Compiler Hopefully, all the things that the compiler itself needs to know are summarized in the `Target` module. In `system/compiler/misc`, create Target.i3.*target* and `Target.m3.`*target*, using the contents of the other versions of these files as a templates.

Runtime routines The file `system/runtime/M3machine.h.`*target* (which you need to create) is included in all the C files that are produced by the compiler. You can compare the existing versions to get a flavor of what goes there.

Threads On `DS3100` systems, thread switching is implemented by standard C code. This code should work on other architectures as well, provided that `_longjmp` does not insist that the new stack pointer should be lower in the stack than the current stack pointer.

VAX/Ultrix 3.1 does not allow this liberal use of `_longjmp`, so there is a piece of assembly code from which you can start if your *target* architecture is also restrictive.

RandomReal The `RandomReal` module (in `system/corelib/random` depends on the architecture. Create a `libs/misc/random/RandomReal.m3.`*target*, taking the existing files as an example.

Loader The loader (`system/loader/m3ld.c`) also depends on the system, because it has to rebuild C names from their translation by the C compiler as they appears in object files. For example, the VAX C compiler adds an underscore in front of C names, while the DS3100 compiler doesn't. The proper behavior is achieved in `system/loader/m3ld.c` by conditional compilation.

Ultrix Library You are running a version of Unix for which there is no supported library, you can make one if you want. This is tedious, because is means putting most of `/usr/include` in Modula-3 syntax. However, this not needed by the compiler and basic runtime itself. The core library has an interface `CoreOs` (in `system/corelib/coreos`) that defines just the things that are needed by the system itself.

Good Luck!

- chose a name for the new system, say *target*. Add it to the `TARGETS` and `CLONES` definitions in `dist-1.5/makefile`.
- whenever you create a new source file, for example `dist-1.5/util/machine.`*target*, you need to create some links in `cross` and `port`:

  ```
  $ (cd cross/util; ln -s .SRC/machine.target)
  $ (cd port/util; ln -s .SRC/machine.target)
  ```

- create the file `dist-1.5/util/machine.`*target*, with the following lines in it:

  ```
  MACHINE = target
  #define target_ARCHITECTURE
  ```

  Don't forget the links in `cross` and `port`.
- configure both hierarchies:

  ```
  $ (cd cross; make target)
  $ (cd port; make target)
  ```

- modify the compiler, the runtime and the core library as described below Whenever, you need to introduce system-dependant part in a C source, you can use a conditional of the form
- add `.ROOT/util` to your path (in each subdirectory of the system, there is a link named `.ROOT` to the top of the hierarchy)
- build the cross-compiler, i.e. the compiler that generates code for the *target* system and runs on the *host* system:

  ```
  $ (cd cross/system/compiler; m3all)
  ```

- produce the C files in the port:

  ```
  $ (cd port/system; m3toC)
  ```

- on the *target* system, finish the construction of the compiler (it produces code for the *target* system and runs on the *target* system):

  ```
  $ (cd port/util; cc -o imake imake.c)
  $ (cd port/system; m3fromC)
  ```

  This compiler is stored in `compiler/m3compiler.target` and the associated runtime and core library in `runtime/m3run.target.o` and `lib/m3lib.target.a` respectively.
- you can run the test programs to see if everything works:

  ```
  $ (cd port/tests; m3all test)
  ```

- if all went well, you have a working system for *target*. Otherwise, you have to determine what's wrong and go back to constructing the cross-compiler
- send us (`m3-request@src.dec.com`) your modifications for the *target* architecture, so that we can incorporate them in the next release.

It does whatever cleanup is necessary and passes control on up the stack to the next handler until the exception is actually handled.

Reference types are represented at runtime by a "typecell". Due to separate compilation, opaque types and revelations, it is not possible to fully initialize typecells at compile time. Typecell initialization is finished at runtime prior to the execution of any user code. A typecell contains a type's typecode, a pointer to its parent typecell, the size of the types referent and method list if any, the type's brand, the number of open array dimensions, the type's fingerprint, and procedures to initialize the typecell, initialize new instances of the type, and print instances of the type.

Initialization of a Modula-3 address space occurs in several steps. First, all types are registered. That is, a global array that points to all typecells is built. Next, the runtime verifies that all opaque types have been given concrete representations. Then, the initialization of typecells is finished. Then, all types with the same brand and fingerprint are identified with the same typecode. Then, a check is made to ensure that distinct types have distinct brands. Then, all global variables are initialized. And finally the main bodies of the modules are invoked. The skeletal code that ensures that every module is initialized is generated by `m3ld`(1).

Other parts of the runtime, such as threads, are actually implemented in the core library.

Thread switching is implemented with `setjmp` and `longjmp`. A timer interrupt (signal) is used to cause preemptive thread switching. The global variable `ThreadSupport.self` points to the currently running thread. The integer `ThreadSupport.inCritical` is used by the runtime to prevent thread switching during garbage collection and other "atomic" runtime operations.

7.3 Porting to another machine

Anyone who is interested in porting this compiler is encouraged! We would like to know how it goes. The primary concerns when doing a port will be the size and alignment constraints of the target machine and the runtime. We tried to avoid suspicious C constructs, but we doubt that we were completely successful.

The directions in this section are somewhat sparse. We tried to make the installation of SRC Modula-3 smooth, but it is another story to make the development of ports smooth. Please bear with us and tell us what we can do to improve this section.

