NxM

"N by M": Creating Software Distributions Consisting of N Different Projects, Built M Different Ways

by Frank Sheeran

The master copy of this document resides in src/doc/NxM_Make_System.doc but is re-rendered to PDF andGithub Markdown format periodically for your convenience. The master copy has diagrams and other useful formatting that are missing in the Markdown version.

1 Executive Summary

Goals

- build a distribution-ready SDK (software development kit), with multiple binaries, multiple libraries, and potentially many scripts, documents, headers, and example programs
- build that distribution on different architectures (OS, CPU), with different configurations (debug, optimized, different compilers, etc.)

Solution

- a methodology of separating information in Makefiles based on what information varies by Solution, Project, Architecture, Configuration, Site, and Machine.
- a methodology of separating source directories, compilation directories, and final distribution directories
- a set of Makefiles for the software builder "GNU make" that implement these methodologies

Applicability

- primarily for Linux and Unix
- primarily projects with at least some C++ and C
- minimum project size: if you create at least two libraries or binaries, or need to build at least two different ways (different OS or OS release, debug vs. optimized, etc.)
- maximum project size: not designed for systems so big that they have a Solution, composed of multiple Projects, that themselves are broken into yet smaller Sub-Projects.

2 Terminology and Notation

Understanding the terminology of NxM will take you much of the way to understanding NxM.

NxM "N by M" refers to the fact that we're building N Projects each a total of M times (for various Architectures and Configurations).

Solution Used in the Microsoft Visual Studio sense: a group of Projects that will comprise a product Distribution.

Project Used in the Microsoft Visual Studio sense, although a bit bigger: a group of files that will create any combination of

1. one (or more) big executables composed of multiple source files;
2. many small executables each composed of a single source file;
3. a big library composed of multiple source files;
4. many small libraries each composed of a single source file; *and/or*
5. some number of scripts, headers, example files, and/or documents

Visual Studio projects are limited to either a single library or single binary each, so if you have a lot of unit test programs you have a lot of Visual Studio projects. In contrast, NxM lets you group such things together in a single Project: a library project could also have 20 little unit test binaries, for instance.

Architecture An OS and CPU combination. This could be widened to different choices of compiler or OS release (old and new versions, or GNU and clang, for instance) but that can also be handled with Configuration.

Configuration Used in the Microsoft Visual Studio sense: different ways to build the given projects on the given architecture, such as debug vs. optimized, or using different compiler and linker options including preprocessor settings.

Site A set of computers that are administered the same way, will have various third-party software installed in the same locations, and so on.

Machine The actual computer. We use the output of `hostname` as proxy for this.

Distribution All Projects, compiled for all Configurations of all Architectures, along with the Projects' non-compiled product (headers, scripts, examples, docs, etc.) placed into a single directory heirachy, ready for acceptance testing, and dispersal to the customer/user base. NxM allows you to keep several Distributions side by side if you want.

Partial Makefile A sub-Makefile intended to be included as part of main Makefile. For instance, it may have information very specific to a single Project, and know nothing about other Projects, nor about the Solution in general, Architecture, etc. NxM is based on this technique.

When the above terms are capitalized in this document, they are being used in the specific senses given here.

When the above terms are used inside curly braces, they are meant as stand-ins for the name of an specific name of the given class. In other words, they're variables. Where you see `{Project}`, it is probably the name of one major binary or library in your distribution.

3 Overview

NxM is implemented entirely with makefiles and "partial makefiles" for **gmake**. A "partial makefile" is gmake input that isn't a standalone makefile, but rather must be included by an actual makefile.

NxM's makefiles fall into two types. The first, simple type isn't involved directly in compilation but rather just cleanup, mainly, and recurses to all subdirectories as appropriate. Every directory has such a **Makefile**, except the directories where Projects are actually compiled.

The directories where Projects are compiled instead have the other **Makefile** type, which is the key to NxM. It is able to compile several big and many small binaries, one big and many small libraries, and copy those, along with headers, example programs, and so on, into a directory ready for acceptance testing and distribution.

The **Makefile** used for this is trivial and generic. First, it sets a **VPATH** to point to a cousin directory which has the source code in it. This is done because we'll compile in a different directory than where the source is. One copy of the source is thus shared between myriad Architectures and Configurations.

Then, it includes a list of Partial Makefiles located around the directory heirarchy. Here is the entire functional text of this **Makefile** as of this writing:

```
ProjName  = $(notdir $(shell pwd))
MachName  = $(shell hostname)
VPATH     = ../../../src/$(ProjName)

include ../../../src/make/Make.$(MachName)
include ../../../src/$(ProjName)/Make.Project
include ../../../Make.Solution
include ../../Make.Architecture
include ../Make.Configuration
include ../../../src/make/Make.Common
include ../../../src/make/Make.Project.Common
include $(wildcard $(DependMakefiles))
```

These included Partial Makefiles do the following job:

Make.Solution — Defines anything that is common to all Projects, but not machine-specific, etc. Perhaps all projects need to use MySQL, for instance.

