The Emerald System[1]

USER'S GUIDE

Eric Jul
Rajendra K. Raj
Norman C. Hutchinson

Emerald Project
Department of Computer Science
University of Washington
Seattle, WA 98195

Emerald Document
Version 1.3 (UW)
(Revised Nov 1988)

Abstract:

The Emerald Project is an on-going effort to demonstrate that the object-based style of programming can be incorporated both elegantly and efficiently in the distributed programming environment. As part of this project, the Emerald language and kernel were designed and implemented. The main objective of this guide is to provide introductory information for users to facilitate compilation, execution and testing of programs written in Emerald.
Details about installing and updating new releases of Emerald are also included for the Emerald installer, and may be ignored by the typical user.

[1] This work was supported in part by the National Science Foundation under Grants No. MCS-8004111, DCR-8420945 and CCR-8700106, by Københavns Universitet (the University of Copenhagen), Denmark under Grant J.nr. 574-2,2, by a Digital Equipment Corporation External Research Grants, by an IBM Graduate Fellowship, and a dissertation award from Microsoft Corporation.

Contents

1 Getting Started

This guide describes how to compile, run and debug Emerald programs at several installations of Emerald. As Emerald is still an on-going project (as of Nov 1988) and is continually being revised, the reader is cautioned that this guide may be slightly out-of-date.

This guide has been written for the person who has a reasonable knowledge of the Emerald programming language; this background can be obtained by reading [Raj 88]. Hutchinson's dissertation [Hutchinson 87a] provides an excellent rationale for the design and implementation of Emerald, and the Emerald language report [Hutchinson 87b] describes its significant features. The published literature on Emerald includes the following articles [Black 87,Black 86,Jul 88].

The rest of this section presents a general introduction to the Emerald system. Section 2 shows how simple Emerald programs can be written, compiled and executed. Although the Emerald system does not have a dedicated debugger, the compiler and the kernel support tools that facilitate debugging; these tools are discussed in Section 3. Appendix A summarizes information about the commonly used Emerald utilities. Appendix B comprises a preliminary installer's guide, and provides information about the installation of new releases of the Emerald compiler and kernel; this may generally be ignored by Emerald users.

1.1 Overview

The Emerald system consists of a number of Ultrix programs of which the two most important are the Emerald compiler and the Emerald kernel (henceforth merely called the compiler and the kernel). These programs run on top of Unix/Ultrix systems as normal Ultrix user programs. The hardware base consists of one or more VAX computers connected by an Ethernet. Normally, there is one kernel running (in the background) on each computer. Each such kernel is contained in a single Ultrix process that stores all objects resident on that computer within its single Ultrix address space. Any Emerald processes executing within Emerald objects on the node are time-multiplexed by the kernel.

To create and execute an Emerald program, use your favourite editor to generate the Emerald program in a Ultrix text file, with a `.m` suffix. Next, the program file needs to be cross-compiled from the Ultrix world to create the corresponding Emerald objects by using the Emerald compiler. An easy of way doing this is to make the the compiler both produce the executable code files and then request the kernel to execute the produced files; the kernel reads the code files, performs the necessary linking, and executes them. Alternately, the compiler may be halted after producing a `.g` output file; subsequently, the utility program, `runec`, can be used to pass the resulting `.g` file to the kernel for execution.

The compiler stores the executable code files in a dedicated directory that is not accessed by the user. When execution is requested (e.g., by `runec`), the kernel looks for the code files in this directory on the local machine; if not found, a network-wide search is initiated, and the corresponding directory is searched on all other accessible Emerald machines. Thus an Emerald program may be compiled on one machine and executed on another machine. A simple input/output facility is available via the builtin types `stdin` and `stdout` that provide simple character input and output corresponding to `stdin` and `stdout` in C/Ultrix programs.

1.2 Machines

The following machines will usually have the necessary software for running Emerald: Freja, Roar and Bjarke. Note that Freja, Roar and Bjarke are used for development, and might run a different

version of Emerald than the other machines. Usually there is a kernel running on each of these machines; if not, follow the instructions in Section 1.4 to boot a kernel.

1.3 Directories

The Emerald software is kept in the Ultrix directory `/usr/projects/emerald`[0] and the executable programs are kept in the directory `/usr/projects/emerald/bin`. It is suggested that you keep this directory in your Ultrix search path.

