Object Tools

http://www.object-tools.com

EiffelS
for Macintosh CodeWarrior

Installation and Usage

Guide

"Think Different ... Think Eiffel!"

VERSION: 1.0

CONTENTS

1. WELCOME

Welcome to Object-Tools EiffelS for CodeWarrior on Macintosh. Included in this release are the EiffelS 2.0 compiler, the EiffelS CodeWarrior preference panel, Eiffel Kernel libraries, the *Mac OS Toolbox Eiffel Library* (*MOTEL*), and example projects. Follow these instructions to install these files in the correct places, and then how to set up your own Eiffel projects.

1.1. EiffelS for CW Licences

If you have loaded this off a free CD or from the Internet you may use this under the Lite licence. With this licence you can use EiffelS for CW on a trial basis. You should not use the Lite form for extended periods, and may not use it to release commercial software, or to deploy software for use of others within your company. The Lite licence limits the size of systems that you can compile, so once your application grows to a certain size, you will get a syntax error and not be able to generate code. We want to make it as easy as possible for you to use EiffelS, but in order to provide support and better future releases, we are dependent on your support.

The commercial licence entitles you to full compilation facilities, and to deploy and resell software created with EiffelS for CodeWarrior. Contact Object-Tools at

i.joyner@acm.org (Ian Joyner Australia)

or

fm@object-tools.com (Frieder Monninger Europe)

gfrank@object-tools.com (Gudrun Frank US)

to obtain a full licence.

Also see the Object Tools Web site at:

http://www.object-tools.com.

The full licence costs $US149. This licence includes the full compiler, and the MOTEL library. This is for a single seat licence. For site licences and University department licences, please contact Object-Tools. Full support is also not included with the Lite version, however, we will accept comments and problems that you encounter, especially since you will probably want any problems you have with your installation ironed out before you pay for a full licence.

If you do not have Internet access, please send International Money Order (personal cheques not accepted except in Australian dollars) to:

Object Tools
attn. Ian Joyner
14 Summerhaze Place
Hornsby Heights
Australia 2077
Phone +61 2 9477 3474

Object Tools
attn. Gudrun Frank
418 Parkview Way
Newtown, PA 18940
Phone 215-504 0854

Object Tools
attn. Frieder Monninger
Nordstr. 5
D 35619 Braunfels
Phone: 6472 911 030 Fax: 6472 911 031

Please enclose your name and address (email address) for us to return a full licensed version.

1.2. Suggested Configuration

PowerPC Mac with at least 32MB memory. CodeWarrior Pro 2 (should work on 1, might work on earlier).

2. INSTALLING

2.1. Installation Procedure

You will have downloaded two files, *EiffelSCWCore.sit* and *EiffelSCWSupport.sit* both files are required.

1. Copy the compiler (*EiffelS*) to your *CodeWarrior Plugins:Compilers* folder.

2. Copy the EiffelS preference panel (*EiffelS Panel*) to your *CodeWarrior Plugins:Preference Panels* folder.

3. Copy the *EiffelS2* libraries folder to your *MetroWerks:MetroWerks CodeWarrior* folder.

4. Copy the *Eiffel* folder from the *EiffelS2:(Project Stationary)* folder to your *MetroWerks:MetroWerks CodeWarrior:(Project Stationary)* folder.

Your EiffelS installation is complete.

2.2. Example Projects

There are several example projects included in the release, *HelloWorld*, *SillyBalls*, and *Mondrian*. You should examine and compile these first, since this is a good introduction working with EiffelS for CodeWarrior and to programming with the *MOTEL* library. Note the first compiles will take a long time because all libraries must be compiled for each project. Subsequent compiles are much faster.

3. Creating your own CodeWarrior Eiffel Project

EiffelS for CodeWarrior has been developed from the UNIX version of EiffelS to be as closely integrated with CodeWarrior as possible. To this end, we have changed quite a bit from the UNIX version to do things in a more Mac-like way and to make things more convenient for the programmer. Follow the instructions and guidelines in this section in order to set up your Eiffel projects with as little fuss as possible.

Those familiar with creating projects with CodeWarrior will find creating an Eiffel project straightforward because there is very little difference. Most of the information in this section can be skipped, but we include it for completeness.

3.1. Creating a project

- Choose **New Project** from IDE File menu

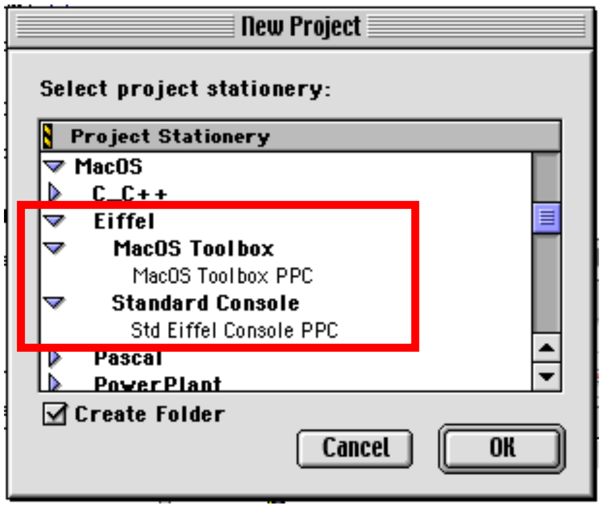


Figure 1. New Project Panel

- Open the **MacOS/Eiffel/** path and one of:

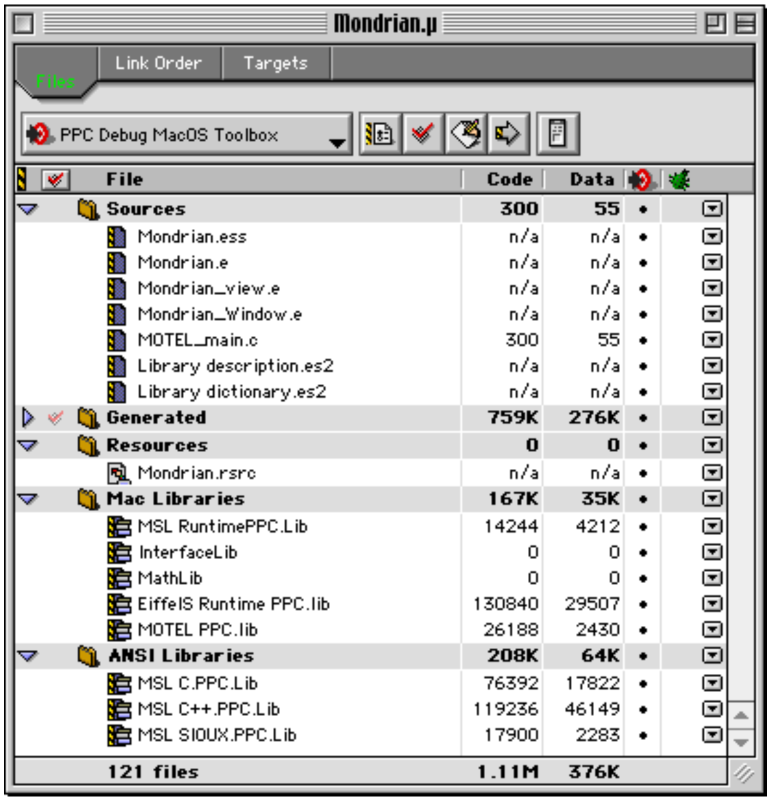
MacOS Toolbox PPC for a Toolbox project

Standard Console PPC for a text based console project.

Note: if the Eiffel section does not appear in the New Project Panel, make sure you copy the *Eiffel* folder from the *EiffelS2:(Project Stationery)* folder to the *Metrowerks CodeWarrior:(Project Stationery)* folder.

- Choose a project name. This name must match the project name you will later enter in the **EiffelS 2.0 Language panel**, so this name should not contain any special characters or blanks.
- Ensure the path is in the *MetroWerks:MetroWerks CodeWarrior* folder and choose **save**. (The Eiffel project folder must be in here because CodeWarrior makes this the starting directory for the compiler, so it is important for the compiler to be able to find things.)

You now have a default Eiffel project that looks as in figure 2.



File	Code	Data	
Sources	**300**	**55**	•
Mondrian.ess	n/a	n/a	•
Mondrian.e	n/a	n/a	•
Mondrian_view.e	n/a	n/a	•
Mondrian_Window.e	n/a	n/a	•
MOTEL_main.c	300	55	•
Library description.es2	n/a	n/a	•
Library dictionary.es2	n/a	n/a	•
Generated	**759K**	**276K**	•
Resources	**0**	**0**	•
Mondrian.rsrc	n/a	n/a	•
Mac Libraries	**167K**	**35K**	•
MSL RuntimePPC.Lib	14244	4212	•
InterfaceLib	0	0	•
MathLib	0	0	•
EiffelS Runtime PPC.lib	130840	29507	•
MOTEL PPC.lib	26188	2430	•
ANSI Libraries	**208K**	**64K**	•
MSL C.PPC.Lib	76392	17822	•
MSL C++.PPC.Lib	119236	46149	•
MSL SIOUX.PPC.Lib	17900	2283	•
121 files	**1.11M**	**376K**	

Figure 2. Eiffel Project Panel

3.2. Checking Project Settings

You should check the following are set up correctly, or tailor the settings as desired.

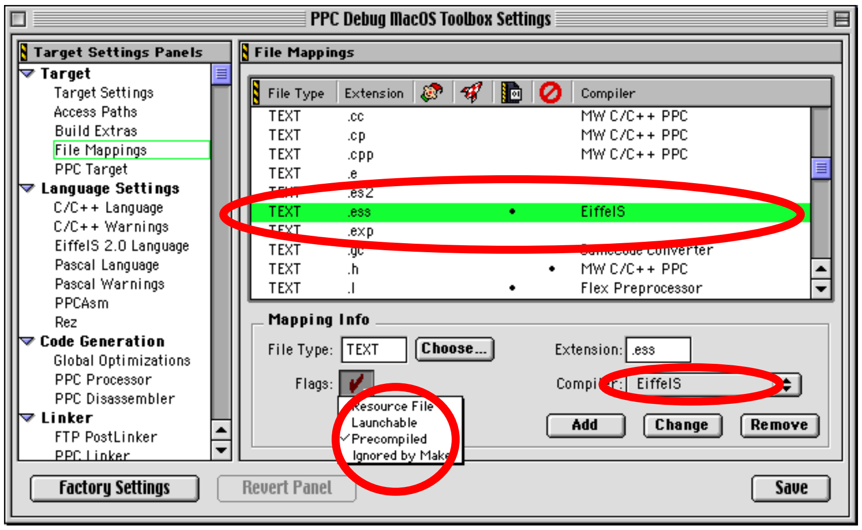
- Choose **<Project> Settings** from **Edit** menu or click **Settings** button in project panel.
- Target Settings

  Leave as is for chosen project or tailor as desired.
- Access paths

  Leave as is for chosen project or tailor as desired.
- Build Extras

  Leave as is for chosen project or tailor as desired.
- File Mappings

  The following settings will be set in the File Mappings panel as shown in figure 3.



