CodeWarrior's Architectural Advantage

Today's software developer faces many hurdles. CodeWarrior's multi-host, multi-language, and multi-target design gives engineers the freedom to choose the best path to their goal.

by Tom Thompson and Jim Trudeau

THE INCREASING COMPLEXITY of software engineering puts real obstacles in an engineer's path. Some of these hurdles deal with the realities of getting a product to market, while others are technical in nature.

First and foremost, software developers face the unavoidable dilemma of cost versus time-to-market. For example, budget constraints may require the use of older hardware or a mix of different computer systems. Meanwhile, market delivery windows hold developers to aggressive coding/debug schedules.

Another hurdle is that a project's specifications can require engineers to master a new processor. An existing body of debugged, time-tested code must be ported or rendered obsolete. Worse, the engineers will have to spend time learning a new suite of development tools. All of these situations can wreak havoc on a project schedule.

As a next-generation Integrated Development Environment (IDE), Metrowerks' CodeWarrior helps software developers clear many of these hurdles. By combining an editor, compiler, linker, and debugger into a single application (all controlled by an easy-to-use GUI), CodeWarrior accelerates the development process.

CodeWarrior is platform-agnostic: it runs on popular Windows 95/NT, Mac OS, and Solaris operating systems. Versions that run on QNX and Linux are available or under development. Virtually the same GUI is presented to the engineer regardless of host, and the operation of the integrated tools is nearly identical. In addition, CodeWarrior source and project files can be exchanged among host systems without any conversion. This host-neutral capability helps engineers clear the cost hurdle mentioned earlier, because it allows them to use a mix of different computers to tackle a time-critical software project.

CodeWarrior also offers a choice of several high-level programming languages (C, C++, Object Pascal, and Java), as well as full support for in-line assembly. Stand-alone assemblers are also available for most processors.

Finally, and most importantly, CodeWarrior doesn't limit the choice of processor for that next hot product. It can generate code for an army of desktop and embedded processors:

PowerPC family, MIPS family, 68K family, x86 family, and others.[1] Compilers for several more processors are under development. Because of CodeWarrior's wide reach in this area, developers can port software to a new processor without having to learn new tools or lose an existing code base in the process.

In short, CodeWarrior provides engineers with the ability to pick the host platform, programming language, and target processor that best suits the needs of the project, without having to learn a whole new set of development tools. This power and flexibility is the direct result of design decisions made by the Metrowerks engineers.

To see how CodeWarrior provides these capabilities, it's necessary to take a closer look at the architecture of CodeWarrior itself.

Cycling Faster

The process of creating software can be characterized as a development cycle with four stages, as shown in Figure 1, "The Development Cycle." In this somewhat simplified model, the engineer writes source code, compiles it into machine code (also known as object code), links the resulting objects and any libraries into an executable program image, and then debugs the program to see if anything goes wrong. Depending upon the results, the engineer may repeat any single step or the entire process. The speed at which an engineer moves code through this cycle is important: the faster the cycle goes, the more (and better) code can be written.

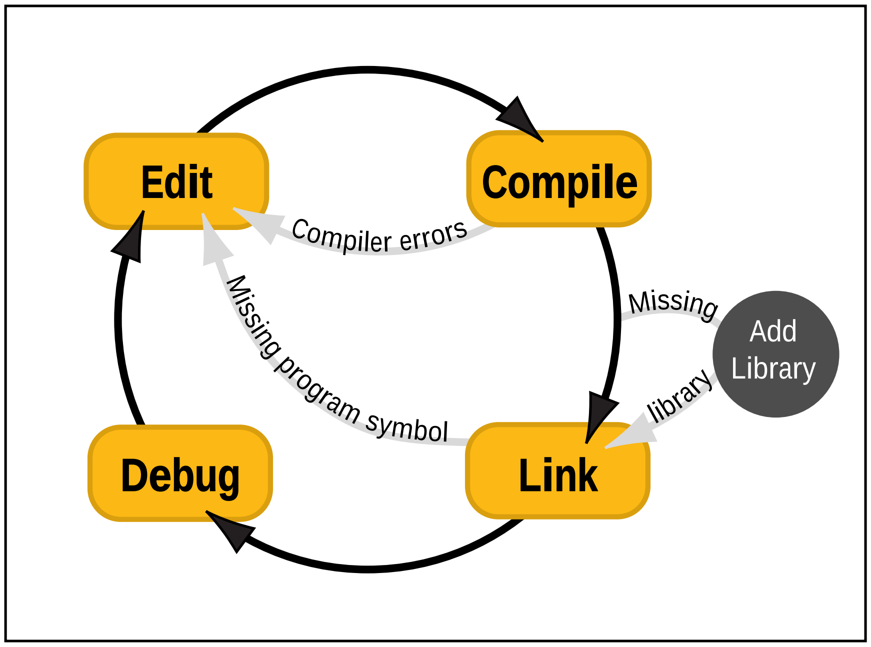


Figure 1: The Development Cycle

[1] *All hosts and languages are not available for every target. Table 1, "Supported CodeWarrior Targets," later in this paper provides the details.*

An engineer can launch the CodeWarrior IDE application once, and stay there throughout the entire development cycle. The IDE keeps state information, symbol tables, and object code in memory to speed operations. The integrated environment keeps track of source code changes so that at the next build, files affected by these changes are automatically compiled and linked into the final executable image. These capabilities let engineers quickly write and test software. The IDE requires 32 MB of RAM.

