Release Notes for *GPM/CLR*: Preliminary (Jan.04)

John Gough

January 22, 2004

1 Introduction

Gardens Point Modula-2 (*gpm*) is a compiler for the Modula-2 language, it conforms closely to the ISO Modula-2 standard. The compiler was ported to a very large number of platforms between the late 1980s and the mid 1990s. Most versions produce native code on their host platform. This new version produces *ECMA Common Intermediate Language* (*CIL*), and may be executed on the *Common Language Runtime* (*CLR*). Throughout this document, when it is necessary to distinguish between versions, we refer to the new compiler as "*GPM/CLR*".

Why bother breathing new life into an old piece of code? For the best possible answer that I can think of at the moment, see Section 4.1.

Implementation Technology

The compiler and associated tools are all written in Modula-2, except for a few hundred lines of *C#* in some runtime support libraries. The compiler compiles itself on its host platform, and has been cross-compiled from both the FreeBSD and SPARC/Solaris versions of *GPM*.

The *CIL* produced by *GPM/CLR* is a textual assembly language file, which is afterward assembled by the *IL*-assembler "`ilasm`". The compiler enforces the type-safety of the source language, but the *CIL* code that is produced is not verifiable. Furthermore, the dynamically allocated data is *Unmanaged*. This means that the responsibility for allocation and deallocation of dynamic data rests with the programmer. The built-in garbage collector of the *CLR* is not used. Thus, all of the traditional issues for languages of this type: dangling pointers and memory leaks appear in this version as for previous implementations.

The implementation relies on a small number of runtime support libraries that are written in *C#*. Some of these make use of the *.NET* base class libraries, while others use *C#*'s interop facilities to interface to functions from the ANSI C runtime. In particular, the implementation of the standard *Storage* module depends on `malloc` and `free` from the ANSI C implementation in Microsoft's `msvcrt.dll` library.

The compiler is able to generate "*PInvoke*" code that calls directly from Modula-2 to functions in unmanaged dynamic link libraries. However the main runtime support and interoperation libraries were implemented in "unsafe" *C#*, as a convenient bootstrap pathway.

Requirements

GPM/CLR requires its host to have the *Microsoft .NET* software development kit installed. It has been tested against the first production release, the 2003 “Everett” release, and the preliminary alpha release of “Whidbey”, i.e. the PDC-alpha. The Everett/Visual Studio 2003 version should be treated as the reference.

As is typical for *unverifiable* code for the *CLR*, programs compiled with *GPM/CLR* are not portable between implementations. This is because of certain target dependencies built in to the front-end-determined offsets of fields in explicit layout structures. With native code versions of *GPM* these issues are solved by parameterizing the frontend with a target configuration file. We expect to use the same mechanism for *GPM/CLR*, but *this release only supports Intel iap86 targets*. It has not been tested against Rotor yet.

Completeness

The preliminary release of *GPM/CLR* correctly runs all programs in the *GPM* regression test suite, except for the coroutine tests. Couroutines, although a central feature of Modula-2, have not been implemented at this time. It seems possible that a technically correct implementation of coroutines may be possible by subverting the *.NET* thread system, but this is not a priority at this stage.

Most of the language extensions of previous versions of *GPM* have been implemented. For example, the type-safe extensible array types have been implemented (and are very heavily used in the compiler itself). The compiler does not support the exception handling facilities of the ISO Modula-2 standard, at this stage. The interprocedural precondition checking mechanisms that some previous versions of *GPM* supported are present in *GPM/CLR* but have not been activated for this first release.

2 The Quick Start

2.1 Installing the Compiler

The compiler comes with an *Install-shield* installer. The distribution installs in a directory conventionally named `GPM` and has a number of subdirectories. The structure, contents and purpose of the file hierarchy is detailed in Section 2.6. The distribution contains all of the sources for the compiler and its libraries and support tools.

2.2 Installing from the Installer

Run the `setup.exe` executable from Windows Explorer. Choose an installation path, and answer the questions. The default install path is `C:\gpm`, but in most cases a different device, such as `D:`, is a better choice.