Make.Project — Defines what big and small binaries, and big and small libraries, that should be built, from what, and which headers, example programs, and scripts should be installed. Nothing here should be specific to the machine, compiler choice, and so on.

Make.Architecture Defines anything that is specific to this *kind* of computer: not things like where third-party software is installed, which can vary by Machine, but rather things specific to, say Linux, or Fedora Linux, or Fedora release 40.

Even if two OS releases have the same settings, you would still define two architectures if building on the two would produce different results that cannot be mixed and matched. For instance, software built on Fedora 40 may not run on Fedora 39, because it may depend on various shared libraries whose versions were incremented in the move to 40 and not present on 39. In this case you'd probably need two architectures, even though identical `Make.Architecture` files might work fine.

There is a grey area as to whether some settings pertain to Architecture or to Configuration. For instance, you might define your Architecture to use GNU compilers, and have two Configurations for Debug and Optimized builds. Or, you might have your Architecture limited to settings that are common to both GNU compilers and clang, then have a Configuration that does a GNU build, and another Configuration that does a clang build.

Make.Configuration Best to explain by example: Debug and Optimized builds, or GNU and clang builds, or regular and trace-enabled, 32 and 64 bit, and so on. On a single given Architecture, we're building the software two or more ways, usually with differing compiler flags, though NxM is probably able to cover other differences as well.

Make.Site Settings common to a group of machines. For instance, all the machines in an office may have MySQL version 1.2.3 installed in the same directory. That location and version can be set here, not repeatedly in all the `Make.{Machine}` files.

Make.{Machine} Gives all settings related to this particular machine, such as the locations and versions of various software packages. These settings have nothing to do with the Solution, Projects, etc., except in as much as the locations and other compiler options needed for of any third-party software those rely on should be stated here, not elsewhere.

Make.Common This defines all variables that will be common to all directories of the directory heirarchy. This includes 1) the name of the Distribution, 2) the destinations within the distribution of various file types, and 3) rules to make targets clean, veryclean, and pristine that we want in all directories.

Make.Project.Common This defines all variables that will be used for all projects on all architectures and configurations, for all sites and machines. For instance, it produces a list of compiler flags that includes all those required by the Solution, Project, Machine, Architecture, and Configuration, producing a final list of compiler flags the compiler will actually use.

.depends/*.d GNU compilers and clang support the output of dependencies for a given source file while compiling. Those are stored in a subdirectory out of the way, and ensure that any time a header changes, all code depending on that header will be recompiled while nothing else will.

Note that several of these come in pairs:

- Solution holds elements common to multiple Project's
- Architecture holds elements common to multiple Configuration's
- Site holds elements common to multiple Machine's

4 Case Studies

Algorithm Research, Testing, and Demonstration

The author of this paper is writing another paper on the performance of certain new algorithms. The code must be compiled on a minimum of two different platforms, Windows and Linux, and perhaps subvariants such as 32-bit vs. 64-bit, and maybe with or without certain CPU instructions. It must also be compiled in Debug and Optimized flavors.

Besides the library with a reference implementation of the new algorithm, the project contains unit tests of basic functionality, extended high-load tests, timing tests, documentation, example programs, and some associated scripts.

Since we're using C/C++, making at least two "things," (library and performance test, plus other minor executables) and making them on at least two architectures and/or configurations, use of NxM is indicated.