PPC Debug MacOS Toolbox Settings

Target Settings Panels
- Target
  - Target Settings
  - Access Paths
  - Build Extras
  - File Mappings
  - PPC Target
- Language Settings
  - C/C++ Language
  - C/C++ Warnings
  - EiffelS 2.0 Language
  - Pascal Language
  - Pascal Warnings
  - PPCAsm
  - Rez
- Code Generation
  - Global Optimizations
  - PPC Processor
  - PPC Disassembler
- Linker
  - FTP PostLinker
  - PPC Linker

File Mappings

File Type	Extension					Compiler
TEXT	.cc					MW C/C++ PPC
TEXT	.cp					MW C/C++ PPC
TEXT	.cpp					MW C/C++ PPC
TEXT	.e					
TEXT	.es2					
TEXT	.ess			•		EiffelS
TEXT	.exp					
TEXT	.gc					GameCode Converter
TEXT	.h				•	MW C/C++ PPC
TEXT	.l			•		Flex Preprocessor

Mapping Info

File Type: TEXT
Extension: .ess
Compiler: EiffelS
Flags: Resource File, Launchable, ✓Precompiled, Ignored by Make

Figure 3. File Mappings Preference Panel

.e files allow you to add Eiffel files to your project for convenience. CodeWarrior does not pass these files directly passed to the compiler. The EiffelS compiler searches for and processes all *.e* files in the libraries and your project in a single execution. This is the reason for the dummy file (*.ess*)—it fools CodeWarrior to start the Eiffel compiler; the compiler does not use this

file (hence it is completely arbitrary). The *.e* files do not need to be added to your CodeWarrior project for the compiler to process them.

The *.es2* extension allows the *library dictionary* and *library description* files to be added to your project.

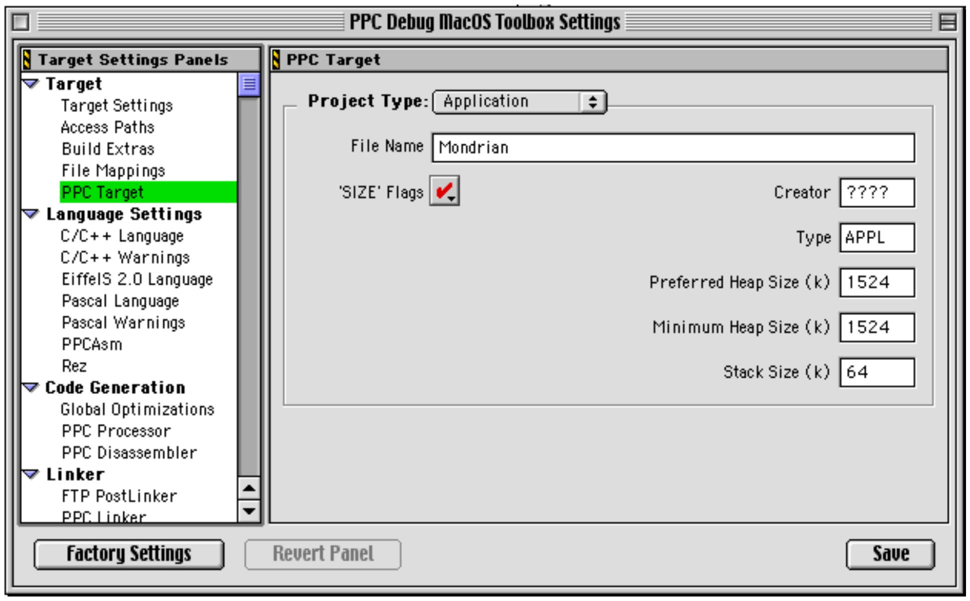
Note that an Eiffel compiler must process all files in one execution because the definition of the Eiffel language requires many cross module consistency checks. Most other languages leave such checks to a linker that often gives obscure errors (if the linker catches them at all, and if not you have to debug such errors at run time). Hence an Eiffel compiler will take longer compiling than other compilers, but the thorough checking will save much more debugging time and frustration. EiffelS also keeps much information from its first and second passes in database files, so once the files are compiled the first time, subsequent compiles are a lot faster. The third pass generates the C files, and these are regenerated only when the source file has been changed, so once Pass 3 is completed, subsequent compiles are again much faster.

This also removes the need to create module header files separately, and to specify module dependencies (via *make*, etc.)

- **<Processor> Target**

These settings are shown in figure 4. In the **File Name** put whatever file name you want.

The **Preferred** and **Minimum Heap Size** should be set to at least 684K. This is because Eiffel performs automatic memory management for you, and it reserves 512K memory blocks for its objects. Thus the smallest block must be at least 512K plus a little extra. The Eiffel runtime will automatically allocate extra 512K blocks as needed if the space required for active objects increases beyond 512K. You should increase your memory sizes for your application appropriately if this happens.



PPC Target

Project Type: Application

File Name: Mondrian

'SIZE' Flags

Creator: ????

Type: APPL

Preferred Heap Size (k): 1524

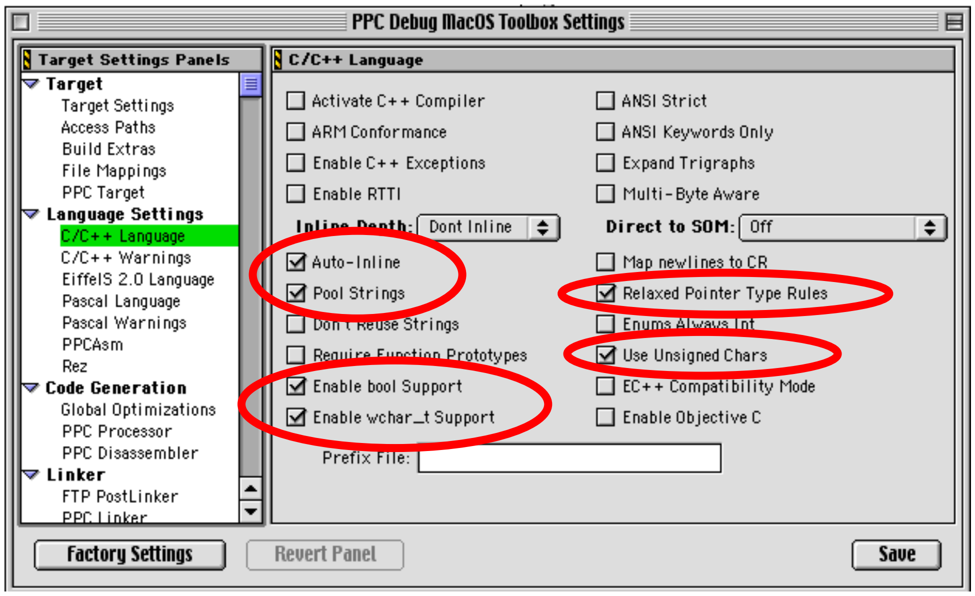
Minimum Heap Size (k): 1524

Stack Size (k): 64

Figure 4. PPC Target Preference Panel

- **C/C++ Language**

  These settings are shown in figure 5. Everything in this panel should be disabled, except **Relaxed Pointer Type Rules**, **Use Unsigned Chars**, and **Enable bool Support** - these must be checked. **Prefix File** must be blank. **Auto-Inline**, **Pool Strings**, and **Don't Reuse Strings** may be set optionally, but will effect your memory requirements.



C/C++ Language

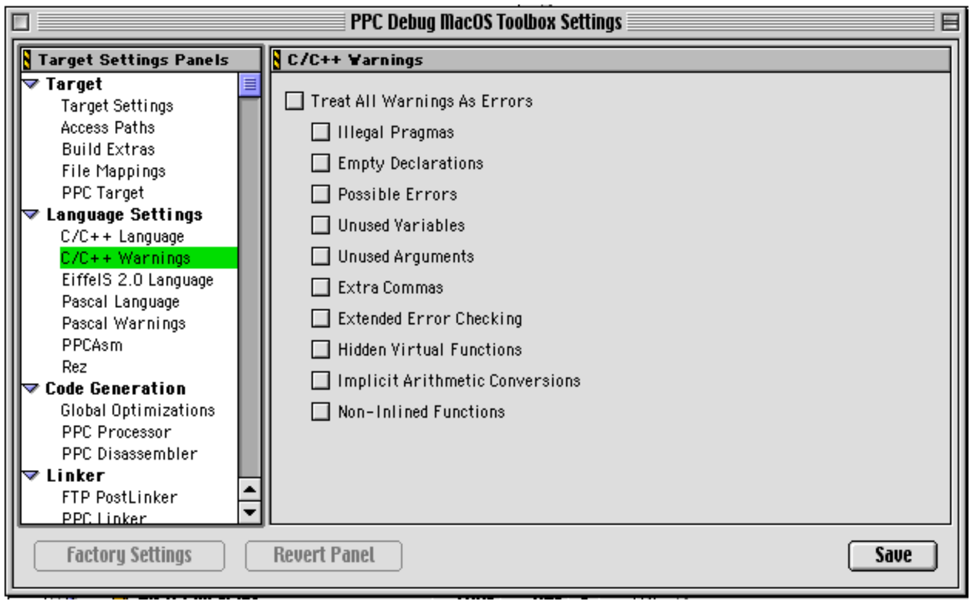
☐ Activate C++ Compiler	☐ ANSI Strict
☐ ARM Conformance	☐ ANSI Keywords Only
☐ Enable C++ Exceptions	☐ Expand Trigraphs
☐ Enable RTTI	☐ Multi-Byte Aware
Inline Depth: Dont Inline	Direct to SOM: Off
☑ Auto-Inline	☐ Map newlines to CR
☑ Pool Strings	☑ Relaxed Pointer Type Rules
☐ Don't Reuse Strings	☐ Enums Always Int
☐ Require Function Prototypes	☑ Use Unsigned Chars
☑ Enable bool Support	☐ EC++ Compatibility Mode
☑ Enable wchar_t Support	☐ Enable Objective C

Prefix File:

Figure 5. C++ Language Preference Panel

- **C/C++ Warnings**

These are all reset—don't worry Eiffel has done many more checks to ensure your code is correct than C/C++ will ever do. This is shown in figure 6.



PPC Debug MacOS Toolbox Settings

Target Settings Panels
- Target
  - Target Settings
  - Access Paths
  - Build Extras
  - File Mappings
  - PPC Target
- Language Settings
  - C/C++ Language
  - C/C++ Warnings
  - EiffelS 2.0 Language
  - Pascal Language
  - Pascal Warnings
  - PPCAsm
  - Rez
- Code Generation
  - Global Optimizations
  - PPC Processor
  - PPC Disassembler
- Linker
  - FTP PostLinker
  - PPC Linker

C/C++ Warnings

☐ Treat All Warnings As Errors
- ☐ Illegal Pragmas
- ☐ Empty Declarations
- ☐ Possible Errors
- ☐ Unused Variables
- ☐ Unused Arguments
- ☐ Extra Commas
- ☐ Extended Error Checking
- ☐ Hidden Virtual Functions
- ☐ Implicit Arithmetic Conversions
- ☐ Non-Inlined Functions

Figure 6. C++ Warnings Preference Panel

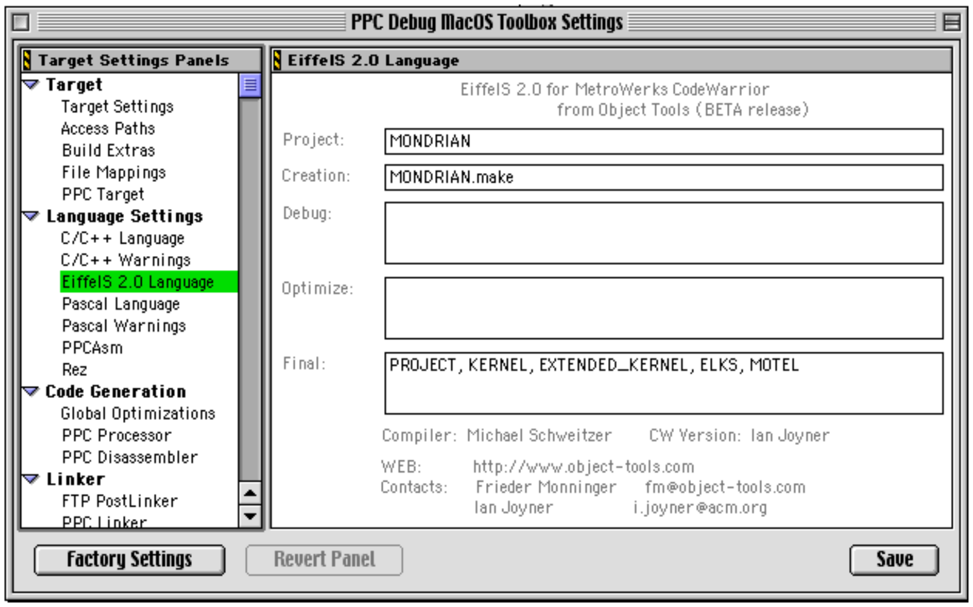
- **EiffelS 2.0 Language**

Enter the **Project** name. This must match the folder name where you saved the project when you created the project. The name must not contain special characters or blanks.

