



Elektrobit

EB tresos Bootloader Generic documentation



Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

<https://www.elektrobit.com/support>

Legal disclaimer

Confidential information.

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Elektrobit Automotive GmbH.

All brand names, trademarks, and registered trademarks are property of their rightful owners and are used only for description.

Copyright 2021, Elektrobit Automotive GmbH.

Table of Contents

1. Overview of EB Bootloader documentation	5
2. Compilers supported	6
2.1. Compiler options for GCC 6.2.0	6
3. BL release notes	7
3.1. Overview	7
3.2. Scope of the release	7
3.2.1. Configuration tool	7
3.2.2. EB Bootloader modules	7
3.3. Module release notes	7
3.3.1. Platforms module release notes	7
3.3.1.1. Change log	8
3.3.1.2. New features	9
3.3.1.3. EB-specific enhancements	10
3.3.1.4. Deviations	10
3.3.1.5. Limitations	10
3.3.1.6. Open-source software	10
3.3.1.6.1. Open-source software in software executed on the ECU	10
3.3.1.6.2. Open-source software in software used for the development infrastruc- ture	10
4. BL user's guide	12
4.1. Overview	12
4.2. Application template and application demos	12
4.2.1. Overview	12
4.2.2. Getting an application demo to run	12
4.2.2.1. Prerequisites for starting a new project	13
4.2.2.2. Importing the application demo	13
4.2.2.3. Adapting the build environment	15
4.2.2.3.1. Changing the compiler	16
4.2.2.3.2. Changing the board settings	16
4.2.2.4. Building the application demo	17
4.2.2.5. Taking further steps	17
4.3. Build environment	18
4.3.1. Overview	18
4.3.2. Background information	18
4.3.2.1. Improve build speed with GNU Make command line parameters	20
4.3.2.2. Folder structure of a project	20
4.3.2.3. Basic concepts of the makefiles	21
4.3.2.4. <code>Make</code> plug-ins	22
4.3.2.5. Board support packages	24

4.3.3. Configuring the build process	24
4.3.3.1. Setting the application path in <code>util/launch.bat</code>	25
4.3.3.2. Configuring the makefiles	26
4.3.3.2.1. Starting the command shell	28
4.3.3.2.2. Defining the name of the project	28
4.3.3.2.3. Defining C files to be compiled	29
4.3.3.2.4. Defining a list of assembler files	29
4.3.3.2.5. Building libraries of compiled object files	30
4.3.3.2.6. Linking libraries to a project	31
4.3.3.2.7. Linking binary objects to a project	31
4.3.3.2.8. Extending the include path with project-specific path names	32
4.3.3.2.9. Selecting a board support package	32
4.3.3.2.10. Selecting a toolchain/a different compiler	33
4.3.3.2.11. Extending compiler and assembler options	33
4.3.3.2.12. Adding preprocessor defines	34
4.3.3.3. Compiling the application	35
4.3.3.3.1. Locating output directories	36
4.3.3.3.2. Other <code>make</code> targets	37
5. BL module references	38
5.1. Overview	38
6. Bibliography	39



1. Overview of EB Bootloader documentation

Welcome to the EB Bootloader (BL) product documentation.

This document provides:

- ▶ [Chapter 2, “Compilers supported”](#): compiler options used for EB Bootloader QP
- ▶ [Chapter 3, “BL release notes”](#): release notes for the BL modules
- ▶ [Chapter 4, “BL user's guide”](#): containing background information and instructions
- ▶ [Chapter 5, “BL module references”](#): information about configuration parameters and the application programming interface

2. Compilers supported

This release of EB tresos AutoCore supports the following compilers:

- GCC Version 6.2.0

2.1. Compiler options for GCC 6.2.0

The compiler options summarize under which conditions the module is to be built. The module is tested using these compiler options. If you change the compiler options, consider the module *untested*.

Compiler	Options
GCC Version 6.2.0	<code>-c -std=iso9899:199409 -ffreestanding -Wpedantic -Wall -Wextra -Wdouble-promotion -Wnull-dereference -Wshift-negative-value -Wshift-overflow -Wswitch-default -Wunused-parameter -Wunused-const-variable=2 -Wuninitialized -Wunknown-pragmas -Wstrict-overflow=2 -Wno-unused-local-typedefs -Warray-bounds=1 -Wduplicated-cond -Wtrampolines -Wfloat-equal -Wdeclaration-after-statement -Wundef -Wno-endif-labels -Wshadow -Wbad-function-cast -Wc90-c99-compat -Wc99-c11-compat -Wcast-qual -Wcast-align -Wwrite-strings -Wdate-time -Wjump-misses-init -Wfloat-conversion -Wno-aggressive-loop-optimizations -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -Wnested-externs -Wvla -Wno-long-long -fno-sanitize-recover -fno-ident -g -O3 -fno-strict-aliasing -fsanitize=undefined -fstack-usage -DNOGDI -D_X86_ -D_WIN32X86_ -C_GCC_</code>

3. BL release notes

3.1. Overview

This chapter provides the BL specific release notes.

3.2. Scope of the release

3.2.1. Configuration tool

Your release of EB Bootloader is compatible with the release of the EB tresos Studio configuration tool:

- ▶ EB tresos Studio: 28.0.0 b210315-0853

3.2.2. EB Bootloader modules

The following table lists modules which are part of BL release.

Module name	Module version	Supplier
Platforms	4.0.0	Elektrobit Automotive GmbH

Table 3.1. Modules specified by OEM specification

3.3. Module release notes

3.3.1. Platforms module release notes

- ▶ Module version: 4.0.0.B419173

- ▶ Supplier: Elektrobit Automotive GmbH

3.3.1.1. Change log

This chapter lists the changes between different versions.

Module version 3.0.1

2018-07-11

- ▶ Updated compiler options:
 - ▶ Added `-fno-ident`
 - ▶ Added `-Wpedantic`
 - ▶ Added `-Wno-unused-local-typedefs`
 - ▶ Removed `-pipe`
- ▶ Updated compiler options: Replaced `-Wstack-usage=200` by `-fstack-usage` to avoid unnecessary warnings and get stack usage information from all files

Module version 3.0.0

2017-09-22

- ▶ Added rule for manifest file to prevent User Access Control warning
- ▶ Added support for GNU C Compiler version 6.2.0
- ▶ Added CIF library
- ▶ Added Compiler abstractions for the `asc_Crypto` and `asc_Crylf` modules
- ▶ removed compiler specific files and compiler settings, they are moved to the new Compiler module update module version 3.0.0

Module version 2.1.0

2015-07-08

- ▶ Add templates for `MemMap.h` and `Compiler_Cfg.h`.
- ▶ Use public OS API for concurrent access locks
- ▶ Add support for GNU C Compiler version 4
- ▶ Make `Compiler.h` compatible to AUTOSAR release 4.0 [R001]

- ▶ Shortened output messages of make calls
- ▶ Use .i suffix for preprocessor files when using the gcc toolchain
- ▶ Added support for CONSTP2FUNC compiler abstraction macro.
- ▶ Improved make build environment to be more lean and faster by updating makefiles to refactoring of `global.mak` of Make plugin
- ▶ Added compiler abstraction for module SecOC.
- ▶ Added support for GNU C Compiler version 4.8.1
- ▶ Removed support for older GNU C Compilers (before version 4.5.1)
- ▶ Added support for Dlt module

Module version 2.0.0

2010-02-11

Module version 1.0.2

2010-10-08

- ▶ Make `Compiler.h` compatible to AUTOSAR release 4.0 [R001]

Module version 1.0.1

2010-04-30

- ▶ Add templates for `MemMap.h` and `Compiler_Cfg.h`.
- ▶ Use public OS API for concurrent access locks
- ▶ Add support for GNU C Compiler version 4

Module version 1.0.0

2010-01-05

- ▶ Initial setup of Platform module for WIN32X86.