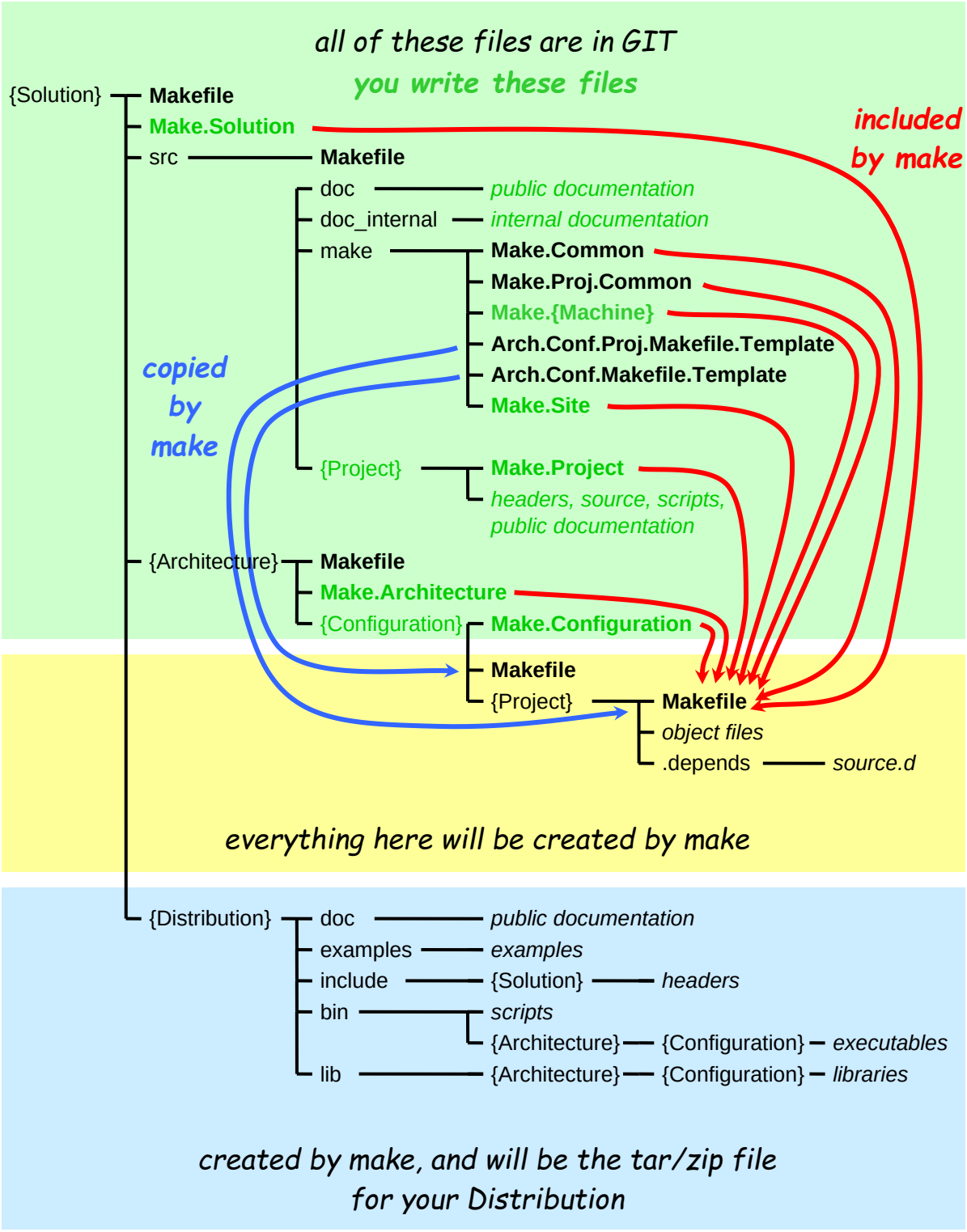
Game Software

We're building a multi-player game with a Linux server; clients for Linux, Windows and iOS; with debug and production versions of all; and which are all based on a portable library used on all three platforms.

5 File Heirarchy and Files

The diagram on the following pages shows: one Solution, with one Project, that is built on one Machine, for one Architecture, for one Configuration. One Distribution is shown. NxM's complexity is overkill when we have only one of each such entity, but should make sense if you imagine having multiple or even many such. (Not all files are shown.)

Note the diagram is visible in the Word 2003 and PDF versions of this document but not in the Markdown.



5.1 {Solution}

This will have:

- `Make.Solution`, which defines variables for the Solution name (deriving it from the directory name) and version number (for you to set by hand)
- `Makefile`, which has rules to create a tar, tar.gz etc., and recurse clean target
- `src` (everything under source code control)
- `{Architecture}` for each architecture you use
- current `{Distribution}` and *optionally* old ones
- *optionally* compressed tar files of one or more Distributions
- *if you're using Git,* a `.git` directory and `.gitignore` file

5.2 {Solution}/src

Everything inside this directory should normally be under source code control, and nothing should normally be generated. The only exceptions would be editor backup files and the like.

You will never compile under this heirarchy. You will never manually copy things from here to distribute to users.

5.3 {Solution}/src/doc and {Solution}/src/doc_internal

Solution-wide document files. Those under `doc` will be copied to the distribution. Those under `doc_internal` will never be. Per-project document files can be put here, or in their relevant {Project} directory.

5.4 {Solution}/src/make

To the extent that NxM is software, this is where all the software can be found.

This subdirectory of src has a master or "template" copy of several `Makefile`'s that the system will copy to new build directories: those that go under {Solution}/{Architecture}/{Configuration} and {Solution}/{Architecture}/{Configuration}/{Project}.

It also holds some common Partial Makefiles: `Make.Common` is included by all other `Makefile`'s, while `Make.Proj.Common` is included by all `Makefile`'s called {Solution}/{Architecture}/{Configuration}/{Project}/Makefile .

There are also Partial Makefiles for the Site and Machine: the optional `Make.Site` or machine-specific `Make.{Machine}` files. Arguably these should be in a directory outside

of this tree, but that would assume that multiple Solutions could leverage the same `Make.Site` or `Make.{Machine}` files, an expectation outside the scope of NxM.

5.5 {Solution}/src/{Project}

You'll have such a directory for each Project. A Project will consist of the following:

- 1-4 binaries produced from multiple C/C++ files *and/or*
- any number of small binaries produced from one C/C++ file each *and/or*
- 1 library produced from multiple C/C++ files *and/or*
- any number of small libraries produced from one C/C++ file each *and/or*
- any number of scripts, public headers, example programs, and documents
- `Make.Project`, a Partial Makefile defining all of the above

The demo contains one Project called `HelloWorld`, which will product a big and small library, plus a big and small binary, and also includes a script and a document.