Enter the **Creation** class and routine of your Eiffel system. This will be the name of the class that is first executed, and the routine that the runtime executes. The format for this is *<class>.<routine>*, for example, *HELLO.make*, *MY_APPLICATION.get_going*, etc.

Enter *PROJECT* in the **Debug**: text box. Enter the names of the Eiffel libraries you will be using in either the **Debug**, **Optimize**, or **Final** box. These boxes enable different levels of runtime checking. **Debug**, for example, enables all assertions in preconditions, postconditions, etc., and enables the Eiffel source language debugger.

Optimize enables only preconditions and postconditions. **Final** disables all assertions. The names of the libraries match the names given in your projects *Library dictionary.es2* file (see later). These names will normally match the library folders in the *EiffelS2:Library* folder, for example *ELKS*, *EXKERNEL*, *RUNTIME*, *CONTAINER*, *MOTEL*. You can also develop your own libraries that other projects will use. The EiffelS preference panel is shown in figure 7.



EiffelS 2.0 for MetroWerks CodeWarrior
from Object Tools (BETA release)

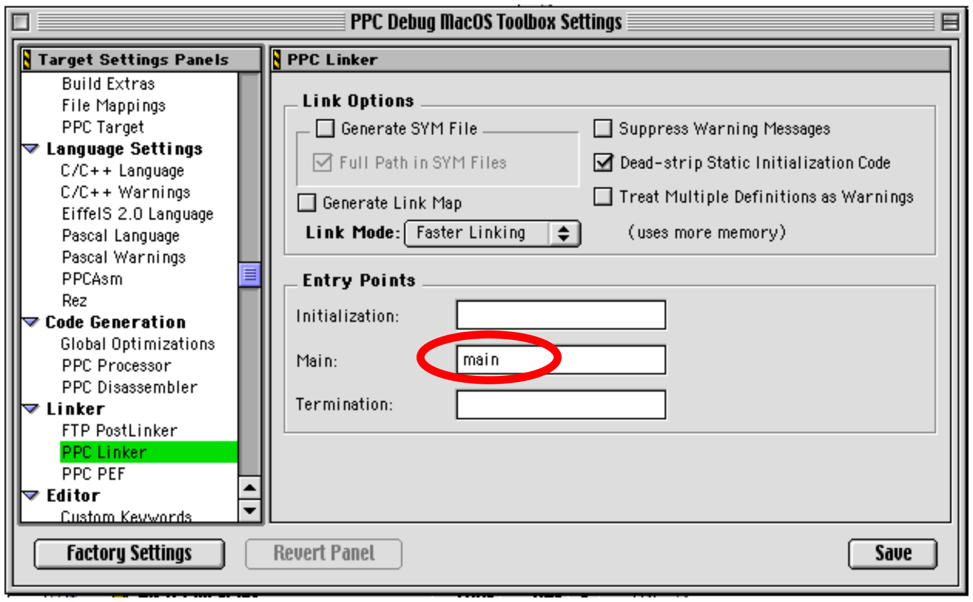
Project: MONDRIAN
Creation: MONDRIAN.make
Debug:
Optimize:
Final: PROJECT, KERNEL, EXTENDED_KERNEL, ELKS, MOTEL

Compiler: Michael Schweitzer CW Version: Ian Joyner
WEB: http://www.object-tools.com
Contacts: Frieder Monninger fm@object-tools.com
Ian Joyner i.joyner@acm.org

Figure 7. EiffelS Preference Panel

- **<target> Linker**

The linker panel for your target has **Main** set to execute *main*. Ensure **Initialization** and **Termination** are blank. The linker preference panel is shown in figure 8.



PPC Linker

Link Options

- [ ] Generate SYM File
- [✓] Full Path in SYM Files
- [ ] Generate Link Map
- [ ] Suppress Warning Messages
- [✓] Dead-strip Static Initialization Code
- [ ] Treat Multiple Definitions as Warnings

Link Mode: Faster Linking (uses more memory)

Entry Points

Initialization:	
Main:	main
Termination:	

Figure 8. PPC Linker Preference Panel

- **Custom Keywords**

  These are setup in the project stationery—if you don't like the defaults, you can change them. You can even change them in the default project in the *(Project Stationery)* folder.

3.3. Setting up EiffelS Compiler Information Files

- In this step you will set up two files particular to your project, the *Library dictionary.es2* and the *Library description.es2* files. There is one dictionary file for each project and one description file for each project and each library; libraries do not have a dictionary file.
- The *Library dictionary.es2* and the *Library description.es2* files go in your project folder. The compiler will automatically create *.pub* and *.bin* files to go in the *ES2 data* folder.

  You may throw out the *.pub* and *.bin* files—the compiler will regenerate them.
- The *Library dictionary.es2* file.

  This file contains the path information about where the libraries are. These will normally be in the EiffelS2 folder, and are path relative to your *MetroWerks CodeWarrior* folder (or whatever folder is the default folder for CodeWarrior.) The *Library dictionary.es2* file looks like:

```
---------------------------------------------
-- Standard library dictionary
HELLO        "HELLO"
ELKS         "EIFFELS2:library:elks"
RUNTIME      "EIFFELS2:library:runtime"
EXTENDED_KERNEL   "EIFFELS2:library:exkernel"
CONTAINER    "EIFFELS2:library:container"
MOTEL        "EIFFELS2:library:MOTEL"
---------------------------------------------
```

Note the library names match the names you put in the **EiffelS 2.0 Language** preference panel. The first entry is your own project, so this will match the project name in the panel, and the folder name for the project.

You can also use different versions of the libraries by using different paths in the *Library dictionary.es2* file. (See section 8 for a full description of the *Library dictionary.es2* file.)

- The *Library description.es2* file.

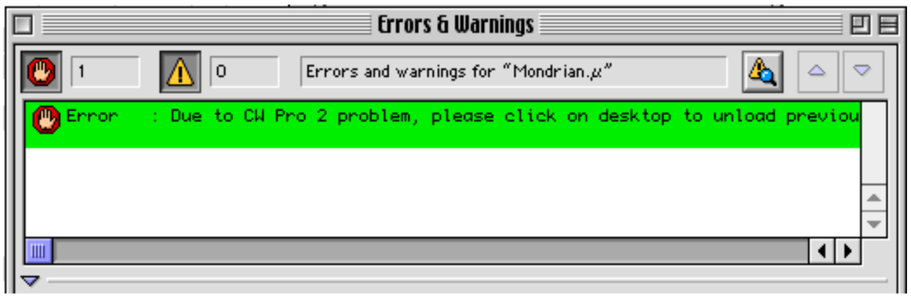
  This file contains project information about the libraries that this project uses. For your project, the special library name *PROJECT* is used. The other library names given in the use clause must match the libraries given in your projects *Library dictionary.es2* file.

```
---------------------------------------------
-- Library description for the
-- project library
open source library PROJECT
use      ELKS, MOTEL
clusters     ""
end -- library PROJECT
---------------------------------------------
```

Each library also has its own *Library description.es2* file. They will be supplied with the library, so you need worry no more about them. (See section 8 for a full description of the *Library description.es2* file.)

4. Compiling Your Eiffel Project

(IMPORTANT NOTE: There is a problem with CW Pro 2/3 that causes the compiler to not be reinitialized between runs. This means you will get an error if you run the compiler a second time. All you need to do is force CodeWarrior to unload the compiler by switching out of CodeWarrior to another application: clicking on the desktop for instance. The workaround is simple, but in our opinion annoying—sorry about that. MetroWerks have accepted this as a bug, and it will probably be fixed in CW Pro 4.)



```
Error   : Due to CW Pro 2 problem, please click on desktop to unload previou
```

Figure 9. Compiler unloading error

(Another important note: *.e* files must not be open in the IDE when EiffelS is compiling. This is also an annoying environmental factor that we will attempt to fix in a future release. Make sure *.e* files are closed when compiling, and that they are also not open in the CW Errors & Warnings window.)

4.1. Compiling Eiffel Files

- Select the *.ess* file in the project window;
- Chose compile from the project menu (Cmd-K). Compiling the *.ess* file will cause the compiler to be invoked, and it will automatically compile *.e* files that need recompiling, based on updated files and the dependencies between classes;
- The EiffelS compiler now reads the preferences from the EiffelS preference panel;

- If the compiler cannot create the *Project description.es2* file in the *ES2 data* folder, you will get the error:

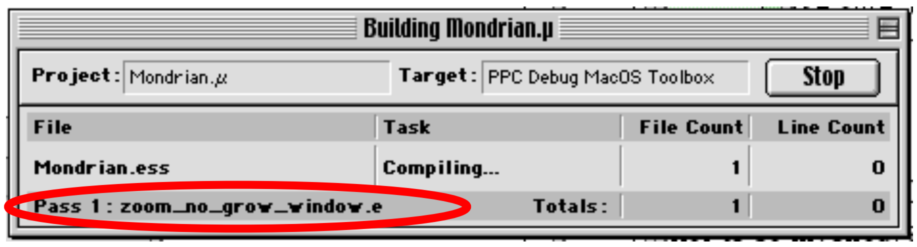
  ```
  Error: Can't create Project description.es2
  ```

  in the CW IDE error window.

The compiler then compiles your *Library dictionary.es2*, *Library description.es2* files and *Library description.es2* files from the libraries you have listed in the **use** clause. If any problems are found in these files, they are reported as syntax errors in the IDE errors window.

The compiler reports progress in the IDE build window. The compiler works in four passes. The first pass checks the source files in your project and libraries to see if they or any files they have dependencies on have been updated since the information stored in the *dbase* files. If so the second phase of the first pass compiles these files.

The name of each file as it is being checked and compiled appears in the IDE build window:



Building Mondrian.µ

Project: Mondrian.µ Target: PPC Debug MacOS Toolbox Stop

File	Task	File Count	Line Count
Mondrian.ess	Compiling...	1	0
Pass 1 : zoom_no_grow_window.e	Totals:	1	0

Figure 10. First pass

As each file is compiled the line count is updated in the IDE build window. Unfortunately, the total line count remains at 0 because the IDE updates this only after a compile is complete. The line count is updated only at the end of each file, so don’t expect to see the line count going up rapidly on large files.

Since Pass 1 checks and then compiles files, you will see twice the name of the files that are recompiled (if you can read that fast)

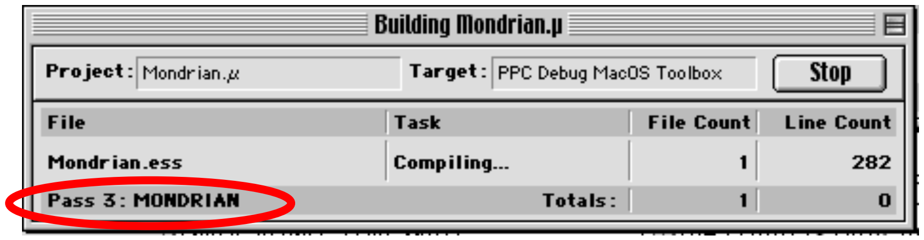
Syntax errors in pass 1 are linked to the source file, so double clicking on the error will open the source file. Unfortunately, in subsequent passes, the original file is not used, so only the library name, class name, line, and column are reported, but this is enough information to pin point each error.

