# The BLDEV Build System

The build system described by this document is designed for C++ development, for use by programmers familiar with make, specially gnu make. The build system is intended to fill a niche in the development community: *A minimalist cross-platform build system for which only Makefiles and simple utilities are needed.*

The document, BUILDING.md [10], has BLDEV installation details.

CMake [4] is very popular, produces a hierarchical build system, and possesses a broad ability to "pull in" necessary packages, automatically. A drawback of CMake is the layer of a cmake scripting language, between the user and the Makefiles that are generated. Bazel [8] is a build system, similar to cmake, and incorporates testing. See [9] for a comparison of bazel, cmake and make.

The GNU atuotools [6] produces a hierarchy of Makefiles, while also including powerful automated configurability for platforms and packages. However, autotools is not well-suited to Windows, is quite complex, and has files with special configuration syntax.

The build system of the current document, BLDEV, is only as complex as the Makefiles that the programmer crafts, including some make rules that are part of the build system delivery. These make files, however, may be completely discarded, and new ones written. The programmer retains "full control".

BLDEV has no "auto-find" for packages, and does not explicitly support testing. With BLDEV, the user must specify where to find header files and libraries, and which libraries to use on the link line. There are Make macros to help with this, and to encapsulate such definitions. As such, BLDEV is particularly useful if you do not want to learn another language, and are eager to acquire a facility with make files. ☺

There is no gui, although a gui would somewhat ease the initialization of new projects within the build system. A functioning build system consists of a folder hierarchy with a Makefile in each folder. There are a few simple utilities. There are also script files (Bourne shell for linux, and Batch files for Windows), which the user can write to set the run-time environment for each project. The build system is, thus, available to be extended to other platforms, and requires nothing more than a facility in using make files, and elementary scripting in the native scripts of the platform.

In addition to describing the build system, this document also outlines some utilities that are useful in any development environment, which were constructed during the development of BLDEV, and are delivered with the build system:

- A utility to scan C++ source code, and create C++ source code for reading/writing enums and simple structs/classes to/from disk. (iogen §5.4)
- A utility to scan C++ source code, and convert a set of preprocessor macros to enum values (meconv §5.5)
- A wrapper infrastructure for incorporating external logging libraries (Logging §6)

# 1 Introduction

A build system enables a software engineer to write source code and to convert the source code into programs/deliverables, while managing the complexities of the various goals and requirements faced by the developer:

1. Platforms[1] e.g. Windows and Linux
2. Flavours e.g. Debug and Release
3. Projects e.g. Different subsets of the source code

The build systems described in this document have the design goals:

- Use build tools, such as compilers and linkers, in conjunction with Makefiles. An understanding of Makefiles gives a full understanding of the structure of the build system.
- Multiple developers can access the same copy of the source code, by using different build folders. Build folders can be on different machines, for example, by attaching a Windows drive letter to a linux source folder, if the source code is on a linux machine.

The build system is independent of the software repository being used to manage the source code. In this context, the build system only ever sees a working copy (checked-out version) of the source code.

In this document, all the action happens on the development machine. The source machine, where the source code lies, may be different from the development machine, but otherwise plays no role. The setting of environment variables and the running of utilities, all happen on the development machine.

# 1.1 Overview

The build system is comprised of various tools:

- C++ build utilities
- Development windows – Xterm (linux), and Command prompt (Windows), with properly initialized environments.
- A few native scripts – Bourne shell (linux), and Batch files (Windows), which are used to initialize the development windows' environments.

Since the build utilities are C++, a bootstrap process is needed, to achieve a functioning build system. On any one machine, it is not necessary to build different flavours of these C++ build utilities. Consequently, the system uses an environment variable, PLATBIN, pointing to a folder, to hold build utilities that are created during the bootstrap process. The PLATBIN variable should be part of the PATH variable. This is most easily accomplished in the login script (linux), or Environment Variables interface (Windows). For subsequent build systems on this machine (different flavours and projects), the bootstrap process is not needed. Of course, once the build system is set up, the build utilities, themselves, may be modified/rebuilt, as needed.

The source code, which the build system works on, is assumed to lie in a hierarchy of folders, as shown in Figure 1. Different build environments on the same machine are in separate xterms (linux) or command terminals (windows), with their appropriate definitions of DEVTOP and run-time initialization.

