Platform SDK Build System Architecture

Requirements

Gnu make version 3.81 or later is required.

build_sdk

The primary interface to the SDK build system is through the `build_sdk` script in the `tools/` directory. It provides an easy way to build the SDK while validating configuration selections. It can also build all available configurations, or just a subset, in one invocation. Of course, building a single configuration is also supported.

Running make directly

If you want more control, you can run make directly:

```
make target=mx6dq board=evb boardrev=a test=caam
```

`boardrev` defaults to 'a' if not present. `test` is only used by sdk_unit_test. These variables can also be set in the environment if you don't want to pass on the command line. For instance:

```
$ export target=mx6dq
$ export board=sabre_ai
$ export boardrev=b
$ make
```

These variables are actually aliases for the variables `TARGET`, `BOARD`, `BOARD_REVISION`, and `TEST`. They are meant to make it easier to type make command lines in a shell. Either style will work equivalently.

Additional variables you can pass on the command line or via the environment:

- `BUILD_SDK_COLOR` enables or disables colorized output on terminals. A value of 1 enables and a value of 0 disabled. The default is 1. Even if color output is enabled, the color escape codes will only be printed when outputting to a terminal.
- `VERBOSE=1` turns on printing the full command line for compilation, etc.

Supported top-level targets:

- all: build sdk lib and all apps (default)
- sdk: just the sdk library, no apps
- board: build only the board specified by configuration variables
- <*app name*>: build just one app and the sdk lib
- clean_sdk
- clean_board
- clean_<*app name*>

The targets can also be passed to the -a option of build_sdk. But you can pass multiple targets to make, and it will execute each one in order. So you can say, "make clean_board sdk_unit_test …" to rebuild the board lib and relink sdk_unit_test.

However, you can't build an app but not the sdk lib, because the apps depend on the sdk. But unless you are changing files in the sdk, it should take less than a second to check dependencies. (One major benefit of having just a few makefiles, i.e., only one for all of sdk/drivers, is that builds are significantly faster. Recursive make is very slow.)

Makefile structure

All makefiles should follow this structure:

```
include $(SDK_ROOT)/mk/common.mk

### Define control variables here ###

include $(SDK_ROOT)/mk/targets.mk
```

The first include of `common.mk` defines a bunch of common variables and process configuration settings. These include paths, compiler info, compiler flags, and so on. The `targets.mk` file included at the bottom of the makefile contains all of the recipes that actually do things like compile source files and link the application.

In between the two include directives, you declare variables that tell `targets.mk` what to do. To compile source files, set the `SOURCES` variable. To add objects to a library, set the `TARGET_LIB_NAME` variable. And so on.

Only the top-level makefile cannot use the `SDK_ROOT` variable to locate `common.mk`. This variable is available to all child makefiles, as `common.mk` exports it as soon as it is set.

Adding an application

The steps involved in adding a new application to the SDK are fairly simple.

First, create the new application's directory under `apps/`. The new app directory should have a `src` directory under it.

Second, copy the `Makefile` from the power_modes_test application into the root of your new app directory. This is in most cases a better example makefile than the one for sdk_unit_test, as it is much simpler. It assumes that the only source files compiled directly into the application are listed locally in the `SOURCES` variable.

Update your new app's `Makefile` to include all the necessary source files and libraries. For most apps, `LIBSDK` and `LIBBOARD` will be the only libraries you need, and these are already present in the power_modes_test makefile.

Finally, add the app's name to the `ALL_APPS` variable in the top-level `Makefile`. This is necessary to be able to build your app.

Makefile variables

Configuration variables

These variables are passed into make on the command line.

Variable	Description
`TARGET`	Chip name, i.e. "mx6dq", "mx6sl", etc.
`BOARD`	The board name.
`BOARD_REVISION`	Board revision, i.e., "a", "b", etc. Defaults to "a" if not provided.
`TEST`	Used by the sdk_unit_test application to choose which test is run. The name of the unit test function to execute is this variable's value plus "_test". If not provided, this variable defaults to "ALL".
`target`	Alias for `TARGET`.
`board`	Alias for `BOARD`.
`boardrev`	Alias for `BOARD_REVISION`.
`test`	Alias for `TEST`.

