Gardens Point Component Pascal — Release Notes

John Gough

April 10, 2019

> **This document applies to GPCP version v1.4.08 for .NET**
> **(Microsoft Common Language Runtime)**

1 Introduction

Gardens Point Component Pascal (*gpcp*) is an implementation of the *Component Pascal* Language, as defined in the Component Pascal Report[1] from Oberon Microsystems. It is intended that this be a faithful implementation of the Report, except for those changes that are explicitly detailed here. Any other differences in detail should be reported as potential bugs.

The distribution consists of four programs, and a number of libraries. The programs are the compiler *gpcp*, the make utility *CPMake*, a module interface browser tool *Browse*, and a tool for extracting public symbol metadata from assemblies written in other *.NET* languages *PeToCps*.

The compiler produces either *.NET* Common Intermediate Language (*CIL*) or *Java* byte-codes as output. The compiler can be bootstrapped on either platform. These release notes refer to the Microsoft.NET platform. Current versions of the compiler require V4.5 or later of the .NET runtime.

Modules compiled with *.NET* version V2.0 up to V3.5 appear to interwork seamlessly. However earlier executables cannot work with V4.0 binaries. A separate build of *gpcp* is required for version V4.0.

There are a number of syntactic extensions to the *Component Pascal* language accepted by the compiler that have been introduced to allow interworking with the native libraries of the underlying platform. The guiding philosophy in such cases is to not significantly extend the semantics of the constructs that form part of Component Pascal, but rather to provide syntax for accessing features of other languages, which have no direct counterpart in *Component Pascal*.

---

[1]The defining document is simply referred to throughout this document as *the Report*.

2 Overall Structure

2.1 Input and Output files

In normal usage the compiler creates between two and four output files for every source file. If the file "`Hello.cp`" contains the module *Hello*, and is compiled, then the output files will be "`Hello.cps`", "`Hello.il`", and either "`Hello.dll`" or "`Hello.exe`". The "`*.cps`" file is the symbol file which contains the metadata that describes the facilities exported from the module. The "`*.il`" file contains the Common Intermediate Language (*CIL*) representation of the program. The program executable will be "`*.exe`" if the program contains an entry point (i.e. if the module imports *CPmain*), otherwise the compilation will create a dynamic link library "`*.dll`". By default, all of these files are created in the current directory. However command line options allow alternate destination for all the output files. If a listing file is created it will have filename extension "`.lst`".

Be aware that the stem name of all the output files comes from the *module* name, and not from the source-file name. Thus if the source file "`Hello.cp`" contains module *Foo* then all of the output files will have stem name "`Foo`".

It is possible to invoke the compiler so as to produce just the intermediate language file, and to then manually invoke the intermediate language assembler "`ilasm`". The assembler may then be used to produce any of its possible output formats.

2.2 Invoking the compiler

The compiler is invoked from the command line using the command syntax —

> $> gpcp [*options*] *files*

The options are given in Figure 1.

UNIX-style options "–" may also be used. In the *JVM* versions the "–"form is the expected default. Any number of files may be added in a white-space separated list.

2.3 Target choice

The compiler may choose its output language at runtime. The default output when running on the *.NET* platform is *.NET* assembler (*CIL*). The recognized options are —

>   /target=net   *this is the default* CIL *format*
>   /target=jvm   *this causes* Java *byte codes to be emitted*

The *Java* output option produces *JVM* class files directly.

Output files

Running the compiler with the `/nosym` flag causes the input files to be parsed and type-checked, but no output files are created except possibly a listing file.

If the compiler is run with the `/noasm` flag, the input files are parsed and type-checked, and a symbol file is produced for each input file. No assembly language or program executable file output is produced however.

If the compiler is run with the `/nocode` flag, the input files are parsed and type-checked, and a symbol file and one *CIL* assembly language file is produced for each input file. No executable files are produced in this case.

If the compiler is run without any flags, the input files are parsed and type-checked, and a symbol file, and a program executable (*PE*) file (either *.DLL* or *.EXE*) is produced

General Options	
/copyright	display the copyright notice
/dostats	emit timing and other statistics
/help	emit this usage prompt
/list	create an output listing if there are errors (default)
/list+	always create an output listing
/list-	never create an output listing
/quiet	make *gpcp* run silently whenever possible
/verbose	chatter on about progress during compilation
/version	emit version information
/warn-	suppress warning messages from the console
/nowarn	same as -warn-
Environment Options	
/cpsym=*XXX*	use environment variable *XXX* instead of *CPSYM*
/hsize=*N*	set hashtable size, with N (0 .. 65000)
/special	used for creating symbol files for foreign interfaces
/strict	disallow non-standard language constructs
Output Options	
/bindir=*X*	place *PE* (and pdb) files in directory "*X*"
/debug	emit debugging symbols (default)
/noasm	don't create asm (or object) files
/nocheck	don't perform arithmetic overflow checks
/nocode	create il output, but do not assemble
/nosym	produce no output files, not even a symbol file
/symdir:*X*	place symbol files in directory "*X*"
Code Generation Options	
/ilasm	use ilasm even with /nodebug
/target=*X*	emit assembler output for platform "*X*"
/unsafe	allow import of *SYSTEM* functions
/vserror	errors are in Visual Studio format
/xmlerror	errors are in *XML* format

Figure 1: *gpcp* options

for each input file. If the default /debug flag is in effect then a textual *CIL* file (extension .il) is produced and the *PE*-file is created by the ilasm tool. The default behaviour in the presence of the /nodebug option was to directly produce a *PE*-file using the *PERWAPI* component. This option is no longer available. The next release will introduce a new file writer.

Output files with "/target=jvm" option

If the compiler is run with the /target=jvm flag, the input files are parsed and type-checked, and a symbol file and one or more class files will be produced. These class files are written directly, using the legacy class file writer from *gpcp* v1.3.

If, in addition, the /nocode flag is used, then *Jasmin* assembly language (*.j) files will be produced, but *Jasmin* will not be invoked.