1.4 Booting a kernel

The current version of the kernel is kept in `/usr/projects/emerald/bin`. The kernel is booted by executing the `em` command. The common practice is to redirect the kernel output to a log file:

```
em >& kernel.log
```

The kernel outputs bootup and shutdown messages and any relevant error messages to its standard output. Furthermore, trace messages (see Section 3.1) and user output (see Section 1.7) may also be written to standard output.

The kernel accepts a number of parameters; of these, a few are useful to the the user, but most are meant for kernel debugging (cf. Section 3).

1.5 Compiling and running a program

A program is compiled and run by:

```
ec program.m
```

Alternately, the program may be compiled using the `-C` flag which prevents the compiler from executing the program immediately. Thus to compile a program for later execution, use

```
ec -C program.m
```

The program may be executed later by using

```
runec program.g
```

The compiler requires an *Emerald name server* to be running on Freja. If the name server is not running on Freja, then the compiler attempts to start it. This is a temporary limitation on the distributed nature of Emerald, one that will hopefully be corrected in a future version.

The compiler has a number of optional flags, for a description of them, use `ec -h`. Note that the compiler cannot produce a listing; use, e.g., `cat -n program.m` instead. The error messages produced by the compiler are generally self-explanatory.

1.6 Builtins

A number of builtin objects and types are available in the Emerald system; these include traditional types such as **Integer**, **Real**, etc., and newer ones such as **InStream**, **Node** and **OutStream**. A description of these objects is beyond the scope of this guide; the reader is referred to the Emerald Language Report [Hutchinson 87b].

[0] This is the directory structure used at DIKU. See Appendix B for the directories used at other sites.

1.7 Input/Output

Every compiled object is provided with two builtin references (`stdin` and `stdout`) to i/o streams similar to Ultrix character i/o. These streams have already been opened and can be used immediately. The example in chapter 2 illustrates the usage of streams.

When a program uses character i/o, the `-i` option should be used when executing it (either directly using the compiler or with `runec`). This option causes the compiler/`runec` to use its own standard input/output as the input/output of the program. When used, these streams must be specifically closed before program termination, or the terminal will hang.

When the `-i` option is not used, input operations on `stdin` return end-of-file immediately; output from `stdout` is piped into the standard output of the kernel that received the request to execute the program. It is interesting to note that this output will always be bound to the original kernel, even when the actual program objects using `stdin` and `stdout` move to other machines.

1.8 Run-time Errors

A number of different errors may occur at run-time. As a rule, all such error messages appear on the standard output (or error output) of the kernel. Therefore, when things go wrong (or the program hangs), the first place to look is in the kernel log. If this does not help, starting both line number and failure tracing is recommended (see Section 3.1); this can be done by:

```
emtrace -T LineNumber -T Failure
```

There are several reasons why a user process may fail; these include the following:

- an attempt to invoke `NIL` (usually caused by an uninitialised variable).
- an assertion failure, e.g., `assert FALSE`.
- an assertion failure due to index out of bounds when executing an operation on an Array or Vector.
- read or write operations on a closed stream, or repeated attempts to read past end-of-file.

1.9 Restrictions and Peculiar Features

As stated elsewhere, Emerald is an on-going research project and is continually being improved and "fixed." Some of the present restrictions include:

- the `move` primitive does not work correctly for remote objects; this can be avoided by moving oneself to the remote site, performing the move, and then returning oneself to the original site.
- the `fix`, `unfix` and `refix` primitives are not implemented as of Nov 1988.
- garbage collection has not been implemented as yet. So the kernel breaks when it runs out of virtual memory. In addition, the disk fills up with old code files (when recompiling a program, the old code files are not removed). The utility program `newTreeVersion` can be used to remove *all* Emerald code files; note that if this program is used, it will be necessary to recompile *all* user programs so use this utility with great care.
- message forwarding sometimes does not work properly, and the status *unavailable* may be returned. For highly mobile objects, a simple way of avoiding this error is to *locate* it before invoking a highly mobile object.

2 Examples

This section presents two examples of Emerald programs. The first, an extremely simple program, provides a rather detailed introduction to the Emerald kernel and compiler. The second illustrates the interaction between the compiler and the kernel environment.

