From Makefile to Project: CodeWarrior's Makefile Importer Wizard

A Metrowerks White Paper

By Rick Grehan

Copyright © 1999 Metrowerks, Inc. All Rights Reserved.

From Makefile To Project: CodeWarrior's Makefile Importer Wizard

CodeWarrior's Makefile Importer Wizard overcomes what may be the steepest barrier facing developers who want to move their projects into a visual development environment.

By Rick Grehan

THE MAKE UTILITY is a well-known software project management tool that got its start in the Unix world and continues to enjoy wide use today. The `make` utility reads a "makefile"—a source file that the developer creates—and, guided by the information in that makefile, compiles and links all the source code files to build the application's executable. That's a simplified description, you can do much more with the `make` utility than automate compiling and linking, but that's the gist of it.

In this paper we discuss some of the capabilities of a makefile, and explore the parallel features of a CodeWarrior project. The bulk of this paper is devoted to illustrating the process of importing a makefile into CodeWarrior using the Makefile Importer Wizard. This Wizard manages the translation process, and takes most of the sting out of moving a makefile-based project into CodeWarrior.

What Is A Makefile?

A makefile is, among other things, a list of all the source files, library files, resource files, etc., that are used to build an application. In one sense, then, a makefile is like a batch file. When it's time to rebuild an application, a developer doesn't have to remember all the files that comprise that application; the developer doesn't have to type line after line of compile commands and link commands. The developer simply launches the `make` utility, which reads the makefile and issues all the compile commands and link commands automatically.

In addition, the makefile records file dependencies. The makefile "knows" which source files must be compiled together to create a single object file, and which object files must be linked to build each executable (a single makefile can build more than one executable). However, dependency information can be more complicated than a linear list of files to be compiled and linked. For example, a makefile can record the dependency that C source code files have on header files, so that if a header is changed, all effected C source code files are recompiled when a build is requested. In short, the makefile records all the files that comprise an application in such a way that if *any* of the files so recorded is modified, rebuilding the application (with the `make` utility) causes the proper files to be processed.

Finally, the makefile also includes the command-line switches that specify the options passed to the compilers and linkers. These switches are important, because a developer must set them properly for the compilers and linkers to work correctly. However, these switches are difficult to keep track of; there are so many variations, and they tend to be single-letter mnemonics whose meanings are easily forgotten.

Now, if you look at the three kinds of information that a makefile contains – filenames, dependencies, and command-line options – you can see that a makefile is an automated recipe for building an application from its constituent files. A single `make` command will read the makefile and manage the complete compile and link process.

But, more important, because the makefile carries file dependencies, the `make` utility can be intelligent about what it rebuilds. If, for example, a developer has changed only a single source file and wants to rebuild the application, the `make` utility can determine which components of the application depend on the altered source file, and will compile only the affected parts. For even moderately large projects, this can save significant time over the course of the project development.

Finally, though the main use of `make` is automating the compile-and-link process, it's not restricted to dealing with compilers and linkers. A makefile can also work like a batch file. For example, programmers familiar with `make` frequently define a target called "clean." The clean target does not build any executable; rather, clean is defined in such a way that building it causes a "delete" (or "remove") command to clear all the object files from the current directory. So, if the developer wants to build the executable "from scratch" – using source files only – the developer first commands `make` to build the clean target. With object files thus wiped out, the developer can then command `make` to build the actual executable target, confident that only the latest source will be used.

(Note: A single makefile can build more than one target. A command-line switch allows the developer to command `make` to build a specific target in a given multi-target makefile.)

CodeWarrior Projects

The CodeWarrior integrated development environment (IDE) manages code projects visually. Where a makefile is a text document edited by a word processor, a CodeWarrior project appears to the developer as a window into which files can be dragged and dropped. Actions that, in a makefile, must be typed in as text commands can, in a CodeWarrior project, be activated by choosing an item from a picklist or selecting a checkbox. The CodeWarrior IDE 's capabilities are apparent to the user through its visual interface. With a makefile, the user must either memorize commands, or search for them in a reference manual.

A CodeWarrior project performs many of the same duties as does a makefile. First, it works in conjunction with the CodeWarrior IDE as a process manager. It manages the building of a target application from its constituent parts in the same way that a makefile (with the help of the `make` utility) does.

Like a makefile, a single CodeWarrior project can include several targets. In addition, targets can be nested. That is, a given target might actually be composed of multiple dependent targets. Building the parent target causes its offspring to be built as well. We will illustrate this with examples later.

Second, it works like a container. Files are "put into" a project. A programmer can add files to a project simply by dragging their icons from the desktop and dropping them into the project window. This visual representation of the project-as-container is easy to grasp, and works well with the desktop paradigm presented by the operating system.

(Note: In reality, however, it's not a container; it's a list, in the same way that files in a makefile are not really in the makefile, files that appear to be in a CodeWarrior project are really pointers that reference the actual files on disk. But the container abstraction is maintained so well by CodeWarrior that the developer needn't think about the plumbing underneath.)

Associated with a project are parameters that specify which compiler to use (i.e., C/C++, Pascal, or Java), the target CPU, compiler optimizations, stack size, and so on. These parameters are easily modified through settings panels.

Some of the settings correspond to the options and switches that are specified among input arguments to command-line compilers and linkers. Other settings correspond to makefile commands. For example, in one of the CodeWarrior IDE's panels you can associate a compiler to a particular file extension. This tells the IDE that files ending in ".PAS" should be compiled with the Pascal compiler, files ending in ".C" or ".CPP" are to be compiled with the C/C++ compiler, and so on. This is analogous to the makefile

rules for converting a given source-code file to object code using the source-code file's extension to identify the correct compiler (what is known as an "implicit suffix rule").