The installer will automatically add *BaseFolder*`\bin` to the *PATH* environment variable, where *BaseFolder* is the installation base directory that you chose. The installer will also define the *CLRSYM* and *M2SYM* environment variables. The first of these is the path searched first for metadata (symbol) files when compiling a file targetting the *CLR* platform. The second is the path searched for symbol files when compiling for non-*CLR* targets, or if no *CLR*-specific file is found on the *CLRSYM* path. As noted in Section 3, for a majority of libraries the same symbol file is used for all targets,

and are found on the *M2SYM* path. The installer should also declare the environment variable *EXTENDGPM=TRUE*.

2.3 Installing from a Zip-archive

Choose a base directory, and unpack the zip archive into this directory. usually something like `D:\gpm` will be a suitable choice. The `bin` subdirectory in this base directory must be added to the *PATH* variable.

The following environment variables must be set up, assuming the typical base directory —

set `CLRSYM=D:\gpm\gpmd\clrsyms`
set `M2SYM=D:\gpm\gpmd\GPMsym;D:\gpm\gpmd\ISOsym`
set `EXTENDGPM=TRUE`
set `GPNAMES=Mixed`

There are optional environment variables that set the string table size and the global (closed) hash table size. These default to 63kB and 5987. It should not be necessary to change these values.

2.4 The Compiler

GPM/CLR may be activated from the command line and has a large number of command line switches. Figure 1 shows the usage prompt that results from typing just "**gpx**

```
D:\gpm> gpx /?
Gardens Point Modula-2 (GPM-CLR) version of <no version time>
Copyright 1996-2004 Queensland University of Technology (QUT)
Faculty of Information Technology, Gardens Point, Brisbane, AUSTRALIA

Usage: gpx [options] filename(s)
Options may be in any order, and in one or more groups
Wildcards in filenames are permitted. gpx will warn if no files found
Unix style options with prefix ’-’ are also accepted
/a  turn off assertion checks        /Bn allocate ’n’ buffer entries
/c  .NET CIL produced (same as -m)
/Cn, where n~[’0’..’8’]: rewrite CASE tables to keep density > n/8
/D  D-Code output only               /d  dangerous: turn off warnings
/f  filename used as outname         /g  add debugging information
/I  interactive mode with editor     /i  turn off index checks
/m  .NET CIL produced (same as -c)
/l  listing name.lst is created      /n  no object code produced
/N[cflpr] turn off dgen optimisation /O0 turn off all optimisations
/O1 default optimisations (= -Oc)    /O2 turn on all optimisations
/Of optimise for speed               /r  turn off range checks
/S  assembler output only            /s  turn off stack checks
/t  turn off overflow checks         /v  verbose compile messages
/V  super-verbose compile messages   /x+ language extensions turned on
/x- language extensions turned off   /X  verbose error explanations
/?  output this usage message
D:\gpm> _
```

Figure 1: Usage prompt for *GPM/CLR*

/?". Many of these switches control the behaviour of the native code generator and do not apply to *GPM/CLR*. In particular, if the **/c** (*CIL*-output) option is chosen then the following options have no effect: `/B`n`, /D, /N[cflpr], /O[012fc], /s`.

Suppose we start with the canonical "hello, world" program in Modula-2, presumed to be in a file `hello.mod`

```
MODULE Hello;
  FROM InOut IMPORT WriteString, WriteLn;
BEGIN
  WriteString("Hello GPM/CLR World!"); WriteLn
END Hello.
```

From the command line we type "**gpx /cg Hello.mod**", and the following sequence of events takes place

- the driver program `gpx.exe` is invoked with the given arguments. From the **/c** argument *GPM* knows that it is compiling for the *CIL* target. **/g** requests a debuggable program executable
- gpx spawns the compiler proper, `gpmx.exe`, which creates two output files `hello.il` and `hello.rfx`, then returns to gpx
- gpx detects successful compilation of a program module (as opposed to compilation of an implementation or library module), and spawns the assembler `ilasm` with the `/EXE, /debug` and `/quiet` flags[1]
- `ilasm` creates the files `hello.exe` and `hello.pdb`. The second of these is the program data base for the debuggers
- gpx deletes the intermediate language file, and inter-process message file. The compilation is now complete.