2.1 A Simple Example

Figure 1 shows a program that computes the machine precision of the builtin type *Real*. Let us assume that the program is stored in `/usr/projects/emerald/demo/testReal.m` and that no kernel is running. First, on Freja we boot a new kernel redirecting its output to a log file.

```
em >& kernel.log&
```

The kernel boot should take about 5–10 seconds. We can check whether it has indeed booted or not by:

```
snapshot -n Whatisup
```

This snapshot attempts to connect to the kernel, and requests the local kernel to reveal some information about the kernels (including itself) that are currently up and running, or dead. If the kernel has not booted properly, the `Whatisup` snapshot will output:

```
snapshot: connect: Connection refused
```

Any boot problem will be reported in the file `kernel.log`. When the kernel is fully booted, the `Whatisup` program outputs something like:

```
Em36: KMD Connected to host 54
Calling Whatisup on taber, param: 0 (0x00)
Emerald network according to node 54 on Wed Nov  9 12:19:53 1988
 LNN  Incarnation      Physical     State      Last state change
 ------------------------------------------------------------------
  51  Nov  9 11:54:21  whistler     Alive      Nov  9 11:54:34
  54  Nov  9 11:52:17  taber        Alive      Nov  9 11:53:17
  55  Nov  9 12:17:21  roskilde     Alive      Nov  9 12:19:39
*** End of Snapshot ***
Snapshot done.
```

This output indicates that a kernel has been successfully booted on "Freja," and that its logical node number (LNN) is "54"; the LNN is extracted from the tail digits of the node's Internet number. This snapshot also shows that kernels are running on machines Roskilde and Whistler.
Now ensure that we are in the right directory by:

```
cd ../demo
```

The program is compiled by

```
ec -v -gt -gd -gm -C testReal.m
```

The `-v` flag makes the compilation verbose and shows the execution of the different compiler phases. The `-gt -gd -gm` flags are used to include code that helps in line number tracing, debugging and generating statistics. The `-C` flag requests the compiler to store the executable object's ID in the file `testReal.g` rather than immediately execute the program. The compiled object can be executed later by

```
% Tests reals, standard output by computing the precision of the builtin type Real
const precisionTester ==
   object pTester
      process
         var a: Real
         var precision: Integer
         a ← 1.0
         precision ← 0
         loop
            stdout.PutReal[a]
            stdout.PutString["\^J"]
            a ← a / 2.0
         exit when 1.0 = 1.0 + a
            precision ← precision + 1
         end loop
         stdout.PutString["Precision is "]
         stdout.PutInt[precision, 1]
         stdout.PutString[" bits.\^J"]
         stdout.close
         stdin.close
      end process
   end pTester
```

Figure 1: Emerald program for testing precision of the builtin type **Real**

```
runec -i testReal.g
```

The `-i` option makes the output go to the standard output of the shell executing the `runec` command:

```
1
0.5
0.25
0.125
0.0625
0.03125
0.015625
0.0078125
0.00390625
0.00195313
0.000976563
0.000488281
0.000244141
0.00012207
6.10352e-05
3.05176e-05
1.52588e-05
7.62939e-06
3.8147e-06
1.90735e-06
```

```
9.53674e-07
4.76837e-07
2.38419e-07
1.19209e-07
5.96046e-08
Precision is 24 bits.
```

Note that at the end of the program `stdin` and `stdout` are closed so that the `runec` command completes and the shell prompts the user for another command.

2.2 Another Example

This example consists of two program files that are compiled separately: the first defines a polymorphic stack and the second illustrates its usage. While the details about the actual Emerald program are beyond the scope of this document (see [Hutchinson 87a] for a discussion of polymorphism in Emerald), we are interested in the interaction of the compiler and kernel environments here.

Figure 2 defines the Stack object, and permits the definition to be exported to the Emerald environment path (`StackEnv`) on the current machine. Let us assume that this definition has been entered in a textfile, say `polystack.m`. It can be compiled as follows:

```
ec -v -gt -gd -gm -C polystack.m
```

The use of these compiler flags has been explained above. The compiled Stack of Figure 2 is tested in the program of Figure 3. If this program has been entered in the file, `testpolystack.m`, it can be compiled as:

```
ec -v -gt -gd -gm -C testpolystack.m
```