5.6 {Solution}/{Architecture}

Nothing should be under source code control, except 1) the directory `{Architecture}`, 2) the file `{Architecture}/Make.Architecture`, 3) a subdirectory for each Configuration you need to support, and 4) for each of those the file `{Architecture}/{Configuration}/Make.Configuration` . Optionally you can have more, such as if you need to hand-write a `Makefile` . I don't think a normal project would need to, as the job of this makefile is simply to iterate and build all the Projects.

A huge number of files will be generated here, including but perhaps not limited to:

- `{Architecture}/Makefile` will normally copy each Configuration's `{Architecture}/{Configuration}/Makefile` from src/make/Arch.Conf.Makefile.Template
- `{Architecture}/{Configuration}/Makefile` will normally generate a directory `{Architecture}/{Configuration}/{Project}` and copy from src/make/Arch.Conf.Proj.Makefile.Template to `{Architecture}/{Configuration}/{Project}/Makefile` , to mirror every Project `src/{Project}` that contains a file called `Make.Project`
- all object files (`*.o`)
- dependency files produced by a compiler and used by gmake

5.7 {Solution}/{Distribution}

This can be entirely erased, then entirely rebuilt by running gmake 1) in the src directory then 2) under each `{Architecture}` directory.

Nothing under this directory will be under source code control.

You shouldn't need to run any commands here.

Nothing should be here that you aren't distributing to clients/users.

All executables, libraries, and shared libraries are copied here.

All public documents, scripts, and example programs are copied here.

Header directories are often called `include`, so that is what people look for and would be nice to have. However, it is usually good to put headers in a directory called `{Solution}` (or `{Project}`), so that you can set `-I` to the parent of that directory and write `#include <Solution/Foo.h>` to ensure the compiler doesn't grab a same-named `Foo.h` from some other SDK or library. `Make.Common` thus copies all headers to `include/{Solution}` , but you can edit that `include` or `include/{Solution}/{Project}` or whatever you need.

To create a distribution, `{Solution}/Makefile` is set up with example targets `tar`, `gz`, and `zstd` that will tar up the current state of the current Distribution with commands such as:

```
tar cvz --owner=nobody --group=nobody Solution.1.2 >Solution.1.2.tar.gz
```

6 Running the Demo

You should be able to download and build the demo with the following commands.

Get the source code from github:

```
> git clone https://github.com/FrankSheeran/NxM.git
```

You should now see a new child directory, NxM.

```
> ls
> cd NxM
```

6.1 Walkthrough of the Source

Let's look at the source. The directory **doc** contains a public document we want in our final distribution, plus some license files the rest of the code refers to.

doc_internal has a document that will *not* be distributed.

HelloWorld is our single Project, which contains a script **hello.pl**; source for a binary **main.cpp**, which is also configured to serve as an example program; another public document; library headers to be placed in the distribution; and an example small library and big library again to be compiled and placed in the distribution. (Note a project may have many small libraries but each will have one source file; it may also have a single large library with many source files. In this demo, both libraries are single files, but both the small and large library method are demonstrated.)

```
> find src|sort
src
src/doc
src/doc_internal
src/doc_internal/Secret.txt
src/doc/LICENSE
src/doc/LICENSE.NxM
src/doc/Makefile
src/doc/NxM_Make_System.doc
src/doc/NxM_Make_System.pdf
src/HelloWorld
src/HelloWorld/hellolib.cpp
```

```
src/HelloWorld/hellolib.h
src/HelloWorld/hello.pl
src/HelloWorld/hellosmall.cpp
src/HelloWorld/main.cpp
src/HelloWorld/Makefile
src/HelloWorld/Make.Project
src/HelloWorld/PerProjectDoc.txt
src/make
src/make/Arch.Conf.Makefile.Template
src/make/Arch.Conf.Proj.Makefile.Template
src/make/Arch.Makefile.Template
src/Makefile
src/make/Make.Common
src/make/Makefile
src/make/Make.hostname-here
src/make/Make.Proj.Common
src/make/Make.Site
```

6.2 Walkthrough of a single Architecture

Rename the example architecture directory **Fedora40-64** *if you wish*, or leave as is. The name is used in naming subdirectories in the final distribution tree, but has no other significance. The name is deduced by the Makefiles by examining the current working directory, so you don't need to edit it into a file anywhere.

```
> cd Fedora40-64
```

You will see it's set up to compile with two different configurations, **Debug** and **Optimized**. Run find to see what files are needed to build our distribution—not many:

```
> find . | sort
.
./Debug
./Debug/Make.Configuration
./Make.Architecture
./Makefile
./Optimized
./Optimized/Make.Configuration
```