If, instead of /nocode the /jasmin flag is added, *Jasmin* assembly language

files are produced for each input file. The *Jasmin* assembler is no longer available so no class files are created. Thus the jasmin option is only useful for those who are curious to have a human-readable form for the JVM output.

2.4   Overflow checking

Ordinarily the compiler produces code that performs arithmetic overflow checks on all operations. Narrowing assignments (such as assigning a long value to an integer variable) are also range checked. Compiling with the `/nocheck` option removes these checks. There is a very small speed gain if checks are turned off. Checks may also be turned off on a per-procedure basis, as described in Section 4.15.

2.5   Listing output

The compiler, by default, produces a listing file only if there are compile-time errors or warnings. It is possible to force the compiler to produce a listing, using the "`/list+`" option. Equally, it is possible to prevent the creation of a listing file even if there are errors, by using the "`/list-`" option.

The listing file contains the complete listing of the program, with four digit line numbers prepended. Errors are reported in the format shown in Figure 2

```
   1 MODULE BarMod;
   2   IMPORT FooMod;
   3   TYPE
   4     Bar* = POINTER TO ABSTRACT RECORD (FooMod.Foo)
**** ----^ Only ABSTRACT basetypes can have abstract extensions
   5              i,j,k : INTEGER
   6            END;
   7 END BarMod.
```

Figure 2: Example error message

2.6   Statistics output

If the compiler is invoked with option `/dostats` then compile time statistics are produced. Figure 3 is an example, compiling the program *Browse*. The meaning of the values written to the console is as follows.

* The compiler imports symbol files in dependency order, if necessary. The maximum recursion depth for this example turned out to be 0.

* The size of the hash-table, and the number of entries used is shown

* Import time is the time to read and process metainformation for all imports. In this example module *Browse* imports much of the compiler data structures.

* Source time is the time to read the source file into the internal buffer.

* Parse time is the time to parse the buffer, create the syntax tree and resolve all identifiers.

```
E:\gpcp-CLR\work> gpcp /dostats Browse.cp
#gpcp: created Browse.exe
#gpcp: <Browse> No errors
#gpcp: net version 1.3.16 of 1 January 2013
#gpcp: 2613 source lines
#gpcp: import recursion depth 0
#gpcp: 981 entries in hashtable of size 8209
#gpcp: import time      26mSec
#gpcp: source time      34mSec
#gpcp: parse time       67mSec
#gpcp: analysis time    20mSec
#gpcp: symWrite time     4mSec
#gpcp: asmWrite time    92mSec
#gpcp: assemble time    65mSec
#gpcp: total time       308mSec
```

Figure 3: Compile statistics example

* Analysis time is the time to do type checking, and dataflow analysis.

* SymWrite time is the time to write out metatdata to the symbol file.

* AsmWrite time is the time to write out the assembly language (*CIL*) output.

* Assemble time is the time taken to spawn a new process and run `ilasm`. Assemble time is always zero if the direct to *PE*-file output path is selected by `/nodebug`.

2.7   Setting the hash table size

The compiler uses closed hashing internally, with a default number of identifiers of 8209 in the current version. It is possible to increase the number of entries by means of the `/hsize=`*NUMBER* option. Numbers up to 66000 are meaningful to the program.

If the hash table overflows, the compiler gives an error message, with a hint to increase the size. There is a example program with the distribution that creates a program that will break the compiler, so that users may test this feature. The compilation fails with "`/hsize=4000`", but succeeds with the default table size.

2.8   Choosing the Output and Symbol Directories

By default all output files are created in the current directory. This behavior may be overridden with the options `/bindir` and `/symdir`. The output symbol files are placed in the directory specified by the option `/symdir=`*target-directory*. Note carefully that if a target directory is chosen that is not on the *CPSYM* path then *gpcp* will not be able to find the symbol files automatically.

Program executable directories, and debug files in the case that debugging symbols are being created may be placed in a specified directory using the `/bindir=`*target-directory* option.

If the *JVM* target has been chosen then the `/symdir` option still applies, but `/bindir` option does not. Instead, the root of the output class file hierarchy may be specified using a syntactically similar `/clsdir` option.

By default the compiler searches for the symbol files of imported modules along the *CPSYM* path. This environment variable is set during installation, and should always begin with the current directory ".". Under some circumstances it is necessary to override the default path. For example, when compiling against the compact framework foreign system modules such as "`mscorlib`" must be placed in a different directory, since the module names clash with those of the desktop framework. If the environment variable *CPCMPCT* is set to the correct path for the compact framework, then the command line switch */cpsym=CPCMPCT* will override the default. Typical settings might be —

```
> set CPSYM=.;C:\gpcp\libs;C:\gpcp\libs\NetSystem
> set CPCMPCT=.;C:\gpcp\libs;C:\gpcp\libs\CompactSyms
```

All of the non-foreign libraries supplied with *gpcp* will work with either framework, so the "`libs`" directory would typically live on both paths.

2.9   What Libraries are in NetSystem?

For the v1.4.08 release the following system libraries are included in the *NetSystem* folder: *mscorlib.dll, System.dll, System.Configuration.dll, System.Data.dll, System-.Drawing.dll, System.Net.dll, System.Security.dll, System.Windows.Forms.dll, System-.XML.dll*. The total number of modules is 145.

2.10   The Make Utility

The compilation process with *Component Pascal* guarantees type safety across separately compiled module boundaries. Since interface meta-information resides in the symbol files which *gpcp* creates, modules must be compiled in an order that respects the partial order induced by the global importation graph. For complex programs, this may be difficult to determine manually.

The utility *CPMake* reads symbol files, and if necessary source files, in order to determine a valid order of compilation. The syntax for invocation is —

```
$> CPMake [options] moduleName
```

The module name may be given with or without a file-extension, and is usually the name of a module which imports module *CPMain* or *WinMain*, that is, the name is usually that of a *base module*. However, since version 1.3.8 the program accepts any module as the starting module and issues a warning if that is not a base module. The module name given to *CPMake* is case sensitive.