3.3.1.2. New features

- ▶ No new features have been added since the last release.

3.3.1.3. EB-specific enhancements

This module is not part of the AUTOSAR specification.

3.3.1.4. Deviations

This module is not part of the AUTOSAR specification.

3.3.1.5. Limitations

This chapter lists the limitations of the module. Refer to the module references chapter *Integration notes*, subsection *Integration requirements* for requirements on integrating this module.

- For this module no limitations are known.

3.3.1.6. Open-source software

The software that is delivered with EB tresos AutoCore Generic can be classified into the following two categories:

- Software that is executed on the electronic control unit (ECU).
- Software that is used for the development infrastructure (configuration, generation, building) and thus executed on the development platform.

All license text files are located in your module delivery in the tool-specific sub-folders of the sub-folder `<installed module location>\tools\`.

3.3.1.6.1. Open-source software in software executed on the ECU

No open-source software that runs on the ECU is delivered with `Platforms`.

3.3.1.6.2. Open-source software in software used for the development infrastructure

The following open-source software that is used in development is delivered with `Platforms`:

- MinGW GNU C compiler
6.2.0

<https://sourceforge.net/projects/mingw-w64/files/Toolchains%20targetting%20Win32/Personal%20Builds/mingw-builds/6.2.0/>

List of licenses:

- ▶ GNU GENERAL PUBLIC LICENSE Version 3, 29 June 2007
`mingw-gcc-6.2.0/README.txt` (as stored in the folder mentioned above)

List of copyrights:

- ▶ Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Copyright (C) 1989, 1991 Free Software Foundation, Inc.

4. BL user's guide

4.1. Overview

The BL user's guide provides information about common modules that are always available as part of the EB tresos Bootloader Generic product.

- ▶ [Section 4.2, “Application template and application demos”](#) describes an application template and example programs (application demos).
- ▶ [Section 4.3, “Build environment”](#) describes the build environment and how to adapt this for your project needs.

4.2. Application template and application demos

4.2.1. Overview

The idea behind the application template and application demos is to give an example of how to start a new project. Everything is kept as simple as possible. Keep in mind that application demos are only a rudimentary starting point and must not be used as the basis for a real ECU.

The application template and application demos chapter provides you with background information and the set-up of the specific application demos.

The application template and the various application demos are EB tresos Studio projects. These projects can be built and loaded into the ECU target, where they can then be executed. Instructions on how to build and run the application demos are provided in later sections of this chapter.

Certain application demos are provided directly with EB tresos Bootloader Generic and you can import them directly into EB tresos Studio. In this chapter, you will learn the general steps from importing an application demo into EB tresos Studio to building the application demo.

4.2.2. Getting an application demo to run

4.2.2.1. Prerequisites for starting a new project

Before you start using an application demo, please install the compiler supported by EB tresos Bootloader.

Please refer to the `EB tresos Bootloader release notes` for further information about the compiler and compiler version for your architecture.

NOTE



The installation path must not contain any blanks

Ensure that you install the compiler in a path without any blanks in the pathname. Due to limitations in the build environment, pathnames containing blanks will not work.

4.2.2.2. Importing the application demo

The application demos are delivered as EB tresos Studio project. You need to import the project into your EB tresos Studio workspace, e.g. at `$TRESOS_BASE/workspace`. The `$TRESOS_BASE` is the directory into which you installed EB tresos Studio, e.g. `C:/EB/tresos`.

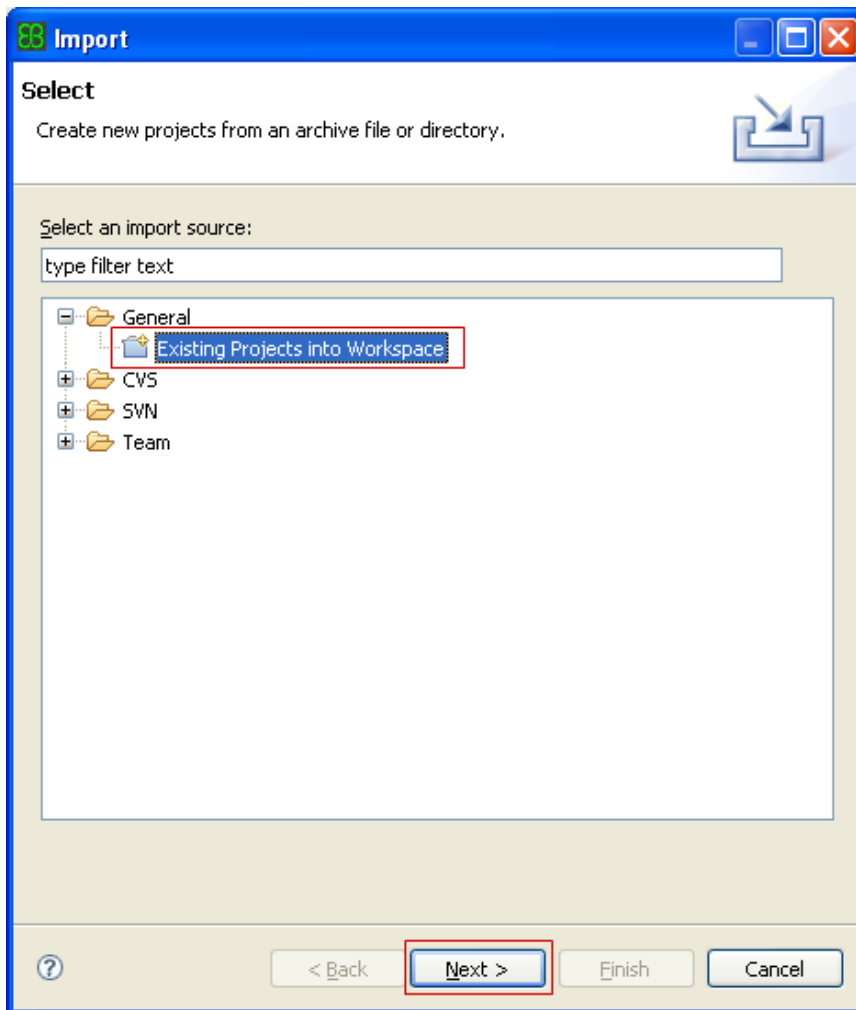
To import the application demo into your workspace:

- ▶ Locate `$TRESOS_BASE/bin/tresos_gui.exe`.
- ▶ Double-click `tresos_gui.exe`.

EB tresos Studio opens up.

- ▶ In the **File** menu, select **Import**.

The **Import** dialog opens up.