During the compilation of `hello.mod` the compiler will read the metadata "symbol" file for module *InOut*. If you want to watch progress of the compilation and see where `gpmx` is finding the metatdata, you should invoke the driver with the verbose (**/v**) flag, using "**gpx /cgv Hello.mod**". This will turn off `/quiet` for `ilasm` as well. Superverbose **/V** is even more chatty.

If an error occurs during the compilation, the compiler will issue one or more error messages and produce a listing `hello.lst` in this case.

You may place any number of source file names on the command line, including names with "`*`" wildcards. For a definition module, *foo*`.def`, `gpmx` will create a corresponding symbol file *foo*`.syx`. For an implementation module, *foo*`.mod`, `gpmx` will create a create a *CIL* file *foo*`.il` and a reference file *foo*`.rfx` just as for a program (base) module. However, in this case `ilasm` will be invoked with the `/DLL` flag rather than the `EXE` flag.

Running Programs

In order to run an executable program produced by *GPM/CLR*, it is necessary to have all of the libraries accessible in the directory of the base module. Even in the case of "`hello`" this will be at least the runtime support files `M2RTS.dll` and `m2iop.dll`, plus the explicitly imported `InOut.dll`. This is not an ideal situation, and can be solved by bundling together all the standard libraries into a "strongly named assembly"

[1] In the 2003 framework the /quiet flag is not sufficient to completely prevent ilasm from chattering on about what it is doing. The flag is more effective on *Whidbey*.

which is signed and deposited in the *.NET* “global assembly cache”. This strategy may be adopted in future releases.

Of course, it would also be possible to copy `hello.exe` to the “`\gpm\bin`” directory where all the dll libraries live anyway.

Other Option Tricks

If you wish to view (or even edit) the *CIL* file, you may use the **/S** flag. In this case `ilasm` is not invoked. Of course, you may invoke the assembler manually later.

In order to be able to single step through the *CIL* file line-by-line, instead of line-by-line through the source file, you should invoke gpx with the **/S** flag but without the **/g** (debugging) flag. In this case the *CIL* file does not have any embedded line numbers. Hence, when you invoke `ilasm` with the `/debug` flag the line number symbols in the pdb file will refer to the assembler source file. Neat!

Invoking the compiler with the **/I** (interactive) flag produces sometimes useful behaviour. If the compiler detects an error, instead of attempting to continue it queries the user to quit, continue, ask for more help, or go to the editor. If the editor is specfied the compiler sends a line-number message back to the driver, which spawns the editor specified by the *GPMEDITOR* environment string. The editor defaults to “vi” and pops up on the correct line[2]. When the editor quits the compiler re-spawns `gpmx`.

If you want a quick check of semantic and syntactic correctness, but not produce output files, then the **/n** (no-output) flag does the trick.

The **/x-** (language extensions off) flag turns off the recognition of language extensions. This is a portability assist for anyone who is still using any other ISO conformant Modula-2 compiler. The complementary **/x+** (language extensions on) turns the extensions on. The default state of these flags is determined by the the environment variable *EXTENDGPM*. It is strongly recommended that you choose to have the extension on by default, by setting —

EXTENDGPM=TRUE

The compiler, as distributed, has all of the modules necessary to produce the “*D-Code*” intermediate form used by native-code versions of *GPM*. The backend program `DGen` that processed this form to the various native assembly language formats is not included in this distribution. The file extension for this *IL* is “`.dcf`. This output is invoked by the **/D** option. If you use **/S** option the driver will try to spawn *DGen* to produce a “`.s`” file, and then try to spawn the assembler “`as`” to produce a “`.o`” file. *DGen* compiles under *GPM/CLR* and if this were added to the distribution in future releases it would be possible to bootstrap *GPM* from the *CLR* platform.

2.5 Other Tools

There are a number of other tools shipped with the distribution. The most important of these are *GPMake* and *GPScript*. These are two versions of the “smart make” facilities of *GPM*.