General. If you want to a port to an unsupported system you should:

- install SRC Modula-3 for multiple architectures (see section 4.3). One of them should be for a supported architecture so that you can build a bootstrap compiler. In the following, let `installed/m3` be the pathname of the installed driver (not the compiler per se).
- make a clone of the source tree for the cross-compiler, say in `cross` and another one for the target compiler, say `port`. You can go in your `dist-1.5` directory and make the targets `cross` and `port` to create these clones, that will reside next to `dist-1.5`.
- establish a link to the installed version:

```
$ (cd cross/system/driver; ln -s installed/m3 m3.bootstrap)
```

The compilation process retains the usual phases. Symbols are scanned as needed by the parser. A recursive descent parser reads the entire source and builds the internal syntax tree. All remaining passes simply add decorations to this tree. The next phase binds all identifiers to values in scopes. Modula-3 allows arbitrary forward references so it is necessary to accumulate all names within a scope before binding any identifiers to values. The next phase divides the types into structurally equivalent classes. This phase actually occurs in two steps. First, the types are divided into classes such that each class will have a unique C representation. Then, those classes are refined into what Modula-3 defines as structurally equivalent types. After the types have been partitioned, the entire tree is checked for type errors. Finally, the C code is emitted. C's requirement that declarations precede uses means that the code is generated in several passes. First, the types are generated during type checking. Then, the procedure headers are produced. And finally, the procedure bodies are generated.

The compiler implementation is in the `compiler` directory. Within that directory the following directories exist:

builtinOps	`ABS`, `ADR`, `BITSIZE`, ...
builtinTypes	`INTEGER`, `CHAR`, `REFANY`, ...
builtinWord	`Word.And`, `Word.Or`, ...
exprs	`+`, `-`, `[]`, `^`, `AND`, `OR`, ...
misc	main program, scanner, symbol tables, ...
stmts	`:=`, `IF`, `TRY`, `WHILE`, ...
types	`ARRAY`, `BITS FOR`, `RECORD`, ...
values	`MODULE`, `PROCEDURE`, `VAR`, ...

7.2 A tour of the runtime

The runtime itself implements the garbage collector, Modula-3 startup code and a few miscellaneous functions. The runtime exists in the `runtime` directory.

The interface between the compiler and runtime system is embodied (and very sparsely documented) in `M3Runtime.h`, `M3Machine.h` and `M3RuntimeDecls.h`. Every C file generated by the compiler includes these files.

The allocator and garbage collector are based on Joel Bartlett's "mostly copying collector". The best description of his collector is

> Joel F. Bartlett, "Compacting garbage collection with ambiguous roots," Digital – Western Research Laboratory, WRL Research Report 88/2, February 1988.

Since that paper, we've made a few modifications to support a growing heap and to use extra information that the Modula-3 compiler generates.

Exceptions are implemented with `setjmp` and `longjmp`. The jump buffers and scope descriptors are chained together to form a stack. The head of the chain is kept in `ThreadSupport.handlers`. There is a distinct chain for each thread. When an exception is raised, the chain is searched. If a handler for the exception is found, the exception is allowed to unwind the stack, otherwise a runtime error is signaled. To unwind the stack, a `longjmp` is done to the first handler on the stack.

Chapter 7

Internals

This section contains a brief introduction to the internal structure of the compiler and runtime system. This introduction is neither comprehensive nor tutorial; it is merely intended as a stepping stone for the courageous.

7.1 A tour of the compiler

The compiler has undergone much evolution. It started as a project to build a simple and easy to maintain compiler. Somewhere along the way we decided to compile Modula-3. Much later we decided to generate C. In hindsight, Modula-3 was a good choice, C was at best mediocre.

The initial observation was that most compilers' data structures were visible and complex. This situation makes it necessary to understand a compiler in its entirety before attempting non-trivial enhancements or bug fixes. By keeping most of the compiler's primary data structures hidden behind opaque interfaces, we hoped to avoid this pitfall. So far, bugs have been easy to find. During early development, it was relatively easy to track the weekly language changes.

The compiler is decomposed by language feature rather than the more traditional compiler passes. We attempted to confine each language feature to a single module. For example, the parsing, name binding, type checking and code production for each statement is in its own module. This separation means that only the `CaseStmt` module needs to know what data structures exist to implement `CASE` statements. Other parts of the compiler need only know that the `CASE` statement is a statement. This fact is captured by the object subtype hierarchy. A `CaseStmt.T` is a subtype of a `Stmt.T`.

The main object types within the compiler are: values, statements, expressions, and types. "Values" is a misnomer; "bindings" would be better. This object class include anything that can be named: modules, procedures, variables, exceptions, constants, types, enumeration elements, record fields, methods, and procedure formals. Statements include all of the Modula-3 statements. Expressions include all the Modula-3 expression forms that have a special syntax. And finally, types include the Modula-3 types.

The compiler retains the traditional separation of input streams, scanner, symbol table, and output stream.

6.4 The data structures library

The library `data` provides Additional data structures

There is a set of interfaces to provide standard access to and operations on basic types: `Char` `Boolean`, `Cardinal`, `Integer`, `Real`, `LongReal`, `Address`, `Refany`, `Root`, and `Cast`.

A first cut a richer data structures:

`STable`	Sorted tables, implemented by 2-3-4 trees
`SIntTable`	`STable` applied to `INTEGER`
`STextTable`	`STable` applied to `Text.T`

6.5 Math

The library `math` provides mathematically oriented types and functions.

`Math`	sin, cos and friends
`Point`	2-D integral points
`Interval`	Open integral intervals
`Axis`	horizontal/vertical
`Rect`	2-D integral rectangles
`Transform`	2-D transformations
`Stat`	statistics

6.6 Miscellaneous

`misc` contains the things that don't fit elsewhere.