The IDE application is multi-threaded and contains a set of tightly-coupled code modules that represent the development tools, as shown in Figure 2, "The IDE Architecture." Four of these modules—the editor, compiler, linker, and debugger—implement the four stages of the development cycle, while the remaining modules assist in code navigation and build control. The project manager module controls the entire process. The set of modules consist of:

- an editor
- a source browser
- a search engine
- a build system
- a debugger
- a project manager

Editor

The editor is memory-resident for speed. It colors the keywords for various high-level programming languages to allow easy recognition and navigation. Custom keyword sets can be defined. The editor can automatically verify the balance of parentheses, brackets, and braces. Pop-up menus in each editor window provide instantaneous navigation to the start of any function's code, or into the header files that the program uses.

The editor has a graphic difference engine that displays the differences between two source code files. The contents of the two files are shown side-by-side. As Figure 3, "Differences Displayed" shows, bands of gray painted across the text make any changes stand out. Differences can be easily applied from the source file to the destination file. As

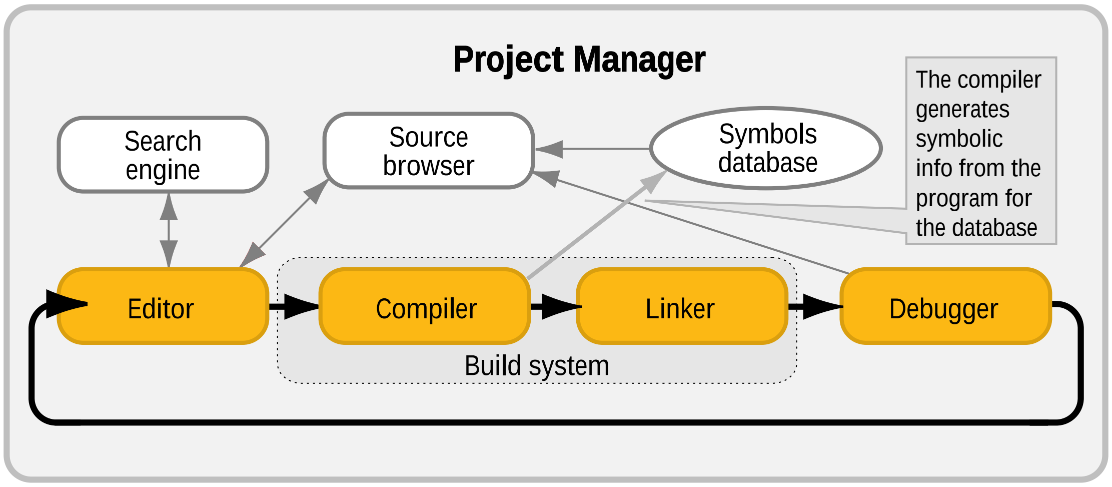


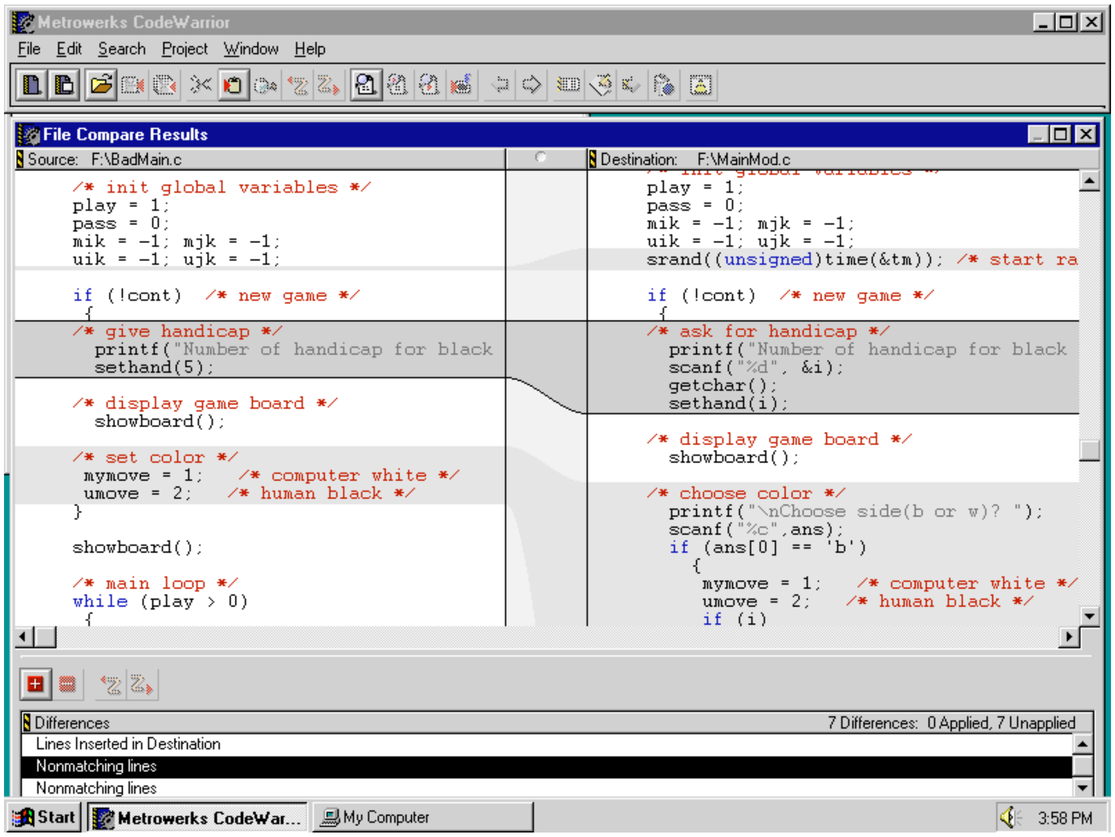
Figure 2: The IDE Architecture

the engineer searches through the code, the difference engine automatically adjusts the scroll rate so that sections of identical text are placed adjacent to one another in the window's two panes. The graphic display and synchronized scrolling of matching text allows the engineer to easily follow changes in the source file.