6.3 Compiling a single Architecture

Let's build everything that depends on this architecture.

```
> gmake
```

Let's look at the build tree now. You'll see that under both configurations, it made shadow copies of each project under ../src, and created dependencies and object files.

```
> find . | sort
.
./Debug
./Debug/HelloWorld
./Debug/HelloWorld/.depends
./Debug/HelloWorld/.depends/hellolib.d
./Debug/HelloWorld/.depends/hellosmall.d
./Debug/HelloWorld/.depends/main.d
./Debug/HelloWorld/hellolib.o
./Debug/HelloWorld/main.o
./Debug/HelloWorld/Makefile
./Debug/Make.Configuration
./Debug/Makefile
./Make.Architecture
./Makefile
./Optimized
./Optimized/HelloWorld
./Optimized/HelloWorld/.depends
./Optimized/HelloWorld/.depends/hellolib.d
./Optimized/HelloWorld/.depends/hellosmall.d
./Optimized/HelloWorld/.depends/main.d
./Optimized/HelloWorld/hellolib.o
./Optimized/HelloWorld/main.o
./Optimized/HelloWorld/Makefile
./Optimized/Make.Configuration
./Optimized/Makefile
```

6.4 Adding Non-Architecture-Dependent Files to Distribution

Let's also run gmake under src. We'll see it copies everything under src/doc that isn't called Makefile, isn't a core, Microsoft Word temporary file, or emacs backup file, to the distribution.

```
> cd ../src
> gmake
```

6.5 Walkthrough of the resulting Distribution

Finally, let's look at the resulting distribution. It contains the example script, an example document, a coding example, and include files. The include file path includes the workspace name a second time, allowing `#include <NxM/hellolib.h>` syntax, so that your headers won't clash with identically-named headers from other distributions. Finally, in directories named after the architecture and configurations, we have copies of static and dynamic versions of large and small libraries, as well as binaries. You should be able to tar up `NxM.1.0` and give to acceptance testers then customers.

```
> find ../NxM.1.0 | sort
../NxM.1.0
../NxM.1.0/bin
../NxM.1.0/bin/hello.pl
../NxM.1.0/doc
../NxM.1.0/doc/LICENSE
../NxM.1.0/doc/LICENSE.NxM
../NxM.1.0/doc/NxM_Make_System_1a.doc
../NxM.1.0/doc/PerProjectDoc.txt
../NxM.1.0/example
../NxM.1.0/example/main.cpp
../NxM.1.0/Fedora40-64
../NxM.1.0/Fedora40-64/bin
../NxM.1.0/Fedora40-64/bin/Debug
../NxM.1.0/Fedora40-64/bin/Debug/hello
../NxM.1.0/Fedora40-64/bin/Optimized
../NxM.1.0/Fedora40-64/bin/Optimized/hello
../NxM.1.0/Fedora40-64/lib
../NxM.1.0/Fedora40-64/lib/Debug
../NxM.1.0/Fedora40-64/lib/Debug/libhello.a
```

```
../NxM.1.0/Fedora40-64/lib/Debug/libhellosmall.a
../NxM.1.0/Fedora40-64/lib/Debug/libhellosmall.so
../NxM.1.0/Fedora40-64/lib/Debug/libhello.so
../NxM.1.0/Fedora40-64/lib/Optimized
../NxM.1.0/Fedora40-64/lib/Optimized/libhello.a
../NxM.1.0/Fedora40-64/lib/Optimized/libhellosmall.a
../NxM.1.0/Fedora40-64/lib/Optimized/libhellosmall.so
../NxM.1.0/Fedora40-64/lib/Optimized/libhello.so
../NxM.1.0/include
../NxM.1.0/include/NxM
../NxM.1.0/include/NxM/hellolib.h
```

7 How To

7.1 Create a New Solution

If you don't have a copy of NxM, get one:

```
git clone https://github.com/FrankSheeran/NxM.git
```

You can then rename it to your solution:

```
mv NxM MySolution
```

Or, if you wish to keep an original reference copy of NxM and simply base a new solution off it:

```
mkdir MySolution
cd NxM
tar cvf - * | ( cd ../MySolution ; tar xvf - )
```

If the name of your distribution directory should indeed be *MySolution.SoluVersion*, *MySolution*/Make.Solution will find that name from its directory path. Otherwise, edit *MySolution*/Make.Solution and set SoluName directly (there's a commented-out example showing how), and comment out the GNU make commands that deduce it from the path.

7.2 Create a New Project

Make the directory `{Solution}/src/{Project}` .

Copy a `Make.Project` from an already-existing, similar and/or familiar project.

Add your source code, scripts, project-specific documents, and code examples if you already have them, or write some mostly-empty starter files for the project.

Edit `Make.Project` to contain your source file list for the various targets.

`{Solution}/src/Makefile` and `{Architecture}/{Configuration}/Makefile` both search for this `Make.Project` and and recursively make it upon your next make. The later will even make the directory and copy `src/make/Arch.Conf.Proj.Makefile.Template` to it. So, you do *not* need to add it to any other file or variable.

7.3 Modify a Project

Big Binaries

Make.Common currently supports making up to four binaries composed of multiple source files. Define the sources, link options, and name of the binary with BigBinSrcs, BigBinLink, and BigBinTgt. Should you need more than one, suffix 2, 3, or 4 to the those variables.

```
BigBinSrcs   = main.cpp part1.cpp part2.cpp
BigBinLink   = -lhello
BigBinTgt    = $(BinDir)/hello
```

If you're producing a binary for internal use, you can just write BigBinTgt = hello and it will be created in the current directory instead of being built in the Distribution directory.