Log control variables

Variable	Description
`BUILD_SDK_COLOR`	Set to 1 to turn on color output for terminals. Logging to a pipe or redirection to a file will never use color.
`VERBOSE`	Setting this to 1 will cause the full command lines of compile and link commands to be logged.

Path variables

Many of these variables depend on the configuration variables.

Variable	Description
SDK_ROOT	Top level of the SDK source tree. Dir containing sdk, apps, board, mk, doc, etc.
SDK_LIB_ROOT	The sdk source directory.
APPS_ROOT	The apps directory.
BOARD_ROOT	The board directory.
OUTPUT_ROOT	Target-specific output directory, like ./output/mx6dq.
LIBS_ROOT	Libs dir in OUTPUT_ROOT.
LIBS_OBJ_ROOT	obj dir for libraries.
APP_OUTPUT_ROOT	Dir containing binaries for the current app. Depends on APP_NAME and BOARD_WITH_REV.
APP_OBJS_ROOT	obj dir for the current app.

SDK_ROOT is exported into the environment, so it is available to all child invocations of make prior to actually including common.mk.

There are also a few variables that are set to the full path to common libraries.

Variable	Description
LIBSDK	libsdk.a path
LIBBOARD	libboard_<*board*>_rev_<*rev*>.a

Control variables

These variables are used by targets.mk to control actions such as compilation, adding objects to a library, and linking an application.

Variable	Description
SOURCES	List of source files to compile.
APP_NAME	Name of the current app being built.
TARGET_LIB_NAME	Name of the library being built. Not a full path.
SUBDIRS	List of relative or absolute directories to invoke make in.
LINK_APP	Set to 1 to enable linking of the app executable.
ARCHIVE_APP_OBJECTS	Enables placing compiled objects in an archive even when not building a library.

The SOURCES variable is set to a list of absolute or relative paths to source files. The files can be located anywhere underneath the SDK source tree. All types of source files (.c, .cpp, .s, etc) go into this one list, and targets.mk sorts out how to build them.

The definition of the APP_NAME variable determines whether an app or a library (.a) will be the target of the makefile.

If APP_NAME is not set, then the target is a library determined by TARGET_LIB_NAME. Libraries in the SDK are by definition not board specific. They go in the output/<target>lib/ directory.

If APP_NAME is set, then we're building an application. The LINK_APP variable must be set (to "1", or "true", etc) in order to actually link the application into an executable .elf file.

Because you may want to include sources from one or more child makefiles in your application, the build system supports an application library. Setting ARCHIVE_APP_OBJECTS to 1 will enable this. APP_NAME must also be set. In this mode, sources are added to an archive file in the application output directory. If ARCHIVE_APP_OBJECTS is set and LINK_APP is set, the application library is linked into the application .elf.

The SUBDIRS variable contains either relative or absolute paths to directories holding makefiles that will be executed as children of the current makefile. Subdirectories are processed before any items in the current directory, such as source files or linking an application.

Compilation variables

Variable	Description

CFLAGS	C compiler flags.
CXXFLAGS	C++ compiler flags.
DEFINES	C preprocessor macro definitions.
INCLUDES	C include paths.

Linker variables

Variable	Description
LD_FILE	Linker script used to link application.
LIBRARIES	List of libraries to link into the application.
LDADD	List of "-l" arguments passed to linker to add standard libraries.
LDINC	List of "-L" library include paths passed to linker.

The LD_FILE, LIBRARIES, LDADD, and LDINC variables are used in the linker command line to link the application. LD_FILE is the path to the .ld linker script.

If the LD_FILE name ends with ".S", the file will be passed through the C preprocessor to produce the final linker script. The DEFINES and INCLUDES variables are passed on the command line for the C preprocessor, so you can make use of the same preprocessor macros available to C code.