Source Browser

Working with a database of the program's symbols, the source browser is useful for all supported languages, both object-oriented and procedural. For object-oriented programs, the source browser can present a graphic display of a program's object hierarchies, or show lists of classes and templates. For procedural programs, it catalogs functions, macros, global variables, and other symbols.

The browser greatly expedites code navigation. Every symbol in the database becomes a link to other locations in the code related to that symbol. A right-click on a symbol's name in any editor window or pane in the IDE (including debugger source views) allows the engineer to jump to any of several useful locations related to the symbol. For example, right-clicking a macro name has the editor open the appropriate file and places the insertion point at the start of the macro definition. Like a Web browser, the source browser also maintains a history of views that the engineer can use to return to previously-visited locations in the code.



Source: F:\BadMain.c

```
    /* init global variables */
    play = 1;
    pass = 0;
    mik = -1; mjk = -1;
    uik = -1; ujk = -1;

    if (!cont)  /* new game */
     {
    /* give handicap */
      printf("Number of handicap for black
      sethand(5);

    /* display game board */
      showboard();

    /* set color */
     mymove = 1;   /* computer white */
     umove = 2;   /* human black */
    }

    showboard();

    /* main loop */
    while (play > 0)
     {
```

Destination: F:\MainMod.c

```
    play = 1;
    pass = 0;
    mik = -1; mjk = -1;
    uik = -1; ujk = -1;
    srand((unsigned)time(&tm)); /* start ra

    if (!cont)  /* new game */
     {
    /* ask for handicap */
      printf("Number of handicap for black
      scanf("%d", &i);
      getchar();
      sethand(i);

    /* display game board */
      showboard();

    /* choose color */
      printf("\nChoose side(b or w)? ");
      scanf("%c",ans);
      if (ans[0] == 'b')
        {
         mymove = 1;   /* computer white */
         umove = 2;   /* human black */
         if (i)
```

Differences — 7 Differences: 0 Applied, 7 Unapplied

```
Lines Inserted in Destination
Nonmatching lines
Nonmatching lines
```

Figure 3: Differences Displayed

The source browser's ability to work in concert with the editor and debugger makes the IDE an exceptionally useful maintenance tool. For example, a new engineer can obtain a high-level graphic display of an object-oriented program to determine how its classes are organized and call one another. The engineer can then have the browser summon the editor to display a particular class' source code.

Search Engine

The search engine enables an engineer to locate a specified text string, implements find and replace operations, and performs search operations using regular expressions. It can find a single instance, present a list of all instances, or perform a global replace operation. The search engine can execute in the background as the other tools run. Searches can cover a single file, or a user-defined set of files. In addition, an engineer can easily include or exclude several common groups of files: source files, system header files, and project headers.

Build System

The build system contains the compiler and linker modules. It is responsible for the code generation and object linking that produces the final executable program image. The settings for these two tools are adjusted by graphic controls located in windows of the IDE. Like the search engine, the build system can execute as a background process.

Debugger

CodeWarrior supports several symbolics file formats, including CodeView, Debug With Arbitrary Records Format (DWARF), and SYM. The debugger uses standard symbolics to provide source-code level debugging. The debugger can also display code in assembly, or mixed source and assembly. An engineer can set breakpoints and single-step through the program's code. The contents of variables, arrays, and structures can be monitored. Control variables can be altered on-the-fly to force the thread of execution through a particular section of code. The debugger can also debug threads. A Processes window lets you switch between an application's threads to examine the interactions among them.

The debugger can trap on C++ and Java exceptions. It supports conditional breakpoints (execution stops if a condition is true) and thread-aware breakpoints (execution stops if a particular thread is using the code). The debugger also handles watchpoints. (A watchpoint is a range of memory that's monitored by the debugger until its content changes.) This capability can be used to catch out-of-bounds memory accesses, or detect when a global variable is overwritten.

For difficult problems, the debugger can display the program code as assembly language, and it can view sections of memory. A separate Registers window displays the processor registers, whose contents can be modified by pointing and clicking. For embedded programs, the debugger is capable of tracing through the device's bootstrap code, and it can debug interrupt handlers.

Project Manager

The IDE gathers source, library, graphic resource, and other files into a *project*. Information about the project is stored in a project file, and is manipulated through the project manager.

While the project manager is mentioned last on this tool list, it is responsible for the integrated environment. It manages the relationships among the various files, and tracks critical state information about them (such as when a source file was last edited and compiled). The project manager uses this information to orchestrate the operation of the tools throughout the development cycle.

The project manager is arguably the most revolutionary tool in CodeWarrior. In effect, it generates automatic makefiles. For any given build, the project manager tracks:

- files and libraries
- link order
- dependencies
- compiler, linker, and other settings

The sum of this information is called a *build target*. As the engineer works, this information changes. In response, the project manager updates the build target automatically. When a make command is issued, the project manager uses all the information in the build target to build the program. It invokes the tools with the desired settings, and in the proper order. There is no need to write a separate makefile.