This program extracts the previously-compiled Stack from the environment path `StackEnv`.
If this program is executed using

```
runec -i testpolystack.g
```

it will produce the following output

```
Testing the Stacks.
Pushing: 0 on both stacks
Pushing: 1 on both stacks
Pushing: 2 on both stacks
Printing in Reverse Order.
Next number: 2
Next string: 2
Next number: 1
Next string: 1
Next number: 0
Next string: 0
End of test.
```

```
% Export the definition of Stack from present compilation environment
% to the Emerald environment on the current machine.
export Stack to "StackEnv"
% A 3−level Emerald object definition to facilitate a polymorphic Stack
const Stack == immutable object Stack
    export of
    function of[eType: AbstractType] → [result: NewStackType]
        where
            NewStackType == immutable type NewStackType
                function getSignature → [Signature]
                operation Create → [NewStack]
            end NewStackType
            NewStack == type NewStack
                operation Push[eType]
                operation Pop → [eType]
                function Empty → [Boolean]
            end NewStack
        end where
        result ← immutable object aNewStackType
            export GetSignature, Create
            function getSignature → [r : Signature]
                r ← NewStack
            end getSignature
            operation Create → [r: NewStack]
                r ← object aStack
                    export Push, Pop, Empty
                    var s: Array.of[eType] ← Array.of[eType].create[0]
                    operation Push[n: eType]
                        s.addUpper[n]
                    end Push
                    operation Pop → [n: eType]
                        n ← s.removeUpper
                    end Pop
                    function Empty → [result : Boolean]
                        result ← s.empty
                    end Empty
                end aStack
            end Create
        end aNewStackType
    end of
end Stack
```

Figure 2: Emerald program defining a polymorphic stack object

```
% This extracts the Stack definition from "StackEnv" for use in this compilation.
import Stack from "StackEnv"
const stackTester == object stackTester
    process
        const intStack : Stack.of[Integer] ← Stack.of[Integer].create
        const strStack : Stack.of[String] ← Stack.of[String].create
        const Max == 3
        var i : Integer ← 0
        stdout.PutString["Testing the Stacks.\^J"]
        loop
            stdout.PutString["Pushing: " || i.asString || " on both stacks\^J"]
            intStack.Push[i]
            strStack.Push[i.asString || "\^J"] % Catenate a newline character
            i ← i + 1
            exit when i = Max
        end loop
        stdout.PutString["Printing in Reverse Order.\^J"]
        loop
            var j: Integer ← intStack.Pop
            var k: String ← strStack.Pop
            stdout.PutString["Next number: " || j.AsString || "\^J"]
            stdout.PutString["Next String: " || k]
            exit when intStack.Empty
        end loop
        stdout.PutString["end of test.\^J"]
        stdout.close
        stdin.close
    end process
end stackTester
```

Figure 3: Emerald program to test the polymorphic stack

3 Debugging tools

There are several debugging tools available. These tools were designed for kernel debugging, but many of them provide high level information that may be useful to Emerald programmers.

3.1 Traces

A trace facility provides for dynamically starting and stopping tracing of kernel events. The trace program is called as a normal Ultrix user program. Multiple traces can be active at any given time. Traces are terminated by killing them (use `^C` or the `kill` program) or when the traced kernel crashes or shuts down. For example,:

```
emtrace -l 5 -T MMTraceMsg
```

starts full tracing of the Ethernet message module.

The `-T` option is used to give the name of the trace. Multiple `-T` options may be present resulting in the simultaneous tracing of several different traces. Some traces have multiple levels of output. The default level is three and is in effect until another level is given using the `-l` option, e.g., `-l 5` raises the level to five. In general, higher numbers give more output according to the following scheme:

1 gives only the most serious error messages.

2 gives any uncommon event such as the retransmission of a message on the Ethernet. Thus levels below 2 should give no output during normal, error-less operation.

3 gives a message for every normal event, e.g., the sending or receipt of an Ethernet message.

4 gives more detail about the event, e.g., the message id, the destination, etc..

5 gives all relevant details.

6 & above gives all relevant and irrelevant details.

The `emtrace` program connects to the kernel and performs the desired trace indefinitely – it is stopped by killing it.

3.1.1 Traces for information about Emerald processes

The following traces provide user level information about the execution of processes:

LineNumber outputs a message every time an Emerald process starts or stops execution. If the object being executed was compiled with the `-gt` flag then a message is also output before attempting to execute a (non-empty) line of source code. The reference contains the name of the object [1] being executed as well as the line number in the original source text.

ProcessSwitch outputs a line for every process switch.

Failure outputs information about every failure that occurs. Higher test output levels give more detailed information about stack unwinding and register restoring.

[1] The name of the object is the identifier following the keyword `object` in the object literal that defined the object.

Invoke outputs information about remote invocations. Higher levels give a great deal of detail and should only be used by kernel debuggers.

HOTSUpDown outputs a message every time the kernel detects a reboot or crash of another kernel. Higher levels give details of the associated table updates.