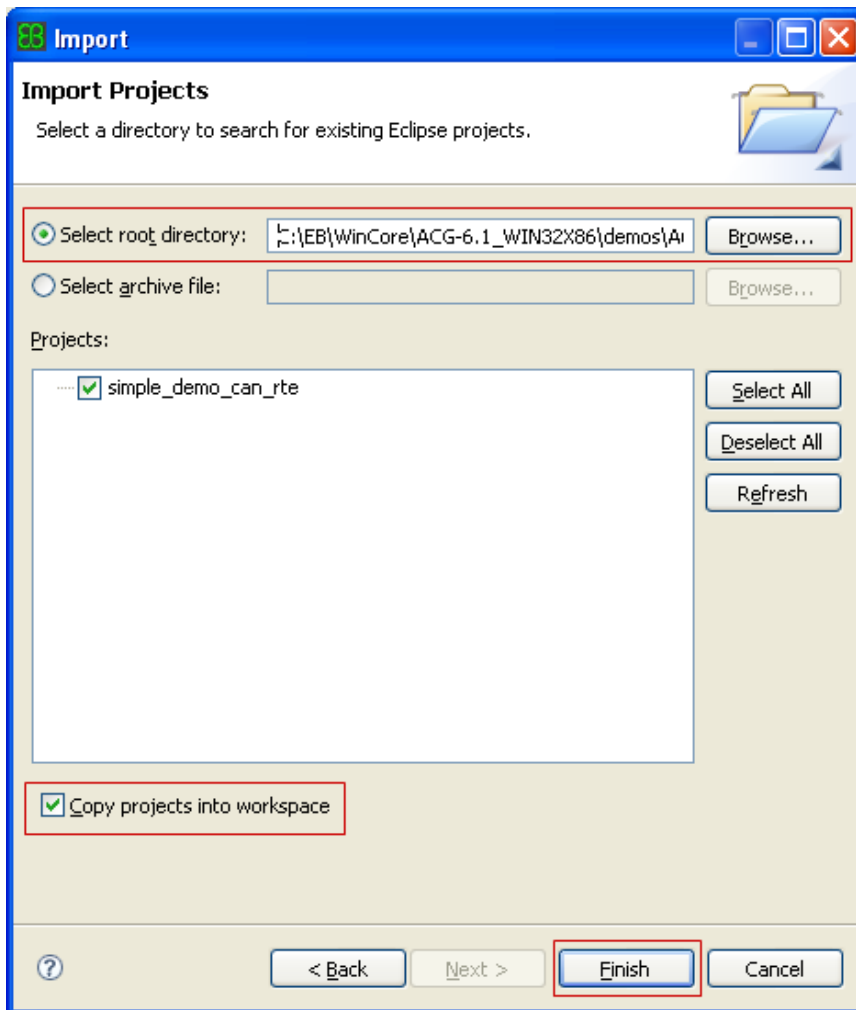


- ▶ Select **Existing Projects into Workspace**.
- ▶ Click **Next**.
- ▶ Select **Select root directory** and browse to `$TRESOS_BASE/demos/<VARIANTS>/<$DERIVATE>/<name of application demo>`.

And choose one of the available application demo, e.g. `simple_demo_psa_dc_ti` in order to work with `simple PSA_DC_TI demo application demo`.

- ▶ Click **OK**.

The project appears in the **Projects** window of the **Import** dialog.




- ▶ Select **Copy projects into workspace**.

The project is now copied to the default workspace at `$TRESOS_BASE/workspace`.

- ▶ Click **Finish**.

You are done.

In the **Project Explorer** view you now see the project:

- ▶ Open it by double-clicking on the project name.
- ▶ Open the configuration by clicking the gray chip symbol .

4.2.2.3. Adapting the build environment

This sections describes how to adapt the build environment in order to build the application demo. For a more detailed description of the build environment please refer to [Section 4.3, “Build environment”](#).

To change the settings or the path of the compiler please follow the steps below:

1. Locate the project folder in the workspace directory, e.g. `$TRESOS_BASE/workspace/<name of application demo>`.
2. Open the file `util/launch.bat` in a text editor.
3. Set the environment variable `TRESOS_BASE` to the directory, into which you have installed EB tresos Studio and save the file.
4. Open the file `util/<target>_<derivative>_<compiler>_cfg.mak` in a text editor.
5. Change the variable `TOOLPATH_COMPILER` to the actual compiler installation path. For example: `TOOLPATH_COMPILER ?= C:/WindRiver/diab/5.5.1.0`.
6. The variable `CC_OPT` in the same file contains the compiler options. If you need any specific settings, adjust `CC_OPT`.

4.2.2.3.1. Changing the compiler

NOTE



For EB tresos WinCore, no compiler has to be configured

EB tresos WinCore is delivered with the MinGW development environment, which is installed automatically when you build the application demo. The MinGW development environment also contains a compiler. Therefore, you only need to follow these instructions when you want to configure a different compiler.

If multiple compilers are supported on your architecture you can change the compiler. To change the compiler:

1. Locate the project folder in the workspace directory, e.g. `$TRESOS_BASE/workspace/<name of application demo>`.
2. Open the file `util/<target>_<derivative>_Makefile.mak` in a text editor.
3. Set the variable `TOOLCHAIN` to the name of the toolchain, e.g. `TOOLCHAIN = dcc`.
4. Set the variable `COMPILER` to the compiler, e.g. `COMPILER = MPC_MPC5567_dcc`.

4.2.2.3.2. Changing the board settings

If you want to use a different board, you need to change the board settings.

The directory `$PROJECT_ROOT/source/boards/<board name>` contains board-specific make and source files.

In order to use a different board, please follow the steps below:

1. Locate the project folder in the workspace directory, e.g. `$TRESOS_BASE/workspace/<name of application demo>`.

2. Open the file `util/<target>_<derivative>_Makefile.mak` in a text editor.
3. Change the variable `BOARD`.

The value of the variable has to be the same as the name of the board directory, e.g. `BOARD = EvaBoardMPC5567`.

4.2.2.4. Building the application demo

In order to build an EB tresos Bootloader project, you need to perform the following three steps in the order they are described:

1. Generate the project.
2. Create the project dependencies.
3. Build the project.

To build the application demo, make sure that EB tresos Studio is not running anymore. Then follow these steps:

1. In the `$TRESOS_BASE/workspace/<application_demo_name>/util` directory, double-click the file `launch.bat`. The first start of `launch.bat` takes some time.
2. Type **make generate** and hit the **Enter** key.
3. Type **make depend** and hit the **Enter** key.

The project is being generated.

4. Type **make** and hit the **Enter** key.

Your application demo is being built.

If you work with EB tresos WinCore, the MinGW development environment is being installed with the build of the application demo.

You find the resulting binary file in the `$TRESOS_BASE/workspace/<application_demo_name>/output/bin` directory.

4.2.2.5. Taking further steps

1. Find more information about about EB tresos Studio in the EB tresos Studio user's guide.

This is located at `$TRESOS_BASE/doc/2.0_EB_tresos_Studio/2.1_Studio_documentation_users_guide.pdf`.

2. Find further information about the build environment used for the application demo at [Section 4.3, “Build environment”](#).
3. For more information about EB tresos Bootloader modules have a look into [Chapter 4, “BL user's guide”](#)

4.3. Build environment

The build environment helps you to compile complete projects and create an executable program that can be loaded to the target. It consists mainly of the two EB tresos Studio plug-ins `Make` and `Platforms`. The `Make` plug-in is a set of GNU makefiles, which is located in the directory `$(TRESOS_BASE)/plugins/Make_TS_<version string>`. The `Platforms` plug-in, located in the directory `$(TRESOS_BASE)/plugins/Platforms_TS_<version string>`, contains the platform and toolchain specific GNU makefiles whereas the `Make` plug-in is platform independent.

NOTE



Make plug-in is incompatible with older Platform plug-ins

The `Make` plug-in version 4.0.8 released with product release 7.3 has been refactored and is not compatible anymore with `Platforms` plug-ins released before product release 7.3. So if you update the `Make` plug-in, be sure to also update the `Platforms` plug-in.

In the following chapter, build environment and `Make/Platforms` plug-ins are used as synonyms.

4.3.1. Overview

The following sections in the chapter build environment will show you how the *user build environment* is configured and used. The information helps integrators in setting up projects, which use EB tresos Bootloader modules together with application code.