Big Libraries

To leave a big library in the current directory, you would write BigLobTgt = libhello.a . To put it in the distribution, write: BigBinTgt = $(LibDir)/libhello.a .

Small Binaries and Libraries

There is currently no way to specify that little binaries and libraries should go: they always go to the distribution.

Headers, Scripts, Docs, and Examples

Headers, scripts, docs and examples are copied to the distribution if and only if they're mentioned in IncludeTgt, ScriptTgt, DocTgt, and ExampleTgt. Private files simply need not be mentioned and instead used where they stand.

There are additional src/doc and src/doc_intern directories for documents. The first has a makefile and will install all files regardless of file type into the distribution DocDir . The second has no makefile so is just a free-form location for files.

7.4 Rename or Remove a Project

Rename with mv {Solution}/src/{ProjectOld} {Solution}/src/{ProjectNew} .

Remove with rm -rf {Solution}/src/{ProjectOld}

Temporarily skip building with mv src/{Project}/Make.Project src/{Project}/Make.Project.skip .

In all three cases, rename or remove the project's output from under {Distribution} (or simply remove then rebuild the full {Distribution}, which is easy and sure). You may also want to remove {Architecture}/{Distribution}/{ProjectOld} though it's not

strictly necessary: without the **src/{ProjectOld}/Make.Project**, nothing will be built or placed in the distribution any more.

7.5 Create a New Architecture

Copy an existing similar and/or familiar architecture file heirarchy (after doing a make pristine, for efficiency). The name can be freeform and isn't actually referred to: the Make system never attempts to CD or otherwise traverse into this directory, only you do.

Then, proceed to the following section, to make at least one configuration.

7.6 Rename or Remove an Architecture

Rename with **mv {ArchitectureOld} {ArchitectureNew}** .

Move with **rm {ArchitectureOld}** .

To temporarily skip building it… simply stop running gmake there. An architecture is only ever built because you manually run gmake under **{Architecture}**.

7.7 Create a New Configuration

Create an empty directory under {Architecture} for every Configuration you wish to build.

Then, copy the Make.Configuration from the Debug or Optimized example that ships as part of NxM, and modify it as needed.

Running gmake under **{Architecture}** will copy **../make/Arch.Conf.Makefile.Template** to **{Architecture}/{Configuration}/Makefile**, and that will in turn make copies of projects.

7.8 Rename or Remove a Configuration

Rename with **mv {Architecture}/{ConfigurationOld} {Architecture}/{ConfigurationNew}** .

Remove with **rm -rf {Architecture}/{ConfigurationOld}**

Temporarily skip building with **mv {Architecture}/{Configuration}/Make.Configuration {Architecture}/{Configuration}/Make.Configuration.skip** .

In all three cases, rename or remove the configuration's output from under **{Distribution}** (or simply remove then rebuild the full **{Distribution}**, which is easy and sure).

7.9 Compilation

In order to compile a:

Single Project Most of your compilations will probably be while doing the initial coding on a single architecture, with a debugging configuration. This command will also install headers, example programs, documents, and so on, in the Distribution, though that may be premature at this point.

Run gmake in the directory:
`{Solution}/{Architecture}/{Configuration}/{Project}` .

Single Configuration If the code you're working on spans multiple projects, run gmake one level higher, in:
`{Solution}/{Architecture}/{Configuration}` .

Entire Architecture Once the code runs correctly, you may wish to build it in debug and optimized configurations for performance measurement. You can run gmake one level higher yet, in:
`{Solution}/{Architecture}` .

Entire Distribution **This is not automated, nor easily automatable.** Run gmake in `{Solution}/src` . (This will install the per-Solution document files, at a minimum.) Then, on every platform, follow the instructions for Entire Architecture, above.

7.10 Making the Distribution

After compiling the entire Distribution (see above), go up one level to the `{Solution}` directory and `make gz` or `make zstd` to make a tarred and compressed version of the Distribution, which should be suitable for sending to the QA team for acceptance testing, and thence on to customers.

7.11 Editing Makefile Templates

If you have to edit them, consider whether the edit will be needed by one Project or all, one Configuration or all, and so on. If all Projects will need the same edit, then edit the master copy.

7.12 Changing Locations of Files in Distribution

The beginning of `Make.Common` states the directories where everything will go in the Distribution. For instance, `BinDir` is set to `$(DistDir)/$(ArchName)/bin/$(ConfName)` . You may prefer bin to be earlier or later in that heirarchy. Or, you may only be using one

Architecture or one Configuration and thus don't need them in the path. You can simply edit this one file to fit your requirements.

7.13 Customization

NxM *as shipped* isn't that flexible, in terms of being able to radically alter its behavior by setting variables or using command flags. However, since it's implemented as `Makefile`'s, you can edit the files as you wish. Use its design as inspiration, and modify it for your solution's needs. There's always a worry about "forking" software, including NxM, and not being able to automatically benefit from future developments on the main branch. However that needn't be a huge worry: NxM is as much about the approach to using make as the specific directory heirarchy and Makefile's themselves. Recommendation: fork away.