1 Linux (fedora 30 ) and Windows (Windows 10) on an x86-64 machine, have been tested with BLDEV.

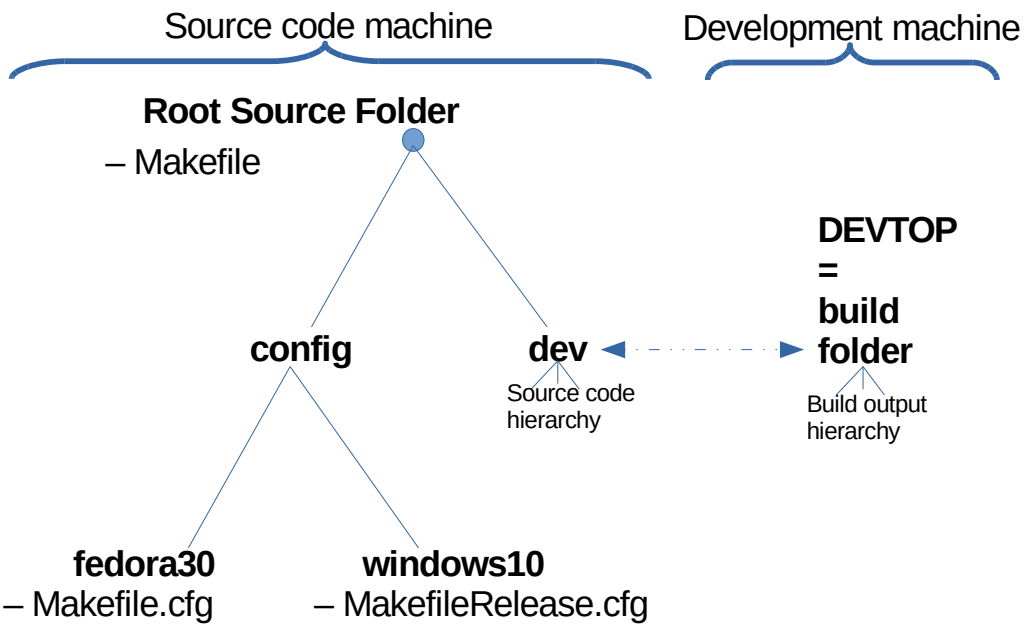


Figure 1: Source tree and Build tree

The Makefile in the Root Source Folder is used to bootstrap the basic development utilities. The DEVTOP environment variable points to the build folder, which is the root of the build tree, and parallels the **dev** folder in the source hierarchy. The config folder contains files for constructing Makefiles in the build hierarchy, and scripts for initializing the environment of command line interfaces. The development machine needs to be able to "cd" to the source hierarchy.

The following section describes the boot process.

# 2 Bootstrapping

This section explains how to build the first build system on a platform. Subsequent build systems, for different flavours and projects, can be constructed directly (§3), without a bootstrapping process, since the bootstrapping process creates the following utilities:

*Essential*

- **bldev** §5.1– Hierarchically build a development tree, paralleling a source tree.

*Recommended*

- **setdev** §5.2– Create scripts that create and initialize Windows command terminals and Linux xterms.
- **setpalette** §5.3– Built on Windows, only, to set command terminals' colours.

*Optional – source code generators*

- **iogen** §5.4– Create C++ insertion/extraction source code for C++ enums and "bare" structs/classes.
- **meconv** §5.5– Convert a set of C++ preprocessor Macros to C++ Enum values.
- **appendvk** §5.6– Used in Windows, to augment meconv for using keyboard codes.

Only bldev is essential for managing the build system. Setdev (and setpalette on Windows) are recommended, as they make convenient the separation of different build environments. *Iogen* may be used to generate IO source code, including header files, so it is needed only if the user is employing such generated IO source code. Similarly, *meconv* and *appendvk* are needed only if the user wishes to convert some macros to enumerations e.g. keyboard codes.

These utilities all have help screens. Other than for setpalette, the command line flags *-h* or *–help* bring up the help screen. These utilities are more fully described in §5.

There are dependencies for building the above utilities:

- Boost libraries [2]– *file system, log*
- CLI11 library [7]– Command line parameter processing
- MSys2 [3] (Windows only) – GNU make, g++, Bourne shell and utils
- Clang/LLVM libraries [1]– *C++ parser*

Please note that the Clang/LLVM libraries are not needed outside of the bootstrap process – they are used only for building the *iogen* utility. If you do not want to build iogen, Clang/LLVM is not needed – you will be able to fully manage a build environment, but not build all the software that is part of the BLDEV package.

A caveat is in order, with regards to the two source code generators, *iogen* and *meconv*. With complex source code, involving default compiler header files and default compiler macros, it is possible for the two source code generators to create source code that is inconsistent with that built by the compiler of your development environment. *Iogen* uses *clang/llvm* routines to parse source code, and *meconv* uses the *gcc* compiler. To date, this has not been a problem, but these two routines could be upgraded to be more careful in this regard.

The following sections describe the bootstrapping steps.

# 2.1 Platform Bin Folder

Set the environment variable PLATBIN to a folder, which will contain various utilities and scripts. This folder should be added to the PATH environment variable. It is suggested that PLATBIN be set up as a normal part of the logging in procedure.

# 2.2 Configuration Files

In the *config* folder, create a new platform folder, using one of the existing ones (fedora30 or windows10) as a template. Suppose your platform folder is called, MyMachine . There are up to 4 files to edit in MyMachine:

1. Makefile.cfg
2. Platform.inc
3. platformInit.bat (using .bat extension to indicate a script file – not needed on linux)
4. projectInit_TAG.bat (if platformInit does not apply to all your projects)

As explained in §4, the TAG corresponds to the development window that you will be creating, §2.4. You may not need either of these two Init scripts. An init script that is not needed, need not be present in the platform folder.

The Makefile.cfg is typically quite short – e.g. debug and optimization flags, and perhaps library names specific to these flags.