Important: Since EiffelS opens the source files directly, and not through the IDE, unfortunately all *.e* files must be closed when compiling with EiffelS. If a file is open in the error window, the error window must also be closed in order to save the file (this seems to be a bug in CW Pro 2). If a *.e* file is open when it is compiled, an obscure error is reported on the first line of the file.

Pass 2 checks some consistency information in each class. You will see the class names as they are compiled:

```
Pass 2: MY_APPLICATION
```

Pass 3 does more consistency checking but also generates the C code. When regenerating C files, this is the longest pass. Progress information is the same as in Pass 2:



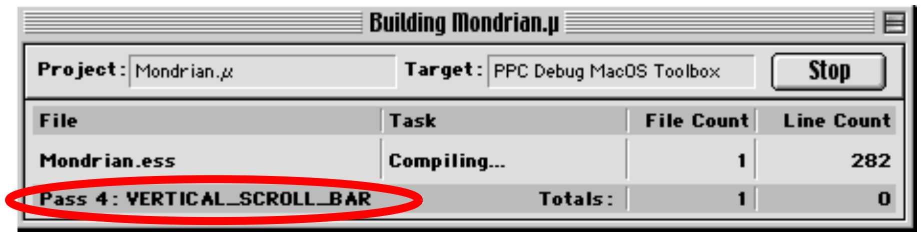
Building Mondrian.µ

Project: Mondrian.µ Target: PPC Debug MacOS Toolbox Stop

File	Task	File Count	Line Count
Mondrian.ess	Compiling...	1	282
Pass 3: MONDRIAN	Totals:	1	0

Figure 11. Third pass

Pass 4 is a clean-up pass, and results in no errors. Again it reports progress on each class:



Building Mondrian.µ

Project: Mondrian.µ Target: PPC Debug MacOS Toolbox Stop

File	Task	File Count	Line Count
Mondrian.ess	Compiling...	1	282
Pass 4: VERTICAL_SCROLL_BAR	Totals:	1	0

Figure 12. Fourth pass

Pass 4 is very fast. You should check that the compiler reaches pass 4 since, very occasionally, a database error causes the compiler to silently give up in pass 2 or 3. In this case you should remove the *ES2 data* folder as described in the section 10. If the compiler should crash, an error file, *Eiffel2.err* will usually be placed in the *MetroWerks CodeWarrior* folder.

The first EiffelS compile will take a long time because it must process all the sources and generate all the C files. However, subsequent compiles do not do this so each compile is done in an acceptable amount of time, proportional to the number of changes you have made.

4.2. Compiling C and Linking

Unfortunately, CW IDE does not provide facilities to add files to a project programmatically, or to schedule the .c files for automatic compilation. In your Eiffel project there is a *Generated* group that will originally be empty.

Now add all the files that the EiffelS compiler generated in the *ES2 Data:Code* folder in your project folder. These files are named *c0.c*, *c1.c* ... *cn.c*, and contain all the C files for all classes compiled from your own project, as well as the libraries you use. Adding all the files from this directory also adds several *.h* files that are used for your Eiffel compilation options. These files mean that the compiler does not have to regenerate any *.c* files if you change the libraries from debug to optimize to final, etc.

The EiffelS compiler has also generated the *main.c* and *emain.c* files. If you are building a very simple program that does only console I/O, also keep the *main.c* file in the project. If you are building a toolbox application using the MacOS library, then remove *main.c* from the project, since the file *MacOS_main.c* from the *EiffelS2:Library:MOTEL* folder contains the main function. If neither of these do what you want, you can provide your own *main*, and add it to the sources group. Do not store your own main in the *Code* folder because the compiler will overwrite it.

Now all you need to do is run the C compiler and linker.

Subsequent Eiffel compiles flag to the IDE that C files have changed, so you should observe the changed red check come up against a C file because the EiffelS compiler regenerates it. However, the IDE is not 100% accurate at picking up these files, so it is best to just compile with EiffelS first, and then build later (at least with CW Pro 2).

Also if you add classes (and therefore *.e* files) to your Eiffel system, new *.c* files will appear. You will also need to add these to your *Generated* group.

Note that the EiffelS compiler always overwrites the correct *.c* file if the source has changed. This information is kept in the dbase files. Thus if for any reason you lose or throw out the code files, make sure you also discard the admin and dbase folders which are all in the *ES2 data folder*.

5. Creating your own CodeWarrior Eiffel libraries

Read the section *Creating a Library* in section 8. To accomplish the steps described there, you should set up your own IDE project for the library. No special settings need be given, this project will never be compiled, and is only a convenient repository for your Eiffel files. Create a real project as described above, and include your library name in your project's *Library dictionary.es2* file, and **EiffelS 2.0 Language preference** panel. You can then manipulate the library files using the library project.

The library does not have a dictionary file, but it does have a description file that tells the compiler which other libraries the library uses. You should also add the library into the uses clause in your project's description file.

If you strike any problems remove the *ES2 data* folder from the library folder.

6. ASSERTION MONITORING

You can enable/disable assertion checks on a per library basis. To accomplish this you create a file *rcl* in the directory where the main project files are. The compiler processes this file and creates an *rcb* file that the runtime uses to determine the assertion settings. The syntax for the *rcl* file is:

require	*library_name.class_name*
ensure	*library_name.class_name*
check	*library_name.class_name*
loop_invariant	*library_name.class_name*
variant	*library_name.class_name*
invariant	*library_name.class_name*
debug [(*key_list*)]	*library_name.class_name*

Comments are indicated by the standard Eiffel --.

All clauses are optional. *library_name* is either the name of a library or the reserved keyword **all** (meaning all libraries used in the program). *class_name* can also be either the name of a class or the keyword **all**. The order of the clauses is irrelevant and each can be repeated as often as desired.

debug has an optional list of keys that are strings (see *Eiffel: The Language* for a detailed description of **debug** keys). For example:

```
debug ("TEST", "ANOTHER_TEST")  KERNEL
```

enables the debug instructions labeled with the keys TEST or ANOTHER_TEST in all classes of the library KERNEL.

Note:

By default all assertion checks are off. Assertion monitoring can also be enabled and disabled in the EiffelS source level debugger.

An example *rcl* file is:

```
require PROJECT.all
ensure PROJECT.all
invariant PROJECT.all
--debug MOTEL.all
--debug ("events") MOTEL.all
--debug ("event_disk") MOTEL.all
--debug ("event_idle") MOTEL.all
--debug ("event_disk") MOTEL.all
--debug ("event_setup_menu") MOTEL.all
--debug ("event_menu_command") MOTEL.all
--debug ("event_key_down") MOTEL.all
--debug ("event_high_level") MOTEL.all
--debug ("event_key_up") MOTEL.all
--debug ("event_mouse_down") MOTEL.all
--debug ("event_mouse_up") MOTEL.all
--debug ("event_null") MOTEL.all
--debug ("event_OS") MOTEL.all
--debug ("event_activate") MOTEL.all
--debug ("event_update") MOTEL.all
--debug ("event_window") MOTEL.all
```

Notice that *MOTEL* gives many trace options that can be enabled by removing the comment characters --. Because of this level of tracing, *MOTEL* helps you find out exactly what is going on in your application, as well as helping you understand how *MOTEL* works.

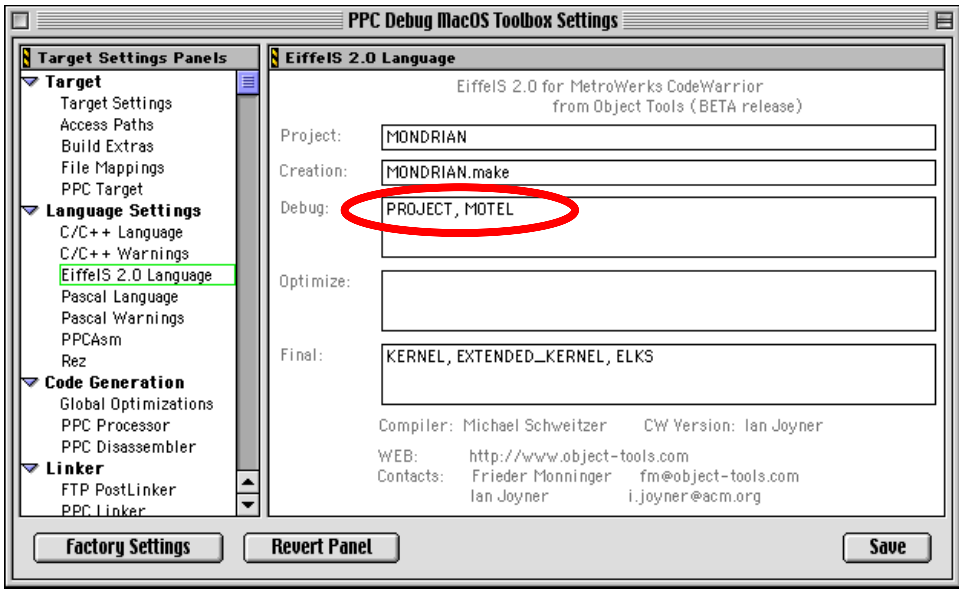
7. Eiffel Source-Level Debugging

You can use the CodeWarrior debugger, MWDebug. However, even if you understand the EiffelS runtime, this proves very tedious. For this reason, a source-level debugger is provided with EiffelS. Note that while the CW debugger has a very nice interface, the EiffelS debugger is very text oriented, and does not provide a pretty interface. Hopefully, this is made up for by the power of the debugger, which has some advanced features not found in any other environment.

In summary, the MW debugger will really help only to debug the EiffelS runtime environment, which you should never need to do, whereas the EiffelS debugger helps debug your applications at the Eiffel level. There is a third even higher level of debugging provided in the *MOTEL* library, which is enabled using the debug keys in the *MOTEL* library that are enabled in the *rcl* file as previously explained. This level can tell you exactly what has happened in the *MOTEL* library, which events have happened, and any other important items at the *MOTEL* level.

7.1. Enabling the debugger

As shown in section 13 to enable the debugger, make sure that one or more libraries are listed in the debug text box on the EiffelS preference panel. Recompile with Eiffel with the .ess file as the source.



EiffelS 2.0 Language

EiffelS 2.0 for MetroWerks CodeWarrior
from Object Tools (BETA release)

Project: MONDRIAN

Creation: MONDRIAN.make

Debug: PROJECT, MOTEL

Optimize:

Final: KERNEL, EXTENDED_KERNEL, ELKS

Compiler: Michael Schweitzer CW Version: Ian Joyner

WEB: http://www.object-tools.com
Contacts: Frieder Monninger fm@object-tools.com
Ian Joyner i.joyner@acm.org

Figure 13. Enabling the debugger

Enabling debug changes the debug option in the *<project>.h* file. When debug is set, the flag in the *.h* file should be:

```
        #define EDEBUG_2 1
```

(You do not have to set this, but if you have problems, check that the *.h* file has been regenerated with this define.)

Then recompile with C.