3.1.2 Traces for kernel debugging

The following traces are intended for kernel debugging only:

MMTraceMsg traces activity in the Message Module that is used for sending and receiving messages on the Ethernet.

Code traces the loading of object code files.

Create traces the creation of Emerald objects specifically storage allocation.

Node traces operations executed on the Node abstraction.

LM traces assembly and disassembly of messages of unbounded length. Long messages are broken into a sequence of fixed size packets for Ethernet transmission via the Message Module.

TT traces insertions into the Translation Table that is sent along with remote invocations and object moves.

InvokeQueue traces updating of the invoke queues.

SI is a very low level trace of the multiplexing of Ultrix signals.

AbCon traces the construction of AbCon vectors. AbCon vectors map abstract operation numbers onto code addresses.

Translate traces the object to kernel linkage phase of object code file loading.

Monitor traces monitor entry and exit that are handled by the kernel. Note that many monitor entries and exits are handled directly by compiled code and are not traced.

StackSegment traces the allocation and deallocation of Stack segments.

Emalloc traces the allocation of Emerald object storage.

GC traces garbage collection.

Item traces the packing and unpacking of submessages from a single Ethernet message.

Locate traces the location protocol that finds objects given only their OIDs.

Conform traces the conforms algorithm. Note, that this algorithm was derived directly from the algorithm in the compiler.

DebugMsg traces miscellaneous kernel stuff.

View traces the execution of the `view` statement.

UserIO traces Emerald process output operations on `InStream` and `OutStream`.

FixMe produces a message each time certain parts of the kernel code is executed. These points are considered to be flawed in some way. The intent is for some future kernel programmer to either fix the problem or at least be aware of the potential for disaster.

Stack traces the dynamic expansion of stacks.

OT traces insertions into the object table.

Move traces mobility related code.

3.2 Snapshots

The snapshot program provides for dynamically requesting data about the kernel and for performing simple kernel management tasks. The snapshot program is run as an Ultrix utility program. To obtain a list of available snapshots and traces, use one of the following:

```
snapshot
snapshot -n Menu
snapshot -n Help
```

Snapshot names are given using the `-n` option followed by the name. Some snapshot require an integer value. These values may be specified using the `-d 31` option (use `-h 1f` to give hexadecimal values). Some snapshots require a string value which is specified using the `-S StringValue` option.

An alternate snapshot program performs repeated snapshots and displays the first screen-full of the result. The program is called `wstats` and has an option `-t 10` to specify an alternate interval in seconds for each snapshot. The default is 5 seconds of waiting between two successive snapshots.

The following sections describe the available snapshots sorted according to usage. Snapshots marked with a star(*) are intrusive and change the state of the kernel – they should be used with a great deal of caution and are intended for kernel hacking (except for the snapshots in Section 3.2.2 which are intended for general use).

3.2.1 Helpful Snapshots

The following snapshots are intended for use by Emerald programmers to aid in the use of the debugging facilities.

Help prints a help message.

Menu prints a menu of traces and snapshots.

Snapshots prints a menu of snapshots.

Traces prints a menu of traces.

Flush flushes the kernel standard output and standard error.

NOP prints a message (this can prove that the kernel is up and alive).

3.2.2 Emerald Process Manipulation

ps prints a summary of the state of the Emerald processes.

StopProcess* stops the execution of an Emerald process. The execution of the given process is suspended until a StartProcess snapshot re-enables execution of the process. The process may be both blocked and stopped. If it becomes unblocked (e.g., the monitor for which it was awaiting entry becomes available) then it will not proceed until unstopped. The process id may be obtained using the `ps` snapshot and is commonly given in hexadecimal using the `-h` option to snapshot.

StartProcess* restarts the execution of the given process.

3.2.3 Kernel Status

The following snapshot is used for printing out the kernel's idea of the current state of other Emerald kernels.

Whatisup prints the state of the other known Emerald kernels.

3.2.4 Statistics

The following snapshots are used for obtaining statistics about the Emerald kernel.

MMStats prints message module statistics.

EtherStats prints message module statistics for communication with the node whos LNN is given as an integer parameter.

EMDumpStats dumps the statistics about the execution of Emerald programs.

EMResetStats* resets the counters used by EMDumpStats.

3.2.5 Kernel Data Structure Dumps

The following snapshots produce dumps of kernel data structures.

HOTSDump dumps the HOTS table. If an integer parameter is given then only the entry for the given LNN is dumped.

EmDataDump dumps the dynamically allocated memory containing real and pseudo Emerald objects.