Third, the CodeWarrior IDE tracks dependencies among files in a project. For example, if a developer changes the content of one of a project's source-code files, the IDE marks that file as "touched." When the developer commands the IDE to rebuild the project's target, only those files that have been marked as touched—as well as any files dependent on touched files—are recompiled.

For example, though header files need not be explicitly included in a CodeWarrior project, the IDE nonetheless tracks them. If the programmer alters a header file, the IDE identifies all affected source files and recompiles them when the application is rebuilt.

So, in many ways the CodeWarrior project is a visual makefile. Though the analogy is not complete, it is close—sufficiently close that it would be reasonable to have an automated method for translating a makefile into a CodeWarrior project. A developer who is moving to CodeWarrior but doesn't want to lose the time and information already invested in using makefiles, could use such a translator to quickly transform a makefile into a CodeWarrior project.

This is precisely the function of CodeWarrior's Makefile Importer Wizard. It reads a makefile, parses it to identify source files, target names, dependencies, and builds a CodeWarrior project.

Working the Wizard

CodeWarrior's Makefile Importer is one of several project wizards that help the developer "jump-start" a new project.

When activated, the Makefile Importer Wizard leads the developer through a series of dialogs that prompt for:

- The makefile's name and location.
- The toolset used in the makefile. Currently, the Makefile Importer Wizard can digest any makefile compatible with Microsoft's `nmake` utility or with the GNU `make` utility. (Note: GNU `make` is compatible with the POSIX 1003.2 standard. Any makefile that is compatible with the POSIX 1003.2 standard can therefore be processed by the Makefile Importer Wizard.)
- The toolset to convert to. This selection identifies the target processor. Currently, the Makefile Importer Wizard can build projects for x86 processors, MIPS processors, or Java.

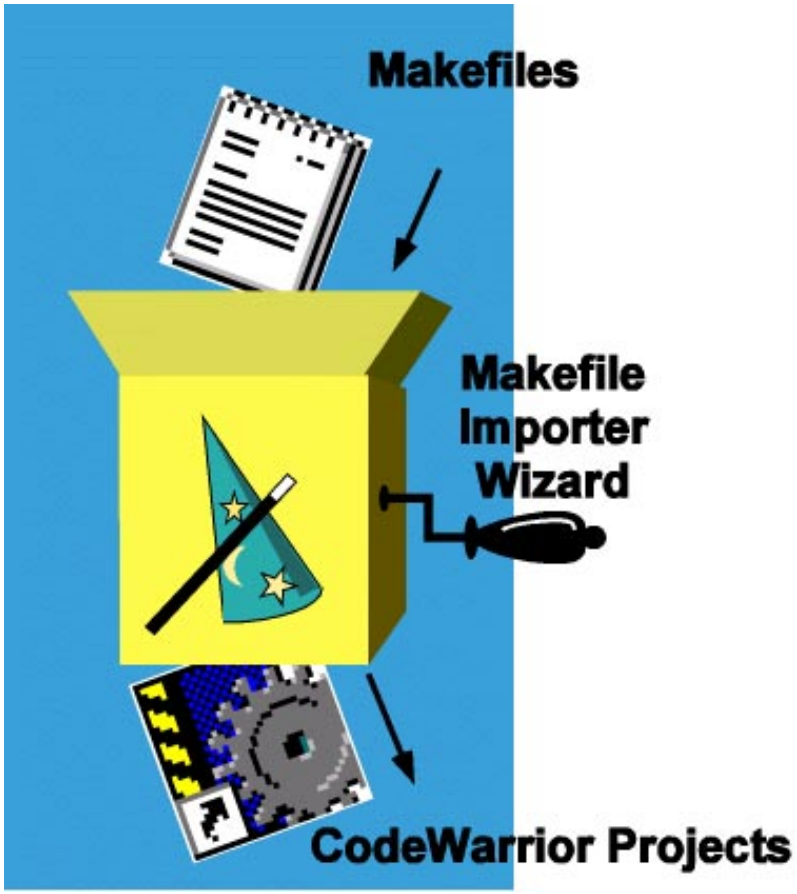

Figure 1. Metrowerks' Makefile Importer Wizard translates a text makefile into a CodeWarrior project. The wizard significantly speeds the process of moving a development project from the command-line based makefile to the CodeWarrior visual environment. Also, because the Wizard automates the translation, there's less likelihood of errors (mistyped filenames, missed dependencies, etc.)

The Importer reads the makefile, identifies source files and libraries, and loads them into the project. The Importer builds CodeWarrior targets that correspond to targets in the makefile. Files are grouped based on file type; source files are placed in source groups, library files in library groups.

In addition, the Makefile Importer wizard handles makefiles that include multiple targets. Since CodeWarrior already supports projects with multiple targets, the translation from multi-target makefile to multi-target CodeWarrior project is straightforward.

Defaults

Just as the make utility includes a set of default rules that can be referenced without definition in a makefile, the Makefile Importer Wizard provides default information for the projects it builds. This default information includes compiler and linker names, values for the preference panels of the new project, and so on.