In general, when source files of a program have been modified only a subset of the modules have to be recompiled. *CPMake* is able to work out which modules must be recompiled by checking the date stamps on the files, and also checking the module hash-keys ("magic numbers") in the symbol files. If a module has been edited, but the public interface of the module has not changed a recompilation should compute a new magic number that is the same as that expected by any previously compiled, dependent modules. In this case *CPMake* detects that the dependent modules are still consistent and do not require recompilation. This "domino-stopping" feature of the program ensures that a conservative minimum of modules are recompiled.

CPMake works out the order in which to perform a compilation by constructing a directed *imports graph* of the program rooted at the given starting module. The graph will contain nodes corresponding to all the modules that are imported either directly or indirectly by the starting module. A valid order of compilation is found by performing a *topological sort* on the directed graph. The set of modules considered for re-compilation includes all the modules on which the starting module depends. If there are any modules that depend on the starting module then these will *not* be recompiled, and the program will probably fail at runtime. This is why it is normal to start with a base module. Starting *CPMake* with a non-base module therefore should be only done after careful consideration.

The options accepted by the program are exactly the options accepted by *gpcp*, except for an additional option -all. This option forces compilation of **all** modules in the local directory that are in the transitive closure of the "imports" relation with the base module, irrespective of date stamps and magic numbers. magic numbers.

> Hint:
> If you use *CPMake* to bootstrap the compiler, be aware that output file-creation will fail if the output would overwrite any file of a loaded assembly. This means that you cannot bootstrap *gpcp* using an instance of the compiler from the same directory, unless you use the "/nocode" option and then invoke ilasm manually, or use the "/bindir=*directory*" option.

2.11   Module Interface Browser

The program *Browse* reads the symbol file of a module and displays the public inter-

```
MODULE ClassMaker;
IMPORT
    RTS,
    GPCPcopyright,
    Console,
    IdDesc;
 TYPE
  Assembler* = POINTER TO ABSTRACT RECORD
                END;

  ClassEmitter* = POINTER TO ABSTRACT RECORD
                  mod* :  IdDesc.BlkId;
                END;

PROCEDURE (self:Assembler) Assemble*(),NEW,EMPTY;
PROCEDURE (self:ClassEmitter) Init*(),NEW,EMPTY;
PROCEDURE (self:ClassEmitter) Emit*(),NEW,ABSTRACT;
END ClassMaker.
```

Figure 4: Browse output from *gpcp* source file *ClassMaker.cp*

face. This public interface is shown in a form similar to a *Component Pascal* module. This "module" shows all the types, variables and procedures that are exported from the specified module. Only the exported fields of record types are shown. Any exported procedures are shown as procedure headers only. The output from *Browse* is not a proper *Component Pascal* module and will not compile using *gpcp*. It simply shows all of the identifiers that may be imported and used by a client module. Figure 4 is the html output from the command "Browse -html ClassMaker".

This program is invoked with the command —

$> Browse [*options*] *moduleName(s)*

The symbol file extension ".cps" may optionally be included in *moduleName*. As with *gpcp*, any number of files may be added in a white-space separated list. The *Browse* program sends its output to the console by default, and has the following options:

/all	// *browse this and all imported modules*
/dst:*dir*	// *place output files in directory dir*
/full	// *display full foreign names*
/file	// *write output to* ModuleName.bro
/hex	// *format integer literals in hexadecimal*
/html	// *write html output to* ModuleName.html
/sort	// *sort procedures and types in alphabetical order*
/verbose	// *chatter on about progress while processing*
/verbatim	// *display internal names of "anonymous" types*

The /all option produces output for all of the modules on the global imports graph of the specified module. The /full option is only meaningful for *FOREIGN* modules where the output from *Browse* will include the full external names for all procedures. The default for *Browse* is to only display the internal (*Component Pascal*) names. See Section 7 for more on Foreign Language Interfaces. The /file option sends the output to the file <*moduleName*>.bro instead of to the console. The /html option produces hyperlinked html text in the file <*moduleName*>.html. In the html output defining occurrences of identifiers are red and are anchored, while module names and external types are blue and hyperlinked.[2]

Browse and Foreign Libraries

Since version 1.4.07 *gpcp* has some extra decoration for html browser files for foreign libraries. In particular, these make plain the semantic differences that apply to certain imported types.

Figure 5 shows the heading of the browse file from the *.NET* foreign library *Systems.Collections*. Things to note about this screenshot fragment are the header comment, which notes the *PeToCps* or *j2cps* version that produced the symbol file, and the library containing the executable content.

Next, note the comment line immediately under the type-header. This comment states that the implementation of the type is (1) a *reference class* rather than a *struct*; (2) has a "no-arg" constructor, and hence can be allocated using the standard *Component Pascal NEW* syntax; (3) has a constructor with arguments, and hence can be called using the alternative "init(...)" syntax; (4) this pointer type cannot be dereferenced in entire assignments as would be expected for (non-extensible) *Component Pascal* record types.

---

[2] The most useful call of the program, in practice, is Browse /html /sort /hex *symFile*

```
(*
 *  Creator PeToCps  version 1.4.07 of 04 March 2018
 *  Executable code is found in mscorlib.dll
 *)
FOREIGN MODULE mscorlib_System_Collections;
    (* version 4:0:0:0 [B77A5C561934E089] *)

IMPORT
    mscorlib_System := "[mscorlib]System",
    mscorlib_System_Globalization := "[mscorlib]System.Globalization",
    mscorlib_System_Runtime_Serialization := "[mscorlib]System.Runtime.Serialization";

TYPE

  ArrayList* = POINTER TO EXTENSIBLE RECORD (mscorlib_System.Object + IList + ICollect
                  (* class | has noArg-ctor | has arg-ctor | no value-assign *)
                  (* Typebound Procedures *)
               STATIC
                 PROCEDURE Adapter*(list : IList) : ArrayList;
                 PROCEDURE FixedSize*(list : ArrayList) : ArrayList;
                 PROCEDURE FixedSize*(list : IList) : IList;
```