`Random`	Random numbers
`RandomPerm`	
`RandomReal`	
`UID`	Generate unique identifiers
`ParseParams`	Parsing of UNIX-style command lines
`ParseShell`	Lower level support
`Formatter`	Formatting of text, for example for pretty-printers

6.7 The X11R4 libraries

There are three libraries that provide a complete interface to the X11R4 system.

The library `X11` implements the `X` interface and provides Xlib-level functionalities.

The library `Xt` implements the interfaces `Xt`, `XtC`, `XtE`, `XtN`, `XtR`, `Xrm`, `Xmu` and `Xct` that provide access to the X Toolkit Intrinsics.

The library `Xaw` implements the Xaw interface that provides access to the Athena Widget set.

6.2 The Unix libraries

A set of interfaces giving access to the Unix system is under construction.

Of course, these interfaces are not the same for all the supported architectures; there is actually one library for each version of Unix. However, these libraries contain the same interfaces that define (whenever possible) the same names. Thus, it should not be more difficult to port Modula-3 programs that use these interfaces than it is to port C programs.

In general, an interface regroups the definitions given by a system include file and the related functions. Eventually, all of sections 2 and 3 should be available.

The libraries are:

`ultrix-3.1`	Ultrix 3.1, seems to work for Ultrix 4.0, SunOS 4.0 and DomainOS 10.2
`ibm-4.3`	IBM 4.3 (aka AOS 4.3)
`aix-ps2-1.2`	AIX/PS2 1.2
`aix-3.1`	AIX 3.1

Here are the interfaces implemented by these libraries:

`Utypes`	Declarations of types name (`sys/types.h`)
`Uerror`	Declarations of error codes (`errno.h`)
`Uipc`	Inter-process communication (`sys/ipc.h`)
`Umsg`	Inter-process messages (`sys/msg.h`)
`Unetdb`	Network database manipulation (`netdb.h`)
`Uprocess`	Process ids
`Uresource`	Resources utilization (`sys/resource.h`)
`Usem`	Semaphores (`sys/sem.h`)
`Ushm`	Shared memory (`sys/shm.h`)
`Usignal`	Signals (`signal.h`)
`Utime`	Time manipulation (`sys/time.h`)
`Uugid`	User and group ids
`Uutmp`	Login names (`utmp.h`)
`Unix`	Other functions (not yet organized in separate interfaces)

6.3 The input/output library

The library `io` provides Additional mechanisms for input/output.

`AutoFlushWr`	buffered writers that flush automatically

`Rd`	Basic operations on readers
`UnsafeRd`	Faster version for non-concurrent access
`RdClass`	To implement new classes of readers
`Wr`	Basic operations on writers
`UnsafeWr`	Faster version for non-concurrent access
`WrClass`	To implement new classes of writers
`TextRd`	Readers that are connected to `Text.T`s
`TextWr`	Writers that are connected to `Text.T`s
`Stdio`	Readers and writers for standard files
`FileStream`	Readers and writers connected to named files
`UFileRdWr`	Readers and writers connected to file descriptors

Fingerprints are built using polynomial arithmetic:

`FPrint`	Compute the fingerprint of a `Text.T`
`PolyBasis`	support for `FPrint`
`Poly`	support for `FPrint`

The core library is poor on data structures:

`List`	Lists of `REFANY`s
`IntTable`	Tables of `INTEGER`s

There is a set of interfaces that give access to the ANSI-C libraries. This collection is under construction.

`M3toC`	support for Modula-3/C communication
`Ctypes`	C-like names for types
`Cstdarg`	obsolete
`Cstdlib`	`stdlib.h`
`Cstring`	`string.h`

Finally, various interfaces, including the mandatory ones.

`Main`	Main program interface
`Text`	Character strings
`TextF`	Reveals to our friends what a `Text.T` is
`Thread`	Control of concurrency
`ThreadF`	Additional control for our friends
`Time`	Time manipulation
`Filename`	File names manipulation
`Word`	Bits manipulation

Chapter 6

The libraries

SRC Modula-3 includes a set of libraries, described in this chapter. It is intended that the interfaces within the library be complete and self documenting.

The library `foo` is in the file `libs/libm3foo.a`, and the interfaces that are implemented by this library are in the directory `libs/foos`, where `libs` is the directory in which the auxiliary Modula-3 software has been installed (usually `/usr/local/lib/m3`.

The key to making Modula-3 successful requires designing, building and sharing libraries. You can send us useful modules or programs and we will include them in the next release as contributed software. You can also announce the availability of your work on the SRC Modula-3 mailing list (see the previous chapter about this mailing list).

6.1 The core library

The `core` library contains some basic interfaces and modules. This library is always included when linking Modula-3 programs, and the interfaces are accessible using the default path.

Conversion of representation:

`Convert`	Basic binary/ASCII conversion of numbers
`Fmt`	Formatting to `Text.T`
`Scan`	Parsing from `Text.T`

Input/output is achieved using readers and writers:

System Research Center
Digital Equipment Corporation
130 Lytton Avenue
Palo Alto, CA 94301-1044

`m3-request@src.dec.com`

Needless to say, this implementation probably has many bugs. We are quite happy to receive bug reports. We don't promise to fix them, but we will try. When reporting a bug, please send us a short program that demonstrates the problem.