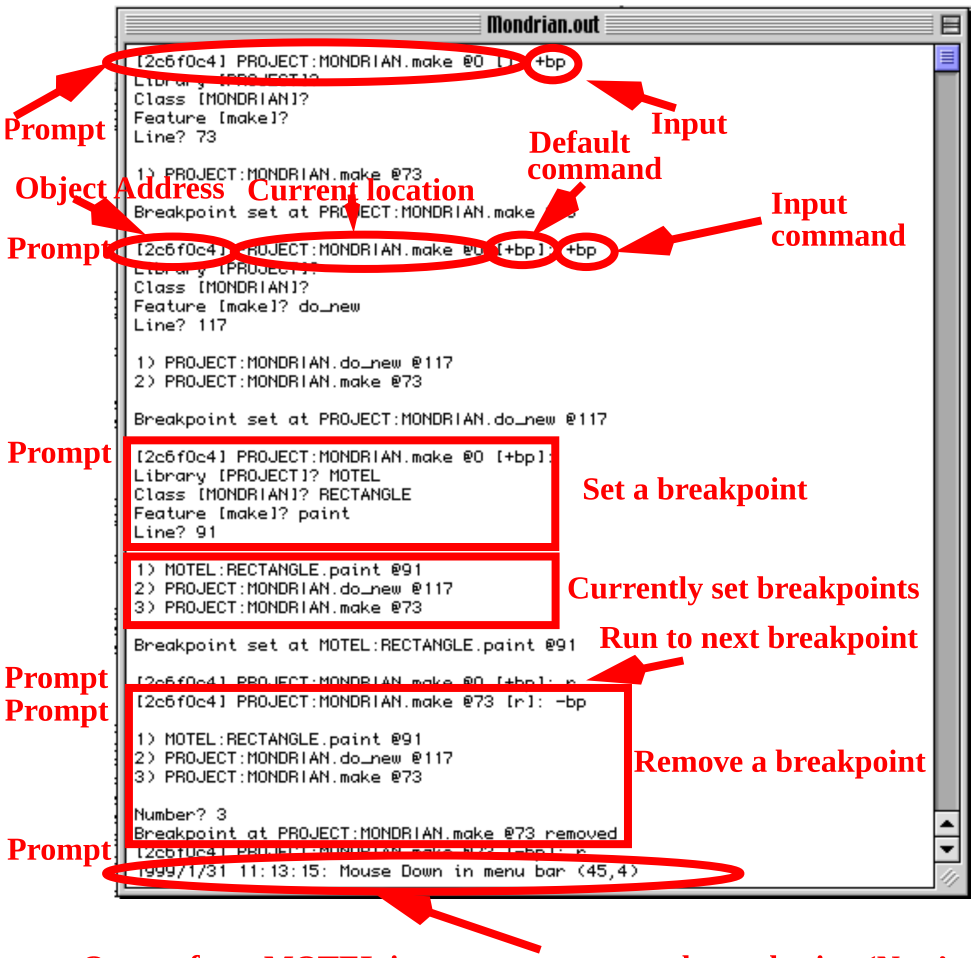
You should ensure that CodeWarrior recompiles all the *.c* files for the library you are debugging so that the option is picked up.

When you run your application, the debugger is automatically started in a CodeWarrior SIOUX window, when the first routine in the library being debugged is called. You will be prompted to run, step, enable breakpoints, etc.

(NOTE: SIOUX has a current limitation of 32K in the window, after which it dies. There is a new library announced that will enable standard output windows of greater than 32K. For this reason, more voluminous outputs are sent to a trace file that is placed in the same directory as the application.)

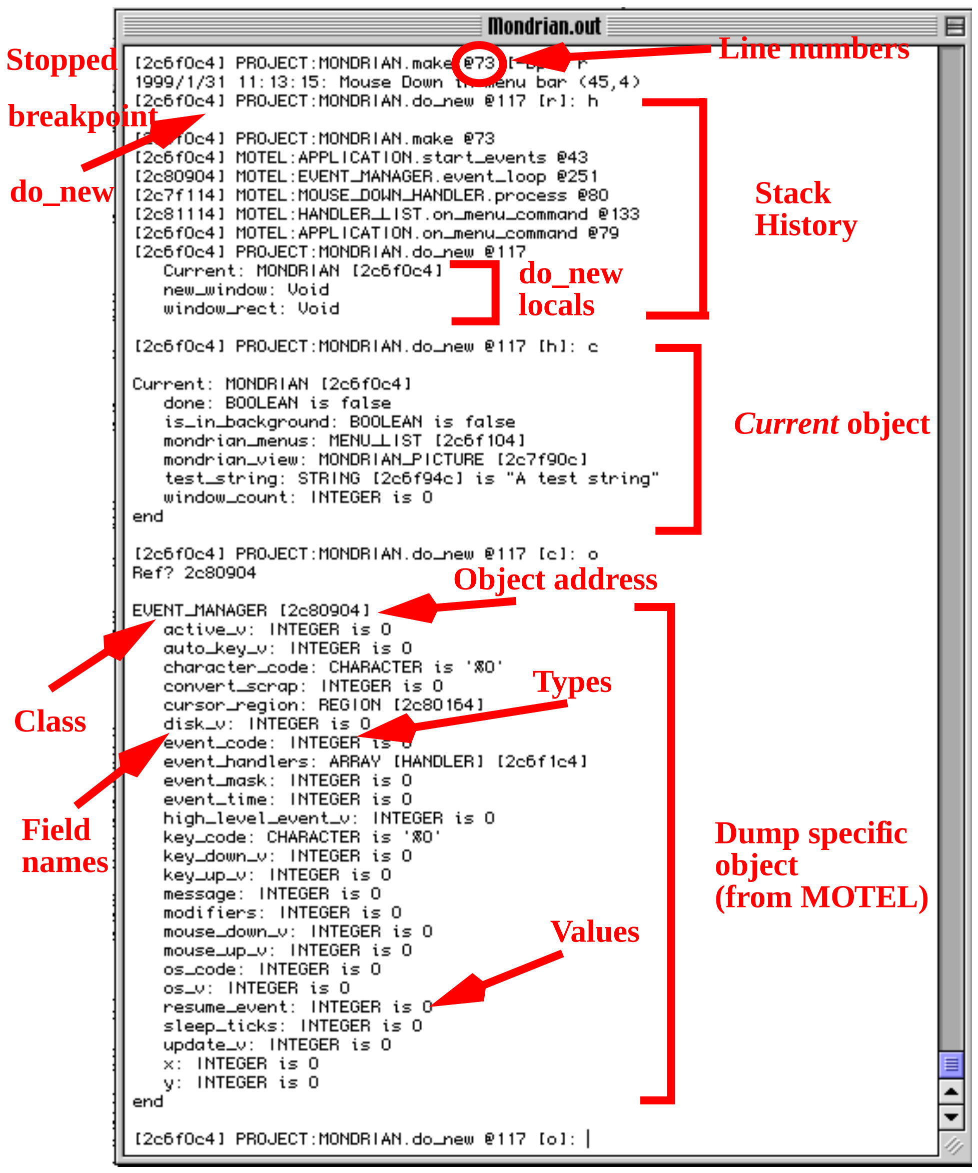
7.2. Example Debugger Session

The following figures show a debugger session, which illustrates the points talked about in the following discussion.



```
[2c6f0c4] PROJECT:MONDRIAN.make @0 []: +bp
Library [PROJECT]?
Class [MONDRIAN]?
Feature [make]?
Line? 73

 1) PROJECT:MONDRIAN.make @73

Breakpoint set at PROJECT:MONDRIAN.make @73
[2c6f0c4] PROJECT:MONDRIAN.make @0 [+bp]: +bp
Library [PROJECT]?
Class [MONDRIAN]?
Feature [make]? do_new
Line? 117

 1) PROJECT:MONDRIAN.do_new @117
 2) PROJECT:MONDRIAN.make @73

Breakpoint set at PROJECT:MONDRIAN.do_new @117
[2c6f0c4] PROJECT:MONDRIAN.make @0 [+bp]:
Library [PROJECT]? MOTEL
Class [MONDRIAN]? RECTANGLE
Feature [make]? paint
Line? 91

 1) MOTEL:RECTANGLE.paint @91
 2) PROJECT:MONDRIAN.do_new @117
 3) PROJECT:MONDRIAN.make @73

Breakpoint set at MOTEL:RECTANGLE.paint @91

[2c6f0c4] PROJECT:MONDRIAN.make @0 [+bp]: r
[2c6f0c4] PROJECT:MONDRIAN.make @73 [r]: -bp

 1) MOTEL:RECTANGLE.paint @91
 2) PROJECT:MONDRIAN.do_new @117
 3) PROJECT:MONDRIAN.make @73

Number? 3
Breakpoint at PROJECT:MONDRIAN.make @73 removed
[2c6f0c4] PROJECT:MONDRIAN.make @73 [-bp]: r
1999/1/31 11:13:15: Mouse Down in menu bar (45,4)
```

Figure 14. Example debug session



```
[2c6f0c4] PROJECT:MONDRIAN.make @73 [illegible] r
1999/1/31 11:13:15: Mouse Down in menu bar (45,4)
[2c6f0c4] PROJECT:MONDRIAN.do_new @117 [r]: h

[2c6f0c4] PROJECT:MONDRIAN.make @73
[2c6f0c4] MOTEL:APPLICATION.start_events @43
[2c80904] MOTEL:EVENT_MANAGER.event_loop @251
[2c7f114] MOTEL:MOUSE_DOWN_HANDLER.process @80
[2c81114] MOTEL:HANDLER_LIST.on_menu_command @133
[2c6f0c4] MOTEL:APPLICATION.on_menu_command @79
[2c6f0c4] PROJECT:MONDRIAN.do_new @117
   Current: MONDRIAN [2c6f0c4]
   new_window: Void
   window_rect: Void

[2c6f0c4] PROJECT:MONDRIAN.do_new @117 [h]: c

Current: MONDRIAN [2c6f0c4]
   done: BOOLEAN is false
   is_in_background: BOOLEAN is false
   mondrian_menus: MENU_LIST [2c6f104]
   mondrian_view: MONDRIAN_PICTURE [2c7f90c]
   test_string: STRING [2c6f94c] is "A test string"
   window_count: INTEGER is 0
end

[2c6f0c4] PROJECT:MONDRIAN.do_new @117 [c]: o
Ref? 2c80904

EVENT_MANAGER [2c80904]
   active_v: INTEGER is 0
   auto_key_v: INTEGER is 0
   character_code: CHARACTER is '%0'
   convert_scrap: INTEGER is 0
   cursor_region: REGION [2c80164]
   disk_v: INTEGER is 0
   event_code: INTEGER is 0
   event_handlers: ARRAY [HANDLER] [2c6f1c4]
   event_mask: INTEGER is 0
   event_time: INTEGER is 0
   high_level_event_v: INTEGER is 0
   key_code: CHARACTER is '%0'
   key_down_v: INTEGER is 0
   key_up_v: INTEGER is 0
   message: INTEGER is 0
   modifiers: INTEGER is 0
   mouse_down_v: INTEGER is 0
   mouse_up_v: INTEGER is 0
   os_code: INTEGER is 0
   os_v: INTEGER is 0
   resume_event: INTEGER is 0
   sleep_ticks: INTEGER is 0
   update_v: INTEGER is 0
   x: INTEGER is 0
   y: INTEGER is 0
end

[2c6f0c4] PROJECT:MONDRIAN.do_new @117 [o]:
```

Figure 15. Debugger output

7.3. Some conventions

Object addresses are output in square brackets, for example: [2acee74]. You can copy and paste this address into other commands to find out further information about the object at this location. Command and other defaults are also presented in square brackets. If the default is acceptable, just press return—that is the input. For example

```
[2acee74] PROJECT:MONDRIAN.make @56 [s]:
```

is presented as the prompt at the current execution location. Here the first address [2acee74] is the address of the *Current* object. [s] at the end of the prompt indicates that single step is the default command; if you press enter, another single step will be taken. The debugger also outputs other defaults in square brackets to save you much typing.

Line numbers are indicated by the @ character.

The rest of the command prompt indicates the project, class, feature and line number in the file where the debugger has halted program execution. The line number corresponds to the file line number in the CodeWarrior IDE editor. The CW IDE indicates this line in the panel at the bottom left of editor windows. You can position an IDE window at this line by clicking this line number panel and entering the number directly. Thus it is easy to relate what the debugger tells you to your Eiffel source.