Figure 5: Browse page header

In order to make this clearer, each component of this comment can be clicked on to create a popup window spelling out the semantics of each attribute. Figure 6 shows

```
  DictionaryEntry* = RECORD (mscorlib_System.ValueType)
                  (* valueclass | has arg-ctor *)
                  (* Typebound Procedures *)

                      The has arg-ctor property asserts that this reference
                      class has one or more constructors that take arguments.
                      These constructors can be called from Component Pascal
                      by calling an init(...) method from the static list
                      here. Component Pascal types derived from this type
  Hashtable* = POINTE   can define constructors with arguments which call the
          (* cl         super-type constructor.
          (* Ty         See 'Interfacing to constructors' in the Release Notes.
        STATIC
            PROCEDURE init*(d            : IDictionary;
                            loadFactor : INTEGER) : Hashtable,CONSTRUCTOR;
            PROCEDURE init*(d            : IDictionary;
                            hcp          : IHashCodeProvider;
                            comparer : IComparer) : Hashtable,CONSTRUCTOR;
```

Figure 6: Browse page with Popup

one of the popups for the *DictionaryEntry* type from the same module. In this case the type is a valueclass and *can* be used in an entire assigment and passed by value. Foreign libraries written in *Java* have no equivalent semantics, although *Component Pascal* record-types *implemented* on the *JVM* do allow value-assignment.

Popups are discarded by clicking on them.

2.12 MakeIndex Utility

Version 1.4.05 includes a utility *MakeIndex* that creates an index for all of the files in HtmlBrowseFiles directory.

Usage from the command line is —

```
MakeIndex [/verb[ose]] /dst:directory
```

The program creates an index for all the html browse files in the given directory. The index file is placed in the same directory, and has name `index.html`. All of the browse files are accessible from this index.

2.13   Symbol File Generator PeToCps

This program is a symbol file generator that first appeared with version 1.3 of *gpcp*.[3] The symbol files produced by the program provide access to all the static features of the basic types, such as *System.Char*. The syntax of the symbol files produced by *PeToCps* is extended so as to allow program access to *.NET* functionality not present in *Component Pascal*. In particular, see section 7.3.

PeToCps generates symbols files corresponding to *.NET* assemblies. Taken together with the *Browse tool*, this makes the libraries of the *.NET* framework accessible to *Component Pascal* users. The *gpcp* distribution includes symbol files for the most commonly used *.NET* libraries, and their corresponding browser files.

To generate symbol files for one or more program executable libraries the usage is —

```
$> PeToCps [options] assemblyName ...
```

where current options are:

```
/big         allocate a very large symbol table
/copyright   display the gpcp copyright message
/help        display this usage message
/verbose     chatter on about progress
```

Each specified assembly will produce one or more symbol "`*.cps`" files. For example, the command —

```
$> PeToCps /big System.Windows.Forms.dll
```

creates seven symbol files. This program executable module (*PEM*) requires the large symbol-table option.

The creation of symbol files from the *mscorlib* system library requires separate handling in version 1.4.06 and higher.[4]

```
$> PeToCps [options] /mscorlib
```

where the options are:

```
/copyright   display the gpcp copyright message
/help        display this usage message
/verbose     chatter on about progress
```

For *mscorlib V4.0* the command —

```
$> PeToCps /mscorlib
```

creates 55 symbol files.

---

[3] Release 1.4.06 has a new version of *PeToCps* which uses the *System.Reflection* library to read program executable files.

[4] *System.Reflection* cannot load the *mscorlib* assembly, since every *.NET* program has this library already loaded before it begins. The previous version simply read and decoded the binary `dll` file, the new version must reflect on its already loaded *mscorlib* assembly.

2.14 Canonicalization of Names in PeToCps

PeToCps determines the names of its output symbol files from the name of the program executable module (*PEM*), and the namespaces that it contains. The program attempts to compress the names to avoid repetition of the same name stem in the assembly name and the namespace. If in doubt, running *PeToCps* with the verbose switch writes the output file names to the standard output stream.

The current version of *PeToCps* uses a canonicalization of assembly file names that avoids problems when the name contains characters that would be illegal in *Component Pascal* identifier names. This was a particular problem for users of Mono, as that system uses hyphen characters freely in assembly names. The new canonicalization of assembly names changes *ALL* non-alphanumeric characters to the lowline "_".

From version 1.3.6 it is possible to avoid using the canonicalized names for foreign module imports, by using an extension to the import list syntax as described in section 4.7.

2.15 Handling of Primitive Types by PeToCps

The common type system of *.NET* defines an abstract class associated with each primitive type. For example the abstract type *System.Char*, defined in the *mscorlib* library, is associated with the primitive Unicode character type. This abstract type provides a convenient hook on which to hang static methods of the primitive type, and also allows that the primitive type can override methods of *System.Object*.

PeToCps provides access to the static methods of these abstract types, but cannot provide access to the instance methods. Figure 7 is an example program accessing a static method of the *System.Char* class.

```
 1 MODULE CharProps;
 2   IMPORT ... ,
 3     Sys := mscorlib_System,
 4     Glb := mscorlib_System_Globalization;
 5   VAR chr : CHAR;
 6       int : INTEGER;
 7 BEGIN
 8   chr := !"\u00A5";  (* The Yen symbol *)
 9   int := Sys.Char.GetUnicodeCategory(chr);
10   ... (* int is 26, CurrencySymbol *)
 7 END CharProps.
```

Figure 7: Accessing static methods

The program imports *mscorlib_System* so that the static methods of *System.Char* are visible to the compiler. *mscorlib_System_Globalization* must also be imported so that the compiler knows that *UnicodeCategory* is an enumeration and thus values of the type are assign-compatible with the integer type. Finally, note the use of the *BangString* format3.2 to enter the Unicode character literal.

The *instance* methods of the *System.Char* class are not visible in the symbol file. An attempt to call, say, `chr.GetHashCode()` will provoke an error "`chr is not of RECORD type`".

