A. Introduction

1. This build system can be run on Windows and Linux hosts. Usually the hosts is chosen according to the tools and utilities that are used in specific project.
2. This build system is based on Makefiles, so shell(Linux) or command line(Windows) is natural environment to run it. You can use any IDE that support custom build command. In custom build command you just need to call 'make' utility with proper target. (will be explained in details in one of following sections)

B. Mandatory requirements.

1. Make utility

   Build system is based on GNU Make utility. The required version is at least 4.1.

   i. Windows:

      Make utility that built in Windows OS is very limited and so not supported!

      Usually, the Make utility for windows is located in:

      $(YOUR_PATH)/uCWorkspace/tools/windows/make

      for example: $(YOUR_PATH)/uCWorkspace/tools/windows/make/make4.1

   ii. Linux:

      Usually make utility is sytem wide. Only make sure that version is as required.

   You can change the default location of Make utility by editing the file

   $(YOUR_PATH)/uCWorkspace/uCProjects/workspace_config.mk.

   Uncomment/add/change variable REDEFINE_MAKE_PROGRAM_DIR and assigning to it the path of Make utility folder.

   For example: REDEFINE_MAKE_PROGRAM_DIR = c:\make4.1 .

   Make sure that this folder contains 'bin' sub-folder with 'make' executable.

2. GIT

   GIT is critical component. It's used not only for source control but for synchronization of repositories.

   Two options available:

   i. install system wide GIT.

      a) Install from official online source.

      b) After installing check that installation succeed: open 'cmd' or 'shell' window and run 'git'command. You should see git help/usage output. If command not found then system PATH needs to be fixed.

   ii. Use GIT located in particular folder.

      Edit file $(YOUR_PATH)/uCWorkspace/uCProjects/workspace_config.mk by uncomment/add variable REDEFINE_GIT_ROOT_DIR and assigning to it the path of GIT folder. For example:

      REDEFINE_GIT_ROOT_DIR = c:\my_git_dir

C. Optional requirements.

1. KConfig utility

   Project configuration is based on .config files.

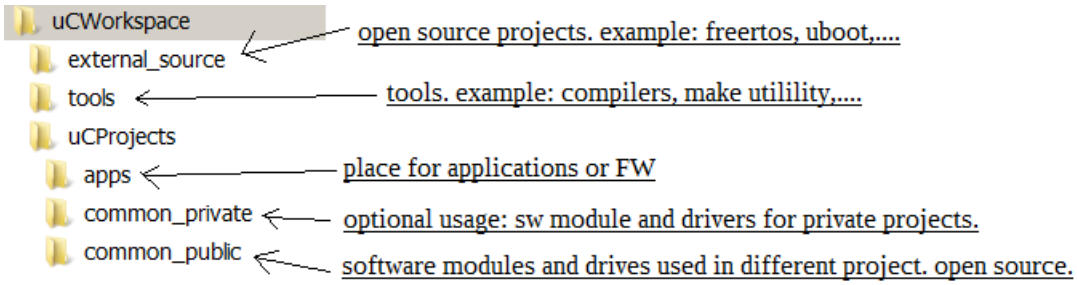
   You can find more about KConfig language in https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html

i. Windows:
   Usually the kconfig utility located in $(YOUR_PATH)/uCWorkspace/tools/windows/kconfig folder for example: $(YOUR_PATH)/uCWorkspace/tools/windows/kconfig/kconfig3.12.0
ii. Linux:
   Usually kconfig is installed system wide during installation of build-essential for kernel. Check according to you distribution.

You can change the default location of KConfig utility by editing the file
$(YOUR_PATH)/uCWorkspace/uCProjects/workspace_config.mk.
Uncomment/add/change variable REDEFINE_KCONFIG_DIR and assigning to it the path of KConfig utility folder.
For example: REDEFINE_MAKE_PROGRAM_DIR = c:\kconfig .
Make sure that this folder contains 'kconfig-mconf' executable.

D. Workspace folder structure.

1. Note for Windows: Don't put uCWorkspace folder in too deep path, because Windows has limitation for length of path name.