For example, if a library is added to the program, the project manager incorporates this fact into its tracking information. At the next build, the new library is linked into the executable automatically. If a header file is modified, the project manager directs the build system to compile only those source files that rely on information in it. The "makefile" information is updated automatically.

As you can see, in CodeWarrior the state of the project manager (preserved in the build target) is the equivalent of a traditional makefile. In this way the project manager functions like an automake facility, but it does so unobtrusively. The information required to make the code is preserved *automatically* as part of the engineer's ordinary use of the tools. No extra effort is required.

The engineer is also freed from both makefile syntax and makefile semantics. Because no makefile must be written, no syntax must be mastered or debugged. And because of the graphical interface, make information can be *seen*. This makes it much easier for the engineer to understand what is going on, and to modify it appropriately.

This bookkeeping on the project manager's part results in a faster turn-around through the development cycle. Since the build system operates only on files that have changed, it reduces the need for lengthy, comprehensive builds. Automatic dependency tracking also minimizes the delays that occur because a program was built with an object made from an outdated source file.

A project can contain multiple build targets. Each build target within the project can include different files, or specify different compiler/linker settings and libraries. For example, a project might contain a "debug" target and a "release" target. If a program build is requested with the debug target selected, that target's settings direct the IDE to generate debuggable code and symbol tables from the source code. Then the diagnostic libraries are linked into the executable, producing a test application for debugging. When the "release" target is requested, its settings order the IDE to produce object code from the same source files, but without the debugging information and symbol tables. The regular libraries are linked in, building a shipping program. These operations are similar to having one makefile that produces a debug version of the program, and another makefile that produces a shipping version of the program, where both use the same source files.

Organizing Files

The project manager also has significant file-management capabilities.

An engineer manages files through the IDE's Project window. By pointing and clicking in this window, files can be added or removed, and organized into groups. The project manager always knows where files are, and how they relate to each other. As a result, the project manager helps an engineer write code more effectively.

For example, when the compiler reports errors, the engineer can select an error message. The project manager tells the editor to display the code in a text pane accompanying the error window, with the insertion point positioned at the offending source code line. Similarly, an engineer can set a breakpoint while writing code in the editor, and the project manager stores the information so that the breakpoint is implemented during subsequent debugging sessions. If a bug is detected, the project manager can open the correct file at the proper spot, and the code can be edited immediately. The engineer stays focused on solving problems, rather than figuring out which file to open.

Subprojects can also organize a large development project's files into smaller, manageable groups. A *subproject* is simply a CodeWarrior project file (complete with potential multiple build targets) incorporated into another CodeWarrior project. This recursive feature allows you to organize arbitrarily complex projects into a hierarchical collection of modules. The IDE can handle projects that contain thousands of files divided into various subprojects. For precise management of files, CodeWarrior uses special code modules called plugins that integrate the IDE with version control systems (VCS). The IDE currently uses VCS products such as Microsoft's SourceSafe; Synergex's PVCS, and others. In addition, many companies have created custom VCS plugins for in-house use.

In addition to storing the details about a program's files and build settings, the project file saves information about where the files are. This information is known as an *access path*. The project manager uses this description of the directory tree to locate files. This arrangement speeds build and search operations because the IDE knows where the source files are, and only has to look in those directories mentioned in the access paths.

The description of the access paths can be absolute or relative. Absolute paths use a complete description of the directory tree to locate a file (for example, C:\MYPROJECTS\CELLFONE\SOURCE\FILTER\FOURIER.C).

A relative path to a file starts *in relation to a particular directory*. The IDE recognizes three types of user-specified relative access paths. These are:

- project relative—the path starts at the project file's directory
- compiler relative—the path starts at the CodeWarrior directory
- system relative—the path starts at an operating system directory