The Importer gets this information from a special file: makefile.dat. The makefile.dat file is simply a text file that can be altered to extend and modify the capabilities of the Makefile Importer Wizard. A portion of a sample makefile.dat file is shown in Listing 1.

```
[Makefile Importer Data File]
[***Makefile Format Definitions***]
MakeFile:Microsoft Visual C++
LinkCommand:link
LinkCommand:link.exe
CompileCommand:cl
CompileCommand:cl.exe
IncludeSwitch:/I
....
[***Metrowerks Compiler / Linker Information***]
[*AccessPaths are relative to CodeWarrior Install*]
Linker:Metrowerks x86
LinkerFullName:Win32 x86 Linker
FileType:c
FileType:cpp
FileType:def
FileType:rc
FileType:res
IntermediateFile:obj
IntermediateFile:h
AccessPath:\Win32-x86 Support
AccessPath:\Metrowerks Standard Library
Library:Advapi32.lib
...
```

Listing 1. A portion of makefile.dat. This file is the Makefile Importer Wizard's knowledge base. Information in makefile.dat identifies the compiler and linker filenames used in the makefile, as well as information used to build the CodeWarrior project.

The first section of makefile.dat includes keywords that identify the compiler and linker filenames in the makefile.

The second section carries information regarding the destination targets (the targets in the created CodeWarrior project), and specifies linker, access paths (to system header files and such), system libraries (names and access paths), and so on. We've only shown part of the second section in listing 1. Not shown are the multiple sections near the end of the file, one section each for the currently supported targets. (At the time of this writing, the Makefile Importer Wizard supported x86 C/C++, Java, and MIPS C/C++ targets.)

Limitations

The Importer wizard cannot always translate the entire contents of a makefile. Some makefile actions do not have counterparts in a CodeWarrior project. Take, for example, the clean target we described earlier. The clean target was a means of making the makefile into a kind of batch file. Currently, there is no equivalent to "batch file" capability in the CodeWarrior IDE. Hence, the Makefile Importer Wizard bypasses such targets.

However, the developer can set an option in the Makefile Importer so that, when it encounters untranslatable elements in a makefile, it emits a logfile that lists those items that were bypassed. The developer can examine the logfile to identify the targets that were not translated.

(Note: The Pro 5 release of the CodeWarrior IDE will support Perl scripts embedded in a project. Shell-command capability could be added manually by someone even moderately familiar with Perl scripting.)

A Simple Example

Let's use a simple makefile to illustrate the Importer's capabilities. Listing 2 shows an elementary makefile that creates a single executable target, `Flight.exe` (a flight simulator program), by compiling two source code files: `Flight.cpp` and `Flightsb.cpp`. Each source code file is compiled and linked into a separate object file; the object files are linked to create the final executable.

This makefile is written for the Microsoft compiler and linkers tools.

```
#
# Simple Makefile Sample
#


# Option switches
CPP_FLAGS = /c /Od
LINK_FLAGS = /NOD

# Libraries
LIBS = kernel32.lib

# Target
all : Flight.exe

# Object files
Flight.obj : Flight.cpp Flight.h
        cl $(CPP_FLAGS) Flight.cpp

Flightsb.obj : Flightsb.cpp Flightsb.h Flight.h
        cl $(CPP_FLAGS) Flightsb.cpp

# Executable
Flight.exe : Flight.obj Flightsb.obj
        link $(LINK_FLAGS) Flight.obj Flightsb.obj $(LIBS)
```

Listing 2. A simple makefile that creates a single target, the executable file `Flight.exe`*.* `Flight.exe` *is built by compiling two source code files, which are then passed to the linker.*

We begin by activating the Wizard via the "New" selection on the CodeWarrior IDE's "File" menu. This brings up a dialog (shown in Figure 2) that displays the available new project Wizards. We choose the Makefile Importer Wizard.