Although it is possible to integrate EB tresos Bootloader modules with other build environments, EB strongly recommends using the build environment integrated in EB tresos Bootloader.

4.3.2. Background information

The background information section is going to introduce you to the concepts of the build environment. It provides you the knowledge that is necessary to understand the instructions for configuring the build process. Step-by-step instructions are available starting in [Section 4.3.3, “Configuring the build process”](#).

[Figure 4.1, “From ECU configuration to code generation”](#) shows an overview of the workflow from the configuration of an ECU to its code generation:

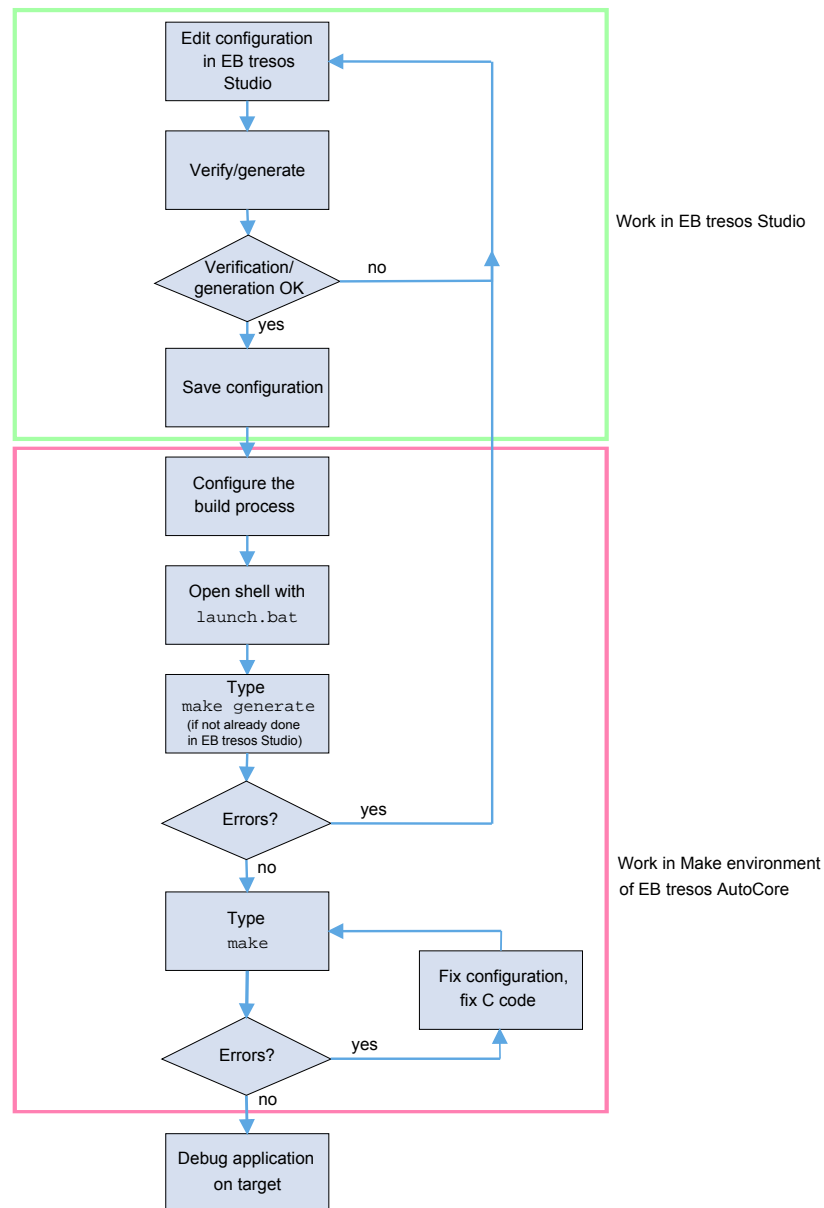


Figure 4.1. From ECU configuration to code generation

4.3.2.1. Improve build speed with GNU Make command line parameters

The build environment is designed for distributing the build steps over as many CPU cores as possible to maximize build speed. To use this feature, run `make` with the following parameters:

```
make -j -O
```

The `-j` parameter triggers `make` to spawn new processes as long as the CPU load on all cores is less than 100%. This parameter is recommended for maximum speed.

If you want to compile files that are stored on drives with slow file system access, e.g. network drives or ClearCase dynamic views, and you run `make -j`, your system may become unresponsive. This behavior occurs because the CPU load will never reach 100% due to the file system bottle neck. In this case you should limit the amount of concurrent processes with the `-jN` parameter. `N` is the amount of concurrent processes. For example, `make -j16` spawns 16 concurrent processes.

The `-O` parameter is a feature which came with GNU Make v4.0. Use this feature to synchronize the output from different processes to avoid that the output is scrambled as seen in GNU Make v3.82.

4.3.2.2. Folder structure of a project

Every project consists at least of the following folders:



Figure 4.2. Project directory structure

The folders contain the following:

- ▶ **MyProject:** This is the main folder of the project. This path is stored in the `make` variable `PROJECT_PATH`.
- ▶ **.prefs:** Contains the file `preferences.xdm` which holds all project settings.
- ▶ **config:** Contains all configuration files. The files are stored in the XDM format by EB tresos Studio.
- ▶ **source/application:** Contains the application source files.
- ▶ **source/boards:** Contains the board support package files. The board support package contains e.g. the linker scripts and parts of the startup code. For further information on the board support package, refer to [Section 4.3.2.5, “Board support packages”](#).

You may need to manually adapt some files in the boards directory. It is not possible to use EB tresos Studio to configure the necessary parameters.

- ▶ `util`: This directory is the starting point for using the build environment. If you want to compile an application, start the batch file `launch.bat` from the `util` directory to open a command shell and setup some environment variables. All compile runs have to be started within this command shell.

For instructions how to compile your application, please refer to [Section 4.3.3.3, “Compiling the application”](#).

Within the `util` directory, you can find the main makefiles `Makefile.mak` and `common.mak`. Moreover, it contains the architecture and compiler-specific makefiles.

4.3.2.3. Basic concepts of the makefiles

Every EB tresos Bootloader module comes with a set of makefiles:

- ▶ `Xxx_defs.mak`: Defines several variables, for example the path to the module implementation or the output path for the generator.
- ▶ `Xxx_rules.mak`: The main purpose of these files is to define the files which have to be compiled for the module libraries.

All configuration-dependent files of EB modules are mapped to the library `Xxx_src_FILES` by default. The following example shows the `Det_src_FILES`:



Example 4.1. `Det_src_FILES`

```
Det_src_FILES      := \  
$(Det_CORE_PATH)/src/Det.c
```

After defining the file sets, the next step is to configure the build environment to actually create those libraries.

The variable `LIBRARIES_TO_BUILD` holds a list of libraries that are to be build.

In the following example, `Det_lib` and `Det_src` are added to `LIBRARIES_TO_BUILD`.



Example 4.2. `LIBRARIES_TO_BUILD`

```
LIBRARIES_TO_BUILD += Det_lib Det_src
```

The build environment can also directly compile and link source files without adding them to a library.

The variables `CC_FILES_TO_BUILD`, `CPP_FILES_TO_BUILD`, and `ASM_FILES_TO_BUILD` contain a list of C, C++ and assembler files which are to be compiled and linked.