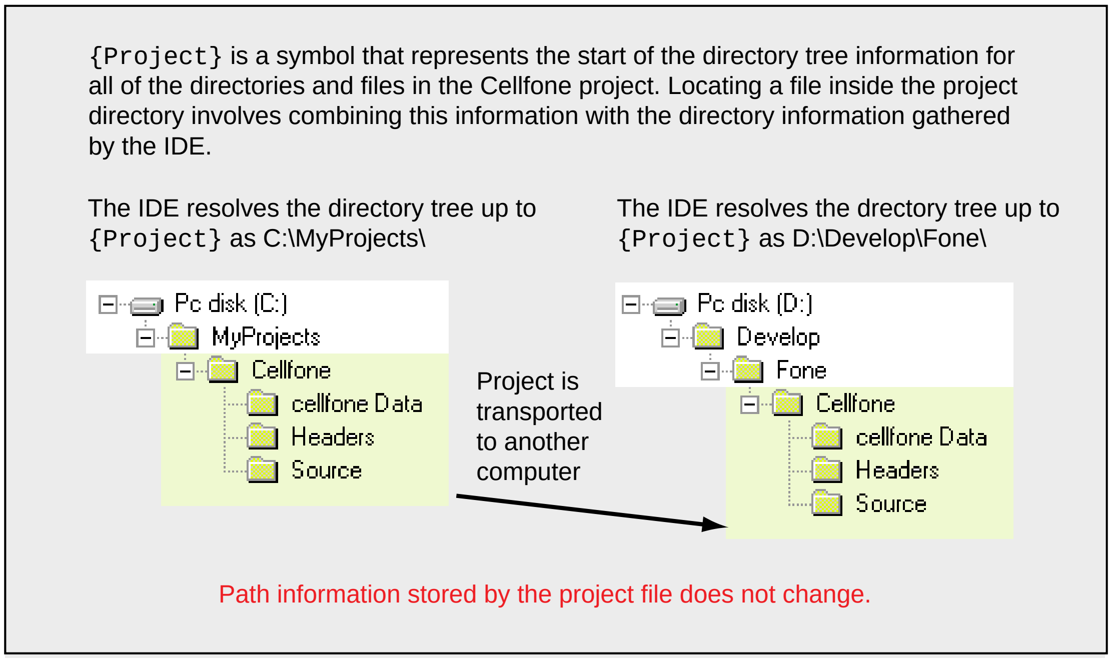

{Project} is a symbol that represents the start of the directory tree information for all of the directories and files in the Cellfone project. Locating a file inside the project directory involves combining this information with the directory information gathered by the IDE.
The IDE resolves the directory tree up to {Project} as C:\MyProjects\
The IDE resolves the drectory tree up to {Project} as D:\Develop\Fone\
Pc disk (C:)
MyProjects
Cellfone
cellfone Data
Headers
Source
Pc disk (D:)
Develop
Fone
Cellfone
cellfone Data
Headers
Source
Project is transported to another computer
Path information stored by the project file does not change.

Figure 4: Relative Access Paths

Using relative paths makes the project file and all of a program's directories easily transportable. Say that there's a directory named CELLFONE on a server. This directory contains the CELLFONE project file, a source file directory for the CELLFONE program, and a custom header directory. If several engineers copy CELLFONE to their computers, they can open the project file with the IDE and begin working. That's because the IDE first resolves the root of the directory tree for the host computer (it might be C:\MYPROJECTS\ on one computer and D:\DEVELOP\FONE\ on another). Then it completes the path description by adding in the relative information obtained from the project file, as shown in Figure 4, "Relative Access Paths." The IDE can interpret DOS, UNIX, and Mac OS path names. Since the project file stores information about the program's build target, settings, and access paths in one format, the project file can copied to another platform and the resident CodeWarrior IDE recovers the project information. This capability enables project directories to be easily shared among programming teams, regardless of their choice of platform or how a computer's directories are organized.

The absolute path feature is useful in those situations where it's desirable to have the IDE read header files from a server, or output a library file to a server. However, absolute paths for local files are almost certain to break if a project or file is moved to a new location.

Plugin and Play

So far, this paper has discussed how the IDE provides a fast mechanism for writing and testing code, all while offering a choice in platform. Thanks to access paths, this flexibility comes without creating problems when files are shared among different computers.

The IDE can lower another hurdle during a device's initial design phase by providing freedom of processor choice, without having to worry about learning how to use a new set of development tools. As mentioned in the introduction, CodeWarrior offers engineers a choice in programming languages and processor targets. This capability is a result of the IDE's plugin architecture, especially as it applies to the build system.

A *plugin* contains software that provides specific services to the IDE on demand. This software is stored in a file separate from the IDE application, typically as a DLL for Windows, and as a shared library for Mac OS and UNIX. CodeWarrior plugins behave like the plugins used by Web browsers or Adobe Photoshop. That is, if the plugin file is present when the IDE starts, it automatically augments the IDE's capabilities.

CodeWarrior currently recognizes several different kinds of plugins: compiler, linker, pre-linker, post-linker, preference panel, and version control plugins all exist today. APIs for debugger and other kinds of plugins are available or under development. Metrowerks has almost 200 different plugins in use or in the works.

Each kind of plugin communicates with the IDE through a platform-neutral, fully documented, Application Programming Interface (API). This enables third-party developers to create fully-integrated additions to the CodeWarrior armory of tools.

The build system uses preference panel, compiler, linker, pre-linker, and post-linker plugins to provide flexibility at many levels. By combining the CodeWarrior IDE with the appropriate plugin tools, the same environment compiles and links code for a multitude of processor targets.

Preference Panel Plugins

Preference panel plugins supply the IDE with the graphic interface the engineer uses to control a particular tool or set of tools. The C/C++ Language panel, for example, has settings that enable language features, control function in-lining, and check for conformance to the ANSI C standard. Other panels control code generation, processor-specific optimizations, and how the linker structures the machine code for use with a particular operating system.

Anatomy of a Compiler

A compiler plugin is a single entity that encompasses the entire CodeWarrior compiler architecture. On the outside, a CodeWarrior compiler looks like any other. It is specific to a particular language and target processor. On the inside, however, the compiler is very different. It is this difference that is the key to CodeWarrior's flexibility.

The compiler plugin combines a front-end language parser with a back-end code generator. The language parser converts the source code into an internal, language-neutral format known as the Intermediate Representation (IR), as shown in Figure 5, "CodeWarrior Build Architecture." The IR consists of data tokens that represent a tree-