Many of the variables in Platform.inc will need to be edited. Any flags that you are not using, you can set to empty e.g. those involving QT or QWT . More details are in the associated docuemtn, BUILDING.md [10].

Unless you are creating a platform other than linux or windows, you will likely not need to edit ../Makefile.inc, which contains the rules by which to invoke compilers, linkers and other tools.

### Logging

The build system currently supports two logging options – Simple and Boost. Logging is discussed in more detail in §6. The source code for Simple logging is part of the build system source code, so setting *SSRD_LOGGING=Simple* in the Platform.inc configuration file, will reduce the risk of complications in the bootstrap process.

## 2.3 Boot

cd to the top level of the source tree. This folder contains the files: Makefile and Makefile_boot. The command, *make help*, prints a help screen. Issue the command (from the development machine):

*make CONFIG=configFile DEVTOP=devTop boot*

For Windows 10 Release, the invocation could be

*make CONFIG=config\windows10\MakefileRelease.cfg DEVTOP=g:\devTEST boot*

Where, g:\devTest is some location on the Windows 10 machine. This location will be created, if it does not already exist.

This creates the two utilities, bldev and setdev (and setpalette, on Windows), and moves them into the PLATBIN folder. Also, the entire development tree is created, rooted at devTop.

### defs.inc

In every folder of the development tree, there is a file, defs.inc . The defs.inc in devTop looks like this:

*SOURCE_DIR =q:/dev*
*CONFIG_PATH =q:/config/windows10/MakefileRelease.cfg*

SOURCE_DIR is the source folder corresponding to the current development folder. CONFIG_PATH is the configFile specified on the make command line.

In this example, q:\ is the Windows path to the top source folder. Both SOURCE_DIR and CONFIG_PATH have been converted to absolute paths, and back slashes have been changed to forward slashes. The reason for the forward slashes, is that the Makefile uses the Bourne shell to execute commands, and the Bourne shell interprets back slashes as escape characters.

## 2.4 Development Window

To create a development window, the devTop folder must already have been initialized with its defs.inc file, e.g. as in §2.3. Issue the command:

setdev TAG -d devTop -f --bg rrbbgg –fg rrbbgg

where,

- TAG is a short string of your choice, to identify this development environment
- --bg and –fg identify the background and foreground colours of the development window.
- r, b and g are single digit hex values, 0..F, upper case or lower case.

This creates a script, setdev_TAG, that is stored in PLATBIN, and then runs the script. This brings up a development window (linux xterm or Windows command line), whose window title is, TAG, the background colour is a specified, with black type. Amongst other things, the script sets the environment variable, DEVTOP.

If you are not happy with your first choice of colours, kill the window and try again. The -f flag allows an existing setdev_TAG script to be overwritten. The script may subsequently be invoked in two ways:

- setdev_TAG
- setdev TAG

Please note that the setdev utility may not be invoked from a development window, as this will unnecessarily complicate the PATH variable, and may corrupt the development environment. This is guarded against by verifying that the DEVTOP environment variable does not already exist.

## 2.5 Init (optional)

This step is for building iogen (requires Clang/LLVM libraries), meconv and appendvk.

While in a development window, cd to devTop, and issue the command

make init

This moves these built utilities to PLATBIN.

# 3 Build Systems

Once the first build system is created, subsequent build systems are created as follows:

As in §2.3, cd to the top level of the source tree, and issue the command:

make CONFIG=configFile DEVTOP=devTop dev

Please note that the target is “dev”, not “boot”. This creates the development tree, rooted at devTop.

Lastly, create a development window, as in §2.4.

### Note

As an alternative to the above make command, one can issue the following command:

bldev --cfg configFile devTop topLevelSource/dev

This is the command that the above make command uses.

# 3.1 Usage

There are various make commands recognized by the build system, the purpose being to build software products and deliverables. To clean out and rebuild, issue the following command

*make clean incinst libs bins bininst*

.

You might do this, for example, if you change the logging option (§6). Each command is described, in turn:

- **clean**

This command removes all of the outputs of the build system (but not, of course, those that are stored in PLATBIN). Also, the contents of devTop/bin are left intact.

- **incinst**

Most header files are scattered in the source tree. Each Makefile can specify a list of header files to "link" from a common folder. For the header file, foo.h, the "link" consists of writing a 1-line file into the common folder, of the form

#include "full path to foo.h"

Some header files are created (in the build folder) e.g. through the use of iogen, §5.4 . Such header files are simply moved to the common folder.

- **libs**

Each Makefile can specify a library to build/contribute to, and which source files participate. The library is built in-place, in its destination location. The library is not first built locally and moved/copied to its destination.

- **bins**

Each Makefile can specify none, one or more executables to build. Executables are built locally.

- **bininst**

Each Makefile specifies where to install the executable(s) it has created. The executables are moved to the specified location.

## *3.1.1 Windows*

To create a GUI binary, rather than a command line binary, use the command

make SUBSYSTEM_WINDOWS=1 bins

Of course, your code would need the WinMain() entry point.