3   Lexical Issues

3.1   Latin-8 Character Set

Versions of *gpcp* up to V1.3.4 worked correctly with input files that contained only *ASCII* characters. The current version allows any characters from *ISO* 8859-1, the latin-1 extension of *ASCII*. Eight-bit characters may now be used in identifiers as described in the *Report*.

3.2   Unicode Literal Strings

Component Pascal has a literal string format which does not allow any form of character escapes. *gpcp* (from version 1.3.12) allows literal strings to contain any 16-bit Unicode character, including embedded *NUL* characters. Strings which require this extended behaviour are declared using the exclamation point prefix —

$$BangString \quad ::- \quad \text{``!''} \text{``"''} \{ANY\ except\ \texttt{"}\} \text{``"''}.$$

Escape sequences are used to insert characters in the string format. The allowed escape sequences are —

* Any of the single character escapes: \0, \a, \b, \f, \n, \r, \", \\, with the usual *ANSI C* interpretation.

* Two-digit hexadecimal escapes in the form \xhh denoting the same character as "0hhX" in *Component Pascal*.

* Any 16-bit Unicode character may be denoted \uhhhh.

where h is any (case insensitive) hexadecimal digit. In this release only Unicode characters from plane-0 are recognized.

3.3   Non-standard Keywords

In order to provide facilities for the foreign language interface there are a total of six new keywords defined. These are all upper case names and cannot be used as program identifiers.

DIV0	an additional arithmetic operator (C integer division)
REM0	an additional arithmetic operator (C integer remainder)
EVENT	used to declare multicast delegate type for *.NET* events
RESCUE	used to mark a procedure-level exception catch block
ENUM	used in dummy foreign modules in the *.NET* system
INTERFACE	used in dummy foreign modules for defining interfaces
STATIC	used to declare static features in dummy foreign modules

Only *DIV0, REM0, EVENT* and *RESCUE* may be used in normal programs, the remainder are used in dummy foreign definition modules.

The following new predefined identifiers have been added. These can be redefined, but not at the outer lexical level. Definitions for these built-in identifiers are given below.

UBYTE	an unsigned 8-bit integer type
MKSTR	function to convert a *CP* "string" to the native string type
BOX	make a dynamically allocated copy of record or array
TYPEOF	fetch the runtime type descriptor, for reflection
USHORT	convert a value to unsigned byte, with range-check
REGISTER	attaches a procedure to a (*.NET*) multicast delegate
DEREGISTER	detaches a procedure from a multicast delegate
THROW	procedure that (re)throws a native exception object
APPEND	appends a new element to an extensible array (vector)
CUT	shortens an extensible array to the given length
LSH	performs a logical (not arithmetic) shift
ROT	performs a rotation of its argument

There are some other predefined identifiers used in the extended syntax, but these are "*context sensitive markers*" and do not prevent the same names being used for program identifiers.

> **Warning**
> Remember, if you use any of these non-standard keywords or built-in identifiers, your program source will not be portable to other implementations of *Component Pascal*.

3.4   Common Language Specification names

Fully qualified names in the Common Language Specification (*CLS*) comprise four parts.

* ⋆ Assembly name – the assembly in which the class will be found

* ⋆ Namespace name – this specifies the namespace of the class

* ⋆ Class name – the class name

* ⋆ Feature name – the field or method name.

An example might be –

```
[mscorlib]System.Exception::ToString
```

where *mscorlib* is the assembly name, *System* is the namespace, *Exception* is the class name, and *ToString* is a method name.

In this version of *gpcp*, the compiler produces one assembly per module, and one namespace per module. Both the assembly and the namespace names are the same as the module name. Thus a type-bound procedure called *isString*() bound to the type *UnaryX* in module *ExprDesc* would have the *CLS* name —

```
[ExprDesc]ExprDesc.UnaryX::isString
```

Procedures and variables at the module level are declared in the *CLS* as belonging to a synthetic "class" that contains only static data and code. This *implicit static class* has the same name as the module. Thus the module-level variable "xId" in module *Foo* will have the somewhat boring *CLS* name —

```
[Foo]Foo.Foo::xId
```

Users of the compiler should almost never have to deal with explicit *CLS* names.

If you do browse the assembler output of the compiler, you will notice that almost all names are escaped with single quotes like ′this′. This is done to avoid clashes with the many names that are reserved in the assembler.

All aspects of the default naming scheme may be overridden, if required. Such a necessity might arise if the *Component Pascal* code must interface with a framework that has particular naming patterns hardwired in. The details of the mechanisms for overriding are given in Appendix 14.

3.5 Identifier syntax

The identifier syntax for *Component Pascal* allows arbitrary use of the underscore (low-line character). There is a further extension that is specific to the foreign language interface of gpcp.

Occasionally, names that are imported from foreign modules will happen to clash with CP reserved words. In this case, we may escape the reserve word detection by starting the identifier with the back-quote character, "`". Thus, if an imported module has (say) a class with a field named "IF", then the field may be referenced as "`IF" in the source of your program. You may not *define* identifiers using this escape mechanism, except in foreign definition modules. You may however *refer* to imported identifiers using this mechanism.

It may be important to know that the back-quote is stripped at the time that the program is scanned. The presence of the escape simply suppresses the usual check for reserved identifiers that normally follows identifier scanning. Thus the back-quote is not used during any name matching of identifiers. A curious result of this strategy is that if a program escapes an identifier that does not need it, the escaped and non-escaped identifiers will refer to the same name.

4 Semantic Issues

4.1 DLLs and EXEs

The compiler can produce either stand-alone executables (.exe files) or dynamic link libraries (.dll files). Executable files must have an entry point known to the runtime. The entry point method optionally takes an array of native-strings as parameters. Any such command line arguments are accessed through the library *ProgArgs*.

If the source file contains the import of the special module name *CPmain*, then an executable file is produced as output. In this case the module body is named ".CPmain", and begins with a hidden call which saves any command line arguments so that they may be later accessed by calls to the *ProgArgs* library.

If, instead, the source file contains the import of the special module name *WinMain*, then again an executable file is produced as output. However, in this case the *PE*-file produced is a Windows executable, and the module body is named ".WinMain". Windows executables do not start a command window when launched.