OT dumps the object table that maps OIDs to object descriptors.

MallocDump dumps memory allocated by malloc (and calloc).

FLCodeLoadMap dumps the set of currently active code loads.

LOCLocateMap dumps the set of currently active location requests.

FLCompilerLoadMap
dumps the set of currently active compiler load and execute requests.

FLCheatingLoadMap
dumps the set of so-called cheating code loads in progress.

TimerDump dumps the set of dynamically settable timers.

ActiveTimers dumps the set of currently active timers.

FLRemoteLoadMap
dumps the set of remote code loads that are in progress.

CurrentLoad prints out the current Emerald process load.

INVKInitiallyMap
prints out the set of objects that currently are not fully instantiated and have processes awaiting access to them.

3.2.6 Kernel Management

The following snapshots can be used to modify the kernel state and are to be used with extreme caution. *Only seasoned kernel hackers should use these!*

PullUp* the HOTS table entry whose LNN is given as a parameter is changed to indicate that the LNN is up.

PullDown* the given LNN is marked as DEAD in the HOTS table.

FLLoadFile* the code file with the given OID is loaded from disk (this snapshot is obsolete).

FLSendCode* the code file with the given OID is sent to the kernel whose LNN is given as an integer parameter.

FLCreateOneOfCTOID*
the kernel instantiates an object using the code file identified by OID.

SendAlive* send out an "I am alive broadcast".

ResetTimeSlicer* reset the time slicer to the interval given as an integer in units of milliseconds, e.g., 1000 means time slice every second.

Shutdown causes the kernel to commit suicide.

GC start a new garbage collection (not implemented).

3.2.7 Changing the value of kernel integer variables

The following snapshots may be used to inspects and change the value of integer variables in the kernel. Warning: you have better know what you are doing before using these snapshots. Certain kernel variable are intended to be inspected and have names formatted as "cXX...." where XX indicates what module they are defined in (e.g., MM for message module). Other variables may be modified and have names formatted as "vXX....".

Variables prints the current cache of variable names and their values.

PrintVar prints the value of the variable whose name is given as a string (using the `-S` option.

ChangeVar changes the value of the variable whose name is given as a string. The new value is given as an integer (using the `-h` or the `-d` option).

3.2.8 Testing

The following snapshots are used exclusively for testing long Ethernet messages: LMSetPingDataSize, TLMPingTiming, RIPing, LMPing. Refer to the kernel code for the Long Message module (lmcode.c).

3.3 Remote Debugging

Both traces and snapshots may be enabled remotely by using the `-m` option, e.g., by

```
snapshot -n MMStats -m Roskilde
```

The network host name is used to identify the computer where the desired kernel is running. Since there is only one kernel incarnation per computer, no further specification is needed.

A Summary of Helpful Information

This section summarizes the useful information about Emerald. Note that all the Emerald programs are available in `/usr/projects/emerald/bin`, which must be included in your Ultrix search path to preserve sanity.

A.1 Emerald Compiler

The following helpful information about the Emerald compiler used for compiling an Emerald source program and executing it can be obtained using: `ec -h`.

```
Usage:  ec <flags> <filename>
        flags:
                [-why] [-v] [-c] [-C[-]n] [-z] [-h] [-g[dtm]] [-i] [-Z]
                [-T<tracename>[=level]] [-O<optionname>[=value]]
                [-R <rootpath>] [-[Mm] <machinename>]
        maintenance flags:
                [-b] [-t] [-d <oid>]

The Emerald Compiler compiles a single Emerald source program and
executes it on the local Emerald kernel.  The flag arguments are:
    -why        Tell me why conforms checking fails.
    -v          Be verbose about the various compiler passes
    -c          Stop after type checking (except when doing a builtin
                where it means stop after doing exports)
    -C[n]       Stop after n passes (N >= 0), or with n passes to go
                (n < 0) -C-2 means stop after generating code, -C-1 means
                stop before invoking the Emerald kernel to run the program
    -z          Leave the temporary files in the current directory
    -h          Prints this list.
    -gd         Compile code to assist in debugging.
    -gt         Compile code to enable line number tracing.
    -gm         Compile code to gather statistics.
    -i          Interactive - the compiler's stdin/out is passed to the
                created object.
    -Z          Create mutable objects at compile time.
    -T<traces>  Turn on tracing.  The traces argument is parsed as a comma
                separated string of trace names and optional trace values
                with an = between the name and value.  A legal string is:
                    -Tassign,environment=5
                Use -Thelp to see the available traces.
    -O<options> Turn on various options.  The options argument is parsed
                as a comma separated string of option names and optional
                option values with an = between the name and value.
                A legal string is:
                    -Oinvokequeue,comment=5
                Use -Thelp to see the available options.
    -M <name>   Run the compiled program on machine named "name".
    -R <path>   Use path instead of /usr/em as the root of the Emerald
```

```
               Unix subtree.
Maintenance flags:
    -b          Compile the description of a builtin type
    -t          Write the intermediate tree file at interesting
                stages (currently undergoing change)
    -d<oid>     Read the intermediate tree file (internal format)
                and display it
```