Example 4.3. Source files, which are to be compiled and linked

```
CC_FILES_TO_BUILD      += appl.$(CC_FILE_SUFFIX)
CPP_FILES_TO_BUILD     += appl.$(CPP_FILE_SUFFIX)
ASM_FILES_TO_BUILD     += appl.$(ASM_FILE_SUFFIX)
```

4.3.2.4. Make plug-ins

The makefile framework consists of several plug-ins. [Figure 4.3, “Make plug-ins”](#) gives you an overview of the plug-ins:

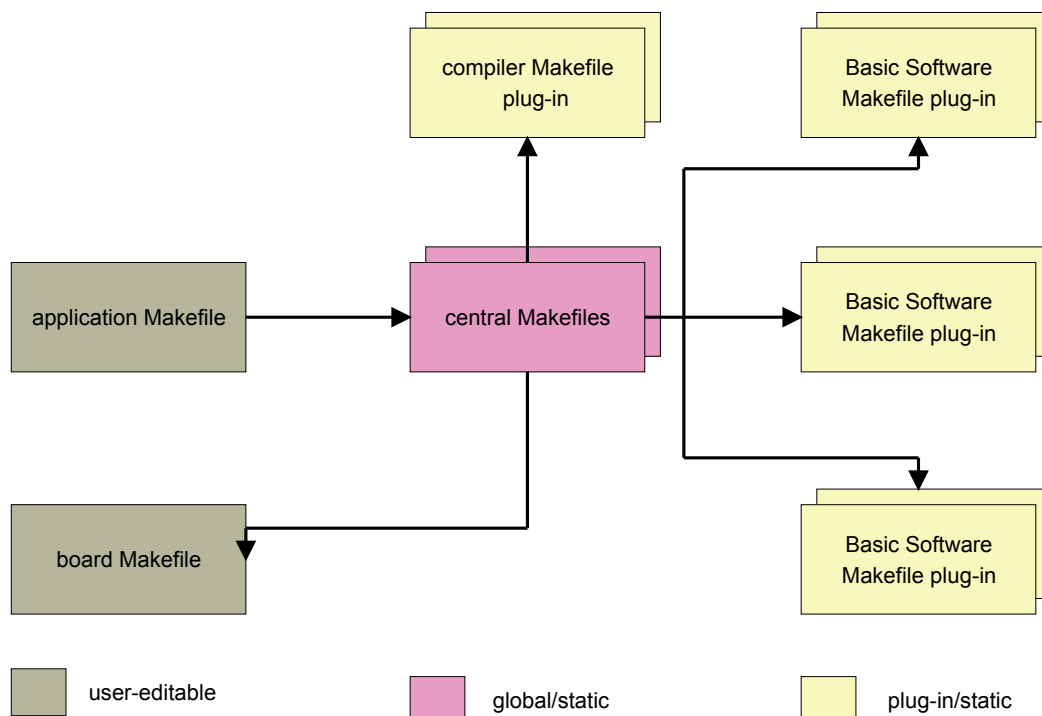


Figure 4.3. Make plug-ins

Starting from the application makefile, the build environment automatically includes several other makefiles. All inclusions are optional.

Every EB tresos Bootloader plug-in *can* implement all makefiles as described in [Section 4.3.2.3, “Basic concepts of the makefiles”](#). However, if some of the files do not exist, this will *not* result in an error message.

On the one hand, this behavior may be confusing, as it does not indicate whether important files are missing. On the other hand, this behavior keeps you from a permanent need to implement the complete set of files, even if the set is not necessary for a specific use case.

Almost all central functionality of the build environment is implemented in the following files:

The file names are sorted alphabetically according to the filename:

Filename	Function
global.mak	Provides all global rules and configuration of the build environment, and includes all other plug-in makefiles. To make use of the makefile framework, include <code>global.mak</code> in your application makefile.
tresos2_rules.mak	Provides rules and configuration for executing the code generation of EB tresos Studio from the command line.

Table 4.1. Central makefiles

Only a few files are located within the project directory. To edit configuration options in the makefiles, you may find the files in `$(PROJECT_ROOT)/util`. Depending on the target and toolchain selected, you may also find the files in `$(PROJECT_ROOT)/source/boards`:

Filename	Function
Makefile.mak	This file is the starting point for the build process, and it includes the <code>common.mak</code> . Usually, you don't need to edit it.
common.mak	This file contains common settings for the make environment such as include paths and C files to build, and it includes the <code>global.mak</code> . You need to edit it if you want to add additional include or source files.
<TARGET>_<DERIVATIVE>_Makefile.mak	This file contains the target-specific settings of the build process. It is included by the <code>common.mak</code> . You need to edit the target-specific options in this file.
<TARGET>_<DERIVATIVE>_<TOOLCHAIN>_cfg.-mak	This file contains toolchain-specific options. For example, you may select the C compiler options here. The file is included by <code>common.mak</code> .

Filename	Function
<code><MODULE>_cfg.mak</code>	Optionally you may add module configuration files in the util directory. For example, if you add a file <code>Det_cfg.mak</code> here, it will override the settings in <code>\$(TRESOS_BASE)/plugins/Det_TS_<version string>/make/Det_cfg.mak</code>

Table 4.2. Application makefiles in `$(PROJECT_ROOT)/util`

Filename	Function
<code><BOARD_MAKEFILE>.mak</code>	This file contains settings for the selected board support package. Depending on the board selected, you may need to edit this file. Check Section 4.3.2.5, “Board support packages” for details.

Table 4.3. Application makefiles in `$(PROJECT_ROOT)/source/boards`

4.3.2.5. Board support packages

A board support package contains code fragments which are necessary to adapt an application to specific hardware, for example a specific startup code. The board support package also contains the linker command file.

You may find all files in the directory `$(PROJECT_ROOT)/source/boards`.

It is not possible to configure the board support package with EB tresos Studio. If you do need to configure the board support package, edit the files in this directory manually.

Depending on the actual target, several boards may be available. To select one of the boards, set the variable `BOARD` in `Makefile.mak`.

When you use the memory abstraction, you must at least edit the linker script to map your sections in `MemMap.h` to the appropriate location. For details on the syntax of the linker script, please take a look at the documentation of your compiler toolchain.

4.3.3. Configuring the build process

You need to configure the build environment in the makefiles listed in [Section 4.3.2.4, “Make plug-ins”](#). It is not possible to edit any configuration of the build environment (EB tresos Bootloader `Make` module) in the EB tresos Studio configuration editor.

The pink field in the workflow overview indicates where in the workflow you currently are:

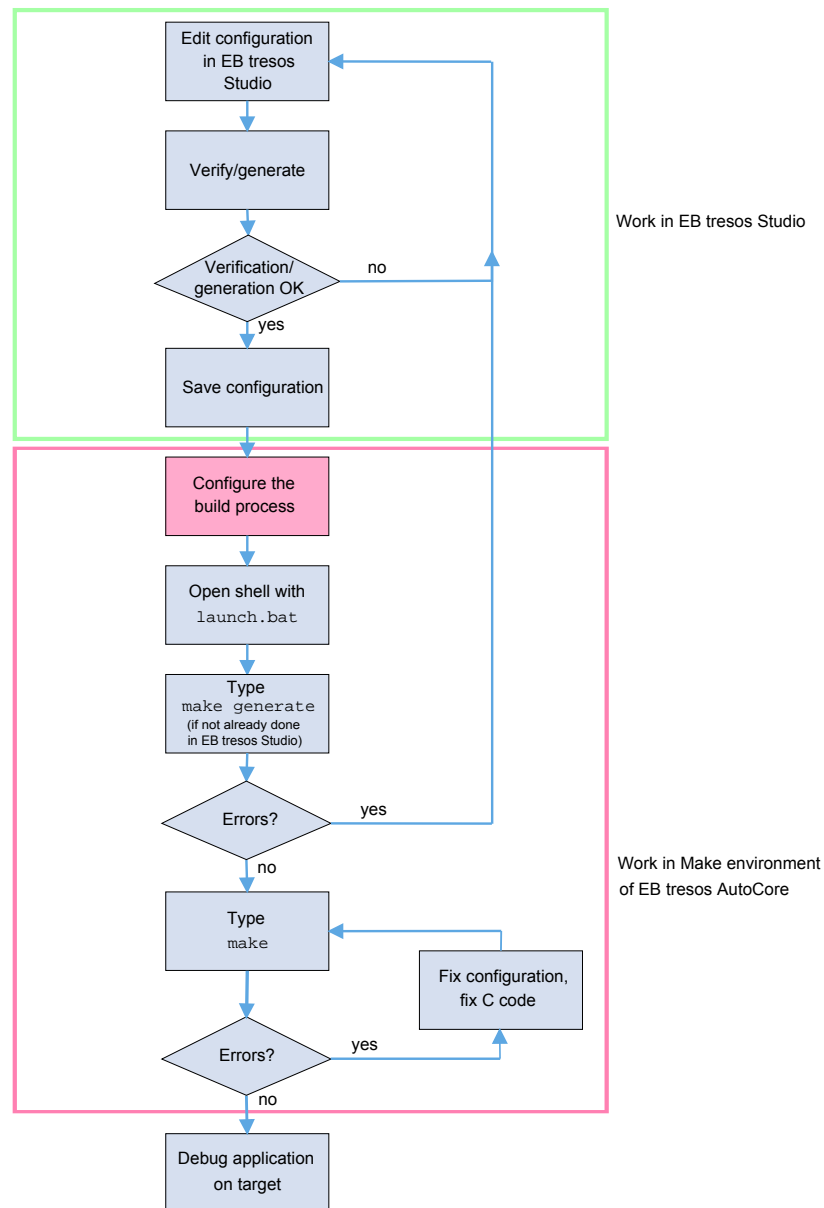


Figure 4.4. Workflow step: configuring the build process

4.3.3.1. Setting the application path in `util/launch.bat`

You need to set up some global parameters, before you can successfully compile your application.

The first and most important global parameter to set is `TRESOS_BASE`, which is located in the `util/launch.bat` batch file. This parameter points to the location of the EB tresos Studio application.

This parameter is usually set during installation of EB tresos Studio by default. If you have deselected this option, you need to manually set the location of EB tresos Studio now. If you have not deselected this option, you do not need to configure the application path and can move on to the next chapter.

The second global parameter you may have to set is `GEN_OUTPUT_PATH`, which is also located in the `util/launch.bat` batch file. This parameter points to the generation path for the code generator. If you did not change this path in your project and kept the default path, you can skip this setting as the default path is set by the makefiles.

To configure the application path manually:

- ▶ Open the `launch.bat` batch file in `util`.
- ▶ Remove the double colon `:` before `TRESOS_BASE` to uncomment the line.

```
:: uncomment line below and add location if you do not set TRESOS_BASE over environment  
::SET TRESOS_BASE=
```

- ▶ Set `SET TRESOS_BASE=` to the pathname, e.g. to `C:/EB/tresos`.

To configure the generation path manually:

- ▶ Open the `launch.bat` batch file in `util`.
- ▶ Remove the double colon `:` before `GEN_OUTPUT_PATH` to uncomment the line.

```
:: uncomment line below and add location if you changed the Generation Path for Code Generator  
::SET GEN_OUTPUT_PATH=
```

- ▶ Set `SET GEN_OUTPUT_PATH=` to the pathname, e.g. to `C:/EB/tresos/workspace/MyProject/output/generated`.

The following code is an example of what the code in the `launch.bat` could look like before you work on it.

```
...  
:: uncomment line below and add location if you do not set TRESOS_BASE over environment  
:: SET TRESOS_BASE=  
:: uncomment line below and add location if you changed the Generation Path for Code Generator  
:: SET GEN_OUTPUT_PATH=  
...
```

Set all other global parameters in the makefile `common.mak`. For a list of configuration parameters refer to [Section 4.3.3.2, “Configuring the makefiles”](#)

4.3.3.2. Configuring the makefiles

The application makefiles, located in the `util` directory in the project folder, contain all global configuration parameters for the build process. You may change configuration parameters by editing variables in these files.

To ensure that all changes take effect, you need to rebuild your project completely.

The following table contains a summary of all configuration options. The table is sorted alphabetically by parameter name. The detailed setup of each parameter is described in the following sections and is linked to from within the list below. The parameter location is based on the location in the (optional) application example delivered with the EB tresos Bootloader.

Configuration parameter	Configuration option	Parameter location
ASM_FILES_TO_BUILD	Define the list of ASM files to build. For instructions, refer to Section 4.3.3.2.4, “Defining a list of assembler files” .	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles
BOARD	Set the name of the board you use for the project. For instructions, refer to Section 4.3.3.2.9, “Selecting a board support package” .	<code>\$(PROJECT_ROOT)/util/\$(TARGET)_\$(DERIVATE)_Makefile.mak</code>
CC_FILES_TO_BUILD	Define the list of C files to build. For instructions, refer to Section 4.3.3.2.3, “Defining C files to be compiled” .	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles
CC/CPP/ASM_INCLUDE_PATH	Extend the include path of the C, C++ or assembler tools. For instructions, refer to Section 4.3.3.2.8, “Extending the include path with project-specific path names” .	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles
CC/CPP/ASM_OPT	Set extra options for the C, C++ or assembler tools.	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles
TOOLCHAIN	Define the name of the compiler tool chain you use. For instructions, refer to Section 4.3.3.2.10, “Selecting a toolchain/a different compiler” .	<code>\$(PROJECT_ROOT)/util/\$(TARGET)_\$(DERIVATE)_Makefile.mak</code>
CPP_FILES_TO_BUILD	This variable is only supported on Windows platforms. Other platforms need adaptation of the rules in <code>compiler_rules.mak</code> in the <code>Platforms</code> plugin.	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles

Configuration parameter	Configuration option	Parameter location
LIBRARIES_LINK_ONLY	Define a list of libraries to link.	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles
LIBRARIES_TO_BUILD	Define a list of libraries to build. For instructions, refer to Section 4.3.3.2.5, “Building libraries of compiled object files” .	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles
OBJECTS_LINK_ONLY	Define a list of precompiled objects to link. For instructions, refer to Section 4.3.3.2.7, “Linking binary objects to a project” .	<code>\$(PROJECT_ROOT)/util/common.mak</code> , module makefiles
PROJECT	Define the name of the project. For instructions, refer to Section 4.3.3.2.2, “Defining the name of the project” .	<code>\$(PROJECT_ROOT)/util/common.mak</code>

Table 4.4. Configuration parameters

4.3.3.2.1. Starting the command shell

The command shell launcher is located at `MyProject/util/launch.bat`.

To open the command shell.

- ▶ Double-click on the `launch.bat`.

To execute any of the makefile commands:

- ▶ Type the command in the command shell opened with `launch.bat`.

4.3.3.2.2. Defining the name of the project

The variable `PROJECT` holds the name of the project. Several other variables in the build environment are derived from the project name. For example, the name of the binary executable starts with the project name.

Usually the project and the name of the base directory of the project are the same.

To manually set up a different name:

- ▶ Set the parameter `PROJECT` to the name desired, e.g. `MyProject` as shown in the example below.



Example 4.4. Renaming the project to `MyProject` with the aid of the `PROJECT` parameter

```
PROJECT = MyProject
```

4.3.3.2.3. Defining C files to be compiled

The `CC_FILES_TO_BUILD` variable allows you to define a list of C files to compile.