7.4. Debugger Commands

The debugger prints out the following list when you input an unknown command or ?.

```
r - run
x - exit
u - run until
s - single step (into sub-routines)
t - single step (over sub-routines)
t/ - run to routine end
+bp - set break point
-bp - remove break point
bp - show break points
+ss - set snap shot
-ss - remove snap shot
ss - show snap shots
h - history
w - where
c - Current object
c/ - deep Current object
o - object
o/ - deep object
a - array
b - bit
" - string_cmd -- "
am - assertion monitoring
```

+d - enable debug
-d - disable debug
+tl - trace lines
+tr - trace routine entry/exit
-tl - trace lines off
-tr - trace routines off
y - status
? - help

r - run

Runs the application up to the next encountered breakpoint, or until the application terminates.

x - exit

Terminates the application.

u - run until

This prompts you for a location in the same format as a breakpoint. The application runs until that location is reached. The breakpoint is removed once it is encountered. (u sets a temporary breakpoint.)

s - single step (into sub-routines)

This single steps the application. Routines are entered as they are encountered.

t - single step (over sub-routines)

This is a single step command as for s, except that encountered routines are not entered.

t/ - run to routine end

This runs the routine until the end, where the debugger stops for further instructions. You can at this point use the history command to find the effect on local variables (arguments cannot be changed in Eiffel), or the c command to examine the state of the *Current* object.

+bp - set break point

This command prompts you for the library, class, feature and line of the breakpoint to set. The default library, class, and feature are the current library, class, and feature.

-bp - remove break point

This command lists the current breakpoints—you can remove a breakpoint by the number.

bp - show break points

This shows the currently set breakpoints.

For example, removing and then showing the remaining breakpoints:

```
[3034e84] PROJECT:MONDRIAN.make @0 [+bp]: -bp
1) MOTEL:MOUSE_DOWN_HANDLER.process @86
2) MOTEL:RECTANGLE.paint @91
3) PROJECT:MONDRIAN.make @73
Number? 2

Breakpoint at MOTEL:RECTANGLE.paint @91 removed
[3034e84] PROJECT:MONDRIAN.make @0 [-bp]: bp
1) MOTEL:MOUSE_DOWN_HANDLER.process @86
2) PROJECT:MONDRIAN.make @73
```

+ss - set snap shot

Snap shot is a little like a breakpoint, except that when the location is reached, a snap shot of the object executing at that location is taken. The debugger does not stop as with a breakpoint. The snapshot taken is a *deep* snapshot; that is, all objects that are connected transitively to the current object are also output. Since you can dump an entire system this way, this snapshot is output to the trace file.

-ss - remove snap shot

This command lists the current snapshots—you can remove a snapshot by the number.

ss - show snap shots

This shows the currently set snapshots.

h - history

This outputs the stack call chain history and outputs the arguments and locals for the routine currently executing.

w - where

This prints out the current location in the application. As this information is output in the command prompt, it is mainly redundant.

c - Current object

This outputs the Current object:

```
[3034e84] PROJECT:MONDRIAN.make @73 [r]: c

Current: MONDRIAN [3034e84]
    done: BOOLEAN is false
    mondrian_menus: MENU_LIST [3034ec4]
    mondrian_view: MONDRIAN_PICTURE [30456cc]
    test_string: STRING [303570c] is "A test string"
    window_count: INTEGER is 0
end
```

```
[3034e84] PROJECT:MONDRIAN.make @73 [c]: s
```

c/ - deep Current object

This outputs the *Current* object and all subordinate objects.

o - object

o allows you to monitor any arbitrary object given an object reference address.

o/ - deep object

This is the same as o, except all connected objects starting from the given address are output.

a - array

a outputs an array.

b - bit

b outputs a type of *BIT N*

" - string_cmd

" outputs a *STRING* object at a given address. This should be redundant since all string values are included in other outputs.

am - assertion monitoring

am enables you to set and reset assertion monitoring at run time overriding the settings in the *rcl* file.

```
[3122e44] PROJECT:MONDRIAN.make @0 []: am
Library [PROJECT]?
Class [MONDRIAN]?
Preconditions? +
Postconditions? +
Checks?
Loop invariants?
Loop variants? -
Class invariants? +
Debugs?
[3122e44] PROJECT:MONDRIAN.make @0 [am]:
```

This example sets precondition, postcondition and class invariant monitoring on the default library and class, *PROJECT:MONDRIAN*. It resets loop variant monitoring and leaves checks and loop variants on their current settings.

+d - enable debug

+d allows you to enable **debug** instructions at run time. For example the *MOTEL* class *HANDLER_LIST* has the routine *event_setup_menus*:

```
setup_menus is
        local
            it: ITERATOR
        do
            -- First setup the menus.

            debug ("events", "event_setup_menus")
                stamp_time (true)
                io.put_string (": Setup Menus%N")
            end

            from
                it := l.iterator
            until
                it.finished
            loop
                l.item (it).setup_common_menus
                it.forth
            end
        end
```

Notice the **debug** instruction: this contains instructions that will be executed when any of the **debug** keys are set. This debug instruction has two keys: *events* and *event_setup_menus*. The *events* key enables many such **debug** instructions, whereas the *event_setup_menus* is more specific to this **debug** instruction.

In the debugger, we set the *event_setup_menus* key as follows:

```
[2c6f0c4] PROJECT:MONDRIAN.make @0 []: +d
Library [PROJECT]? MOTEL
Class [MONDRIAN]? HANDLER_LIST
Key? event_setup_menus
```

-d - disable debug

-d allows you to disable **debug** instructions at run time.

+tl - trace lines

+tl writes a trace for every line executed to the *trace* file. It can produce very voluminous output but will save you much single stepping.

+tr - trace routine entry/exit

+tr writes a trace for every routine entry and exit to the *trace* file. It is less voluminous than +tl. +tl and +tr used with snapshot provide a very powerful tracing facility.

-tl - trace lines off

This ceases output of the line trace to the *trace* file.

-tr - trace routines off

This ceases output of the routine trace to the *trace* file.

y - status

Will give some status information (no information available as yet).

? - help

Output the command list.

8. Description and Dictionary Files

The following sections provide a full description of the *Library dictionary.es2* and *Library description.es2* files should you require the more advanced forms available in these. You will most likely not need this information until you create very large Eiffel projects.

8.1. Conventions

Identifiers

Identifiers must begin with a letter and must be composed of letters, digits and the underscore character only.

Literal strings

Literal strings follow the Eiffel conventions. They must be enclosed in double quotes and Eiffel escape sequences (e.g. %N for the newline character) can be used. Except in Eiffel source files, literal strings may not extend over several lines.

Comments

Comments follow the Eiffel convention. They begin with a double dash (--) and extend to the end of the line.

Eiffel source files

An Eiffel source file must have the file extension *.e*. The name of the file is otherwise arbitrary. The file name does not have to match the class name, but it is best if it does. The compiler will automatically find out which class is stored in the file.

An Eiffel source file must contain exactly one class text.

Paths

Paths must be specified using the MacOS convention (i.e., : as component separator). Paths are always given as literal strings.

Source files vs. binary files

The library dictionary and library description source files (explained in the next sections) may be removed as soon as the compiler has translated them (a *.bin* file of the same name has been produced). For example, if you don't want someone modifying the library description of library *L*, you can remove the *Library description.es2* file from the folder of the library after it has been compiled.

8.2. Library dictionaries

Every Eiffel/S library has a name (an identifier) and its own directory (with a number of subdirectories). The purpose of a library dictionary is to map library names to paths. There is a single library dictionary that you will find in *EiffelS2:LIB_DICT:Library dictionary.es2* Here is a complete example:

```
---------------------------------------------
-- Standard library dictionary
HELLO        "HELLO"
ELKS         "EIFFELS2:library:elks"
RUNTIME      "EIFFELS2:library:runtime"
EXTENDED_KERNEL   "EIFFELS2:library:exkernel"
CONTAINER    "EIFFELS2:library:container"
MOTEL        "EIFFELS2:library:MOTEL"
---------------------------------------------
```

This file tells the compiler that, e.g., library *CONTAINER* can be found in the directory *EIFFELS2:library:container*. Where ever you need the services offered by this library you can request them by simply writing

use CONTAINER

you don't have to remember where this library can be found. Each project has its own dictionary file that enables you to use different versions of libraries for different projects.

The name of a library dictionary file must be *Library dictionary.es2*.

8.3. Library descriptions

Eiffel/S 2.0 offers a structuring mechanism called libraries. A library consists of a set of related classes and is treated by the compiler as an entity in its own right, just as classes. Typical examples of libraries are a container library or *MOTEL*.

The main advantage of libraries is that the user does not need to remember how the library is built, which source files belong to it, where they can be found. All this information is stored in a library description written by the creator(s) of the library. If you want to use library *L* just say **use** *L*.

Every library has its own directory that contains a number of subdirectories. Two of them are important for the user: the subdirectory *ES2 data* and the subdirectory *eiffel source*. The first contains the library description file *Library description.es2* and the second contains the source code (possibly in further sub-directories). The compiler maintains a library database that is stored in the subdirectory *dbase*.

Library descriptions are written in the simple *library description language* (LDL). Instead of giving the formal syntax we explain the clauses step by step. The overall structure is:

```
header clause
[use clause]
[clusters clause]
[hide clause]
end
```

Header Clause

First of all, a library has to have a name. Thus a library description starts with

```
[open] [source] library name
```

where *name* is an identifier - the name of the library. The optional clauses **open** and **source** have the following meaning:

If a library is declared as being open then the EiffelS compiler is allowed to modify the library database. Typically only libraries that are not finished yet are declared as being open. But there are situations in which even a finished library has to be declared as open: suppose *L1* is finished, *L2* is not and some class *C1* in *L1* inherits from some class *C2* of *L2*. If *C2* is modified in a way that would require to modify some database entries for *C1* in *L1* then this would fail if *L1* was not open.

The optional **source** clause simply indicates whether source code for the library is available or not. For a library that is not open, the clause is meaningless to the compiler. However, if the library is **open** and **source** is specified, the compiler will automatically check the source code against the database. If some class *C* has entries in the database but its source file does no longer exist, all entries for *C* will be deleted in the database.

The use of **open** and **source** is explained in the following table:

open	**source**	**use if**
yes	yes	source code is available and may need to be modified
yes	no	no source code available but database may need to be updated

no	no	no modifications necessary or desired
no	yes	don’t use it (same meaning as no/no).

Use Clause

The next clause is the **use** clause. Although it’s optional, it is needed in most cases. It lists all libraries that are used by the current library. To be more precise: a library *L1* uses a library *L2* iff *L1* is different from *L2* and some class *C1* of *L1* is a client or a descendant of some class *C2* of *L2*. If *L1* uses *L2* then the library description of *L1* must include a clause

use *L2*

If it uses several libraries *L1*, ..., *Ln* then the clause becomes

use *L1, L2, ..., Ln*

It is not allowed to mention the same library twice in a **use** clause. The **use** clause can be combined with renaming as follows:

```
use
    L1
        rename
            A as B,
            C as D
        end,
    L2,
    L3
        rename
            U as V
        end
```

A **use** clause with renaming of the form **use** *L* **rename** *A* **as** *B* **end** has the following meaning: Library *L* contains a class named *A*. The current library (i.e. the one in which the **use** clause appears) wants to use this class but under a different name: *B*. Thus if some class in the current library refers to *B* it actually refers to *A*. This mechanism is necessary if two different libraries offer (export) a class under the same name and are both used by a third library. The rules that govern renaming here are the same as those for renaming Eiffel features. However, some classes cannot be renamed because the compiler makes strong assumptions about them. These are:

GENERAL, ANY, COMPARABLE, NUMERIC, HASHABLE, ARRAY, STRING, BOOLEAN, INTEGER, REAL, DOUBLE, CHARACTER, POINTER, BOOLEAN_REF, INTEGER_REF, REAL_REF, DOUBLE_REF, CHARACTER_REF, POINTER_REF, BIT, NONE.