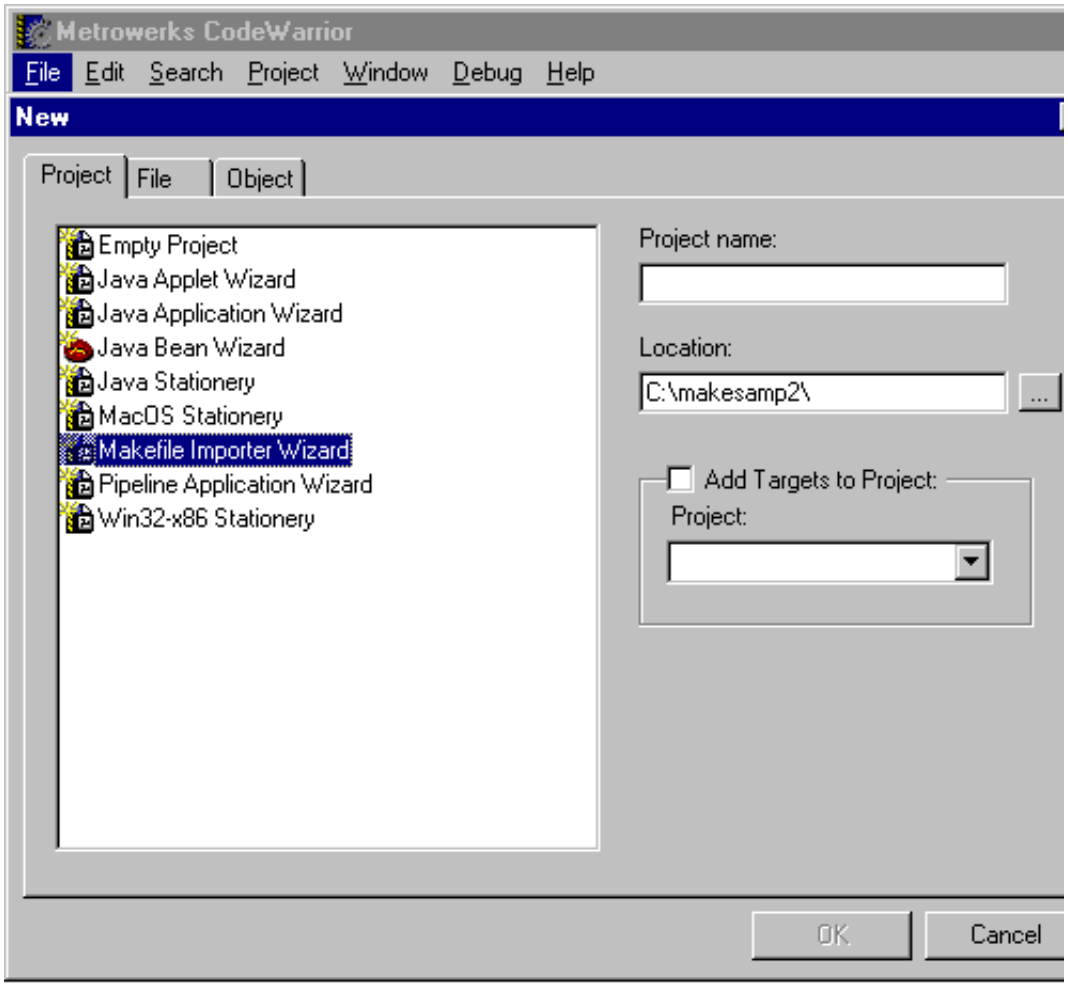

Figure 2. CodeWarrior Pro 5's new project wizards. The Pro 5 version of Metrowerks CodeWarrior includes a host of wizards for jump-starting projects. Among them is the Makefile Importer Wizard, shown here selected.

The Makefile Importer Wizard prompts for information that will guide it as it translates the makefile and builds the project (see Figure 3):

- Makefile Location. This is the path to the makefile that the wizard will process. Note that the makefile needn't be in the same subdirectory as the target that the Importer will build. This lets the developer keep the original makefile-based project separate from the new CodeWarrior project.
- Tool Set Used In Makefile. This refers to the make utility and compilers, linkers, etc. for which the makefile was written. Currently, the Importer offers two choices: Microsoft and Standard Unix. (For this example, we'll select Microsoft.)

- Metrowerks Tool Set. This refers to the compiler, platform, etc. that the project will be built for. Currently, the Importer can create projects for x86 (C and C++), MIPS, and Java. (We'll choose x86.)
- Diagnostic settings. This section allows you to set the level of feedback from the Importer as it's processing the makefile. Selecting "Log Targets Bypassed" will show all the targets that the Importer was unable to move into the project. You can also request that the Importer log all build rules that it could not process. For a more complete picture of those parts of the translation that were ignored, you can request the Importer to show all statements in the makefile that were bypassed.