# 4 Configuration

Configuration occurs at two points in the build system:

- When creating/updating a build tree, §2.3 and §3 – Makefile configuration
- When creating a development window §2.4 – Environment configuration

## 4.1 Makefile configuration

Makefile configuration consists of different pieces, encapsulated in four make files, listed in the order in which they appear in the resulting Makefile:

1. *<Flavour>.cfg* – e.g. compiler flags: debug, release, dynamic, static. This is the configuration file specified on the command lines in §2.3 and §3. The config/windows10 folder has a Makefile.cfg and a MakefileRelease.cfg.
2. *Platform.inc* – e.g. tool locations: explicit path to Bourne shell. Must be in same folder as *<Flavour>.cfg ,*
3. *Makefile.in* – Each folder of the development *source* hierarchy may contain a Makefile.in, which defines what work is to be done in that folder. In this case, the bldev utility creates a Makefile in the corresponding *build* folder.
4. *Makefile.inc* – Makefile rules. Must be in the parent folder of *<Flavour>.cfg .*

The *Makefile.in* files are discussed in more detail in §7.

## 4.2 Environment configuration

Environment configuration may be required for using tools and libraries e.g. to access necessary run-time shared objects. The setdev_TAG script (§2.4) sets up the environment. It always sets the DEVTOP environment variable, and adds DEVTOP/bin to the PATH variable.

Additionally,

If there is a *projectInit_TAG* script next to the Makefile project configuration file, it is used. On Windows, the *.bat* extension is used.

Otherwise, if there is a *platformInit* script, it is used.

Otherwise, no additional environment configuration is done by the *setdev_TAG* script.

### Example

The setpalette utility, §5.3, is in dev/src/boot/setdev/Windows, where the Makefile.in can be reduced to:

```
BIN_INST_DIR =$(PLATBIN)
BINARIES_NAMES = setpalette
```

After running bldev, the Makefile starts off with:

SOURCE_DIR := q:/dev/src/boot/setdev/Windows

The SOURCE_DIR is where the Makefile.in is located. This is followed by the contents of the configuration file:

```
CXX_DBG_FLAGS = -Zi -MDd
LD_DBG_FLAGS = /DEBUG:FULL
BOOST_LIB_FILESYSTEM = libboost_filesystem-vc142-mt-gd-x64-1_72
```

This is followed by the inclusion of the *Platform* file, the contents of the *Makefile.in* and the inclusion of the rules file, *conf/Makefile.inc*.

# 5 Utilities

This section describes the build utilities that are listed in §2.

## 5.1 bldev

bldev is the main tool for creating and updating a build system.

```
Usage: bldev [OPTIONS] [destDir] [srcDir]

Positionals:
  destDir TEXT=.            Must be specified if srcDir is specified
  srcDir TEXT               Can only be specified if destDir is specified

Options:
  -h,--help                 Print this help message and exit
  --nr                      Don't do recursion
  --cfg confiFile           Configuration file.
  --ign                     Ignore bad defs.inc.
 Does two things in DestDir(and subdirs, if recursing):
 -- Creates/updates defs.inc
 -- Assembles a Makefile, if a Makefile.in is present.
 defs.inc has the two lines:
    SOURCE_DIR=<full path source directory>
    CONFIG_PATH=<full path configuration file>
 Ignoring a bad defs.inc is relevant only at the top dir.
 DestDir defaults to the execution directory.
 If configFile is not specified, it is taken from
 defs.inc in DestDir or in the execution directory.
 If SrcDir is not specified, it is set to
 SOURCE_DIR from DestDir/defs.inc, if it exists.
 If this file does not exist, SrcDir is set to DestDir.
 SrcDir is relative to SOURCE_DIR in execDir/defs.inc,
 if it exists, otherwise, SrcDir is relative to
 execDir.
 During recursion, at all directories except the top,
 DestDir, SrcDir and ConfigDir are always specified,
```

so the rules for defaulting are relevant only at
the first (top) directory being configured.
A Makefile is assembled from these parts:
SOURCE_DIR := <full path to source directory>
MAKEFILE_INC := <full path to the included generic makefile>
PLATFORM_INC := <full path to the included platform makefile>
Contents of Makefile.in
Contents of Makefile.cfg

Once a development tree has been constructed, the bldev utiltiy can usually be run in that tree, without any parameters at all, when the tree needs to be refreshed, or expanded to reflect an expanded source tree. This is so, due to the presence of the defs.inc files.

### Example 1

After editing Makfile.in in the current (source) folder, the current build folder may be refreshed with the command *bldev –nr* . The “--nr” flag tells bldev not to work recursively i.e. to refresh only the current folder. In practice, it is usually easier to omit the “--nr” flag, and just let *bldev* work recursively.

### Example 2

The entire build tree may be refreshed with the command *bldev $DEVTOP* , issued from any location (using unix notation). Alternatively, first *cd $DEVTOP* , then issue the command, *bldev* .