5.8.1 Count profiling

This type of profiling reports the number of times each procedure has been called and/or the number of times each statement or line of source code has been executed. UNIX provides two tools to gather the number of executions of a procedure, but no tool to measure the number of executions of a statement. However, these tools also gather time-profiling information and are somewhat problematic in our case (see the next section).

We provide an alternate set of programs that collect both kinds of data. To enable collection of data during the execution of programs, give the `-Z` option to the `m3` command. To interpret the result, run `analyze_coverage`(1).

Note that because of the extensive data collection performed by this mode of profiling, the execution time of the program can be significantly larger when collection is enabled; thus, simultaneous time profiling can produce erroneous results. For the same reason, the profiling data file is rather large; furthermore, as it is augmented by each execution of the program, you may want to compress it from time to time (see `analyze_coverage`(1) for more details).

5.8.2 Time profiling

The standard method to construct time profiles uses a UNIX timer to periodically interrupt the execution of the program and collect the current program counter. We have another use for that timer interrupt (to schedule the execution of threads). Hence, it is generally not possible to gather time profiles. But, programs that have only one thread (and thus, do not use `Thread.Fork`, either directly or indirectly via a library routine), have no scheduling to do. On such programs, using the `-pg` compilation option will gather a time profile that can be interpreted using `gprof`(1).

This situation is only temporary; we plan to offer the full time profiling capabilities standard in UNIX in a forthcoming release.

5.9 Pretty printing

SRC Modula-3 includes a pretty-printer for Modula-3 programs. It is accessible as `m3pp`(1). Read its man page to find out how to use it. (This version may not accept some of the programs that the compiler accepts, in particular because of pragmas. A feature release will fix that.)

5.10 Keeping in touch

Mail sent to `m3@src.dec.com` will be forwarded to all registered users of SRC Modula-3. (You're registered if you sent us a note asking to be added to this list or returned a signed copy of the licensing agreement.) We hope that this mailing list will provide a useful forum for the exchange of ideas, problems and solutions when using SRC Modula-3.

You can contact the SRC Modula-3 implementers at the following address:

Despite the fact that all references turn into `char*`, traced references and objects can be examined with the debugger. To print the referent of a reference or object `ref`, call `print_object (ref)`. Note that using this feature requires your debugger to be able to call procedures in the target address space. Examining untraced `REF`s requires loopholing.

5.7.3 Procedures

Modula-3 procedures are mapped as closely as possible into C procedures. Two differences exist: "large" results and nested procedures.

First, procedures that return structured values (i.e. records, arrays or sets) take an extra parameter. The last parameter is a pointer to the memory that will receive the returned result. This parameter was necessary because some C compilers return structured results by momentarily copying them into global memory. The global memory scheme works fine until it's preempted by the Modula-3 thread scheduler.

Second, nested procedures are passed an "extra" parameter. The first parameter to a nested procedure is a pointer to the local variables of the enclosing block. To call a nested procedure from the debugger, pass the address of the enclosing procedure's local variable named `frame`.

When a nested procedure is passed as parameter, the address of the corresponding C procedure and its "extra" parameter are packaged into a small closure record. The address of this record is actually passed. Any call through a formal procedure parameter first checks to see whether the parameter is a closure or not and then makes the appropriate call. Likewise, assignments of formal procedure parameters to variables perform runtime checks for closures.

`<*EXTERNAL*>` procedures have no extra parameters.

5.7.4 File names and line numbers

Due to liberal use of the `#line` mechanism of C, the Modula-3 file names and line numbers are preserved. Your debugger should give you the right names and line numbers and display the correct Modula-3 source code (if it includes facilities to display source code).

5.7.5 Threads

There is no support for debugging threads. That is, there is no mechanism to force the debugger to examine any thread other than the one currently executing. Usually you can get into another thread by setting a breakpoint that it will hit. There is no mechanism to run a single thread while keeping all others stopped.

5.8 Profiling

SRC Modula-3 provides two types of profiling: count profiling and time profiling.

names like `t182`. The Modula-3 type names are equated to their corresponding C type. The bad news is that variables are declared with the C type names. So, if you ask your debugger "what is the type of `v`?", it will most likely answer, "`t137`". But, if you ask "what is `t137`?" it will most likely give you a useful type description.

The following table indicates the C type corresponding to a Modula-3 type.

Modula-3	C
enumeration	`unsigned char`, `unsigned short` or `unsigned int` depending on the number of elements in the enumeration.
`INTEGER`	`int`
subrange	`char`, `short` or `int`, possibly `unsigned`, depending on the base type of the subrange. Subranges of enumerations are implemented by the same type as the full enumeration. Subranges of `INTEGER` are implemented by the smallest type containing the range. For example, the type `[0..255]` is mapped to `unsigned char` and `[-1000..1000]` is mapped to `short`.
`REAL`	`float`
`LONGREAL`	`double`
`ARRAY I OF T`	`struct { tT elts[n] }`, where `tT` is the C type of `T` and `n` is `NUMBER(I)`.
`ARRAY`^{n} `OF T`	`struct { char* elts; int size[n] }`, where `elts` is a pointer to the first element of the array.
`RECORD ... END`	`struct{ ... }` with the same collection of fields as the original record, except that their names are prefixed by a `_`.
`BITS n FOR T`	Usually `tT` where `tT` is the the C type of `T`. When `T` is an ordinal type and the packed type occurs within a record, it generates a C bit field.
`SET OF T`	`struct { int elts[n] }` where `n` is $\lceil$`NUMBER (T)/sizeof(int)`$\rceil$.
`REF T` `UNTRACED REF T` `OBJECT ... END`	`ADDRESS`, a `typedef` for `char*` defined in `M3RuntimeDecls.h`. Each use of a reference is cast into a pointer of the appropriate type at the point of use.
`PROCEDURE (): T`	Usually `tT *(proc)()` where `tT` is the C type of `T`. If `T` is a record or array, an extra `VAR` parameter is passed to the procedure which it uses to store the return result.

the entry code (as C), compiles it and produces the final linked program. At runtime the generated entry code initializes the program's modules.