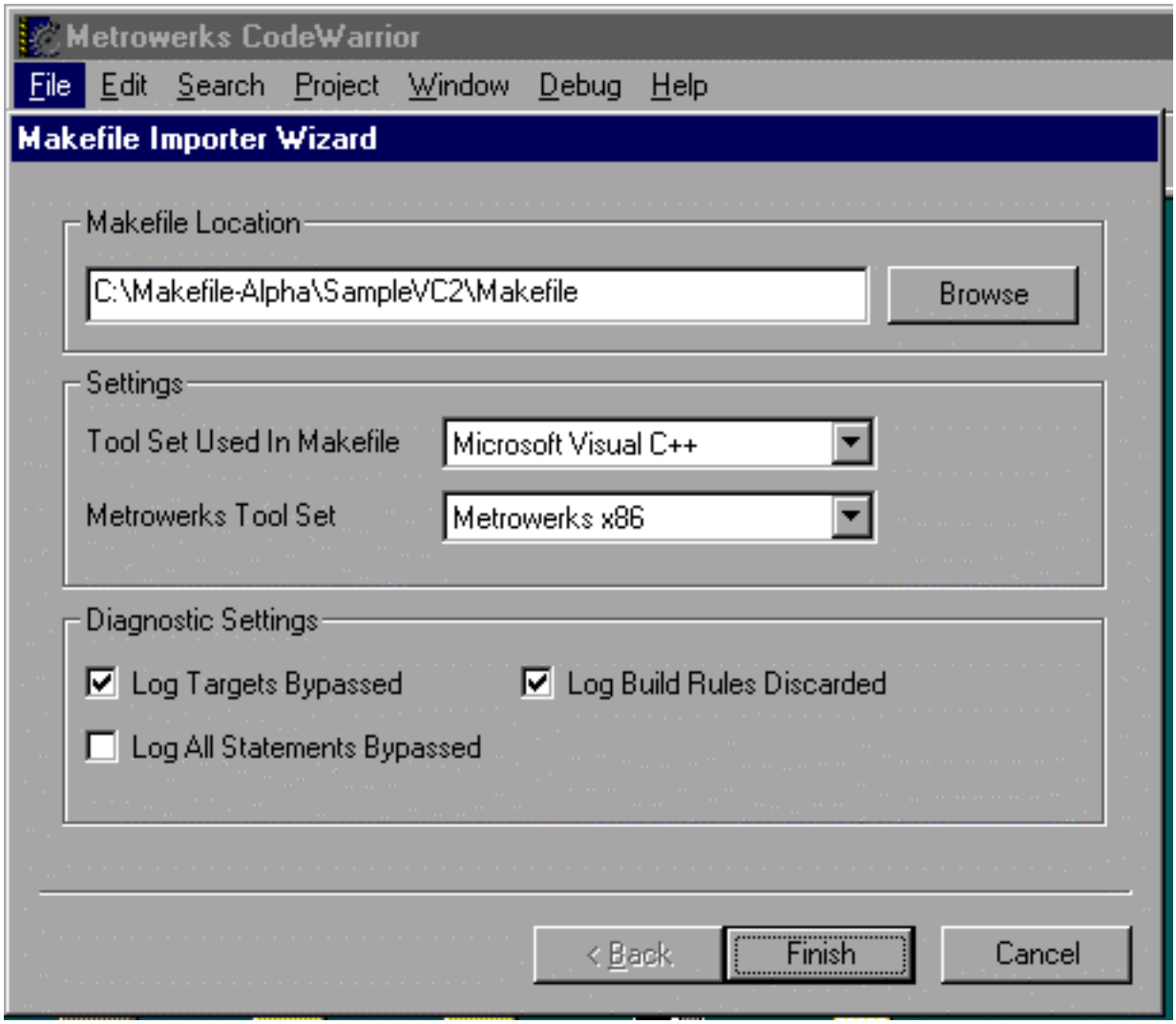

Figure 3. The Makefile Importer Wizard in action. Here, the Makefile Importer Wizard prompts for the location of the makefile, the makefile's intended toolset, and the toolset used for the created project. In addition, the Importer wizard allows you to select the level of feedback it will present as it translates the makefile into a project.

The final project is in Figure 4. Notice that the Importer has automatically created groups for the various project components. The Importer deposits the source files into the `Sources` group (shown open in the figure). It places common libraries in the `Libraries` group, and creates a separate library group for the Debug and Release versions of the target. (The Importer automatically creates a Debug target as well as a Release target. The Debug and Release targets require different libraries, so the Importer properly associates the correct libraries with the correct target.)

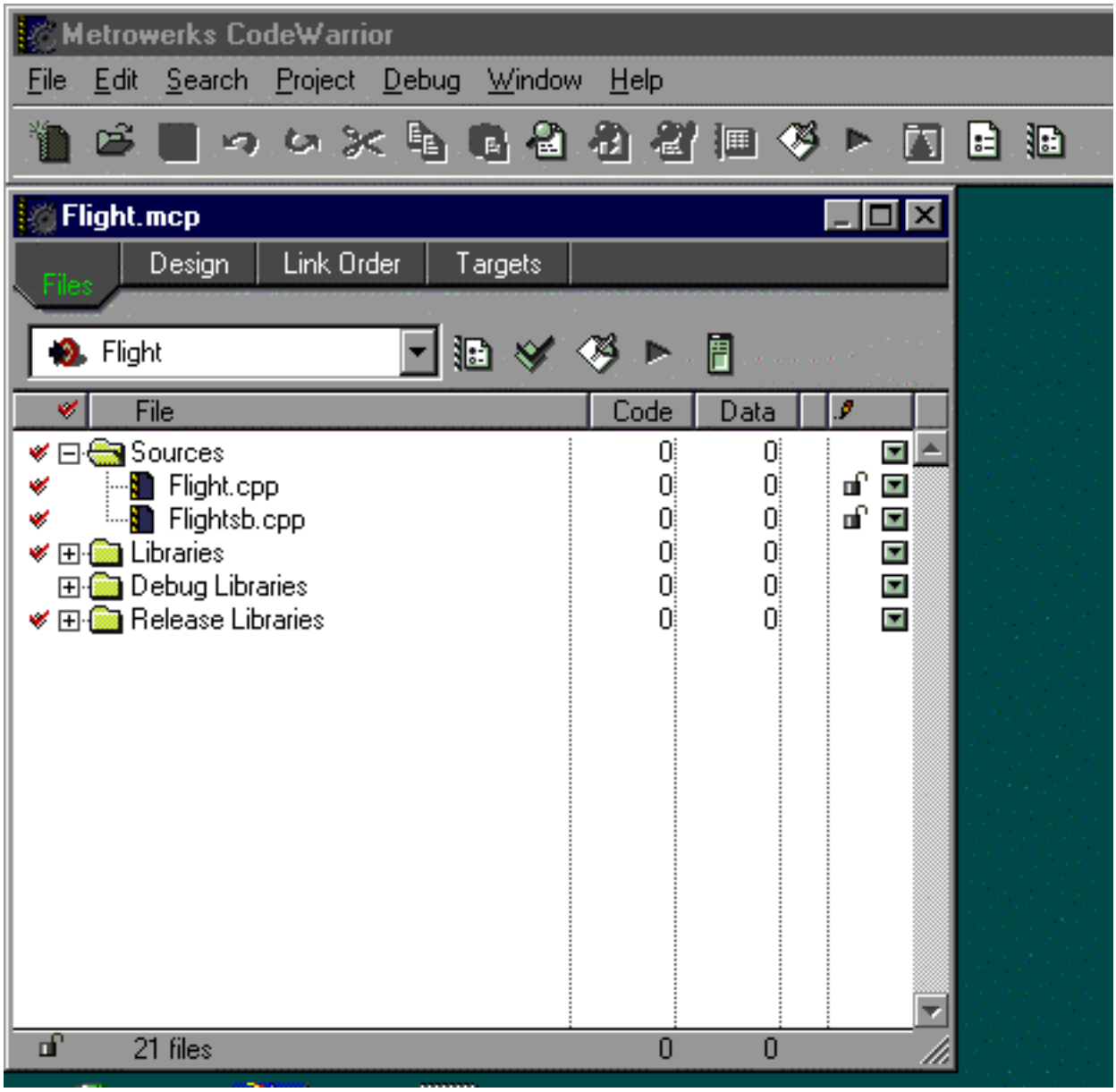

Figure 4. The Final Project. The Makefile Importer Wizard had located the source files within the makefile, created the appropriate groups for them, and loaded them into the project.