## 5.2 setdev

The setdev utility creates development windows with custom colours for the background and text. This is handy especially when multiple development environments are active on a screen, to distinguish them by each being in a separate xterm (linux) or command window (Wihdows) with their own titles and colours.

```
Usage: setdev [OPTIONS] tag

Positionals:
  tag TEXT REQUIRED          Short name for the development environment

Options:
  -h,--help                  Print this help message and exit
  -d,--devTop TEXT           Name of top development directory
  -f,--force                 OK to overwrite script if it exists
  -n,--noexec                Don't execute script after creation


Colours:
  --bg TEXT=FFFFFF Needs: --devTop
                             Colour of the command window background
  --fg TEXT=000000   Needs: --devTop
                              Color of the command window foreground
```

The PLATBIN environment variable must be defined.
It is a folder to hold the initialization scripts.
The PLATBIN folder should be in the PATH variable.

The DEVTOP environmnet variable must NOT be defined i.e.
development scripts/environments must be created/invoked from a
non-development environment.

If devTop or colour is specified, the corresponding script is
assumed not to exist, and it is created:
    $PLATBIN/setdev_tag
This script is executable, and creates a command window where:
-- DEVTOP is defined
-- DEVTOP is in the PATH variable

If devTop is not specified, the script is assumed to exist,
and is invoked. The initialization scripts may also be invoked
directly from the command line, without using this setdev utility.

Each colour is specified as RRGGBB = 6 hex digits

setdev does further initialization, as described in §4.2.

# 5.3 setpalette (Windows)

Setpalette, automatically invoked by *setdev*, sets the background and foreground colours of a Windows command window:

setpalette bgRR bgGG bgBB fgRR fgGG fgBB
where R, G and B are single hex digits

A command window has 16 colours in its palette. Setpalette sets colour 0 to the specified background RGB, and colour 1 to the specified foreground RGB. This is used in conjunction with cmd -t:01 for the colours to be used as intended. For example

*setpalette ff ff cc 00 00 00*

has a pale yellow background with black text.

# 5.4 iogen

Given a C++ struct (or class) or enum, iogen generates C++ source code for the iostream insertion/extraction operators of the struct or enum. Iogen is designed for simple structs (no inheritance), and is designed for automating the use of structs for reading and writing parameter files. Complex parameter files may be obtained using sub-structs.

Usage: iogen [OPTIONS] [tail...]

```
Positionals:
  tail TEXT ...           The remaining arguments, passed to clang parser

Options:q:/dev/src/gui/widgets/TypedSpinBox.hpp
  -h,--help                     Print this help message and exit
  -i,--inFile TEXT REQUIRED     File containing structs, classes & enums
  -n,--noOps                    Don't generate insertion/extraction operators
  -a,--all                      Don't restrict to the main source file
  -s,--subtypes IncludePrefix   Sub-types too.
                                By design, only works with -t flag(s).
                                This switch replaces forward declaration
                                by header inclusion for all types,
                                not just sub-types.
  -o,--outBase outBase REQUIRED
                                Creates outBase.h and outBase.cpp
  -t,--type TEXT ...            Specific types to process only
```

For example, a simplified command line (automatically assembled by the Makefile) from ldev/src/ogging/extra is

*iogen -i GenLoggingExtra.hpp -a -t ssrd_logging::Level -o iog_GenLoggingExtra -- -I/usr/include/c++/9 - -xc++*

Parameters after "–" are for the clang parser. This command line is built by the Makefile system, as part of automating the use of iogen. In this example, insertion/extraction operator source code is created for the C++ enumeration *ssrd_logging::Level .* As the enumeration is not in the top level source file, *GenLoggingExtra.hpp,* the -a flag must used. However, because of the -t flag, only the specified type is processed.

The source code dev/src/examples/src/iogenEx.cpp has an example that compiles to an executable. Build with, *make iogenEx* (append *.exe* on Windows).

# 5.5 meconv

meconv (Macros to Enum conversion) takes a bunch of preprocessor macro definitions, and converts them into source code that can, instead, be used as part of a C++ enum.

```
 Usage: meconv [OPTIONS]

Options:
  -h,--help                     Print this help message and exit
  -i,--inFile TEXT REQUIRED     File containing macros
  -o,--outFile TEXT REQUIRED    Contains list of  enum = value
  -m,--mprefix TEXT REQUIRED    Prefix for macros to convert
  -e,--eprefix TEXT REQUIRED    Replacement prefix for enums
  -2                            Create 2 output files
Converts all macros that begin with the macro prefix.
The input file is scanned by the gnu C++ preprocessor,
```

so the macros do not need to be directly in the input file.
Each enum=value line, except the last, is ended by a comma.

The value may actually be a macro, rather than an integer.
With the -2 option, two output files are created, with 0 & 1
appended to the output file name. The 0-file is identical to
the standard output file (without the -2 flag), and the 1-file
defines an array of { char*, enum }, which can be used to compile
into an executable to write out the values all as numbers, as is
done by the inc2inc executable.

Advantages of using an enum instead of macros are

- Type safety
- The insertion/extraction operators generated for the enum by iogen (§5.4), can read and write the enum string names, as well as the enum numerical values.

Here are the last few lines of example output:

```
VKEY_CRSEL = 0xF7,
VKEY_EXSEL = 0xF8,
VKEY_EREOF = 0xF9,
VKEY_PLAY = 0xFA,
VKEY_ZOOM = 0xFB,
VKEY_NONAME = 0xFC,
VKEY_PA1 = 0xFD,
VKEY_OEM_CLEAR = 0xFE
```

Note that the last line does not end in a comma. The source code dev/src/examples/src/meconvEx.cpp has an example that compiles to an executable. Build with, *make meconvEx* (append *.exe* on Windows).