For every symbol `X.Z` exported or imported by a module, the compiler generates an auxiliary symbol `VS_X__Z__xx`. This extra symbol, `X.Z`'s version stamp, is used to ensure that all modules linked into a single program agree on the type of `X.Z`. The `xx`'s are a string of 16 hexadecimal digits that encode the signature of `X.Z`. Modules that export `X.Z` also define its version stamp. Modules that import `X.Z` also import its version stamp. Modules that import different versions of `X.Z` will require different version stamps. But, since only a single instance of `X.Z` can be linked into the program, only a single version stamp can be defined.

5.6 Garbage Collection

A crucial fact for clients of the garbage collector to know is that *objects in the heap move*. If all references to a traced heap object are from other traced heap objects, the collector may move the referent. Hence, it is a bad idea to hash pointer values. References from the stack or untraced heap into the traced heap are never modified.

5.7 Debugging

Since an intermediate step of the Modula-3 compiler is to produce C code, you may use any debugger for which your C compiler can produce debugging information; in most cases, this means `dbx` or `gdb`.

However, this mechanism has limitations: the C compiler generates source-level information that relates the executable program to the intermediate C code, not to the Modula-3 source code. We attempted to reflect as much as possible of the source-level Modula-3 information into the intermediate C code. But there are still some shortcomings that you should know about.

5.7.1 Names

Every Modula-3 name is prefixed with an underscore. This mutation prevents name clashes between Modula-3 names, names used by the runtime system, and C reserved words.

Global names (i.e. top-level procedures, constants, types, variables, and exceptions) are also prefixed by their module's name. For example, in an interface or module named `X`, the C name of a top-level procedure `P` would be `_X__P`. Note, there is one underscore before `X` and two between `X` and `P`.

Local names (i.e. ones defined within a procedure or block statement) are simply prefixed by an underscore.

5.7.2 Types

Modula-3 is based on structural type equivalence, C is not. For this reason, the compiler maps all structurally equivalent Modula-3 types into a single C type. These C types have meaningless

The character type. The report defines `CHAR` as an enumeration of at least 256 elements that includes the ISO-Latin-1 alphabet. The report gives a lexical method for naming each of the elements of the enumeration (e.g. `'A'` or `'\012'`). Unfortunately the report does not give names to the elements of this enumeration. SRC Modula-3 defines these names:

name	value	name	value	name	value	name	value
NUL	\000	SOH	\001	STX	\002	ETX	\003
EOT	\004	ENQ	\005	ACK	\006	BEL	\007
BS	\010	HT	\011	NL	\012	VT	\013
NP	\014	CR	\015	SO	\016	SI	\017
DLE	\020	DC1	\021	DC2	\022	DC3	\023
DC4	\024	NAK	\025	SYN	\026	ETB	\027
CAN	\030	EM	\031	SUB	\032	ESC	\033
FS	\034	GS	\035	RS	\036	US	\037
SP	\040	DEL	\177				

Hence, `CHAR.NL` and `'\n'` denote the same value. Each of the alphabetic characters a-z and A-Z are also defined in the enumeration, so `CHAR.A` and `'A'` denote the same value.

Assertions. SRC Modula-3 implements an `ASSERT` pragma. The pragma may appear anywhere a statement may appear. It has the form `<*ASSERT expr *>` where `expr` is a boolean expression. At runtime the expression is evaluated. It is a checked runtime error if the result is `FALSE`.

Assertion checking can be disabled with the `-A` compiler switch.

Warnings. SRC Modula-3 generates several types of warning messages (e.g. unused variables, incomplete case statements, and use of non-standard features.) Two pragmas are available to control these warning messages: `<*NOWARN*>` and `<*UNUSED*>`. No warning messages are emitted for lines containing `<*NOWARN*>`. No "unused variable" warnings are emitted for variables declared where `VAR` is immediately preceded by `<*UNUSED*>`. The kinds of warnings emitted can also be controlled by the `-w` switch. See the `m3` man page for the full details.

Obsolete. SRC Modula-3 implements an `<*OBSOLETE*>` pragma. `<*OBSOLETE*>` may appear before any declaration (e.g. `<*OBSOLETE*> PROCEDURE P ();`). A warning is emitted in any module that references an obsolete symbol. This feature is used to warn clients of an interface that is evolving that they are using features that will disappear in the future.

5.5 Linking

SRC Modula-3 requires a special two-phase linker. The Modula-3 linker is normally invoked by `m3`, but may be invoked directly as `m3ld`. The `m3ld`(1) man page describes the arguments and options recognized by the linker.

The first phase of the linker invokes the standard UNIX linker to construct a single object containing everything required by the program except a tiny bit of entry code. The second phase generates

Inlined procedures. The compiler recognizes the `<*INLINE*>` pragma before procedure declarations. The pragma is allowed in interfaces and modules. At the moment, SRC Modula-3 ignores the pragma. Future versions will implement inline procedures.

For example:

```
INTERFACE X;
<*INLINE*> PROCEDURE P (i: INTEGER);
<*INLINE*> PROCEDURE Q ();
END X.
```

declares `X.P` and `X.Q` to be inlined procedures.