At this point, creating the executable is simply a matter of issuing a `Make` command from the CodeWarrior Project menu.

A More Complex Example: Multiple Targets

The preceding example makefile created a single target. What about a more complex situation?

Suppose we had a makefile as shown in Listing 3. In this case, there are two executable targets, `Flight.exe` and `Flightsv.exe`. (Our hypothetical programmer has turned the flight simulator into a client/server application. `Flight.exe` is the client side; `Flightsv.exe` is the "flight simulator server.")

```
#
# Simple Makefile Sample
#


# Option switches
CPP_FLAGS = /c /Od
LINK_FLAGS = /NOD

# Libraries
LIBS = kernel32.lib

# Targets
all : Flight.exe Flightsv.exe

# Object files
Flight.obj : Flight.cpp Flight.h
        cl $(CPP_FLAGS) Flight.cpp

Flightsb.obj : Flightsb.cpp Flightsb.h Flight.h
        cl $(CPP_FLAGS) Flightsb.cpp

Flightsv.obj : Flightsv.cpp Flightsv.h Flight.h

# Executable
Flight.exe : Flight.obj Flightsb.obj
        link $(LINK_FLAGS) Flight.obj Flightsb.obj $(LIBS)

Flightsv.exe : Flightsv.obj
        link $(LINK_FLAGS) Flightsv.obj $(LIBS)
```

Listing 3. A makefile with two targets. This makefile is a more elaborate version of Listing 2. In this case, the makefile specifies that two executables be created: Flight.exe and Flightsv.exe.

If we process the makefile in Listing 3 through the Importer Wizard, the Wizard's first action is to identify the two targets. You can see in Figure 5 that the Importer has identified the two executables in the makefile, and is building Release and Debug targets for each.

(Note: That's a total of four targets in the CodeWarrior project: a Debug and a Release target for `Flight.exe`, and a Debug and Release target for `Flightsv.exe`.)