A.2 Debugging Tools

A.2.1 Tracing

Using: `emtrace usage` prints out the following information:

```
Usage: trace <options>
    -T <name>           Start tracing
    -S <name>           Start the trace; output to kernel stdout
                        (to stop, use -C in a later call)
    -l <level>          Set the trace level for -S & -T
                        (must preceed -T and -S)
    -M <name>           Trace kernel on the named machine
    -m <name>           Trace kernel on the named machine
    -x [<level>]        DebugMsg trace
    -t [<level>]        MMTrace
The program waits until it or the kernel dies.
```

The kinds of traces available can be obtained using the `snapshot` command, as explained next.

A.2.2 Snapshots

Using `snapshot usage` gets the following information:

```
Usage: snapshot [-m <machine name>]
    [-N <LNN> | -X <hexLNN> | -P <pid>]
    [-n <snapname>] [-{d|h} <parameter value> | -S <string>]
```

Using `snapshot -n Menu` gets the following information:

```
The following 50 snapshots are available:
  Menu                HOTSDump            ActiveTimers        TimerDump
  EtherStats          MMStats             PullUp              PullDown
  Whatisup            Snapshots           Traces              Variables
  Flush               Help                NOP                 PrintVar
  ChangeVar           FLLoadFile          FLSendCode          FLCreateOneOfCTOID
  FLCodeLoadMap       FLCompilerLoadMap   FLCheatingLoadMap   FLRemoteLoadMap
  FLUnknownATAbConMa  FLAbConMap96        SendAlive           DumpCondMap
  Shutdown            ps                  StopProcess         StartProcess
  MallocDump          EmallocDump         GC                  ResetTimeSlicer
  CurrentLoad         RIPing              LMPing              LMSetPingDataSize
  INVKInitiallyMap    INVKFrozenMap       OTDump              OTDataDump
  LOCLocateMap        EmDataDump          EMDumpStats         EMResetStats
```

```
  TPingTiming         TLMPingTiming
The following 36 traces are available:
  HOTSUpDown          SI                  DebugMsg            MMTraceMsg
  AbCon               View                Code                Translate
  UserIO              Failure             Monitor             FixMe
  Create              StackSegment        Stack               Node
  Own                 Portability         Vector              Emalloc
  GC                  LM                  Item                ProcessSwitch
  LineNumber          Invoke              OT                  TT
  Locate              Move                InvokeQueue         Conform
  Checkpoint          CPTT                Recover             X
To print variables, try the Variables snapshot
*** End of Snapshot ***
```

A.2.3 Repetitive Snapshots

```
Usage: wstats [-N <LNN>] [-M <Machine Name>]
              [-t <WaitInterval>] [-n <Snapshot>]
```

A.2.4 Short Cuts

Some commonly used snapshots have been placed in /usr/projects/emerald/bin as "full-fledged" programs:

```
Shutdown    is equivalent to        snapshot -n Shutdown
Whatisup    is equivalent to        snapshot -n Whatisup
emps        is equivalent to        snapshot -n ps
empsv       is equivalent to        wstats -n ps
```

A.2.5 The Guru

When all else fails, contact your local Emerald guru.

B Installer's Guide

Emerald is currently operational at the following sites: University of Washington (Seattle), University of Arizona (Phoenix), DIKU (Copenhagen) and Digital Equipment Corporation (Littleton). Since Emerald is an on-going research project, and the Emerald system is constantly being revised, it is likely that newer (and better!) versions of the Emerald system will be released.

This appendix describes how to install new Vax releases of the Emerald system; a future release of this document will deal with the installation of Emerald on the SUNs. There is a four-part procedure that can be used to install the Emerald system:

1. The preliminary initialisation,
2. Making the Emerald Compiler,
3. Making the Emerald Kernel, and
4. Testing the new installation.