External language interface. The compiler also recognizes the `<*EXTERNAL*>` pragma. The pragma may appear immediately prior to a variable or procedure declaration. A variable or procedure that is declared to be external is assumed to be implemented outside the Modula-3 world. The names of external procedures and variables are passed through to the C compiler unchanged. The types of external variables and the formals to external procedures are assumed to be correct and are not checked against their external implementation. Standard calling conventions are used when calling external procedures.

For example:

```
INTERFACE OS;
<*EXTERNAL*> VAR errno: INTEGER;
<*EXTERNAL*> PROCEDURE exit (i: INTEGER);
END OS.
```

allows importers of `OS` to access the standard UNIX symbols `errno` and `exit` through the names `OS.errno` and `OS.exit` respectively.

If several variables are declared within a single `<*EXTERNAL*> VAR` declaration, they are all assumed to be external.

The external pragma may optionally specify a name different from the Modula-3 name. The form `<*EXTERNAL id*>` will use "id" as the external name and the form `<*EXTERNAL "s"*>` will use the text literal "s" as the external name. For example:

```
INTERFACE Xt;
  <*EXTERNAL "_XtCheckSubclassFlag" *>
  PROCEDURE CheckSubclassFlag (...);
  ...
```

defines a procedure named `Xt.CheckSubclassFlag` in Modula-3 and named `_XtCheckSubclassFlag` in the generated C.

5.3 Language restrictions

With a few exceptions, SRC Modula-3 implements the Modula-3 language as defined in the revised report of November 1989. Send a note to `m3-request@src.dec.com` to receive paper copies of the report. A PostScript version of the report is available via `ftp`; see section 4.1.

Arithmetic checking. SRC Modula-3 does not generate any special checking for arithmetic overflow or underflow. You get whatever checking your C compiler gives you. We decided that the runtime checking was too expensive in a compiler that was constrained to produce C. Future versions of this compiler that generate native machine code will perform the checking.

Packed types. Packed types are restricted. `BITS n FOR T` is treated as `T` everywhere except when applied to a field of an ordinal type in a record. In that case, the field is implemented by a *field* of width `n` in a C `struct`. Otherwise, a Modula-3 field is implemented as a *member* of a C `struct`. Consequently, the C representation of a packed field cannot span word boundaries and padding may be added by the C compiler to respect this constraint.

Stack overflow checking. SRC Modula-3 does not reliably detect thread stack overflows. Stacks are only checked for overflow on procedure entry. No checking is done on external procedures. Thread stacks are allocated in fixed size chunks. The required `Thread` interface has been augmented with the —SizedClosure— type to allow arbitrary sized stacks. The default size is given by `Thread.DefaultStackSize`,

Exception semantics. SRC Modula-3 uses C's `setjmp`/`longjmp` mechanism to unwind the stack when raising an exception. A problem can occur: assignments may appear to be undone. For example, consider

```
TRY
  i := 3;
  P ();
EXCEPT E:
  j := i;
END;
```

where `P` raises exception `E`. The compiler generates a `setjmp` at the beginning of the try statement. If the C compiler allocates variable `i` to a register, the assignment of `3` may be lost during the `longjmp` and branch that gets to the handler.

5.4 Language extensions

SRC Modula-3 implements some features not defined in the Modula-3 report.

5.2 An example

Here's a simple program composed of a main module, an imported interface and its implementation. In the file `Main.m3`:

```
MODULE Main;
IMPORT A;
BEGIN
  A.DoIt ();
END Main.
```

In the file `A.i3`:

```
INTERFACE A;
PROCEDURE DoIt ();
END A.
```

In the file `A.m3`:

```
MODULE A;
IMPORT Wr, Stdio;

PROCEDURE DoIt () =
   BEGIN
     Wr.PutText (Stdio.stdout, "Hello world.\n");
     Wr.Close (Stdio.stdout);
   END DoIt;

BEGIN
END A.
```

And finally, in `Makefile`:

```
OBJECTS = Main.mo A.io A.mo
M3FLAGS = -g

program: $(OBJECTS)
       m3 -o program $(OBJECTS)

.SUFFIXES: .m3 .mo .i3 .io
.m3.mo: ; m3 -c $(M3FLAGS) $*.m3
.i3.io: ; m3 -c $(M3FLAGS) $*.i3
```

If SRC Modula-3 is installed properly, running `make` will compile the three compilation units and link them with the standard libraries. The result will be left in the executable file named `program`.

Chapter 5

How to use the system

This section describes each of the tools in the SRC Modula-3 distribution and how to use them. Briefly, the tools include a compiler, linker, pretty printer, and a profiler.

5.1 Compiling

To compile a Modula-3 program, invoke `m3`(1). This driver is much in the style of `cc`(1); the output is an object file or an executable program, according to the options.

`m3` parses the command line and invokes the compiler and linker as required. `m3` tells the compiler where to seek imported interfaces and where to find the Modula-3 runtime library. Arguments ending in `.m3` or `.i3` are assumed to name Modula-3 source files to be compiled. Arguments ending in `.mo`, `.io` or `.o` are assumed to name object files, possibly created by other language processors, that are to be linked with the object files created by `m3`. Arguments ending in `.mc`, `.ic`, or `.c` are assumed to name C source files to be compiled. Arguments ending in `.ms`, `.is`, or `.s` are assumed to name assembly language files to be translated into object files by the assembler. Arguments starting with `-` specify compiler options. Other arguments are assumed to name library files, and are simply passed along to the linker.