To define the list of C files:

- ▶ Set the parameter `CC_FILES_TO_BUILD` to the file names as shown in the example below.



Example 4.5. Defining C files to be compiled with `CC_FILES_TO_BUILD`

```
C_FILES_TO_BUILD += \  
$(PROJECT_ROOT)/source/application/test/file1.$(CC_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/test/file2.$(CC_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/test/file3.$(CC_FILE_SUFFIX)
```

4.3.3.2.4. Defining a list of assembler files

The `ASM_FILES_TO_BUILD` variable allows you to define a list of assembler files, which are processed by the build environment.

To define the list of assembler files:

- ▶ Add all assembler files you want to process to the variable `ASM_FILES_TO_BUILD` as shown in the following example.

The list of assembler files is defined.



Example 4.6. Defining a list of assembler files

```
ASM_FILES_TO_BUILD += \  
$(PROJECT_ROOT)/source/application/test/file7.$(ASM_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/test/file8.$(ASM_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/test/file9.$(ASM_FILE_SUFFIX)
```

The file suffix depends on the compiler makefile plug-in you choose.

The `<filename>_ASM_OPT` variable can be used for ASM files in the same way as for C files.

4.3.3.2.5. Building libraries of compiled object files

It is possible to organize the compiled object files in libraries. Such a library can then for example be delivered to a customer and linked against an application.

To build libraries of compiled object files, follow the instructions in step 1 and step 2.

Step 1: To build a library, the first step is to define a list of libraries that are to be built. The variable `LIBRARIES_TO_BUILD` holds a list of libraries to build.

To define, which libraries are added to the list:

- ▶ Set the parameter `LIBRARIES_TO_BUILD` to the library name(s) as shown in the example below.

The `LIBRARIES_TO_BUILD` now stores the list of libraries to be built.



Example 4.7. Defining the list of libraries

```
LIBRARIES_TO_BUILD += BrakePedalSensor SWC_IndicatorAtomic
```

Step 2: The second step when building a library is to define a list of source files. For each library on your list, you need to define a list of source files in the `<library_name>_FILES` variable. The build environment compiles all files from this list and adds them to the library.

To define the list of source files:

- ▶ Set the parameter `<filename>` you defined in step 1 and add `_FILES =` to the name.
- ▶ Add the source files as shown in the example below.

The list of source files for the first library is now defined.

- ▶ Repeat this step as shown in the example below until you have defined all source files for all libraries.



Example 4.8. Defining the list of source files

```
BrakePedalSensor_FILES = \  
$(PROJECT_ROOT)/source/application/blinker_main.$(CC_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/blinker_init.$(CC_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/blinker_cyclic.$(CC_FILE_SUFFIX)
```

```
SWC_IndicatorAtomic_FILES = \  
$(PROJECT_ROOT)/source/application/light_main.$(CC_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/light_on.$(CPP_FILE_SUFFIX) \  
$(PROJECT_ROOT)/source/application/light_off.$(ASM_FILE_SUFFIX)
```

The file list may contain C, C++ and assembler files.

The result of step 1 and step 2: two library files will be created in the directory `$(PROJECT_OUTPUT_PATH)/lib`:

1. `BrakePedalSensor.<toolchain specific lib extension>`
2. `SWC_IndicatorAtomic.<toolchain specific lib extension>`

In the following, two typical examples of makefile targets are explained in detail. Refer to [Section 4.3.3.3.2, “Other make targets”](#) for a list of other make targets.

To delete the library files in `$(PROJECT_OUTPUT_PATH)/lib`:

- Type **make clean** in the command shell and hit **Enter**.

4.3.3.2.6. Linking libraries to a project

If you want to link binary libraries to your project without building them, add them to the variable `LIBRARIES_LINK_ONLY`.

To link libraries to your project:

- Set the `LIBRARIES_LINK_ONLY` variable to the name of the library to be included as shown in the example below.

The library is now *linked* to your project rather than included.



Example 4.9. Linking a library

```
LIBRARIES_LINK_ONLY += c:/Programme/matlab/libs/matlab.$(LIB_FILE_SUFFIX)
```

The extension of the library file depends on the compiler makefile plug-in you choose.

4.3.3.2.7. Linking binary objects to a project

You may link binary object files to your project by adding them to the variable `OBJECTS_LINK_ONLY`.

To link a binary object to your project:

- ▶ Set the `OBJECTS_LINK_ONLY` variable to the name of the binary object as shown in the example below.

Your binary object is now linked to your project.



Example 4.10. Linking a binary object

```
OBJECTS_LINK_ONLY = c:/Programme/matlab/libs/matlabobj.obj
```

4.3.3.2.8. Extending the include path with project-specific path names

You may extend the include path with project-specific path names by extending the variables `CC_INCLUDE_PATH` for C files, `CPP_INCLUDE_PATH` for C++ files and `ASM_INCLUDE_PATH` for assembler files.

To extend the include path with project-specific path names:

- ▶ Depending on the type of file, set the variable
 - ▶ `CC_INCLUDE_PATH` for C files,
 - ▶ `CPP_INCLUDE_PATH` for C++ files,
 - ▶ or `ASM_INCLUDE_PATH` for assembler files
- to the include path as shown in the example below.

The include path is now extended.

The build environment adds the additional path names when invoking the C-compiler, the C++-compiler, or the assembler compiler.



Example 4.11. Defining the include path with `CC_INCLUDE_PATH`

```
CC_INCLUDE_PATH = \  
$(PROJECT_ROOT)/source/common/include \  
$(PROJECT_ROOT)/source/diag/include \  
$(PROJECT_ROOT)/source/network/include
```

4.3.3.2.9. Selecting a board support package

Via the variable `BOARD` you may select a board support package for your project. For details on board support packages, see [Section 4.3.2.5, “Board support packages”](#). Depending on your target device, one or more board support packages are available.

To select a board support package:

- ▶ Set the `BOARD` variable to the board support package of your choice as shown in the example below.

In the example, the evaluation board `eva168_2` is selected.

The board support package is now selected.

The build environment includes the makefile `$(PROJECT_ROOT)/source/boards/$(BOARD)/$(BOARD).mak` to find instructions specific to this board.



Example 4.12. Selecting a board support package

```
BOARD ?= eva168_2
```

4.3.3.2.10. Selecting a toolchain/a different compiler

Every target defines a default toolchain in the build environment.

You can have different compiler plug-ins for a specific target. Each compiler plug-in supports one toolchain. If you want to select a toolchain, set the variable `TOOLCHAIN`.

To select a different compiler/another toolchain:

- ▶ Set the `TOOLCHAIN` variable to the name of the compiler as shown in the example below.



Example 4.13. Selecting a different compiler

```
TOOLCHAIN := tasking
```

4.3.3.2.11. Extending compiler and assembler options

You can extend the options for the assembler, the C compiler and the C++ compiler by adding values to the variables `ASM_OPT`, `CC_OPT`, and `CPP_OPT`, respectively.

To extend the compiler options:

- ▶ Set either of the following, depending on which compiler options you are extending:
 - ▶ `ASM_OPT` for the assembler
 - ▶ `CC_OPT` for C compilers
 - ▶ or `CPP_OPT` for C++ compilersto the option by which you want to extend the compiler. The following example shows you how to do this.
- ▶ Your compiler option is now extended.

The build environment will send these values to the appropriate tool.



Example 4.14. Extending the C compiler option

```
CC_OPT += -ggdb
```

4.3.3.2.12. Adding preprocessor defines

If you need to define additional preprocessor definitions, you should not add them to `CC_OPT` but use the key/ value approach supplied by the build environment. The reason for this is that in most cases the additional definitions have to be available for the processing of dependencies and the processing of the source files.