B.1 The Preliminaries

Several site-dependent details have to taken care of first. First, the different sites have different defaults for the Ultrix directories needed by the Emerald system:

Digital: /usr/local/em
Washington: /usr/em
Arizona: /usr/norm/em
DIKU: /usr/projects/emerald

The standard Emerald distribution tape contains the directory `em`. For the above specified sites, it should be loaded in the above specified directories; for other sites, it may be loaded as desired, but the installer must make the changes suggested below. This guide simply uses a generic `../` to refer to the Emerald directory; the installer is responsible for using the appropriate directory.

For the proper execution of Emerald, the default Emerald directory on each machine should contain the following sub-directories: `bin, EC, ErrCodes, Nodes,` and `Builtins`. The Emerald directory on the machine designated for the source code should also contain the sub-directories: `Language` and `Kernel`, which contain the required source code; the standard Emerald distribution tape will correctly load the sources onto the chosen default directory. After loading the tape, the following preliminary steps should be taken:

1. Change to the Emerald directory.

```
cd ../em
```

2. Edit the file `fixemdir` to include your machine name, and make appropriate choices for the other alternatives (`emdir, rootdir, emserverhost, emsite`).

3. The following command will take a long time as it finds all appropriate files that have to be edited to include the proper pathname for the Emerald root directory.

```
fixemdir -r
```

4. After the files have been located, the following command can be used to perform the actual editing.

```
fixemdir
```

B.2 Making the Compiler

The following steps have to be taken to make the Emerald compiler:

1. The Operation Name Server has to be created first; this is the server that maps the strings representing operation names into unique integers. In addition, it manages the allocation of OIDs for compiler generated objects. To create this, execute the following:

```
cd ../em/EC/OperationNames.
make
make new
```

2. This step creates the dummy builtin objects that helps start the compilation of the Emerald compiler. The commands to be executed here are:

```
cd ../em/Language/Compiler/DBuiltins
make
```

3. This step takes a fairly long time, and when done, will produce the new Emerald compiler. The Ultrix commands needed are:

```
cd ../em/Language/Compiler/Builtins
../em/bin/newTreeVersion
make fromscratch
cd ..
rm -f ../em/bin/ec
cp -p ec ../em/bin/ec
```

4. The `runec` utility has to be created next. The following commands will accomplish this:

```
cd ../em/Language/Runec
make install
```

B.3 Making the Kernel

The following three steps are needed to create the Emerald kernel, i.e., the run-time system:

1. The first step will require some actual editing of at least two of the kernel source files: `main.c` and `msgCode.c`. In the first file, you should add the names of the machines in your local environment that will be running the Emerald system. While these names are not really necessary, providing the names will enable better, and more meaningful diagnostics.

   In `msgCode.c` you will find some stuff dealing with `ETHERDEV`. The code needs to know the name of the Ethernet devices used by the various local machines. This usually varies from place to place, and from system to system. Edit it to get the appropriate name for your ethernet device. Look in the Ultrix files, `/etc/rc*`, for the name of the Ethernet device. Alternatively, execute

```
/usr/ucb/netstat -n -i
```

and use the first name specified in the result. This device name is needed because the Emerald run-time system requires the ability to do `udp` broadcasts, and needs the device to find the broadcast address.

2. After the above editing, the Emerald kernel can be made by executing the following:

```
cd ../em/Kernel/Em
make depend
make em
mv -f em ../em/bin/em
```

3. The Kernel Measurement and Debugging (KMD) tools need to be made next. They are used to examine/modify the state of the run-time system when it is running. Of these, `emtrace` and `snapshot` are the most useful.

```
cd ../em/Kernel/KmdOps
make -f MakeUtil install
```

B.4 Testing the Installation

The final step in the installation is to ensure that the system has been made properly. There is a minimal set of tests that exercise both the Emerald compiler and run-time system on a single machine. The following commands enable the testing of the Emerald installation:

```
cd ../em/Language/ExecTests
make clean
make test
```

If the output from `make test` looks fine, the Emerald system has probably been installed properly.

References

[Black 86] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. Object Structure in the Emerald System. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986.

[Black 87] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, TR 87-01-01, Department of Computer Science, University of Washington, Seattle, January 1987.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language Report.* Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987. (Revised August 1988).

[Jul 88] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[Raj 88] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. The Emerald Approach to Programming. Technical Report 88-11-01, Department of Computer Science, University of Washington, Seattle, November 1988. Revised February 1989.