8 Interoperability

8.1 Windows

Approach 1: Use Visual Studio inside the directory heirarchy prescribed by NxM

It's not well-integrated into NxM but what works for me is to put Visual Studio's Solution heirarchy under {Solution}, as a sibling to src and distribution. I then make whatever projects are necessary, and point them to the source under src and have them create libraries and binaries under distribution. I'm not sure how much to put under source code control, such as the `*.sln` and `*.vcxproj` files, etc.: how many files are necessary, how many are not?

Approach 2: Use Visual Studio's Compiler, Linker etc. from gmake

The main issue would be translating all the compiler and linker flags to Windows tools' format. A suggestion is to keep the makefiles' contents 100% in "Unix" but have a script that translates a set of flags to Visual Studio's syntax. File paths using / instead of \ as directory separators may be already accepted by Visual Studio tools but you'd turn `-Dfoo` into `/D foo` and so on.

Approach 3: Use g++, clang etc. on Windows

I'm not sure if you can use the Microsoft Standard C library and so on but maybe.

I'm not sure if you can link to third-party libraries on Windows but maybe.

8.2 Java

This framework hasn't been tried with Java projects.

9 FAQ

Q: Automatically update "latest" link to point to a successfully-finished build?

A: We can't do this because a build may require compilation on multiple architectures, none of which will know it's last. It's probably easier for the engineer to manage this link manually than to learn, configure, and trouble-shoot how it would be handled by NxM.

Q: Why does each project have both a `Makefile` and a `Make.Project`? Why not just put the `Make.Project` settings inside the `Makefile` instead of including it?

A: That would be possible, so it's just a question of which is easier and clearer. The `Make.Project` absolutely has to vary per project, and be in source code control. All the files that absolutely have to be in source code control are under src. In contrast the Makefile is hopefully always generic and can be copied from `src/make` to a `{Architecture}/{Configuration}/Project` directory. Further, separating them makes the construction of the actual `Makefile` out of parts as clear as possible, hopefully. That said, a `Makefile` with the `Make.Project` settings in it *would be technically feasible* under `src/{Project}` and simply be run from `{Architecture}/{Configuration}/{Project}` with `../../src/{Project}` .

Q: I have internal documents I don't want in the distribution, but anything I put in src/doc goes there.

A: Just use the sibling directory `doc_intern`. It doesn't have a `Makefile`, so the src Makefile won't recurse into it. Even if you need a Makefile and write one, obviously you wouldn't have it install internal documents.

Q: I've removed or renamed headers, examples, docs, or even entire Configurations, but the Distribution still has them. Running `make pristine` doesn't get rid of them. How can I make a totally clean Distribution?

A: The pristine target removes binaries and libraries, and their Configuration-based directories, from the Distribution. But since headers, examples, and docs are added to the Distribuation by multiple Architectures' builds, no single Architecture knows to remove them all. (And since different Architectures must normally be compiled on different machines, there's no single point where this could happen.) The workaround is simple: remove the `$(DistDir)`, *without* doing a `make clean` (which would delete your object files). Then, re-run make on each Architecture, which will simply relink thebinaries and libraries and make a fresh copy of (and *only* of) the current headers, examples, and docs. Likewise, if you rename a Configuration (e.g., by `mv OldConfig NewConfig`), this method will produce the new Distribution quickly as the object files built in the old

configuration directory name will still make good binaries and libraries under the new name.

Q: How can I control whether static or dynamic libraries are created?

A: For the big library, comment out the corresponding `BigLib` rule in `src/make/Make.Proj.Common`. Search for `.so` or `.a` . For the small libraries, you can only control them as a group but again, same file, same method, the `LtlLib` rules.

Q: My distribution needs other types of files, such as man pages, or something.

A: `{Solution}/src/Makefile` will run a gmake in every subdirectory that itself has a file called `Makefile`. Look at how documents are installed: `{Solution}/src/doc` is a directory and has a Makefile so gmake is run. You can probably make a sibling directory to `doc`, and copy and modify its makefile. You can edit your `Make.Common` to have a new directory variable (such as `ManDir`) and refer to it in the `InstallToDistribution` rule instead of `DocDir` . You could also simply write the path in your new makefile directly.

10 Future Directions, Open Questions

Future Directions

NxM is designed to distribute SDKs (software development kits). It could be used for shrinkwrap software as well, but how to do so should be explained.

Integration with Windows should be experimented with, integrated and documented.

Some useful difference should be implemented between targets clean, veryclean, and pristine. Currently they all do the same thing.

There should be a `src/additional`, all files under which are added to `$(DistDir)` .

There should be a more generalized mechanism than `DocDir`, `ExampleDir`, etc. to copy files from Projects to the Distribution. This could handle things like putting data files into a data directory, man pages into a man directory, and so on. Arguably, `ScriptTgt`, `DocTgt`, and `ExampleTgt` could be removed and that more general mechanism be used.