# 5.6 appendvk

A recent version of Windows 10 does not have VKEY_ macros for all the keyboard codes. Fortunately, this lacuna is only for the alpha-numeric keys 0-9,A-Z, and the key codes are simply the ascii values.

```
Usage: appendvk [OPTIONS]

Options:
  -h,--help                Print this help message and exit
  -o,--outFile TEXT REQUIRED  Contains list of  enum = value
  -e,--eprefix TEXT REQUIRED  Prefix for enums e.g. VKEY_
Append
    VKEY_* = value,
to the file
for * = 0...9, and * = A...Zq:/dev/src/gui/widgets/TypedSpinBox.hpp
where, value = the ascii value of * .
```

Here are the first few lines of the otuput:

```
,
VKEY_0 = 0x30,
VKEY_1 = 0x31,
VKEY_2 = 0x32,
```

There is a leading comma (the first line of the output), as this is intended to be appended to an existing list, which does not end in a comma.

# 6 Logging

Logging has always been an important aspect of software engineering. Even the bare-bones writing of messages to *stdout* has been invaluable. More sophisticated logging systems label the messages according to priority (WARNING, ERROR,…) and print these selectively. Further, messages can be classified according to a generalized concept of originating location, and subsequently sent to respective log files. Moreover, message destinations can be more general than stdout and disk files e.g. GUI windows and socket connections. All this flexibility requires configuration – which loggers send which messages with which priorities to which destinations. Configuration files can be used to do this.

There are many C++ logging libraries to choose from, as a net search will quickly reveal. The logging package in the current build system is simply a wrapper, and has been designed with the following goals:

1. Be able to wrap around external logging libraries, without modifying any of the existing logging calls scattered through the user's source code.
2. Be able to switch between logging libraries by editing a Makefile variable.
3. Be able to remove all logging calls by editing a Makefile variable.

The above design goals seem to imply that only one logging library may be active in user's source code, but, in fact, if multiple logging libraries are desired , this can be done by judicious placement of the macro *SSRD_LOGGING* in the source code. For simplicity, we discuss only the case of a single active logging library, at a time.

## 6.1 Interface

The current interface is rudimentary – prioritized writing to the screen. There is no decomposition according to originating location, no message destination other than the screen, and the only configuration is the setting of the threshold logging level. However, when needed, this interface can be augmented.

There are 9 logging levels, corresponding to the 9 macros, in increasing priority:

- SLOG_TRACE( expr );
- SLOG_DEBUG( expr );
- SLOG_INFO( expr );
- SLOG_NOTICE( expr );
- SLOG_WARN( expr );
- SLOG_ERROR( expr );
- SLOG_CRITICAL( expr );
- SLOG_ALERT( expr );
- SLOG_EMERGENCY( expr );

There is also SLOG_EXCEPTION(expr), which logs an emergency and throws an exception. The *expr* is according to iostream e.g. *“There are “ << numAplples << “ apples”.* The one configuration command is SLOG_SET_LEVEL( level ), where *level* is from the enum ssrd_logging::Level in dev/src/logging/src/GenLoggingUtil.hpp ,

There are currently two logging libraries supported:

- **Simple** (a few macros in dev/src/logging/src/GenLogging_Simple)
- **Boost** (boost_log)

To choose between these two options, set the Makefile variable SSRD_LOGGING to either Simple or Boost. This variable is currently in the Platform.inc configuration files. If SSRD_LOGGING is set to be empty (no value), then all of the above SLOG macros will evaluate to empty, so that the logging statements will have no effect on the run-time behaviour of the user’s source code.

Not all logging libraries have the same number of logging levels, nor the same names for the levels. The way SLOG_SET_LEVEL maps logging levels to boost logging levels, is shown in Table 1.

*Table 1: Boost logging levels*

| **value** | **ssrd_logging** | **boost** |
|---|---|---|
| 0 | lev_TRACE | trace |
| 1 | lev_DEBUG | debug |
| 2 | lev_INFO | info |
| 3 | lev_NOTICE | info |
| 4 | lev_WARN | warning |
| 5 | lev_ERROR | error |
| 6 | lev_CRITICAL | error |
| 7 | lev_ALERT | error |
| 8 | lev_EMERGENCY | fatal |

This captures all the boost logging levels. However, for example, both lev_INFO and lev_NOTICE will be logged as a boost(info) message, the difference being that, in the case of lev_NOTICE, the message string will contain, “NOTICE”, as well.

This design decision was taken so as to keep the wrapping code light, but does also have the effect of respecting the boost design decision of fewer logging levels.

When using the CLI11 library for command line parsing, the header file dev/src/logging/GenLoggingCLI.hpp has the macro CLI11_LOG_PARSE which can be used instead of CLI11_PARSE. This has the effect of adding the optional command line option

--logLevel <*level*>

and calls the SLOG_SET_LEVEL macro. The default level is INFO. *level* is an integer from 0 to 8, as in the table. Integers outside this range are clamped to either 0 or 8.