If the source file does not import either *CPmain* or *WinMain* then the module body becomes the "class constructor" which is executed at the time that the dynamic link library is loaded on demand.

If the compiler is invoked with the /nocode option, then only the assembler (*CIL*) file is created. In this case the assembler ilasm may be invoked so as to create either

a .dll or an .exe file using the command "`ilasm /DLL`" or "`ilasm /EXE`". Of course, it is an error to try to create an executable file if the source does not contain an entry point.

4.2 The STA Special Module

Windows programs usually need to run in a thread with the *Single Thread Apartment* property set to true. It was previously rather unwieldy to meet this condition by accessing the facilities of the base class library. In V1.3.12 a new "dummy" module *STA* may be imported to set the property.

When a base module imports *STA* the entry point of the executable is a code wrapper. The wrapper saves the program arguments, if any, creates a new thread with the STA property set to true. The no-arg procedure that forms the payload of the thread is the code specified by the base module body. The wrapper then starts the thread.

Many Windows programs require the STA property, but do not otherwise make any mention of the the threading system. In such cases programs may import the *STA* module, and leave *gpcp* to arrange an implicit import of *System.Threading*.

It is an error to import *STA* if the target is the *JVM*. If *STA* is imported into a module which does not import either *CPmain* or *WinMain* then *STA* has not effect and a *gpcp* issues a warning.

4.3 Unimplemented constructs

There are a small number of constructs that are unimplemented or restricted in this release of the compiler. These are —

 ⋆ Module finalizers (unimplemented)

 ⋆ Procedure variables (restricted)

 ⋆ Uplevel access to reference parameters (inexact semantics)

All of these features were implemented in a prototype version of the compiler.

Module finalizers are intended to be run prior to unloading the module code. There is no facility for doing this on either of the *gpcp* target platforms.

Procedure variables are restricted in the current release. Arbitrary procedures of matching type may be assigned to procedure variables, and called in the usual way. However assignment of procedure variables is only permitted if the two sides of the assignment have the same type. That is, assignment of procedure values other than literal procedures requires *name compatibility*, rather than the *structural compatibility* specified in the language Report.

Non-local variable access is permitted in an unrestricted way since release 1.1.6. However, in the case of reference (*VAR*) parameters of unboxed[5] type that are accessed from within nested procedures the semantics of parameter passing are modified because the actual parameters are passed by *copying* rather than by *reference*. The compiler gives an explicit warning in these unusual circumstances.

---

[5] *Component Pascal* types that are unboxed in the *.NET* implementation are scalar values and record types that are not extensible, do not extend another type, and are not defined as the anonymous bound type of a pointer type.

4.4 Additional Arithmetic Operators

The usual arithmetic operators *DIV* and *MOD* in Pascal-family languages have well defined semantics that are different to the division and remainder operators of implementations of C-family languages. In *Component Pascal* the operators *DIV* and *MOD* are defined as follows —

$$i \ DIV \ j = \lfloor i/j \rfloor$$

$$(i \ DIV \ j) \times j + (i \ MOD \ j) = i$$

where i, j are integers, i/j denotes real division, and $\lfloor \ . \ \rfloor$ is the *floor* function.

Notice that *DIV* always rounds toward negative infinity unlike most C-language implementations (which normally round toward zero). The Pascal operators are mathematically preferred, but in case the alternative semantics are required for compatibility reasons, *gpcp* introduces alternatives. *DIV0* denotes integer division with rounding toward zero, while *REM0* denotes the corresponding remainder operation.

$$i \ DIV0 \ j = RTZ(i/j)$$

$$(i \ DIV0 \ j) \times j + (i \ MOD0 \ j) = i$$

where i, j are integers, i/j denotes real division, and *RTZ*(.) is the *Round-to-Zero* function.

> **Warning**
>
> Remember, if you use any of these non-standard operators your program source will not be portable to other implementations of *Component Pascal*.

4.5 Semantics of the WITH statement

The semantics of the *WITH* statement have been slightly modified so as to strengthen the guarantees on the properties of the selected variable. In the code —

```
WITH x : TypeTi DO
   ... (* guarded region *)
| x : TypeTj DO
   ... (* guarded region *)
END;
```

the variable x is asserted to have the specified type throughout the so-called *guarded region*. The base language guarantees that the type of the selected variable cannot be "widened" in the guarded region, but might possibly be narrowed. In *gpcp* if the guarded variable is a pointer type, the pointer value is treated as a constant. If the guarded variable is a record type, then assignment to the fields is allowed, but an attempt to change the variable type by an entire assignment is a semantic error.

4.6 Extensible arrays: the vector types

From version 1.3 there is direct support for extensible array types. Values of these *vector* types are dynamically allocated, and automatically extend their capacity when an append operation is performed on an array that is already full. Vectors may be declared to have any element type, and extend their length using *amortized doubling*.

In most circumstances when a linked list would otherwise have been used the vector types are faster, more memory efficient, and allow memory-safe indexing. Elements of vectors may be accessed using the familiar index syntax, with index values checked against the *active length* of the array, rather than the array *capacity*.

Declaring vector types

Vectors are declared using the new syntax —

$$Type \quad ::- \quad ... \qquad \textit{-- other type constructors}$$
$$| \quad \text{``VECTOR''} \text{``OF''} \textit{Type}.$$

Variables of vector type are not automatically allocated. They must be explicitly allocated using a variant of the built-in *NEW* procedure which specifies the initial capacity. Here is an example –

```
TYPE IntVec = VECTOR OF INTEGER;
VAR  iVec  : IntVec;
   ...
NEW(iVec, 16); (* Allocate vector with initial capacity 16 *)
```

Built-in procedures

There are two new procedures defined on the vector types. The first of these appends a new value of the declared element type to an existing vector. The signature of the procedure is —

```
PROCEDURE APPEND(v : VectorOfEType, e : EType);
```

As noted above, vectors are reference types, so that the first argument may be passed by value. The vector will double its length if there is no further space left in the array.