Have `Make.Project` specify whether headers go under `{Distribution}/`, such as in `include/Solution`, `include/Project`, `include/Solution/Project`, `Solution`, `Project`. To do this, I think we can let `Make.Project` set `IncludeDir`, which is currently set in `Make.Common`, and somehow get the `Make.Project` version if set to take priority. Maybe gmake already makes the first setting take priority?? Or the `Make.Common` setting for this and maybe many other variables can be coded such that it only takes effect if not already set.

Java should be tested and any necessary support added. I think it'd work but perhaps be too trivial for NxM to be needed. On the other hand, a project with some C++ and some Java would be an excellent candidate for NxM.

Support for man pages could be added.

Open Questions

The include order of the Partial Makefiles in `src/make/Arch.Conf.Proj.Makefile.Template` should be examined. The current order is simply based on what has worked for the last many projects, but it may not be a great ordering.

What if we want a nightly build to {**Solution}.{SoluVersion}.nightlycandidate**; then, if it completes without error, move to {**Solution}.{SoluVersion}.nightly** ?

How can we ship Debug versions with source code and object files?

What if we want to distribute only, say, the Optimized build, and have a Debug build built only for internal use?

Is there an easy way to distribute the code via GIT instead of or in addition to pre-compiled? Maybe the GIT user still always gets this deluxe build?

How can this be combined with `dnf` and similar?

How should we create dynamic library versions and version links? E.g., `mylib.so` is a symlink to `mylib.so.3`?

What's the right place in the filesystem for `Make.{Machine}`? I think it should perhaps be in the directory level above `{Solution}` as in theory multiple Solutions could and should share these files. However, I haven't done so because I despair that multiple projects could actually evolve separately yet use compatible variable names and meanings.

Can we extend to support Three-Tier Solutions? E.g., allow SubProjects? Can we make it open-endedly deep?

The previous iterations of this design had Partial Makefiles including other Partial Makefiles. This was a bit more terse, but harder to support and explain and understand. To simplify it at the price of being more verbose, the current model was adopted of having a single level of include: only actual Makefiles include Partial Makefiles.

For instance, in theory, we could make `{Architecture}/{Configuration}/{Project}/Makefile` a one-line file, which includes `../../../src/make/Make.Proj.Common`, and everything in the former could be moved to the latter.

Running make pristine in `{Architecture}`, `{Architecture}/{Configuration}`, or `{Architecture}/{Configuration}/{Project}`, will remove the binaries and libraries. However, making *any* architecture will install headers, examples, documentation and so on, so it's unclear if a make pristine for a single architecture can or should remove these files—they're not needed for this architecture or configuration, the developer is saying,

but perhaps they're needed for others. `$(BinDir)` and `$(LibDir)` are removed, which are typically configuration-specific, but by default in parent directories (e.g., `$(DistDir)/$(ArchName)/bin`) that are create by a build but *not* removed by pristine. As a workaround, before creating a full distribution for testing as as a candidate distribution, you can remove `$(DistDir)` entirely and rebuild it, even if from the existing object files so it needn't take long, but will not have any files files or directories that should no longer exist.

TODO

Checklist for each file:

- license and doc notice in all files
    - Makefile
    - Make.partial
    - C++ files
    - others?
- DoClean is in all files

Move `.PRECIOUS` from `Make.Common` to `Arch.Conf.Proj.Makefile.Template`?

Why is ConfName, SoluName, etc., assigned with := instead of =?

Document how to control the compilation order of projects, for instance to have a library compiled before the binaries that use the library.

Document how to use NxM to make a distribution that's not an SDK: no headers and static libraries; perhaps no debug binaries.

A future release of this document should compare and contrast NxM with these other tools, and explain how to coexist with them.

- Cmake
- Autotools (GNU build system)
- Ninja
- Maven
- Meson
- Gradle

11 Change Log

11.1 Version 1.0

Initial Release.

11.2 History

NxM's roots evolved in the 1990s for a software distribution of an in-memory database SDK, comprising a main library and a dozen auxilliary libraries in debug and optimized variants for Solaris, Linux, VMS, and Windows, with a dozen or so binaries and about the same number of scripts.

This original system didn't use `VPATH`, instead using a forest of simlinks. It copied binaries and libraries instead of simply outputting them in the finished distribution tree. It didn't have `Make.Solution` or `Make.Site`, and in lieu `Make.Configuration` there were instead multiple `Make.Common` . Dependencies were handled by `makedepend` . Other than that, the initial system was much as it is today and the contents of `Make.Architecture` and `Make.Common` largely date to this period.

The same needs subsequently presented themselves at a small software company, a proprietary trading team, a brokerage, and a startup. In each case the make system as it then stood was adopted and adapted. After each of these jobs, the make system was updated with the resulting observations and lessons learned.

For ongoing research projects the same need to build N "projects" M "ways" again presented itself. This build system was factored out, named, more thoroughly documented, and made its own solution. `Make.Solution` and `Make.Site` were added. The previous bizantine include structure was combed out into the current flat structure.

12 Licensing

The license resides in the file src/doc/LICENSE.NxM, but is periodically copied here for your convenience. Should the texts differ, LICENSE.NxM is the definitive copy.