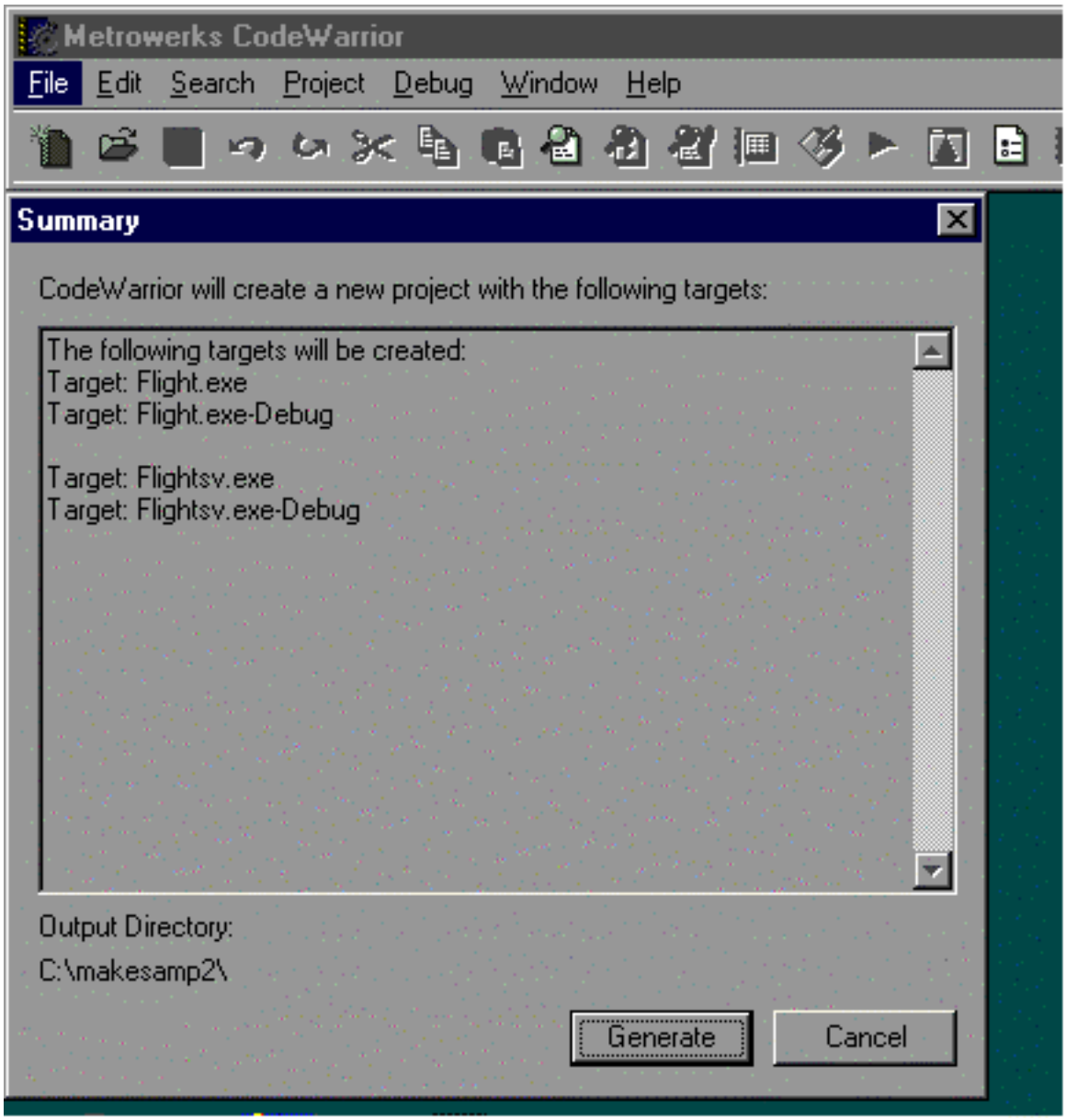

Figure 5. The Makefile Importer sees two targets. As the Importer begins to process the makefile from Listing 3, it identifies the two target executables. It informs us that it will be creating Debug and Release targets in the final project for each.

Once the Importer finishes processing the two-target makefile, it builds the project as shown in Figure 6. As before, the Importer has automatically created groups within the project and deposited each file in its proper group.

A little explanation of Figure 6 is in order for those unfamiliar with the CodeWarrior IDE. It appears to show all the source code files in the same group, `Sources`, and therefore in the same project. However, this is the "Files view" into the project (as evident by the tab at the top of the `Flights.mcp` window), which shows *all* files (in all targets) in the project. The current target is `Flight.exe`; indicated by the entry to the

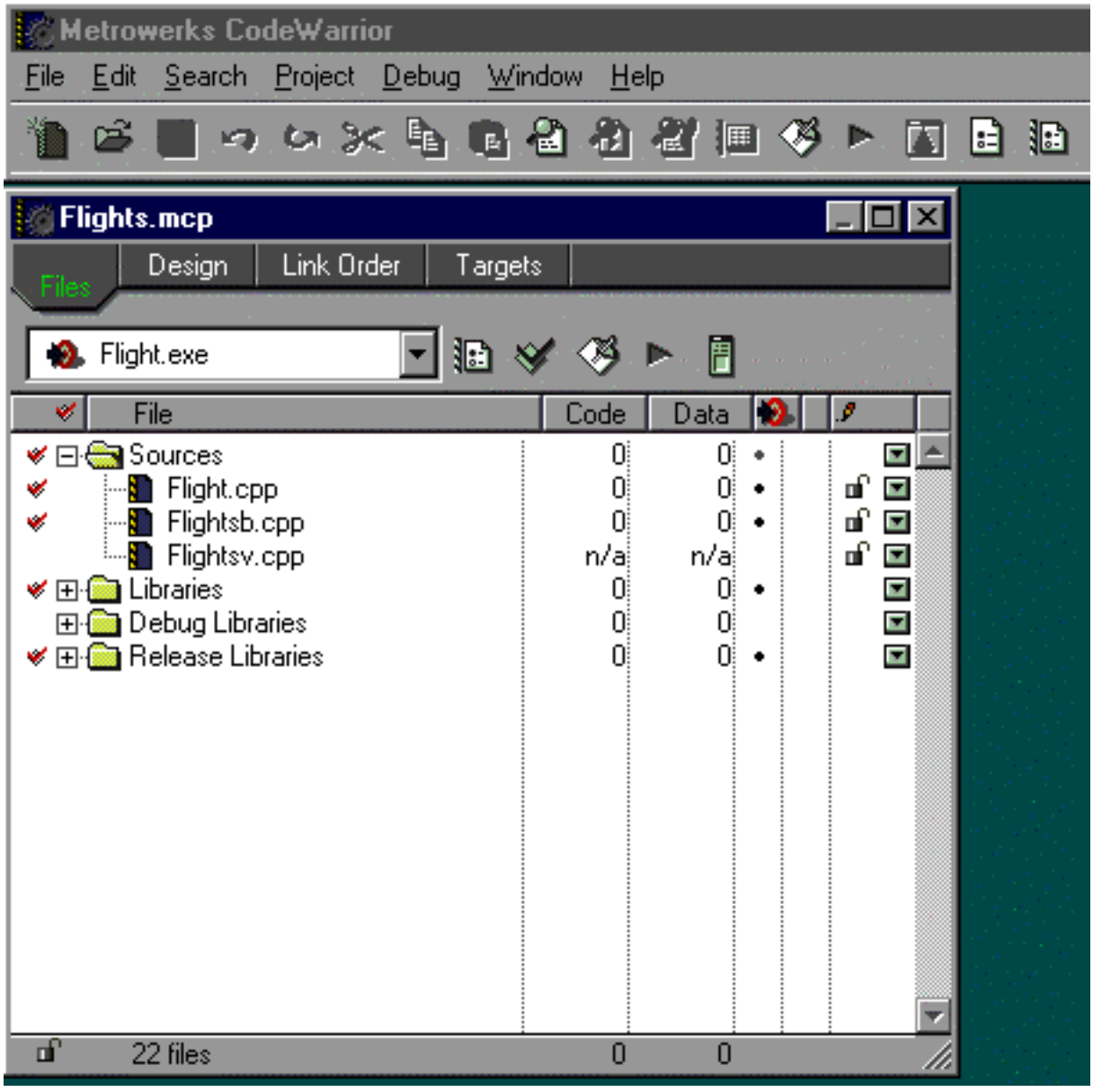

Figure 6. The multi-target project. The Makefile Importer wizard has built the `Flights.mcp` *project from the makefile in Listing 3.*

right of the small bullseye icon in the textbox. We can see that only `Flight.cpp` and `Flightsb.cpp` are in the Flights.exe target, because the Code and Data column entries for `Flightsv.cpp` are labeled "n/a."