Clusters Clause

As we mentioned in the introduction, the source code for a library is stored in the subdirectory *eiffel source* of the library directory and in subdirectories thereof. The **clusters** clause tells the compiler in which subdirectories of *eiffel source* it shall look for Eiffel source files. The clause

clusters "", "c1", "c2:c3"

instructs the compiler to look for Eiffel source code in the subdirectories

'eiffel source', 'eiffel source:c1' and 'eiffel source:c2:c3'

[Recall that the compiler assumes that Eiffel source files have the file extension *.e*].

If the clusters clause is omitted, the compiler expects all Eiffel sources of the library in *eiffel source*.

Hide Clause

Sometimes it is necessary to introduce some utility classes in a library that are for implementation purposes only, and should be visible only to the library itself. One may or may not wish to make these classes visible to other libraries. If not, one can simply hide them by writing

hide C1, C2, etc.

Classes declared as hidden are useable only within their enclosing library. This clause is optional. Note that those classes that cannot be renamed (see above) can also not be hidden.

Flat libraries

If you find it inconvenient that the library description file and the Eiffel sources must be placed in subdirectories of the library directory, you can *flatten* the library as follows:

Create a file with name *flat* in the library directory. Place *Library description.es2* in the library directory. The clusters clause will now be interpreted as being relative to the library directory itself instead of relative to the subdirectory *eiffel source*.

Note that the compiler is not interested in the content of the file *flat*—it merely looks whether such a file exists or not.

Creating a library

In order to create a library you have to do the following:

a choose and create a directory for the library.

b create a library description file *Library description.es2*. Use the keywords **open** and **source** in the description.

c Copy the source code of the library to *eiffel source* folder or appropriate subfolders thereof.

d Enter the name and path of the library in a library dictionary (see: library dictionaries).

e Create a simple project that uses the library (it suffices to mention the library name in the use clause of the project description).

f Compile the project.

[adapting c) and d) to *flat* libraries is straightforward].

As soon as a library has passed pass 1 of the compiler, the source code is no longer necessary in order to use the library. You could then remove the keyword **source** from the library description and remove the source code (of course only if you don’t want to modify it in the future).

Removing a library

In order to make a library inaccessible, simply remove its name and path from any library dictionary that contains it. Then you can safely remove the whole library directory (of course you will normally save the source code and the library description file first).

9. KEYWORD DETAILS

We have divided the Eiffel keywords into four categories that you will find in the files *Primary keywords*, *Secondary keywords*, *Warning keywords* and *Basic types*.

Primary keywords

These are the main keywords in Eiffel. We suggest attaching to the **Custom keyword set 1** with the colour blue.

class, **create**, **creation**, **do**, **else**, **elseif**, **end**, **expanded**, **export**, **feature**, **from**, **frozen**, **if**, **indexing**, **infix**, **inherit**, **inspect**, **is**, **like**, **local**, **loop**, **precursor**, **prefix**, **redefine**, **rename**, **select**, **strip**, **then**, **undefine**, **until**, **variant**, **when**

Secondary keywords

These are the secondary keywords in Eiffel. We suggest attaching to the **Custom keyword set 2** with the colour green.

alias, **all**, **and**, **as**, **Current**, **false**, **implies**, **not**, **old**, **Result**, **or**, **true**, **unique**, **Void**, **xor**

Warning keywords

These are the other keywords of Eiffel. They could be included with the primary keywords, but a different colour will help flag a different to the usual situation. The design by contract keywords (assertions, invariants and exception handling, therefore with words that can be affected in the RCL file) keywords have also been included in this set. We suggest attaching to the **Custom keyword set 3** with the colour brown.

check, **debug**, **deferred**, **ensure**, **external**, **invariant**, **obsolete**, **once**, **require**, **rescue**, **retry**, **separate**

Basic types

These keywords identify basic types. We suggest attaching to the **Custom keyword set 4** with the colour purple.

BIT, *BOOLEAN*, *CHARACTER*, *DOUBLE*, *INTEGER*, *NONE*, *POINTER*, *REAL*, *STRING*, *TUPLE*

10. Trouble Shooting

Error: Can't create Project description.es2

Ensure your project folder is in the *MetroWerks:MetroWerks CodeWarrior* folder. Make sure the project name in the EiffelS 2.0 Language panel is the same as your project folder.

Error: Can't create Project description.es2. Ensure Project name in EiffelS2 Panel matches the folder name.

Ensure the *EiffelS2* folder is in the *MetroWerks:MetroWerks CodeWarrior* folder and that the name is the same as in the EiffelS panel project entry. The project name and the folder name must not contain blanks or special characters.

Error: Dictionary file Library dictionary.es2 doesn't exist or you don't have read permission. Project desription.es2 line 1 project <PROJECT>

Make sure that the *Library dictionary.es2* file is present in the project folder.

Error: The binary file "<DRIVE>:metrowerks:metrowerks codewarrior:ES2 data:Library desctription.bin" doesn't exist.

Make sure that the *Library description.es2* is present in the project folder or *Library description.bin* file is present in the *ES2 data* folder. You do not need a *Library description.es2* file if you have the *Library description.bin* file. In this case a library is closed.

Error: The library "PROJECT" cannot be used because it has not successfully passed ECC (pass 1).

Make sure all the *.e* files for your project are in the *Eiffel source* folder in your project folder.

Error : Lexical error: Unknown symbol " "
Library dictionary.es2 line 1 -- Standard library dictionary

Make sure that the *Library dictionary.es2* is not open, either in CodeWarrior or any other application.

The compiler mysteriously stops during Pass 2 or 3.

Find the file *eiffel2.err* and open it. This will give clues as to why the compiler failed (for example I/O error). (Yes a stack dump on the Mac, not system error or core!)

Try recompiling. If the compiler keeps failing:

Sometimes the generated compiler files get corrupted. Remove the *ES2 data* folder from your project folder. If you are developing a library as well, remove the *ES2 data* folder from the library folder. Recompile: this will cause the regeneration of all files.

Error: Keyword "class" expected reported on line 1 of a .e file

Ensure file is closed and also that it is not open in the IDE Error window.

Error: Compiler reports undefined item, but there is no item like it near the line in the reported class.

This is probably an error in an inherited invariant, precondition or postcondition. Search all ancestors of the class, and check the reported error line in those classes. You will probably find an item with the name at one of those lines. This will probably indicate a problem with inheritance, and you will have to rename/redefine the feature causing the clash.

Compiler reports strange error on first line of a .e file.

Ensure .e file is not open in CW IDE. Ensure the file is also not open in the IDE Error window.

Compiler quickly reports "Error: Due to CW 2 problem, please click on desktop..." as a syntax error.

This is a problem that CW does not unload the compiler after a compile. It is easy to force CW to do this: either click on the desktop to force CW to swap, or do a C compile or link. Object-Tools is working with MetroWerks to get a permanent fix to this problem.

After a seemingly successful compile (compiler has progressed through to Pass 4), compiler reports: "Error: Due to CW Pro 2 problem, please click on desktop..."

In the "File Mappings" preference panel, make sure you have set up a *.ess* file. There should be only one *.ess* file added to your project—its contents does not matter, it is simply used to get CodeWarrior to invoke the Eiffel compiler once. The .ess file must be mapped to the EiffelS compiler in "File Mappings".

Ensure that the compiler in "File Mappings" for *.e* files is None, not EiffelS, otherwise the Eiffel compiler will be invoked for every *.e* file, even though it has already compiled them all. This is because Eiffel compilers do their own dependency analysis, they do not need external help of make files, environments, etc., to work out what to compile. The downside to this is you must put all files where the compiler can find them and put paths to libraries in your *Library dictionary.es2* file. On the upside, all Eiffel projects will be organized in a similar way.

Compiler reports: Class not declared as deferred although it has a deferred feature, but does not report which file is the problem.

This is a known problem we have not been able to ascertain a fix to. All other known errors report the error file correctly in pass 1.

Check recently modified files: if any have deferred features, ensure the class is also declared as deferred. You might also notice the name of the file in error in the CW build panel just before compiler terminates.

Compiler reports errors on paths that do not exist on your machine.

If this is a project that you have copied from another machine, remove the *ES2 data* folder.

Compiler hangs must reboot.

Check that you have not run out of disk space. The compiler has reported that it cannot create or open compiler files to the IDE, but these are not displayed for some reason. Free up space on your disk.

Compiler does not detect VTCG(2)

If you declare a generic type, but do not put any actual generic list, EiffelS fails to report the VTCG(2) error. Obscure C syntax errors will result. Workaround: Supply correct number of actual generics.

C compiler error

A rare and difficult to track error occurs that C compiler reports that a function of the form `_ETexpanded_INTEGER` is not found.

This rarely occurs when an expanded type is used as an actual generic. If this happens and you have a simple example please report it to Object Tools. Workaround—find a copy of `_ETexpanded_INTEGER` in another *.c* file and copy it into the erroneous file.