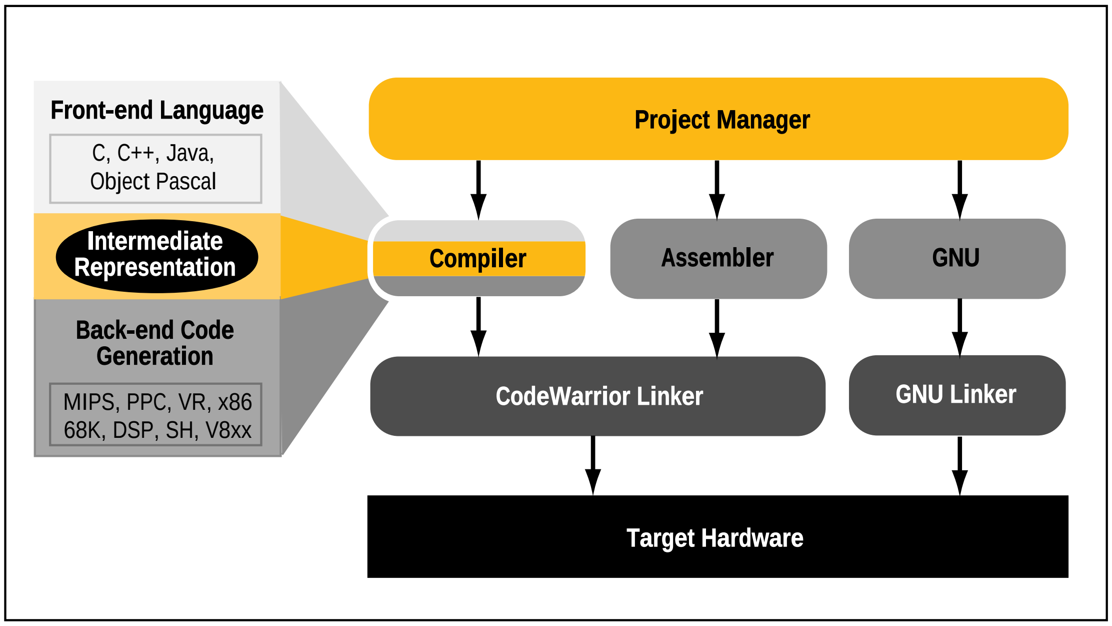


Figure 5: CodeWarrior Build Architecture

based, syntactically complete and accurate expression of the source program. Next, generic algorithmic optimizations are done on the IR, such as dead code removal. The code generator reads the IR and translates it into native code for a particular processor. In addition, the code generator performs any processor-specific optimizations, such as instruction scheduling. Various preference panels control how the compiler operates with respect to language, optimization, and code generation.

By factoring the compiler into three parts—front-end language parser, IR, and back-end code generator—the Metrowerks engineers can readily develop compilers for new targets. And that they have done. Table 1, "Supported CodeWarrior Targets," shows the choices available for targets, and mentions some additional targets under development.

More importantly, this factored design also benefits the developer. If an engineer wants to retarget high-level code to another processor, he uses a compiler with the appropriate back-end code generator. Recall that every CodeWarrior compiler uses the same front-end language parser, so language-related porting problems are nonexistent.

The IDE also supports assembly language. Since an assembler generates machine code directly, the IR stage is not used. The IDE's flexibility even extends to the use of command-line compilers and linkers such as GNU tools. However, it is CodeWarrior's own modular build system for high-level languages that provides the truly significant benefits in compiler technology.

First, any enhancement to the front-end language (such as in the C/C++ parser) or the IR means that all the back-ends benefit from the improvement. For example, because of optimizations in the IR, each back-end can begin its work using highly-optimized code.

Table 1: CodeWarrior Targets (items in *italic* are under development)

TARGET	**HOST**	**HIGH-LEVEL LANGUAGE**
Embedded Processors		
68K family	Windows 95/98/NT	C, C++
DSP (Motorola 56800)	Windows 95/98/NT	C
StarCore	Windows 95/98/NT	C
M•CORE	Windows 95/98/NT	C, C++
MIPS	Windows 95/98/NT, *Solaris*	C, C++, Java
MIPS 16	Windows 95/98/NT	C, C++
NEC VR	Windows 95/98/NT	C, C++
PowerPC	Windows 95/98/NT, Solaris	C, C++, Java
SHx	*Windows 95/98/NT*	*C, C++*
V8xx	Windows 95/98/NT	C, C++
Real-Time Operating Systems [1]		
Nucleus (PowerPC, MIPS, V830)	Windows 95/98/NT	C
Palm OS (68K)	Windows 95/98/NT, Mac OS	C, C++
QNX Neutrino (x86, PowerPC, *MIPS*)[2]	Windows 95/98/NT	C, C++
VxWorks (PowerPC)	*Windows 95/98/NT, Solaris*	*C, C++*
Precise/MQX[3]	Windows 95/98/NT	C, C++
RT XC[3]	Windows 95/98/NT	C, C++
Embedded Linux	*Windows 95/98/NT*	*C, C++*
Proprietary Devices		
AGCS GTD-5 Telecom Switch (x86)	Windows 95/98/NT	Pascal
NewBridge Mainstreet 3600 multiplexor switch (68K)	Mac OS, Solaris	Pascal
Nortel Proximity i wireless switch (PowerPC)	Windows 95/98/NT	C
Nintendo64 (MIPS)	Windows 95/98/NT	C, C++
PlayStation Console (MIPS)	Windows 95/98/NT	C, C++
PlayStationII Console (MIPS)	Windows 95/98/NT	C, C++
Sega Dreamcast	Windows 95/98/NT	C, C++
Desktop Processors/Operating Systems		
Java Virtual Machine	Windows 95/98/NT, Mac OS, Solaris	Java
Linux (x86, PowerPC)	*Red Hat Linux, SuSE Linux*	*C, C++, Java*[4]
Mac OS (PowerPC, 68K[5])	Windows 95/98/NT[6], Mac OS	C, C++, Pascal
Novell NetWare (x86)	Windows 95/98/NT	C, C++
Win32 (x86)	Windows 95/98/NT, Mac OS	C, C++

1 Has support for system-state (kernel-aware) debugging. Other RTOS's can be used with CodeWarrior. Include the RTOS source code (if available) or the RTOS as a library.