All preprocessor definitions that you define as key/ value pairs are promoted by the build environment to every build step.

To set up a key/ value pair:

- ▶ Set the `PREPROCESSOR_DEFINES` to your values, e.g. as shown in the example below.



Example 4.15. Setting up a key/value pair

```
PREPROCESSOR_DEFINES += myDefine  
myDefine_KEY         := MY_DEFINE  
myDefine_VALUE       := TRUE
```

This results in the following command line option:

```
-DMY_DEFINE=TRUE
```

This command line option is passed to the preprocessor *and* to the compiler.

4.3.3.3. Compiling the application

The pink field in the workflow overview indicates where in the workflow you currently are:

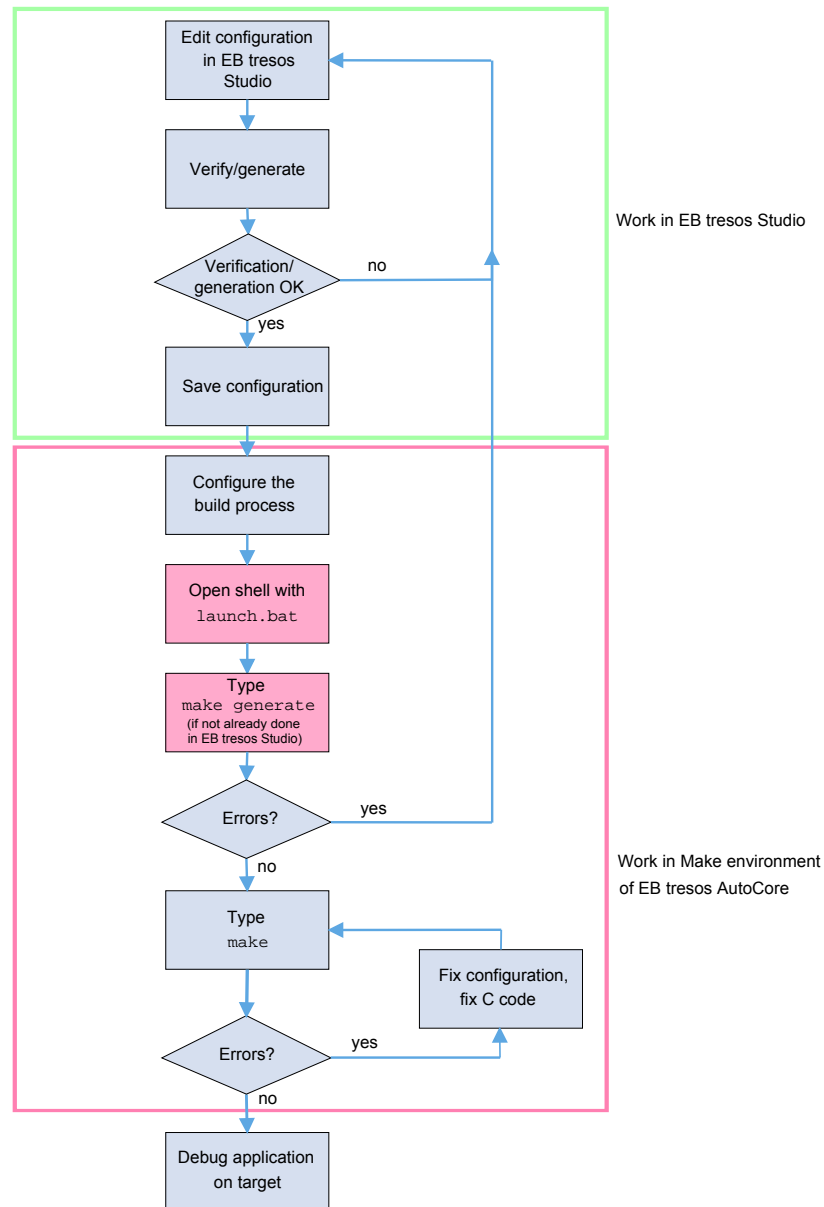


Figure 4.5. Workflow step: compiling the application

To generate, compile, and link your application from scratch:

1. Start the `launch.bat` batch file from the `$(PROJECT_ROOT)/util` directory.

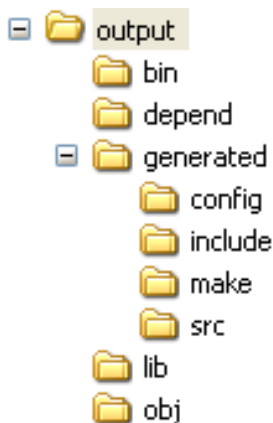
The command shell opens up.

2. To generate all configuration-dependent files, type **make generate** into the command shell.
3. To process the dependency information and build the binary executable, type **make** into the command shell. To speed up compilation, you can use GNU Make's built-in job parallelization by calling **make -j**.

The target **make** processes the dependency information by analyzing all source files by the GCC pre-processor, compiles all objects files, builds all libraries and links the application binary.

4.3.3.3.1. Locating output directories

You may find all folders and files generated by **make generate** and **make** in the `$(PROJECT_ROOT)/output` directory of your project:



The following list gives you a short overview of the content of the different output directories:

- `bin`: Contains the binary executable file.

The format and the filename extension differ depending on the compiler toolchain you choose. Files in the ELF format often have the extension `.elf`. Some toolchains also create other files here, e.g. HEX files that can be used with specific FLASH programming tools.

- `depend`: For every source file that is processed in the dependency processing, a makefile fragment is created in this directory.

This fragment defines on which other files a source file depends. For a file `Can.c` e.g. **make depend** creates a file `Can.mak` in this directory.

- `generated`: All output files of the generator tools go into this directory. The directory contains some subdirectories such as `source` and `include` to organize the generated files.

- ▶ **lib**: This is the default output path for libraries. All libraries are archived in this directory, unless you define an alternate output path for a library.

4.3.3.3.2. Other `make` targets

The build environment provides some more targets that are important for your everyday work. This chapter will give you an short overview of these `make` targets.

make show_rules

make show_rules gives an overview of all available targets. You can register additional messages to display information about specific targets you have implemented.

make single_file SF=filename

make single_file compiles only one certain source file into an object file. This target is useful to verify only a single source file.

make single_lib SL=libname

make single_lib compiles and links only those source files that are required to build a certain single library. This target is useful when you need to recompile only one library because of changed settings.

make single_lib_clean SL=libname

make single_lib_clean removes the library, all object and dependency files that are needed to build a certain single library. This target is useful when you need to recompile libraries because of changed compiler switches or a toolchain update.

make clean

make clean removes files created by the compiler (including dependency information) or by the build environment (deletes folders `bin`, `depend`, `lib`, `obj`, `make` in directory given by `PROJECT_OUTPUT_PATH`). You can use this target to rebuild all your application code with **make** without the need to run **make generate**.

make clean_all

make clean_all deletes all files generated by the compiler, the build environment, or by the code generator.

make clean_all deletes the directory where files are placed created by a code generator. The path is set in EB tresos Studio and is provided as the makefile variable `GEN_OUTPUT_PATH`. If the default path has been changed in your project, the variable must also be set in `launch.bat`.

5. BL module references

5.1. Overview

This chapter provides module references for the BL product modules. These include a detailed description of all configuration parameters. Furthermore this chapter lists the application programming interface with all data types, constants and functions.

The content of the sections is sorted alphabetically according the EB tresos AutoCore Generic module names.

For further information on the functional behavior of these modules, refer to the chapter BL user's guide.

6. Bibliography