The source for a module named `M` is normally in a file named `M.m3`. The source for an interface named `I` must be in a file named `I.i3`. The main program is the module that implements the interface `Main`.

There are options to compile without linking, stop compiling after producing C, emit debugger symbols, generate profiling hooks, retain intermediate files, override search paths, select non-standard executables for the various passes, and pass arguments to individual passes. For the full details, see the `m3`(1) man page.

In a source file, an occurrence of `IMPORT Mumble` causes the compiler to seek an interface named `Mumble`. The compiler will step through a sequence of directories looking for the file `Mumble.i3`. It will parse the first such file that it finds, which is expected to contain an interface named `Mumble`. If no file `Mumble.i3` exists, or if the parse fails, the compiler will generate an error. The particular sequence of directories to be searched is determined by the options passed to `m3`. See the `m3(1)` manual page for full details.

`util`	tools to build and install the system
`system`	
`  compiler`	the Modula-3 to C compiler
`  driver`	glue for the compiler passes: compile , C-compile, link, ...
`  loader`	the Modula-3 specific loader
`  runtime`	the required runtime system
`  corelib`	utility modules forming a core library
`tests`	test programs for the Modula-3 system
`libs`	additional libraries (described in chapter 6)
`tools`	
`  coverage`	a line-based profiler
`  depend`	a tool to detect if a file needs to be recompiled
`  pp`	pretty-printer
`demos`	example programs for the libraries
`contrib`	contributed tools and libraries
`doc`	source for this document and license agreement

Each directory contains a `README` that describes the contents of that directory and an `Imakefile` that rebuilds and installs the contents of that directory.

The table above shows in the "Requirements, build" column the CPU time (in minutes) needed to compile the corresponding distribution of SRC Modula-3.

6. Install SRC Modula-3:

```
$ make install
```

7. Conserve space:

```
$ make clean
```

4.3 Installing for several architectures

If you are installing for more than one system from a single file system, you can share the source files. If you are doing a port, this can be useful.

For each architecture you want to support, go to the top-level directory and:

1. Clone the distribution directory:

   `$ (cd dist-1.5; make` *system*`.obj)`

   where *system* is one of those mentioned in the table on page 7. This will create a directory named *system*`.obj` in the top level directory. The file tree rooted there is similar to the distribution tree, but plain files are symbolic links to the corresponding files in the distribution directory.

2. Go to the newly created directory:

   `$ cd` *system*`.obj`

3. Install that system, following steps 3 through 7 in the previous section.

4.4 What's in the distribution

The distribution hierarchy comprises the following directories:

10. Uncompress the file:

```
$ uncompress m3-1.5.tar.Z
```

11. Extract the files for this `tar` file:

```
$ tar xpof m3-1.5.tar
```

    The `tar` arguments specify the following options:

`x`	extract the files from the `tar` file to the current directory
`p`	restore files to their original modes
`o`	override the original ownership, this makes you the owner of the files
`f`	use the following argument `m3-1.5.tar` as the input file

    You can add the `v` option to see what is going on. This will create a subdirectory named `dist-1.5` in the top-level directory.

12. At this point you may delete the `tar` file to save space (the disk requirements indicated above assume that you do delete this file):

```
$ rm m3-1.5.tar
```

4.2 Installation procedure

If you are supporting multiple architectures from a single file system, skip to section 4.3.

1. Go to the distribution directory

```
$ cd dist-1.5
```

2. Select in the table on page 7 the name of the system you want to build (e.g. `VAX`); call that *system*.

3. Using `util/config.dist-`*system* as a model, create `util/config` to describe your system.

4. Configure the system for you machine:

    `$ make` *system*

5. Compile SRC Modula-3:

```
$ make
```

2. Create a fresh directory for the software. Path names below are relative to that directory, and it will be called the top-level directory:

```
$ mkdir m3
```

3. Change to the top-level directory:

```
$ cd m3
```

4. Open an `ftp` connection with `gatekeeper.dec.com` [16.1.0.2]; give `anonymous` for the name and your login id for the password:

```
$ ftp gatekeeper.dec.com
Connected to gatekeeper.dec.com.
...
Name (gatekeeper.dec.com):  anonymous
Password (gatekeeper.dec.com:anonymous):   your name@your machine
...
```

5. Change to the proper directory:

```
ftp> cd /pub/DEC/Modula-3
```

6. Set the transmission type to binary:

```
ftp> type binary
```

7. Get the distribution in one shot:

```
ftp> get m3-1.5.tar.Z
```

   or in pieces:

```
ftp> mget m3-1.5.tar.Z-*
```

   Other files are also available in the same directory; `Report.ps` is the Modula-3 in PostScript format. `Report{1,2,3}.ps` is the same, but in pieces.

8. Close the connection:

```
ftp> quit
```

9. Put the pieces back together if needed:

```
$ cat m3-1.5.tar.Z-* m3-1.5.tar.Z
$ rm m3-1.5.tar.Z-*
```

Chapter 4

Installation

This chapter describes how to get and install the SRC Modula-3 system.

4.1 Getting SRC Modula-3

SRC Modula-3 is distributed through anonymous `ftp` on `gatekeeper.dec.com`, in a compressed `tar` file named `m3-1.5.tar.Z`. This compressed `tar` file is 4.5 Mb. The table below indicates the resources necessary for the various installations (all of them can be made from the same distribution):

System	Requirements			Hardware	Operating system
	build (Mb)	build (min)	disk (Mb)		
`VAX`	45	30	8	VAX 8800	Ultrix 3.1
`DS3100`	45	25	9	DECstation 3100	Ultrix 3.1 & 4.0
`SPARC`				Sparcstation-1	SunOS 4.0.3
`AP3000`	25	70		Apollo DN4500	Domain/OS 10.2
`AIX386`					AIX/PS2
`IBMR2`				IBM R6000	AIX 3.1
`IBMRT`				IBM RT	IBM/4.3 (AOS 4.3)

We have been told that the `SPARC` version runs on SUN-3's as well (if compiled on a SUN-3, of course).