2 The x86 and PowerPC versions are shipping. The MIPS version is still under development.

3 Third-party kernel-awarness add-ons.

4 CodeWarrior for Linux uses the GNU C/C++ compilers supplied with the Linux installation. The Java tools use Metrowerks' compilers and libraries.

5 CodeWarrior can generate binaries for the 68K Mac OS, but the tools run only on the PowerPC platform.

6 A Mac-specific resource editor and debugger are not available on the Windows platform.

Second, as new processors come onto the market, the IDE can quickly support them. All that's required is the addition of a suitable set of plugins, including the compiler. Much of the work involved in creating a new compiler is already done: the language front-end and the highly-optimized IR are reused in each CodeWarrior compiler.

Linking the Pieces

In addition to preference panel and compiler plugins, the build system also uses linker plugins. The linker plugin reads object code, resolves any references (such as function calls to other objects), and then links these objects and libraries to generate a final binary file. The resulting file might be an executable program, a DLL, a static library, an embedded operating system image, or a driver.

The plugin architecture allows for the possibility of having more than one linker for a particular target. More commonly, the tool chain will use a pre-linker or a post-linker plugin to provide flexibility in the linking process.

A pre-linker plugin performs some sort of processing before the linker works on the object code. For example, the JavaDoc pre-linker reads Java source files and creates standard Java documentation from function comments.

A post-linker plugin performs additional processing after the linker produces a binary file. For example, the FTP post-linker automatically forwards the final binary to a remote site. Some post-linkers convert a standard binary file into a proprietary binary format.

The plugin architecture puts no boundaries on what CodeWarrior can do, and is limited only by the imagination. For example, a compiler is simply an intelligent tool that translates a structured text file (source code) into another format (machine code). Plugin tools can operate on other kinds of data. A "compiler" plugin can convert data from a drawing program into bitmaps for use in a computer game. In this way the IDE can serve as a powerful batch file management system for files of any kind. Seamless plugins have been developed at many companies to solve unusual or proprietary problems.

From a wider perspective, because CodeWarrior presents the same IDE regardless of host, language, or target, many hurdles that confront system designers are lowered. For example, CodeWarrior opens the possibility of a low-cost port of existing code to another processor. The same projects and high-level source code files can be reused, while the IDE uses a different compiler to generate code for the new processor. If some of the code must be modified, the engineers don't have to learn a new set of tools.

Another example is a software project for two devices requiring different processors and hardware. Perhaps a program for a hand-held computer also has a component for a desktop computer. A single CodeWarrior project would contain a build for each target. The two builds share code common to both devices. Any change in the common code affects both builds automatically. Device-specific code or libraries would be included through the appropriate build. Producing code for one device becomes a simple matter of choosing the proper build target.

While this plugin architecture provides choices and flexibility, what does an engineer do when the requirements of a development project push beyond the boundaries of what the IDE offers? CodeWarrior provides options in this area as well.

Extensible Tools

Engineers are prone to using development tools in unanticipated ways. One size does not fit all in software development. To an increasing degree, CodeWarrior is adopting the tool chain model. In this model, loosely-coupled tools process code from inception to completion. An engineer can replace any tool in the chain with any other suitable tool.

While a loosely-coupled tool chain might seem to be the antithesis of an IDE, CodeWarrior is compatible with and supports the concept. Although CodeWarrior maintains a comprehensive and tightly integrated tool set, substitutions are permitted in the CodeWarrior chain of tools. Any time you replace a component in a tightly-integrated system there can be disappointments. Nevertheless, familiarity with the replacement tool can make the substitution worthwhile to a development team.

The CodeWarrior development environment can be modified in two ways: by replacing an existing IDE component, or by adding a component. Figure 6, "Extending the IDE," summarizes the extensibility of the CodeWarrior IDE.

Because the project manager makes up the core of the IDE, it can't be replaced. Nevertheless, the architecture does permit substitutions for other parts, although with reduced integration in some cases. Components that can be replaced include the editor, search engine, source code browser, compiler and linker, or debugger.

Engineers defend their choice of source code editor with evangelical fervor. This comes as no surprise, since they spend most of their time writing code. In recognition of this, the IDE does provide support for third-party editors. An engineer can write code in any editor that saves information as plain text, and CodeWarrior will compile and build the