11. INTERFACING TO C

You can accomplish most interfacing to C in CodeWarrior can be done by using a routine that is declared as **external** (rather than **deferred** or **do**). The *MOTEL* library has many examples of this. However, if you need a more sophisticated interface to C, including being able to create Eiffel objects from C and accessing the internals of objects from C, you should use the following include file in your C libraries. Interfacing techniques and the *Cecil* library are fully described in Chapter 24 of *Eiffel: The Language* by Bertrand Meyer.

```
#ifndef CECIL
/*-----------------------------------------------------------------*/
/* Eiffel/S 2.0 Cecil include file                                 */
/*-----------------------------------------------------------------*/

#define CECIL       1       /* Cecil included       */

/*-----------------------------------------------------------------*/
/* Eiffel <-> C types                                              */
/*                                                                 */
/* Note: Cecil routines which return Eiffel objects of non-basic   */
/*       type return a pointer of type 'EIF_PROXY'. This may or may */
/*       not be a pointer to an Eiffel object. To obtain a true    */
/*       pointer to the Eiffel object, use 'eif_access'.           */
/*                                                                 */
/* Warning:                                                        */
/*       Any assumption about the true nature of EIF_PROXY's is    */
/*       non-portable. Using a proxy as if it were a pointer to    */
/*       an Eiffel object may cause havoc.                         */
/*-----------------------------------------------------------------*/

typedef unsigned char      *EIF_BIT;
typedef unsigned char       EIF_BOOLEAN;
typedef unsigned char       EIF_CHARACTER;
typedef double              EIF_DOUBLE;
typedef long EIF_INTEGER;
typedef unsigned char      *EIF_OBJREF;
typedef unsigned char      *EIF_POINTER;
typedef char               *EIF_PROXY;
typedef double              EIF_REAL;
```

```
/*-----------------------------------------------------------------*/
/* Type descriptor                                                 */
/*-----------------------------------------------------------------*/

typedef unsigned char      *EIF_TYPE_ID;

/*-----------------------------------------------------------------*/
/* Function pointer types                                          */
/*                                                                 */
/* Note: Cecil routines which return routine pointers return a     */
/*       pointer of type 'EIF_xxx_FN' or 'EIF_PROC'. To call the   */
/*       routine, use the 'eif_??_call' features.                  */
/*                                                                 */
/* Warning:                                                        */
/*       Any assumption about the true nature of 'EIF_xxx_FN'      */
/*       pointers is non-portable. Using such a pointer as if      */
/*       it were a routine pointer may cause havoc.                */
/*-----------------------------------------------------------------*/

typedef unsigned char      *EIF_PROC;
typedef unsigned char      *EIF_BOOLEAN_FN;
typedef unsigned char      *EIF_CHARACTER_FN;
typedef unsigned char      *EIF_DOUBLE_FN;
typedef unsigned char      *EIF_INTEGER_FN;
typedef unsigned char      *EIF_OBJREF_FN;
typedef unsigned char      *EIF_POINTER_FN;
typedef unsigned char      *EIF_PROXY_FN;
typedef unsigned char      *EIF_REAL_FN;

/*-----------------------------------------------------------------*/
/* Eiffel/S types needed for the implementation of Cecil           */
/*-----------------------------------------------------------------*/

typedef void (*EIF_S_V)(EIF_OBJREF, ...);
typedef EIF_BOOLEAN         (*EIF_S_B)(EIF_OBJREF, ...);
typedef EIF_CHARACTER       (*EIF_S_C)(EIF_OBJREF, ...);
typedef EIF_INTEGER         (*EIF_S_I)(EIF_OBJREF, ...);
typedef EIF_OBJREF          (*EIF_S_O)(EIF_OBJREF, ...);
typedef EIF_POINTER         (*EIF_S_P)(EIF_OBJREF, ...);
typedef EIF_PROXY           (*EIF_S_PXY)(EIF_OBJREF, ...);
typedef EIF_REAL            (*EIF_S_R)(EIF_OBJREF, ...);

/*-----------------------------------------------------------------*/
/* Eiffel/S runtime routines needed for the implementation of Cecil */
/*-----------------------------------------------------------------*/

extern  void ES2RT_bit_put (EIF_BIT, EIF_BOOLEAN, EIF_INTEGER);
extern  EIF_BOOLEAN         ES2RT_bit_item (EIF_BIT, EIF_INTEGER);
extern  EIF_INTEGER         ES2RT_bit_count (EIF_BIT);
extern  EIF_INTEGER         ES2RT_array_count (EIF_OBJREF);
extern  EIF_POINTER         ES2RT_array_to_external (EIF_OBJREF);
extern  EIF_CHARACTER       ES2RT_array_type_code (EIF_OBJREF);
extern  EIF_INTEGER         ES2RT_string_count (EIF_OBJREF);
extern  EIF_CHARACTER      *ES2RT_string_to_external (EIF_OBJREF);
extern  void ES2RT_mem_adopt (EIF_OBJREF);
extern  void ES2RT_mem_wean (EIF_OBJREF);
extern  EIF_BIT             ES2RT_mem_bit_create (EIF_CHARACTER *);
extern  EIF_PROXY           ES2RT_mem_string_create (EIF_CHARACTER *);
extern  EIF_PROXY           ES2RT_mem_array_create (EIF_POINTER,
EIF_INTEGER,
       EIF_TYPE_ID);
extern  EIF_PROXY           ES2RT_mem_create (EIF_TYPE_ID);
extern  EIF_TYPE_ID         ES2RT_type_type_id (EIF_CHARACTER *);
extern  EIF_TYPE_ID         ES2RT_type_generic_id (EIF_CHARACTER *, ...);
extern  EIF_TYPE_ID         ES2RT_type_bit_type (EIF_INTEGER);
extern  EIF_INTEGER         ES2RT_type_generic_count (EIF_TYPE_ID);
extern  EIF_TYPE_ID        *ES2RT_type_generics (EIF_TYPE_ID);
extern  EIF_TYPE_ID         ES2RT_type_typeof (EIF_OBJREF);
extern  EIF_CHARACTER      *ES2RT_type_baseclass (EIF_TYPE_ID);
extern  EIF_CHARACTER      *ES2RT_type_name (EIF_TYPE_ID);
extern  EIF_TYPE_ID         ES2RT_type_expanded (EIF_TYPE_ID);
extern  EIF_BOOLEAN         ES2RT_type_cecil_conforms_to (EIF_TYPE_ID,
             EIF_TYPE_ID);
extern  EIF_S_V             ES2RT_disp_proc_prep (EIF_PROC);
extern  EIF_S_B             ES2RT_disp_boolean_prep (EIF_BOOLEAN_FN);
extern  EIF_S_C             ES2RT_disp_character_prep (EIF_CHARACTER_FN);
extern  EIF_S_I             ES2RT_disp_integer_prep (EIF_INTEGER_FN);
extern  EIF_S_O             ES2RT_disp_objref_prep (EIF_OBJREF_FN);
extern  EIF_S_P             ES2RT_disp_pointer_prep (EIF_POINTER_FN);
extern  EIF_S_PXY           ES2RT_disp_proxy_prep (EIF_PROXY_FN);
extern  EIF_S_R             ES2RT_disp_real_prep (EIF_REAL_FN);
extern  EIF_POINTER         ES2RT_disp_rdispatch (EIF_CHARACTER *,
     EIF_TYPE_ID);
extern  EIF_POINTER         ES2RT_disp_ddispatch (EIF_OBJREF,
EIF_CHARACTER *);

/*------------------------------------------------------------------*/
/* Predefined values                                                */
/*------------------------------------------------------------------*/

#define EIF_FALSE           ((EIF_BOOLEAN) 0)
#define EIF_TRUE            ((EIF_BOOLEAN) 1)
#define EIF_VOID            ((EIF_OBJREF) 0)

/*------------------------------------------------------------------*/
/* Convenience features for some fundamental classes                */
/*                                                                  */
/* These features allow you to create and manipulate objects of     */
/* type BIT, ARRAY and STRING in a convenient way.                  */
/*------------------------------------------------------------------*/
/* ARRAYs                                                           */
/*------------------------------------------------------------------*/

#define eif_array_count(o)              ES2RT_array_count (eif_access(o))
#define eif_array_make_from_c(s,c,t)
ES2RT_mem_array_create((s),(c),(t))
#define eif_array_to_c(o)               ES2RT_array_to_external
(eif_access(o))
#define eif_array_kind(o)               ES2RT_array_type_code
(eif_access(o))

/*------------------------------------------------------------------*/
/* BITs                                                             */
/*------------------------------------------------------------------*/

#define eif_bit_count               ES2RT_bit_count
```

```
#define eif_bit_make_from_c(b)      ES2RT_mem_bit_create((EIF_CHARACTER
*)(b))
#define eif_bit_item(o,p)           ES2RT_bit_item((o),(p))
#define eif_bit_put(o,x,p)          ES2RT_bit_put((o),(EIF_BOOLEAN)
(x),(p))

/*-----------------------------------------------------------------*/
/* STRINGs                                                         */
/*-----------------------------------------------------------------*/

#define eif_string_count(o)         ES2RT_string_count (eif_access(o))
#define eif_string_make_from_c(s)   ES2RT_mem_string_create(\
      (EIF_CHARACTER *)(s))
#define eif_string_to_c(o)          ES2RT_string_to_external
(eif_access(o))

/*-----------------------------------------------------------------*/
/* Object creation                                                 */
/*                                                                 */
/* Note: The fields of an object created with 'eif_create' are all */
/*       initialized to the proper default values. However, if the */
/*       generating class of the new object has creation procedures */
/*       then you have to call one of them to ensure that the new  */
/*       object satisfies the class invariant.                     */
/*-----------------------------------------------------------------*/

#define eif_create   ES2RT_mem_create

/*-----------------------------------------------------------------*/
/* Object access                                                   */
/*                                                                 */
/* To obtain a true reference to an Eiffel object from a pointer of */
/* type 'EIF_PROXY', you have to use 'eif_access'.                 */
/*                                                                 */
/* Warning: Do not store the result of a call to 'eif_access' for  */
/*          future reuse! The object represented by the proxy may  */
/*          move.                                                  */
/*-----------------------------------------------------------------*/

#define eif_access(p)               ((EIF_OBJREF)(p))

/*-----------------------------------------------------------------*/
/* Object protection                                               */
/*                                                                 */
/* If you want to store an EIF_PROXY for reuse in future activa-   */
/* of a C routine, you must protect it to ensure that the garbage  */
/* collector will not reclaim the object associated with the proxy. */
/* The feature 'eif_adopt' serves this purpose.                    */
/*                                                                 */
/* If protection is no longer needed, use 'eif_wean' to unprotect  */
/* the object associated with a proxy.                             */
/*-----------------------------------------------------------------*/

#define eif_adopt    ES2RT_mem_adopt
#define eif_wean     ES2RT_wean

/*-----------------------------------------------------------------*/
/* Type id's                                                       */
/*-----------------------------------------------------------------*/
```

```
#define eif_base_class              ES2RT_type_baseclass
#define eif_bit_type_id             ES2RT_type_bit_type
#define eif_expanded ES2RT_type_expanded
#define eif_generic_count           ES2RT_type_generic_count
#define eif_generic_id              ES2RT_type_generic_id
#define eif_generics ES2RT_type_generics
#define eif_type_id_of_object(o)    ES2RT_type_typeof (eif_access(o))
#define eif_type_id(n)              ES2RT_type_type_id ((EIF_CHARACTER
*)(n))
#define eif_type_name               ES2RT_type_name
#define eif_type_conforms_to        ES2RT_type_cecil_conforms_to

/*------------------------------------------------------------------*/
/* Routine calls                                                    */
/*------------------------------------------------------------------*/

#define eif_proc(r,t)               ((EIF_PROC_PROXY) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_boolean_fn(r,t)         ((EIF_BOOLEAN_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_character_fn(r,t)       ((EIF_CHARACTER_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_double_fn(r,t)          ((EIF_DOUBLE_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_integer_fn(r,t)         ((EIF_INTEGER_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_objref_fn(r,t)          ((EIF_OBJREF_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_pointer_fn(r,t)         ((EIF_POINTER_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_proxy_fn(r,t)           ((EIF_PROXY_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))
#define eif_real_fn(r,t)            ((EIF_REAL_FN) \
                ES2RT_disp_rdispatch((EIF_CHARACTER *)(r),(t)))

#define eif_proc_call(fp)           (*(ES2RT_disp_proc_prep(fp)))
#define eif_boolean_call(fp)        (*(ES2RT_disp_boolean_prep(fp)))
#define eif_character_call(fp)      (*(ES2RT_disp_character_prep(fp)))
#define eif_double_call(fp)         (*(ES2RT_disp_real_prep(fp)))
#define eif_integer_call(fp)        (*(ES2RT_disp_integer_prep(fp)))
#define eif_objref_call(fp)         (*(ES2RT_disp_objref_prep(fp)))
#define eif_pointer_call(fp)        (*(ES2RT_disp_pointer_prep(fp)))
#define eif_proxy_call(fp)          (*(ES2RT_disp_proxy_prep(fp)))
#define eif_real_call(fp)           (*(ES2RT_disp_real_prep(fp)))

/*------------------------------------------------------------------*/
/* Attribute access and reattachment                                */
/*------------------------------------------------------------------*/

#define eif_boolean_field(o,f)    *((EIF_BOOLEAN *) \
 ES2RT_disp_ddispatch (eif_access(o),(EIF_CHARACTER *)(f))
#define eif_character_field(o,f)  *((EIF_CHARACTER *) \
 ES2RT_disp_ddispatch (eif_access(o),(EIF_CHARACTER *)(f))
#define eif_double_field(o,f)     *((EIF_DOUBLE *) \
 ES2RT_disp_ddispatch (eif_access(o),(EIF_CHARACTER *)(f))
#define eif_integer_field(o,f)    *((EIF_INTEGER *) \
 ES2RT_disp_ddispatch (eif_access(o),(EIF_CHARACTER *)(f))
#define eif_objref_field(o,f)     *((EIF_OBJREF *) \
 ES2RT_disp_ddispatch (eif_access(o),(EIF_CHARACTER *)(f))
#define eif_pointer_field(o,f)    *((EIF_POINTER *) \
 ES2RT_disp_ddispatch (eif_access(o),(EIF_CHARACTER *)(f))
#define eif_real_field(o,f)       *((EIF_REAL *) \
 ES2RT_disp_ddispatch (eif_access(o),(EIF_CHARACTER *)(f))
/*-----------------------------------------------------------------*/
/* Pointer checks                                                  */
/*-----------------------------------------------------------------*/

#define EIF_VALID(p)        (((EIF_POINTER)(p)) != ((EIF_POINTER) 0))
#define EIF_INVALID(p)      (((EIF_POINTER)(p)) == ((EIF_POINTER) 0))

/*-----------------------------------------------------------------*/

#endif /* CECIL */Index
```

INDEX