When using the logging package, include only the header file *GenLoggingCLI.hpp* (or *GenLogging.h* if you are not using the CLI command line parser). You control which logging library is used, with the macro SSRD_LOGGING, which is both a make macro and a CPP macro.

The source code dev/src/examples/src/loggingEx.cpp has an example that compiles to an executable.

# 7 Makefiles

As introduced in §4.1, the Makefile in a folder of the development tree is assembled from various pieces. The piece which tailors the Makefile to specific purposes, is the *Makefile.in* from the corresponding source folder. This section discusses the Make macros, summarized in Table 2, that are available to facilitate writing a Makefile.in. More details are in the corresponding subsections.

*Table 2: Makefile.in Functionality and Macros*

*Any of the macros in this table may be empty, if not needed.*

| **Syntax** | **Summary** |
|---|---|
| USR_CPPFLAGS = -IFolder1 ... | Verbatim CPP flags. Folders of include files for compiler. |
| INC_INST_DIR = Folder | Folder of installed include files: headers, templates, ... |
| HEADERS= header1.h ... | Include files to be linked from INC_INST_DIR |
| HEADERS_MV = header1.hpp ... | Include files to be moved to INC_INST_DIR |
| USR_CXX_FLAGS | Verbatim compile flags |
| USR_LIB_DIRS = Folder1 ... | Folders of libraries for the linker |
| USR_LIB_NAMES = library1 ... | Base names of libraries to be linked with the executable(s) |
| USR_LDFLAGS = ... | Verbatim linker flags |
| USR_LDLIBS = ... | Verbatim linker libes: full path, or -lname, or name.lib |
| LIBRARY_NAME = MyLib | Base name of library being created or updated |
| LIB_INST_DIR = Folder | Folder containing LIBRARY_NAME |
| LIBRARY_OBJ_NAMES = file1 ... | Base names of object modules for LIBRARY_NAME |
| USR_LIBFLAGS = ... | Verbatim librarian/archiver flags |
| BINARY_NAME = binName | Base name of executable file to be built |
| BINARY_OBJ_NAMES = name1 ... | Base names of object modules for BINARY_NAME |
| BINARIES_NAMES = name1 ... | Base names of executable files to be built |
| iog_inFile_FLAGS = ... | Flags for iogen in creating iog_inFile.h and iog_inFile.cpp |
| <No Macro> | QT moc creating moc_inFile.cpp |
| SUBDIRS = subFolder1 ... | Sub-folders that Make should also process |

The design strives for simplicity, and allowing the user full flexibility of the make system. For example, when working in a particular development folder, should it be that you want some executables to be installed to one destination, and some other executables to be installed to another destination, it is recommended to use separate development folders,

rather than complicate the make rules for multiple executable installation destinations from a single development folder.
The same design decision applies, should you want a particular development folder to contribute to two distinct libraries; it is, again, recommended to use separate development folders, each targeting a single library. Lastly, the same recommendation applies, should you want to install header files to more than one destination location.

# 7.1 Header files

Header files need to be dealt with in various ways.

## Finding

Compilers seem to have a common notation for specifying directories where to look for header files. In the Makefile.in, use

```
USR_CPPFLAGS += -IFolder1 -IFolder2 ...
```

The *USR_* prefix is reserved for macros that are intended for use in Makefile.in . The USR_CPPFLAGS is inserted, verbatim, to the C++ command line.

## Installing source header files

If header files from the source folder need to be made available to other packages, put them into the HEADERS macro

```
HEADERS= header1.h  header2.hpp ..
```

and make sure that the macro INC_INST_DIR is set to where the header files should be linked from. The extensions of the header files are arbitrary. For example,

```
INC_INST_DIR = $(DEVTOP)/inc
```

The command make incinst will link the header files from the INC_INST_DIR, as explained in §3.1.

## Installing generated header files

Some header files are generated e.g. by the iogen utility, to create IO source code. This generated header is created in the development folder (not the source folder). If the generated header, *iog_header.h*, is to be made available to other packages, use the macro

```
HEADERS_MV = iog_header.h iog_header1.hpp
```

The command make incinst will move (not link) the listed header files to the INC_INST_DIR folder. The extensions of the header files are arbitrary.

# 7.2 Compilation

Flags for the C++ compiler:

```
USR_CXXFLAGS = …
```

The USR_CXXFLAGS macro is inserted verbatim to the compile line.

# 7.3 Libraries

Libraries need to be dealt with in various ways.

## Finding libraries

Tell the linker where to find libraries, using

```
USR_LIB_DIRS = Dir1  Dir2 ...
```

At link time, this gets converted to -LDir1 (unix) or /LIBPATH:Dir1 (windows).