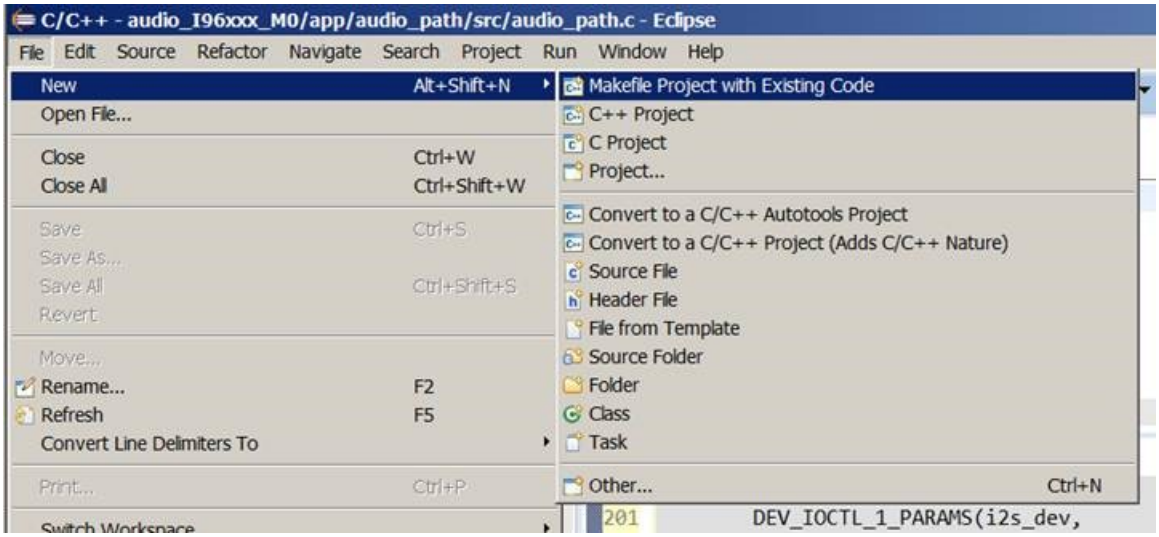
2. For creating folder structure from scratch, you can run create_workspace_tree.bat(in Windows) located in 'common_public' folder.
   Running this .bat file will create workspace_config.mk file with sample configuration.

E. Compilation of project.

1. From command line or shell.
   i. Compiling from command line is highly recommended during bring-up of workspace or if build process from IDE is not performed as expected.
   ii. Go to $(YOUR_PATH)/uCWorkspace/uCProjects/apps/your_app
   iii. Run 'make all'.

   Note 1: For Windows you cannot run 'make all' because Windows limited built-in of make.exe will run, so you need to run GNU make.exe version:
   $(YOUR_PATH)/uCWorkspace/tools/windows/make/make4.1/bin/make.exe all
   Note 2: If you see 'err: shell command is too long on Windows systems' then the most common reason is the location of uCWorkspace is too deep. So the easiest solution is to move it to higher level folder, for example to 'C:\ or C:\work\'

