include a ‘k’ suffix, meaning \*1024. | |

## Command line use

Bmptk is 'driven' by its makefile(s). In most cases you will invoke the Makefile through your GUI, but sometimes it is handy to invoke it directly. The make commands that are recognized are shown in the next table. (On windows, bmptk-make should be used, on Linux make is OK.)

|  |  |
| --- | --- |
| **Command** | **Effect** |
| [bmptk-]make build | Build the application (compile, link, etc.) |
| [bmptk-]make run | Download the application (if needed) and run it (If necessary, build it first) |
| [bmptk-]make clean | Delete all files generated by building the application |

# 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. This file determines the Windows 32-bit application directory, and then includes Makefile.custom (if it exists), or else Makefile.local.

When your list of source file has files with the .cpp 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 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 and stack size) to get a linkerscript appropriate for the target chip. The .o files are linked using this linkerscript to a .elf file, which is in most cases further processed to a .hex file and a .srec file.

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 return key.

# 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** |
| atmega328p | AVR | 32k | 2k | 512 | 20 | DIP28 (S) |
| msp430g2553 \* | mps430 | 16k | 512 | 128 | 16 | DIP20 |
| msp430fr4133 | msp430 | 15k | 2k | 256 | 16 | LQFP64 |
| lpc2148fbd64 | arm7tdmi | 128k | 32k2 | 4k | 48 | TQFP64 |
| lpc810m021fn8 | Cortex-M0+ | 8k | 1k | 256 | 30 | DIP8 |
| lpc812m101jdh16 | Cortex-M0+ | 16k | 4k | 1k | 30 | TSSOP16 |
| lpc1114fn28 | Cortex-M0 | 32k | 4k | 1k | 50 | DIP28 (W) |
| lpc11c14fbd48 | Cortex-M0 | 32k | 8k | 2k | 50 | LQFP48 |
| stm32f051r8 | Cortex-M0 | 64k | 8k | 2k | 48 | LQFP48 |
| stm32l152rc | Cortex-M3 | 256k | 32k | 8k | 32 | LQFP48 |
| stm32f411re | Cortex-M4F | 512k | 128k | 32k | 100 | LQFP48 |
| mkl25z128vlk4 | Cortex-M0 | 128k | 16k | 4K | 48 | LQFP80 |
| xmc1100 | Cortex-M0 | 64k | 16k | 4k | 32 | VQFN24 |
| atsam3x8e | Cortex-M3 | 512k | 96k | 8k | 84 | LQFP144 |
| esf8266 | LX106 | 64k3 | 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. A size must be in a format that is acceptable for the linker, which can include a ‘k’ suffix, meaning \*1024.  2 This chip has some more RAM, but this is not contiguous with the main RAM.  3 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.  \* 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 in bmptk, and it can be used by hwcpp to provide interfaces to board-specific hardware.

|  |  |  |
| --- | --- | --- |
| **Board** | **Chip** 1 | **Download** |
| arduino\_uno | 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. Some means to download an application to the chip from a windows PC, preferably hands-off.
3. Sufficient documentation to write a blink-a-led application, built with an normal GCC, and my own linkerscript and startup (as a second best, the manufacturers toolchain can be used, but strictly in command-line mode).

To support a new chip in bmptk:

1. Add it to Makefile.inc
2. Add the cmsis or equivalent files to a targets subdirectory
3. Add or adapt or write the startup code
4. Document it in the chips table
5. Add and test a blink-a-led example

To support a new board in bmptk:

1. When it is new, add the chip (above steps)
2. Add it to Makefile.inc
3. Document it in the boards table
4. If appropriate, document

To support a new chip in hwcpp:

1. Add it to bmptk
2. Add it to hwcpp.hpp
3. Write a hpp file for it in targets
4. Document it in hwcpp.docx

To support a new board in hwcpp:

1. Add its chip to hwcpp
2. Add it to hwcpp.hpp
3. Write a hpp file for it in boards
4. Document it in hwcpp.docx

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

* PSoC : apparently no cmsis files 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.

1. 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-1)
2. 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-2)