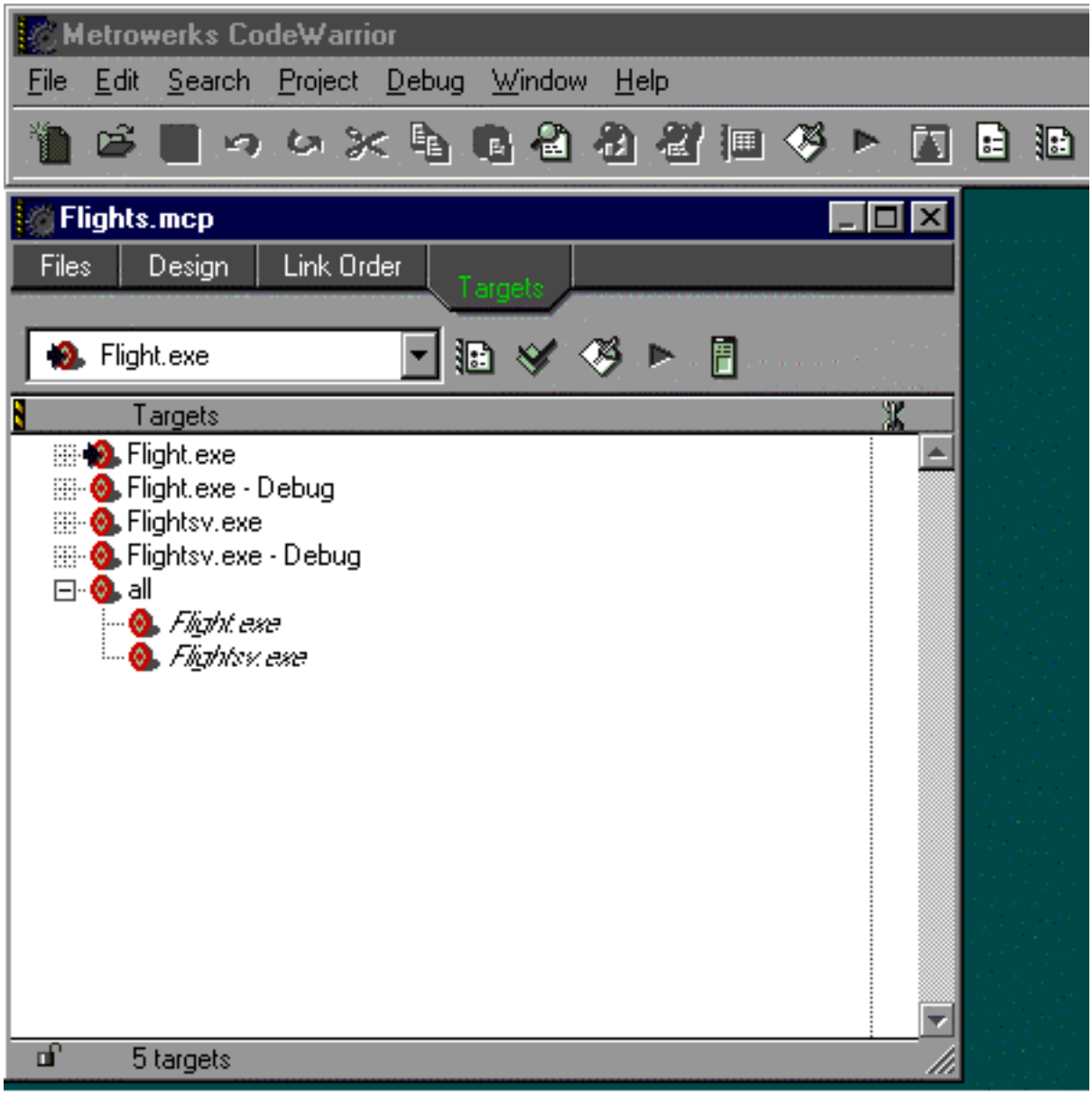

Figure 7. Another view of the multi-target project. This is CodeWarrior's "target" view of the project from the preceding figure. As you can see, both executables – `Flight.exe` *and* `Flightsv.exe` *– are associated with two targets, one a Release target and the other a Debug target. In addition, the Makefile Importer Wizard has also created an "all" target, which is dependent on both* `Flight.exe` *and* `Flightsv.exe`*. Building "all" builds both* `Flight.exe` *and* `Flightsv.exe`*.*

The multiple targets that the Makefile Importer Wizard has constructed in the project are easier to see in Figure 7, which shows the Targets view of the same project. As you can see, the project includes both Release and Debug targets for `Flight.exe` and `Flightsv.exe`. In addition, notice that the "all" target specified in the original makefile – which builds both `Flight.exe` and `Flightsv.exe` – has been added to the project.

Conclusion

Metrowerks' Makefile Importer Wizard—one of a number of project-building wizards in the Pro 5 version of the CodeWarrior integrated development environment—creates an easy path from makefile-based projects to CodeWarrior projects. Developers need no longer be locked-in to text-based tools; the Importer wizard overcomes what may be the steepest barrier facing developers who want to move their projects into a visual development environment.

The wizard accepts the two most prevalent makefile formats: Microsoft's nmake format and the GNU makefile (POSIX-standard) format. The Makefile Importer Wizard already generates CodeWarrior projects for three platforms – Win32 x86, Java, and MIPS. Support for new platforms will certainly be available in the near future.