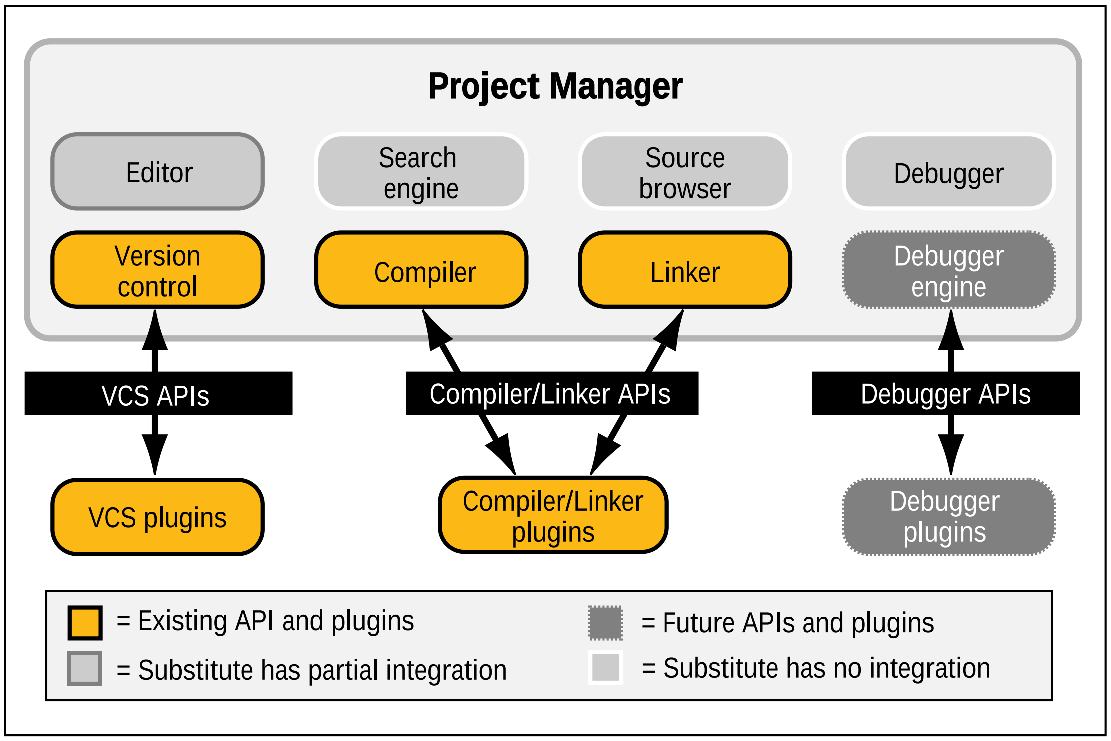


Figure 6: Extending the IDE

code. Integration of the replacement editor with the IDE (such as launching it from within CodeWarrior) will be improved over time. However, the CodeWarrior editor cannot be fully replaced. Editable text panes (using the CodeWarrior editor) appear in many other IDE components such as the source code browser and the search engine.

An engineer may also use a third-party search engine or source code browser as external tools. However, this arrangement lacks any integration with the IDE, which in turn impacts how fast someone navigates through the code. While theoretically possible, practically speaking there should be little reason to replace these components.

CodeWarrior will soon provide support for command-line compilers and linkers, such as GNU tools. An adapter technology is under development that allows them to be incorporated into the IDE's build system without modification. This API defines a default interface panel where the engineer enters the options (or switches) that direct the compiler's code generation and the linker's operation. The adapter pipes this information from the IDE into the GNU tool, and pipes any output from the tool's stdio interface back into the IDE. The command-line adapter API lets an engineer use the IDE's capabilities without having to change compilers or port legacy code. The IDE can even receive and display error messages from the command-line tool. However, there are some features that might be absent. For example, a command-line tool may not generate browser information. However, an engineer could modify the GNU tool to pipe browser information into the IDE through a compiler/linker API, described below.

Finally, an engineer is not limited to the IDE's integrated debugger. The IDE's build system generates symbolic information in standard formats such as CodeView and DWARF. An engineer can use any debugger that supports these standards. At this time, third-party debuggers are not integrated. Some features are lost, such as the ability to use breakpoints set in the CodeWarrior editor.

For debugging, a simple replacement is not always the ideal. It would be best if a fully-integrated tool was available. A debugger API will be published in November 1998 that should enable third-party debuggers to interoperate with the IDE's debugger interface.

For other IDE components, the ideal of a fully-integrated custom tool has already arrived. The Compiler/Linker API was first released in the last quarter of 1995, and the Version Control System API was released in the last quarter of 1996.

The Compiler/Linker API is open and public. It defines an abstract, cross-platform interface that supports compilers and linkers, pre-processing and post-processing stages,

Table 2: Version Control Software plugins for CodeWarrior (items in *italic* are under development)

PRODUCT	PLATFORM	MANUFACTURER[1]
CVS	Mac OS	Electric Fish
CVS	*Solaris*	*Metrowerks*
Perforce	Mac OS	Perforce
PVCS[2]	Mac OS	Synergex
SourceServer/Projector	Mac OS	Metrowerks, Electric Fish
Visual SourceSafe	Windows 95/NT, Mac OS	Metrowerks
VOODOO	Mac OS	Uni Software Plus

1 Some companies have their own in-house plugins for VCS.

2 Version 5.2.10 or later.

and disassembly operations. The API provides access to settings data, so that a plugin receives compile/link options from the user. It documents how object data (generated code) or symbols (created by compilers for the source browser) are stored and retrieved. This API enables a third party to create a fully-integrated compiler and linker for a different language or processor, as well as create pre- and post-linker utilities.

The VCS API is also cross-platform, and its plugins provide support for external version control software, as shown in Table 2. Support for other source control vendors is in progress. The VCS API is available upon request from Metrowerks. It manages functions such as establishing a connection to a file database, file check in/out, determining a file's status, plus logging and displaying a file's modification history. Metrowerks works closely with vendors developing VCS plugins to continually refine the VCS API.

CodeWarrior's integration with third-party VCS plugins enables it to track and maintain the integrity of multi-user development projects, projects whose members might be located across the country, or around the globe.

Conclusion

The CodeWarrior IDE's flexible architecture provides choices at several levels. At the production level, the IDE enables code to be written faster and better with highly-responsive tools. Just as important, CodeWarrior offers choices at the development level: the host platform, the programming language, the processor, and the target operating system. The architecture's extensibility also provides real choices at the tool level: Metrowerks' own integrated tools, integrated third-party tools, non-integrated third-party tools, and the ability to adapt in-house tools.

The many choices available during production, development, and tool selection result in an environment of unique capabilities. This frees engineers to choose the optimal path to their goal, using CodeWarrior to clear the hurdles between an idea and its realization.