In the following, `$` is the shell prompt and `ftp>` is the `ftp` prompt. To get SRC Modula-3:

1. Make sure that you have enough disk space. The table above, in the column "Requirements, build", shows the amount of disk space necessary to build SRC Modula-3, while "Requirements, disk" shows the amount of disk space permanently occupied (if you remove the sources).

Version 1.0 This version is the first public release of the SRC Modula-3 system. It contains a Modula-3 compiler and runtime, a core library, a coverage analyzer, a dependency checker, a Modula-3 pretty printer, and a small test suite of Modula-3 programs. The compiler generates C as an intermediate code.

It is known to run on VAX Ultrix 3.1. We have not tested the software in any other configurations. The software may function correctly on other versions of Ultrix, and if recompiled, may even work on other machines.

Chapter 3

History

Version 1.5 supports four new architectures (`AP300`, `AIX386`, `IBMR2` and `IBMRT`). The libraries have been reorganized, and of course known bugs have been fixed. New demonstration programs and games are included.

Version 1.4 Version 1.4 is the second public release of SRC Modula-3. It uses the new features of version 1.3 and was alpha-tested by several SRC clients. This version added `<*UNUSED*>` and `<*OBSOLETE*>` pragmas, simplified coverage profiling by having the compiler directly generate the counters, reduced the number of `#line` directives in the generated C, added "map" procedures so that the garbage collector can efficiently locate global references, packed enumerations into smaller C types, and fixed several bugs.

Version 1.3 Version 1.3 was for internal use only. This version served to snapshot the massive editing that had taken place since 1.2. This version fixed the variable renaming problems, made `TEXT` a `REF ARRAY OF CHAR`, converted the text implementation to Modula-3, passed nested procedures as closures, used C initialization where possible for constants and variables, added warning messages, and fixed many bugs.

Version 1.2 Thanks to the new technology introduced in 1.1, porting the compiler to other machines is much easier. We have ported it to DECstation 3100 running Ultrix 3.1. A few bugs have been fixed. The driver processes the options `-D` and `-B` in a slightly different way.

The installation procedure is new, and we no longer furnish executables as the intermediate C files are present on the release. Because the intermediate C files vary according to the target machine, there are separate `tar` files for each of the supported machines. However, each distribution contains all of the sources; only the intermediate C files differ.

Version 1.1 This version is for internal use only. The main difference with Version 1.0 is the use of RCS and the use of `imake` rather than the standard `make`.

By signing and returning the commercial license included in this package, further rights to use and distribute SRC Modula-3 are granted. This license is in `doc/agreement.ps`.

Chapter 2

License

SRC Modula-3 is distributed under the terms of this license:

Jorge Stolfi (DEC-SRC) and **Stephen Harrison** (DEC-WSE) were very patient alpha-testers. They gave us invaluable bug reports and also translated some of DEC-SRC Modula-2+ modules to Modula-3.

Jérôme Chailloux (ILOG) developed the X interfaces while visiting DEC-SRC. We also had numerous discussions about the evolution of SRC Modula-3.

The "gatekeepers" (DEC-WRL), in particular **Paul Vixie**, helped with the distribution of SRC Modula-3.

David Goldberg (XEROX PARC) ported SRC Modula-3 to the SPARC machines.

Ray Lischner ported the system to the APOLLO machines.

Richard Orgass (IBM Rochester) ported the system to the IBM machines.

Thanks also to all the people who used SRC Modula-3 and reported bugs.

Chapter 1

Introduction

This document describes the SRC Modula-3 distribution and the terms under which it is distributed.

The distribution contains a Modula-3 compiler and runtime, a set of libraries, a coverage analyzer, a dependency checker, a Modula-3 pretty printer, and a small test suite of Modula-3 programs. The compiler generates C as an intermediate code.

This release is known to work on a variety of machines (see the table on page 7). We have not tested the software in any other configurations. It may function correctly on other versions of Ultrix or on other machines.

The compiler and runtime system was designed and implemented by Bill Kalsow and Eric Muller. Neither of us view this as a finished product. Nonetheless, we thought others might like to use it. The system should be of interest to two camps: those interested in trying out Modula-3 and those interested in compiler hacking.

Other Documents

The Modula-3 language is described in the "Modula-3 Report (revised)", by Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson, published as a DEC-SRC technical report (number 52).

Readers and writers are described in "IO Streams: Abstract Types, Real Programs", by Mark R. Brown and Greg Nelson, published a as DEC-SRC technical report (number 53).

Acknowledgments

Many people contributed to SRC Modula-3, and we would like to thank them.

We use the garbage collector developed by **Joel Bartlett** (DEC-WRL).

John Dillon (DEC-SRC) provided the C version of thread switching.

Mark R. Brown and **Greg Nelson** (DEC-SRC) designed the readers and writers interfaces.

Contents

SRC Modula-3
Version 1.5

Bill Kalsow
Eric Muller

Systems Research Center
Digital Equipment Corporation
130 Lytton Avenue
Palo Alto, CA 94301-1044

June 8, 1990