Modula-2 has the property that the import dependencies between modules may be explicitly and trivially derived from the source code of the modules. In the case of a

[2]OK, ok, so this was designed in 1987! See the full manual set to see how to use the *GPMEDITOR* variable to adapt **/I** (interactive) mode to other editors. If all else fails the setting *GPMEDITOR=notepad %* works, but *NotePad* starts on line 1.

fresh recompilation from source, it is sufficient to read the first few lines of each definition and implementation module in order to construct an "*imports graph*" of a complete application. The subgraph from the definition modules must be acyclic. The resulting complete directed graph can be topologically sorted to define a valid compilation order. This is the task that the *GPScript* tool performs.

In the case of a partial recompilation it is necessary to check the dependencies of the current source files, but also to check the dependencies of the files already compiled. The metadata for these checks is held in the reference, `*.rfx`, and symbol, `*.syx`, files. Each compiled definition file defines a cryptographic checksum on the metadata of its public interface. Every reference and symbol file notes the magic datum on each of its dependencies. The important idea is that if a definition source file is more recent than its symbol file then the source must be recompiled. However, if recompilation yields the same magic as expected by its dependents no further recompilation is necessary.

Part of the reason that this strategy works is because the metadata is held in files related to their modules by a simple naming scheme. In cases where the the executable content of modules is not held in the expected place (for example module *Storage* is actually found in the library *M2RTS.dll*) the location is noted in `storage.def` which begins —

```
FOREIGN DEFINITION MODULE Storage;
  IMPORT IMPLEMENTATION FROM "M2RTS.dll";
...
```

This information is embedded in the symbol file `Storage.syx`, which allows gpmx to produce correct *CIL*, and also stops *GPMake* fruitlessly searching for non-existant reference files. Note the different decisions made in this earlier design compared to the situation with *C#*, for example. In *C#* compiling a file that said "`using Storage`" would require the command line switch `/r:M2RTS.dll`. In Modula this location information is explicit in `Storage.def`. This strategy would not work for *C#*, since the metadata is stored in the very file that the compiler needs a hint to find!

Of course *.NET* executable files contain their own metadata, raising the question as to the necessity of the symbol and reference files in *GPM*. Certainly the reference files are unnecessary, since the equivalent data is held in the program executable files. The situation is less clear in the case of symbol files. Symbol files hold information on the public contracts of modules the executable content of which may not even have been written. In this case there is no program executable in which to embed the metadata. Furthermore, symbol files contain type information, some of which[3] cannot be represented in the common type system of *.NET*. In future we might use the *.NET* metatdata, by defining custom attributes to declare the non-standard parts. In the meanwhile *GPM/CLR* uses the same metadata files and formats as the previous, native-code versions.

GPScript

GPMake is invoked with the name of a base module as argument, together with any options intended for *GPM*. When the program is invoked it opens the base module source file, appending the `.mod` extension, if necessary. This file must be a program

[3]The length of fixed length arrays is one such missing part.

module, it is an error if the file contains either a definition module or an implementation module. The program only attempts to recompile source files in the current directory.

The program creates an "imports graph" for the program, and writes out the commands for a correctly ordered recompilation as a script file. For example, if we give the command "`gpscript /cg hello`" then a script file `Mk-Hello.bat` will be produced. This is not a very interesting example, since *Hello* only imports *InOut*, which will be found among the libraries, and is not written in Modula anyway. The same naming pattern holds for other cases, so that if we give the command "`gpscript /cg GPScript`" then the program responds —

```
Script output file is "Mk-GPScript.bat"
```

The contents of this file is shown in Figure 2. As may be seen, the definition files on

```
REM script for build of module <GPScript>
gpx -cg mkalphab.def
gpx -cg mkinout.def
gpx -cg mkscanne.def
gpx -cg mknameha.def
gpx -cg gpscript.mod
gpx -cg mkalphab.mod
gpx -cg mkinout.mod
gpx -cg mkscanne.mod
gpx -cg mknameha.mod
```

Figure 2: Script for recompilation of *GPScript*

which *gpscript* depends are all compiled first.

In may also be noted that the base name argument to *GPScript* is a *filename*, while the script name is derived from the name of the *Module* that is discovered in the file. These are usually, but not always, the same.