2. From eclipse.
   i. Adding project to Eclipse.
      a) Go to menu -> File -> 'Makefile Project with Existing Code'

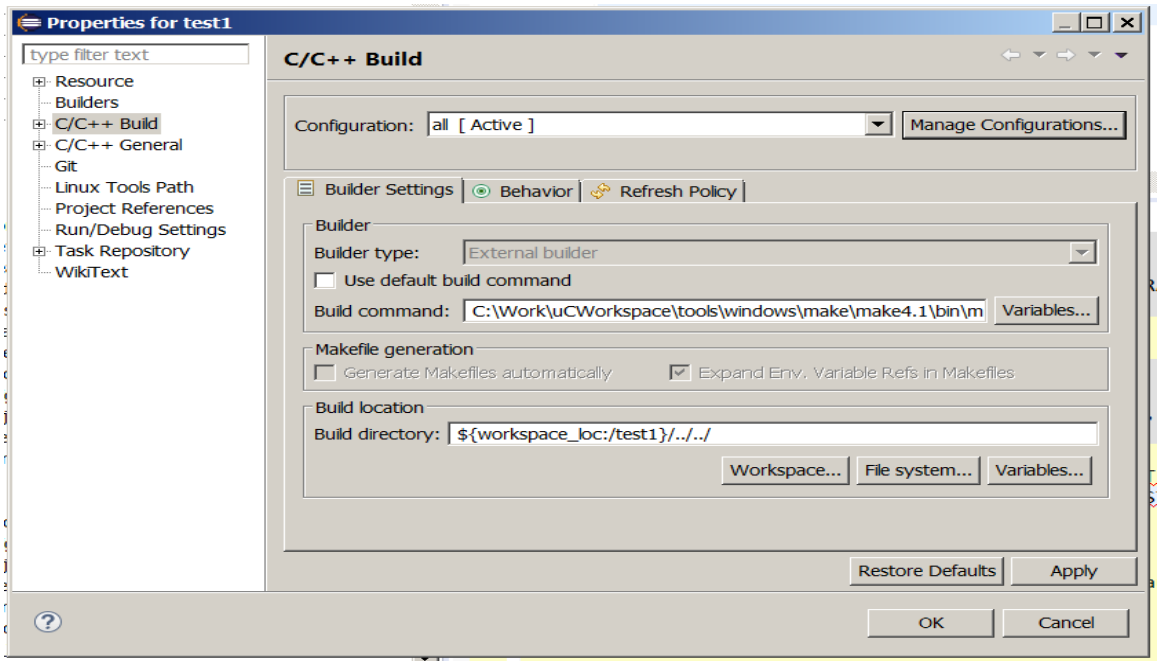


      b) Fill 'Project Name' and 'Existing Code Location'.
      The location should point to folder that will contain eclipse project files, **NOT** application folder.
      The recommended location is in app_path/zIDE/ because zIDE is mentioned in .gitignore file.
      Sometimes the proper project already exists in zIDE folder, then you can try to use it.

ii. Right click on project and select ‘Properties’. In properties window go to ‘C/C++ Build’

a) Unselect ‘Use default build command’. In ‘Build command:’ put ‘make’ in linux or ‘$(YOUR_PATH)/uCWorkspace/tools/windows/make/make4.1/bin/make.exe’ in Windows. To accelerate build process you can add ‘-j4’ flag.

b) Build location should point to application root folder (the one that contains Makefile). It can be relative to eclipse project folder (for example ‘${workspace_loc:/test1}/../../’) or absolute path (for example C:\work\uCWorkspace\uCProjects\apps\test1).

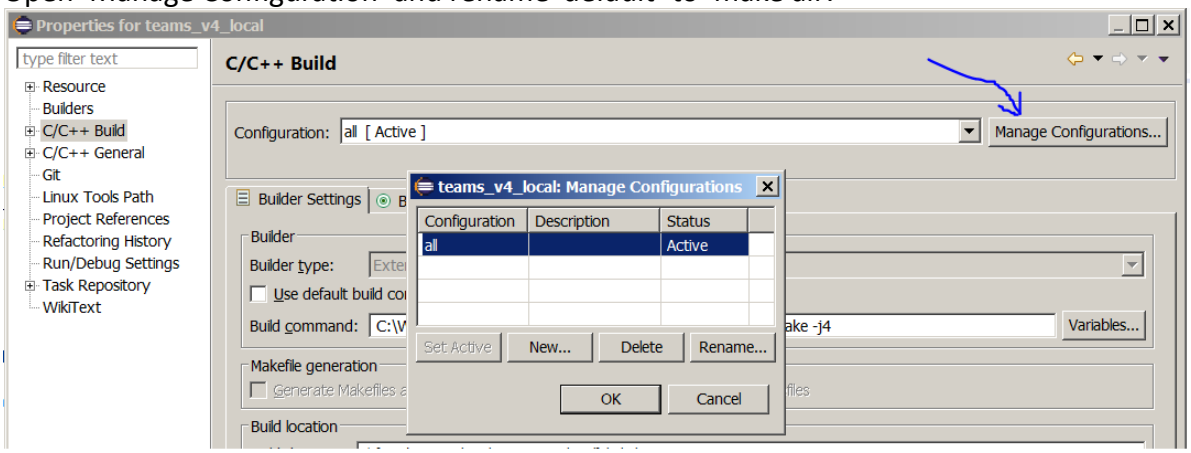


Note: if you got following error pattern when trying to build, then try to use absolute path:

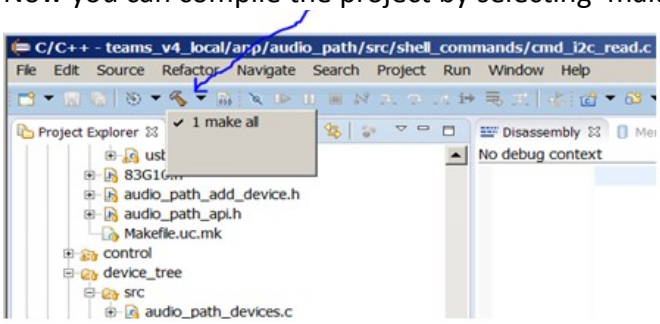
*13:11:52 **** Build of configuration all for project test1 *****
"D:\\make\\make4.1\\bin\\make" -j4 all
*make: *** No rule to make target 'all'. Stop.*

c) Make sure that ‘Generate Makefile automatically’ is un-checked.

iii. Open ‘Manage Configuration’ and rename ‘default’ to ‘make all’.



Now you can compile the project by selecting ‘make all’ in eclipse toolbar->build button:



F. Cleaning of project.

1. Go to $(YOUR_PATH)/uCWorkspace/uCProjects/apps/your_app
2. Run make clean.
   For Windows you cannot run ‘make clean’ because Windows limited built-in of make.exe will run, so you need to run GNU make.exe version:
   $(YOUR_PATH)/uCWorkspace/tools/windows/make/make4.1/bin/make.exe clean

G. Build process.

1. Build system based on Makefiles and starts from Makefile located in application folder. The most of the work is done by makefiles located in common_public/build_tools.

2. During the first steps of build process, the host system is checked, the required tools, utility and files are searched and build variables are set.

3. On second step the **.config** file of application is parsed. This file declare the configuration of project. The structure and creation of this file is based on KConfig utility. You can find more about KConfig language in https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html .

4. Then project name (CONFIG_PROJECT_NAME) is extracted from .config file and following is verified:
   i. The name of the project should be unique. The build process scan all workspace for .config files to check that there is no project with the same name.
   ii. The name of current application git branch should start with project name CONFIG_PROJECT_NAME.
   for example: CONFIG_PROJECT_NAME = "foo_project" then following git branch names are valid:
   foo_project, foo_project123, foo_project_v2021.12.35 (last option is recommended)

5. Then git commit and branch name is tested in common_public. Following is verified:
   i. The current GIT hash number of common_public is compared with CONFIG_COMMON_PUBLIC_GIT_COMMIT_HASH in .config.
   ii. The branch name of common_public is compared to branch name of application.

If hashes are same **and** branch names are same then build process continue, if not, then build process is stopped and warning message displayed.

To continue the build, you need to checkout the commit of common_public required by CONFIG_COMMON_PUBLIC_GIT_COMMIT_HASH. The checked-out branch name of this commit should be the same as current branch name of application git repository.

Comments:

a) The manual checkout is implemented as a protection against mistakenly compiling wrong project.
b) Warning message display hint that you can use to checkout correct branch using command line.
c) If needed common_public commit cannot be found, then you need to update your common_public.

6. Then the workspace is scanned for Makefile.uc.mk. The following folders are scanned: application folder and sub-folders, common_public folder and it's sub-folders, and optional common_private folder and it's sub-folders.

7. In each Makefile.uc.mk the INCLUDE_THIS_COMPONENT variable is checked. If it equal to 'y', then the corresponded SW component will be included in build process. In most components of common folders INCLUDE_THIS_COMPONENT will be set according to configuration in .config file.
   All found components are listed in: $(APP_FOLDER)/z_auto_generated_files/all_found_components.mk .
   All components that will be included in the build are listed in:
   $(APP_FOLDER)/z_auto_generated_files/include_components.mk .

8. After the scan is completed the components the components in include_components.mk file are analyzed, the required GIT repositories are analyzed and required commits are checked-out, additional build variable are set according to .config file and the build process starts.

9. The compilation will visit folder of all included components and compile the files according to Makefile.uc.mk of each component.

H. Adding new files to compilation

1. Adding file to existing component:
   Open Makefile.uc.mk and add additional line: `SRC += new_file.c`

2. Adding new component:
   i. Create new folder.
   ii. Create Makefile.uc.mk.
   iii. In Makefile.uc.mk add INCLUDE_THIS_COMPONENT variable and assign to it 'y' if you want the component to be compiled always, or some another Makefile variable that will create more sophisticated condition when to include the component into build.
   iv. Add and populate SRC, CFLAGS, INCLUDE_DIR. You can look on other Makefile.uc.mk files as reference.