## Linking with libraries

When the executable you are building needs specific libraries, use the macro

```
USR_LIB_NAMES = library1  library2 ...
```

At link time, this gets converted to *-llibrary1* (linux) or *library1.lib* (windows). In the event that library names across different platforms are not the same, you can create appropriate macros in the platforms' Platform.inc files, or do the following in the Makfile.in

```
ifeq ($(OS), Windows_NT)
USR_LIB_NAMES += windowsLibName
else
USR_LIB_NAMES += linuxLibName
endif
```

Finally, if USR_LIB_NAMES does not work for you,

```
USR_LDFLAGS = flags verbatim for the linker
```

```
USR_LDLIBS = libraries verbatim for the linker
```

USR_LDLIBS is appended to the libraries on the link line. The purpose is to allow full path names to libraries. On linux, this can be interspersed with -lName notation, and on Windows with Name.lib notation.

## Creating/Augmenting libraries

Each Makefile.in is set up to be able to create/contribute to a library.

```
LIBRARY_NAME = MyLib
```

This refers to library file libMyLib.a (linux) or MyLib.lib (windows). The library is built in place in $(LIB_INST_DIR). For example,

LIB_INST_DIR = $(DEVTOP)/lib

The object files that get compiled into this library are

LIBRARY_OBJ_NAMES = file1 file2 …

The linker will see *file1.o* (linux) or *file1.obj* (windows).

If LIBRARY_OBJ_NAMES does not work for you,

USR_LIBFLAGS = flags verbatim for the librarian

The command *make libs* inserts the object files into the library.

# 7.4 Executables

A Makefile.in supports two ways to build executables:

1. Build one executable, specify which object modules are to be linked to the executable, and specify the name of the executable.
2. Build several executables. No object modules are specified, other than the one containing main() (and the libraries being used in the link). The base name of the executable is the base name of the object module containing main(), with the appropriate extension appended. This is useful if many simple utilities are being built in the same folder.

Both approaches may be taken in the same development folder.

The executables are created with the command *make bins* . They are installed to BIN_INST_DIR with the command *make bininst* .

## One executable

Set

BINARY_NAME = binName

The file name of the executable is *binName* (linux) or *binName.exe* (windows). Specify object modules to link into the executable as

BINARY_OBJ_NAMES = name1 name2 ...

The file names used are as *name1.o* (linux) or *name1.obj* (windows).

## Multiple executables

Set

BINARIES_NAMES = binName1 binName2 ...

This creates the executables binName1 and binName2 (with appropriate suffixes appended). The only explicit object module linked to binNameI is binNameI.o (linux) or binNameI.obj (windows).

# 7.5 Generate source code with iogen

When invoking via Makefile.in, the resulting command line for iogen is

iogen -i inFile.h -o iog_inFile *Flags* #Could also be -i inFile.hpp

where the *Flags* on the above command line are specified in the Makefile.in as

iog_inFile_FLAGS = *Flags*

The two files that are created by iogen are: iog_inFile.h and iog_inFile.cpp . If you want to share this header file outside the packge, you can add iog_inFile.h to the HEADERS_MV macro (this will also invoke iogen).

To make use of iog_inFile.cpp , and to induce iogen to be invoked, you can add *iog_inFile* to either BINARY_OBJECT_NAMES or LIBRARY_OBJ_NAMES , depending on whether you are building a single executable, or a library, or both.

# 7.6 Generate source code with QT moc

This is only relevant if you are doing QT development. The QT moc utility is used to generate source code that supports the signal/slot model of communication. The details (https://doc.qt.io/qt-5/signalsandslots.html) are beyond the scope of this document. When invoking via Makefile.in, the resulting QT moc command line is:

moc inFile.h -o moc_inFile.cpp #Could also be inFile.hpp

To make use of moc_inFile.cpp , and to induce moc to be invoked, you can add moc_inFile to either BINARY_OBJECT_NAMES or LIBRARY_OBJ_NAMES.

# 7.7 Subfolders

If the Makefile.in has the macro

SUBDIRS = subFolder1 subFolder2 …

then *make* will process each of the sub-folders , in order, before processing the current folder. Usually, the SUBDIRS macro occurs at non-leaf points of the source hierarchy, and there is no other work to do at those locations, other than recurse.

# 8 Addendum

The **BLDEV** system creates another utility, *replace,* which replaces one string in a file with another string (like *sed*), but also allows the two strings to be specified from files as well as the command line, with options for how to handle white space. This was used in conjunction with *find* and *xargs* to manage the **Copyright** and **License** headers that are part of many of the **BLDEV** files.

# 9 References


[1] Clang/LLVM
https://releases.llvm.org/download.html
C++ parser library to create IO source code for enums and bare structs/classes.

[2] Boost
https://sourceforge.net/projects/boost/files/boost-binaries
filesystem, log

[3] MSYS2
https://www.msys2.org/
bash, make, gcc and utilities to support Makefile usage on Windows

[4] CMake
https://cmake.org/

[5] VS Visual Studio
https://visualstudio.microsoft.com/downloads/
VS is accessed solely through the x64 native command window.

[6] GNU autotools
https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html

[7] CLI11
https://github.com/CLIUtils/CLI11
Command line parameter processing

[8] Bazel
https://bazel.build/

[9] Nalys Blog, *Build Systems: Bazel vs Make*
https://nalys-taas-projects.gitlab.io/internal/taas_blog/post/bazel_vs_make/

[10] BUILDING.md, Building BLDEV from source
BLDEV delivery.