GPMake

GPMake is invoked with the name of the base module that needs to be built. It is optional to use the `.mod` file extension. The usage prompt is shown in Figure 3. Most of the options to *GPMake* are simply passed on to each invocation of the compiler.

```
D:\gpm> gpmake /?
#GPMake: Usage: GPMake [(-|/)acdfgIilO[012]mprStvVX?]+BaseModFileName
          -+ causes GPMake to chatter about progress
          -S (non CLR) calls build with Build -s
          -c creates CLR program executables
          -? gives this usage message
         ... other options are for GPM

D:\gpm> _
```

Figure 3: Usage prompt for *GPMake*

When the program is invoked it opens the base module source file, appending the `.mod` extension, if necessary. This file must be a program module, it is an error if the file contains either a definition module or an implementation module. The program

only attempts to recompile source files that are present in the current directory. It is assumed that files found on the library paths will be up to date.

If all is well, the program spawns a separate process *GraphBld* which constructs a flattened file representation of the imports graph of the target program. *GraphBld* attempts to locate both source and metadata files for every node of the imports graph. In the event that symbol files exist, the magic checksum of the target file is recorded in the graph. The graph representation is written to the file *baseName*`.mak`.

If *GraphBld* returns with a success indication, the driver program begins repeatedly spawning the *Decider* program. If *Decider* determines that a file needs to be recompiled, it returns an indication to *GPMake*, which spawns the compiler. Thus *Decider* and *GPM* are spawned alternately until the files are all consistent. The key idea here is that *Decider* evaluates the need for recompilation incrementally. A static evaluation of consistency might find long chains of recompilations that depend on the recompilation of a particular definition module. However, if the public interface of the module has not changed then the recompilation of the definition will yield precisely the magic number against which the dependent module have been previously compiled. In this case no further compilation will be required. This is the "domino-stopper" effect[4].

Other, Other Tools

The program *MkEnumIO* creates reader-writer modules for enumeration types in user-defined modules. For example, if a module *Bar* exports an enumeration type —

```
TYPE Colors = (Red, Green, Blue);
```

then the command "`MkEnumIO /n Bar.Colors`" will read the symbol file for *Bar* and produce two output files: `colorsIO.def` and `colorsIO.mod`. These files contain the definition and implementation of a text-writer for the *Bar.Colors* enumeration constants. The **/n** flag specifies "no input routines".

Other options to the program add input routines, with the choice of exact or case-insensitive matching. A verbose option informs the user of the exact output filenames. The implementation of the writer uses Modula-2's facilities for defining structured constants. It is based on a large constant data array, with another constant array which holds indices into the string table.

The program *Decode* decodes symbol files, unparsing them into something approximating a definition file format. The format is readable but not quite roundtrippable.

The program *MPP* is a simple macro preprocessor for Modula-2. The purpose of the tool is to allow variant code arising from the different argument-passing conventions of different operating systems, and similar variations to be factored from the source code. The `gpmake` sub-directory of the distribution has these tools, together with the preprocessor source files.

The program processes `*.mpp` files to produce `*.mod` files, and `*.dpp` files to produce `*.def` files. The tool writes a header on the output file, recording the time, and the symbol definitions that were passed to *MPP*. Most of the example files include a machine header file "`machine.mpp`" which defines symbols for all the platforms to which *GPM* was ported at one time or another.

Most of the tools in the `gpmake` have some version dependencies. The source files of the tools have been extracted using the command —

[4]Of course, it is clear that this particular form of consistency checking merely guarantees "signature-conformance". This guarantees freedom from missing method exceptions, but does nothing to solve the much more difficult versioning problem caused by *semantic* version changes.

```
mpp /Ddotnet86 *.dpp *.mpp
```

The extracted files contain the header comments similar to that shown in Figure 4.

```
(*
 * =========== macro processed output from MPP  ==========
 *
 * input file  : decider.mpp
 * time stamp  : 2004:01:08::03:51:26
 *
 * output file : decider.mod
 * created at  : 2004:01:10::17:24:51
 *
 * options ... :  /Ddotnet86
 *
 * =======================================================
 *)
```

Figure 4: Header comment in preprocessed file

2.6 Exploring the Distribution

Figure 5 shows the file hierarchy for the current distribution. In this case the distribution has been installed in a base directory named “`gpm-dist`”. There are five subdirectories.