There is another built-in procedure which allows for the *active length* of the vector to be reduced. This has the effect of truncating the array at the given length. The signature is —

```
PROCEDURE CUT(v : VectorOfEType, i : INTEGER);
```

It is a runtime error if the requested new length of the vector is less than zero, or is greater than the current active length.

A new version of the standard built-in function *LEN* returns the active length of the vector. There is no way of querying the current capacity of a vector datum.

As noted above, a new version of the standard built-in procedure *NEW* allocates vectors of the specified initial capacity.

Assignment semantics

Vector values are references, so that an assignment of a vector value creates an alias to the original r-value. If you really do have to make a value copy, here is a coding pattern —

```
    VAR  a,b : SomeVecType;
      ...
    NEW(b, LEN(a)); (* b is barely big enough *)
    FOR i := 0 TO LEN(a)-1 DO APPEND(b, a[i]) END;
```
Note that in this case the value copy *b* will extend at the very next append operation, since its initial *capacity* is the same as the *active length* of *a*.

4.7   Extended import list syntax

The automatically constructed canonical names for foreign modules are a little unwieldy. This is unavoidable given that names in *.NET* have a qualifying assembly file name which is not necessarily a valid *Component Pascal* identifier, and a namespace name which may be a multipart dotted name. In *Component Pascal* module names are simple identifiers. This awkwardness is lessened by an extended import list syntax.

Names in import lists may declare aliases for module names, using assignment syntax. In standard *Component Pascal* the right hand side of the assignment is the module identifier. The extended syntax allows a literal string on the right, with the format—

‘"’ ‘[’ *assembly-file-name* ‘]’ *namespace-name* ‘"’

In this case the assembly file name is the file name without the .dll or .exe extension. The namespace name may be a dotted name. For example, the module corresponding to the *System.Net* namespace is found in the file "system.dll". It may be imported into a *Component Pascal* program with the line—

```
    IMPORT SysNet := "[system]System.Net";
```
gpcp parses the string, and reconstructs the canonical name that *PeToCps* will have assigned to the corresponding symbol file. Within the *Component Pascal* source, the foreign module may be referred to by the alias "*SysNet*" rather than the non-obvious "System_Net"[6]

4.8   Implementing foreign interfaces

Component Pascal types may extend classes from the *.NET CLS*. Types which extend *CLS* classes may also declare that they implement interfaces[7] from the *CLS*. The syntax extension to access this feature has *BNF* —

RecordDecl	::-	"RECORD" [*BaseType*] [*Fields*] "END" ";" .
BaseType	::-	"[" *QualifiedIdent* { "+" *QualifiedIdent* } "]" .

The first qualified identifier, as in the Report, is the class that is extended by the type being defined. Any additional qualified identifiers are the names of interfaces that the type promises to implement. The compiler checks that this contract is honored. In the case that interfaces are implemented, the base type may be left blank, or may be explicitly set to *ANYREC*.

The semantics of type-assertions are also relaxed whenever a reference is asserted to be of some interface type. For non-interface types many erroneous type-checks can be detected at compile time. However, there are almost no cases where an assertion that a dynamically typed object belongs to some interface type can be rejected at compile time.

Thus, interface types may be *used* in *Component Pascal*. However, it is not possible to *define* interface types using *gpcp*.

---

[6]Yes, the canonical name really does have a repeated lowline to mark the implicit duplication of the prefix.

[7]By "interface" in this context, we mean *fully abstract class*.

4.9 EVENT types

Event types are declared in *gpcp* with the same syntax as procedure types, but with the keyword *PROCEDURE* replaced by *EVENT*. Events are implemented as multicast delegate types in the *.NET* framework. If variables are declared to be of some event type, then it is possible to use the new built-in procedures *REGISTER* and *DEREGISTER* to register or deregister callbacks on the multicast delegate.

The usage for registering a callback is —

```
REGISTER(target-variable, callback-method);
```

The target variable is the designator of the object, which must be of some event type. The denotation of the callback method has two forms. If a *static procedure* is to be registered, then the simple procedure name is used. If the callback is intended to invoke a particular type-bound method on some particular object, then the syntax "*object.method*" is used. This works for any type-bound procedure in *Component Pascal*. The usage for deregistering a callback is syntactically identical, but using the non-standard built-in procedure *DEREGISTER* rather than *REGISTER*. A callback may be registered multiple times. The delegated calls are made in order of registration.

4.10 Unsigned Byte Type

The 8-bit type used in the *.NET* Common Language Specification (*CLS*) is an unsigned type. If *Component Pascal* is to be a full consumer of *CLS* libraries then it must be possible to declare variables and fields of such types in *Component Pascal* programs. In order to facilitate this a new built-in type *UBYTE* has been introduced in version 1.2 of *gpcp*. Values of this type may be assigned to variables of larger integral types as required. However, if values of this type are assigned to locations of the signed 8-bit type *BYTE* a runtime range-check is required. Similarly if values of any signed type are assigned to a location of unsigned byte type an explicit narrowing cast is required, using the new built-in function *USHORT*().

4.11 Runtime type descriptors

A new function since version 1.2 returns runtime type descriptors. This allows easy access to the facilities of the *system reflection* libraries. The function is overloaded, and has the following signatures —

```
PROCEDURE TYPEOF(typename): RTS.NativeType;
PROCEDURE TYPEOF(IN s : anytype) : RTS.NativeType;
```

If the target is *.NET*, then *NativeType* is an alias for *System.Type* on the underlying runtime. If the target is the *JVM*, then the return value will be *java.lang.Class*.

The procedure with the first signature takes any type name as actual parameter. The procedure with the second signature takes an actual parameter that is any variable designator. If the type of the designator is statically known (perhaps because it denotes an object of an inextensible type) then the compiler resolves the reference and no call is needed to the runtime function `System.Object::GetType()`.

4.12 Additional built-in functions

There are six additional built-in functions added to the implementation.