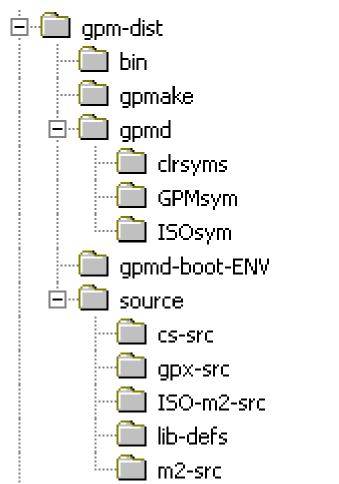

Figure 5: Folder hierarchy for the distribution

The *bin* subdirectory

The `bin` directory holds all of the *program executable* (*PE*) files for the distribution. It contains the *PE* files for the runtime system (`M2RTS.dll, m2iop.dll`) and the standard libraries. You will also find the executable files for the compiler (`gpx.exe,`

gpmx.exe) and all their associated modules (m2*.dll). Finally, all the executable content of the other tools lives here as well.

This directory must be on the executable path for the *GPM* user.

The *gpmake* subdirectory

The gpmake directory holds the sources of all the tools described above. It also contains the preprocessor files for all those tools and libraries that need to be specialized for various targets. This directory also holds the text file from which the compiler's indexed error message file is constructed, and the program (*M2ErrLst*) that does the construction.

The *gpmd* subdirectory

The gpmd directory contains all the metadata files for the libraries. There are three groups of files. The libraries are traditionally split into two groups: those defined by the ISO standard, and those unique to *GPM*. These live in the directories ISOsym and GPMsym respectively. The *GPM* specific libraries include such things as support for generic linked lists, extensible arrays, 64-bit integers and the (*GPM*-specific) convention for shortening and case-mangling of filenames.

With the standard installation the *M2SYM* environment variable specifies searching the *GPM*-specific directory first, followed by the *ISOsym* directory. However, there is one further complication for *GPM/CLR*. Foreign libraries, that is, those that are **not** implemented in Modula-2 have a line in the definition file which states where the linker will find the corresponding object code. For example, in the case *Storage* the source begins —

```
FOREIGN DEFINITION MODULE Storage;
  IMPORT IMPLEMENTATION FROM "storage.o";
...
```

For the *CLR* there is no static linking process, but when a module imports *Storage* the *compiler* needs to know the assembly in which the code is found. In such cases the compiler needs to produce *CIL* including lines like —

```
call void* [M2RTS]Storage.Storage::ALLOCATE(int32)
```

So, for *CIL* output the compiler needs to have the definition —

```
FOREIGN DEFINITION MODULE Storage;
  IMPORT IMPLEMENTATION FROM "M2RTS.dll";
...
```

In the absence of the **/c** flag the compiler needs the previous version of the definition.

The clrsyms directory contains all of the *CLR*-specific versions of the module definitions. If *GPM/CLR* is producing a *CIL* output file it searches this directory first, followed by the directories on the *M2SYM* path. If *GPM/CLR* is producing a *DCode* output file it will search the *M2SYM* path only.

The *gpmd-boot-ENV* subdirectory

The `gpmd-boot-ENV` directory is just a working directory that may be used if you wish to re-bootstrap the compiler. The procedure is as follows —

- copy all of the library *DLL*s from `..\bin` into the working directory. If you wish, copy all the files, then delete the compiler files `m2*.dll`. Do not include any of the `*.exe` files
- copy all the sources `*.def, *.mod` from the `gpx-src` subdirectory
- run the script `gpfullmake.bat`. This will compile `gpx.exe` and `gpmx.exe` as well as all the `m2*` modules
- try *GPMake /cg gpmx*. It should respond that gpmx is up to date
- run the script `gpfullmake.bat` again, and even again!

If you *change* any code in the compiler then it is not a good idea to re-bootstrap starting with the previous executable in the same directory. This is because half way through the bootstrap you will be trying to execute an application that consists of a mixture of old and new *PE* files. In such cases it is best to copy the *PE* files to another directory, and execute the compiler from there. In the case that the code is unchanged it is possible to bootstrap the compiler in a single directory. A small point to note however is that although "`gpmx /cg gpmx.mod`" will succeed, "`gpx /cg gpx.mod`" will fail. It will fail because when gpx spawns the `ilasm` process to assemble `gpx.il` the executable `gpx.exe` will be locked and creation of the output file will fail. "`gpx /cgS gpx.mod`" followed by "`ilasm /debug gpx.il`" still works, of course.

The *source* subdirectory

The `source` directory contains all the source code for the compiler and libraries. Subdirectory `cs-src` has the sources of all of the libraries written in *C#*. Most of the methods in these libraries have signatures featuring unmanaged pointers. The sources are thus full of unsafe *C#* code, and are not to be taken as examples of good practice!

Subdirectory `gpx-src` has the sources of the compiler and its driver program. There are about 3k LOC of definitions and 36k LOC of implementations in the compiler and its driver. About 9k LOC is new code written for *GPM/CLR*.

The final three directories hold the sources of the those libraries that are implemented in Modula-2.

3 Deviations

3.1 Architectural Differences

Traditional versions of *GPM* produce statically bound executables. In that case, the driver program spawns the compiler frontend. The compiler frontend (*gpmd*) produces intermediate language *DCode* and the driver program spawns the code generator *DGen*. The driver then spawns the assembler to generate an object file. As a final step, a separate program *Build* creates an imports graph, checks consistency of the magic checksums, and then issues a command to the system linker to link the various files. The code of *Build* is not present in this distribution.

For *GPM/CLR* the assembler produces a dynamic link library for each module, and there is no equivalent to the checking and linking sequence performed by *Build*. Under some circumstances where *Build* would signal an error at build-time, with *GPM/CLR* an exception would be thrown at runtime. Such errors could be avoided by using *GP-Make*, which performs equivalent consistency checks. Nevertheless, we may consider generating a separate tool for the purpose.

3.2 Mapping the Type System to the *CLR*

All primitive types in *GPM/CLR* have the same range and precision as other versions of *GPM*. The primitive types of Modula-2 are implemented by the corresponding primitive types of the *CLR*. In particular, the *INTEGER* type is implemented as the *CLR* `int32` type, and the *CARDINAL* type by `unsigned int32`. Subranges are implemented by the smallest signed or unsigned *CLR* primitive type will contain the range of values. Enumerations are limited to a maximum of 256 elements, and are implemented by the `unsigned int8` type. The non-standard *HUGEINT* type is implemented by the `int64` type.

The Modula-2 *REAL* and *LONGREAL* types are implemented as *IEEE* double, while the non-standard *SHORTREAL* type maps to *IEEE* single. Once again this is consistent with all other versions of *GPM*.

The Modula-2 *CHAR* type is problematic. The existing codebase of programs all expect that the character type will be a single, unsigned byte. Making this choice leaves the semantics of (badly written) existing programs intact, but cuts off the wonderful world of unicode. Most debuggers will not display the character values of such data.

Array types in Modula-2 are always of statically known length, and are implemented as value classes with explicitly declared size, but no declared fields. The elements of such arrays are accessed by means of pointer arithmetic.

Record types are implemented as value classes with named fields declared at explicit offsets. In the case of union types (that is, *variant records*) there will be overlapping fields. These named fields are thus able to be located by the debuggers. Nevertheless, in the current release the code generator always accesses such fields by address arithmetic. This may change in future.

Pointers to various types are implemented as (unmanaged) pointers to value objects of their bound type. There is a special case for open arrays, also called *conformant arrays*. Such arrays are passed by reference, and since the actual arrays are of unknown length the formals are declared as being of the type “pointer to array element-type”. Just like C, really.

In the case of open arrays of value mode, the copy of the actual parameter array must be made by the *caller*. This rather unusual implementation detail occurs as a necessary consequence of a dubious decision by the ISO standards committee. Array compatability for value-mode open arrays only requires that the *element types* should be assignment compatible. Thus in many cases the array must be copied element-by-element with either widening or narrowing (and range-checking) of element values. Across separately compiled module boundaries only the compiler of the *calling side* will know if such a copy is necessary.

Opaque types in Modula-2 are declared in definition parts, with no implementation detail. Such values can only be elaborated as reference types. Unfortunately, it is possible for such types to be elaborated as some already named reference type. In order to ensure proper signature matching where such values occur, all opaque types

are declared in *CIL* as having type `void*`. Nasty, yes, as it robs the *PE*-file of useful debugger information. Unfortunately there appears to be no easy way around this problem.

Finally, the procedure types (that is, *function pointers*) are implemented as generic pointer-sized values, and are declared in *CIL* as having type `void*`. The semantic rules of Modula-2 ensure that all such uses are safe, even across module boundaries. Unfortunately it is not possible to prove this to the runtime.

3.3 Why the code is not Verifiable

It does not seem possible to implement languages such as Modula-2 on the *CLR* and prove type safety to the satisfaction of the verifier. The heavy reliance on address arithmetic, and the free overlap of reference and primitive fields in union types are major obstacles.

It is possible, under some circumstances, to replace union types by suitably designed subtype hierarchies. Unfortunately, using such an implementation would necessitate the strict checking of "active variants" at runtime. Experience suggests that no Modula-2 (or indeed Pascal) compilers enforce these rules, for the simple reason that such tests would invalidate most existing user code.

4 Notes

The standard documentation that came with previous versions of *GPM* is included in the distribution. Those documents were designed with the predominantly student user group in mind. They are somewhat dated, but still a handy reference on the language. Certainly this is the place to look to explore the possible adaptation of interactive mode to more modern editors that **vi**!

The code of the compiler is common for all versions, including some historical versions that boostrapped via a C-language emitter. There are a number of contortions in the front-end that involve computing attributes that are *probably* not used by any contemporary version, and certainly not by *GPM/CLR*. Please do not ask (for example) about the code that limits the depth of prototype nesting when targetting broken ANSI C compilers. I will remove such redundant code if it ever gets to be more interesting than writing new compilers. On the other hand, I am happy to talk about the new code in `m2clr*.mod, m2cil*.mod` and so on. I will be writing more about the fascinating lessons learned from this new code-emitter "real soon now".

The wider issues of mapping particular programming language constructs to the *CLR* are dealt with in the book "*Compiling for the .NET Common Language Runtime*" by John Gough, Prentice-Hall, 2002. The running example in the book is *Gardens Point Component Pascal* (*GPCP*). This is an open source compiler for the *Component Pascal* language. For the *GPCP* compiler the code produced is fully verifiable, fully garbage collected and object oriented. Various distributions of *GPCP*, including all the source code are available from —

`http://www.citi.qut.edu.au/research/plas/projects/cp_files`

Updates of *GPM/CLR* will appear in the related URL —

`http://www.citi.qut.edu.au/research/plas/projects/`

There are other versions of *GPM* that are freeware, and are available from the same URL. However, little development of the other versions has taken place since about 1996, and none of the other versions are released with source code.

4.1 Why Bother?

The project to port *GPM* to the *CLR* was attempted for two separate reasons. There has been very little written on producing “unsafe” code for the *CLR*. The topic was very lightly skated over in “*Compiling for the .NET Common Language Runtime*”. Therefore there seemed to be a valuable learning experience on offer.

Having chosen to construct an unsafe-*CIL* compiler, the choice of starting point became a little more easy. The source code of *GPM* was pretty thoroughly debugged by many thousands of users over the ten years of its use. Furthermore, the regression test suite for the compiler contains tests of every reported bug over a ten year period, together with all of the pathological test cases that my colleagues could dream up over a number of years. It seemed logical to assume that once a new compiler could pass the test suite it would be a fairly accurate implementation of the language.

The second reason for choosing Modula-2 as a starting point was to ensure the continuing usability of a number of legacy programs. The ability to continue to use these programs on the Windows platform in now assured.