4.12.1 Functions MKSTR and BOX

One allows convenient access to the underlying native string object type. The signature is —

```
PROCEDURE MKSTR(IN s : ARRAY OF CHAR) : RTS.NativeString;
```

Note that it is never necessary to use MKSTR when passing a *literal* string to a formal parameter of native string type. In the literal case the compiler does the conversion for the programmer automatically.

Another handy function takes a record or array type, and makes a value copy onto the heap, returning a pointer to the copy. There is a special case version also, for *CLS* value classes. The signatures are —

```
PROCEDURE BOX(s : CP-type) : POINTER TO CP-type;
PROCEDURE BOX(s : CLS-ValCls) : System.Object;
```

Here, *CP-type* is a *Component Pascal*-defined record, array or string type. The function copies the value so that modification of the boxed value does not affect the original value. The function is particularly convenient for programs that manipulate character data implemented as dynamically allocated arrays. Thus "BOX("hello")" returns a pointer to an array of characters of length 6, while "BOX(ptr1^ + ptr2^)" performs a string concatenation and allocates a destination array of the required length. If the function is applied to an array of fixed length the return value is an open array of the same length. In the case of character arrays the use of the array "stringifier" mark "$" on the argument of *BOX* boxes a copy of the array which is truncated at the position of the "nul" character. Here is an example program fragment —

```
VAR str : ARRAY 16 OF CHAR;
    ptr : POINTER TO ARRAY OF CHAR;
  ...
str := "Hello";
ptr := BOX(str); (* ptr points to an array of length 16 *)
ptr := BOX(str$); (* ptr points to an array of length 6 *)
```

Without the *BOX* function, the construction of a value copy of an open array would require the following tedious construction —

```
VAR a,b : POINTER TO ARRAY OF CHAR;
  ...
NEW(b, LEN(a));
FOR i := 0 TO LEN(a) DO b[i] := a[i] END;
```

Using the *BOX* function, the same effect is achieved by "b := BOX(a^);".

The special case of *BOX* applies to arguments that belong to *CLS* value classes as used in the *.NET* base class libraries. In this case, for compatability with the libraries, the function returns a "boxed" copy of the value class datum. Such boxed values are treated by the system as having type *System.Object*. Boxed values may be unboxed by using the standard type-check syntax. Here is an example —

```
VAR datTim : Sys.DateTime; (* A CLS value type *)
    objRef : Sys.Object;    (* Native object type *)
  ...
objRef := BOX(Sys.DateTime.get_Now()); (* Boxing *)
  ...
datTim := objRef(Sys.DateTime);        (* UnBoxing *)
```

Note that in the final statement of the fragment the type-check unboxes the object to create a reference, and the assignment performs a *value copy* as would be expected for a value type.

4.12.2 Function USHORT

As of version 1.2 a new built-in unsigned byte type has been introduced, for conformance with the *.NET CLS*. In order to coerce values of signed type to the new type a new function *USHORT*(), analogous to the standard *SHORT*() function is also introduced. This function has the signature —

```
        PROCEDURE USHORT(s : AnyNumericType) :  UBYTE;
```

It is a runtime error if the value of the parameter is not within the unsigned byte range.

4.12.3 Function TYPEOF

The fourth new built-in function, *TYPEOF*, allows programs to access the reflection facilities of the underlying platform. The function was described in the previous section.

4.12.4 Functions LSH and ROT

The logical-shift function *LSH* has the following signatures —

```
    PROCEDURE LSH(arg : LONGINT; n : INTEGER) : LONGINT;
    PROCEDURE LSH(arg : INTEGER; n : INTEGER) : INTEGER;
```

The return value is the shifted version of the first argument, and has the same type.

The second argument specifies the number of places that the first argument is to be shifted. Positive values give a left shift. The shift value is range-checked, so that for *INTEGER* (*LONGINT*) shifts of greater than 31 (63, respectively) will return a correctly sized zero value.

The generic bitwise-rotate function *ROT* has the following signature —

```
    PROCEDURE ROT(arg : <anyint>; n : INTEGER) : <anyint>;
```

The return value is the rotated version of the first argument, and has the same type. The *arg* input may be of any integer type, including *LONGINT* or any of the shorter integer types.

The second argument specifies the number of places that the first argument is to be rotated. Positive values give a left shift. The rotation is within the bitwidth of the argument type. Thus if a contains the value 64 then *ROT* (a,2) will return 1 if a is of *BYTE* type, but returns 256 if a is of any of the larger integer types.

4.12.5 Changes for ASH

Prior to version 1.3.16 the builtin arithmetic shift function *ASH* only worked correctly for the 32-bit integer type. From the current version onward two significant changes have been made.

First, the function now has the following available signatures —

```
    PROCEDURE ASH(arg : LONGINT; n : INTEGER) : LONGINT;
    PROCEDURE ASH(arg : INTEGER; n : INTEGER) : INTEGER;
```

The return value is the shifted version of the first argument, and has the same type.

The second change is the behavior when shifts of greater than the data-word width are attempted. Previously, shift amounts were applied modulo-wordwidth, which is the usual semantic for machine instruction sets. The shift value is now range-checked, so that for *INTEGER* (*LONGINT*) shifts of magnitude greater than 31 (63, respectively) will return a correctly sized zero (if *arg* is positive OR the shift is leftward) or negative-one (if *arg* is negative AND the shift is rightward).

4.13 Deprecated features and warnings

The use of procedure variables and of super-calls are deprecated. Both attract compile-time warning messages. Warnings are also issued in the case of procedures that are not exported, and are not called (or assigned as procedure variables) within their defining module. This situation is usually an error arising from failure to mark the procedure for export.

4.14 Program executable verification

Component Pascal is a type-safe language. Every correct program is type-safe in the same sense that is guaranteed by the *.NET* virtual object system's verifier. In principle therefore, all output of *gpcp* should be verifiable.

You may test-verify the output of compilation by running the stand-alone program executable verifier "peverify" over the file. Figure 8 shows the result of running the verifier over an example module.

```
D:\gpcp-CLR\work>peverify Browse.exe
Microsoft (R) .NET Framework PE Verifier   Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001.

All Classes and Methods in Browse.exe Verified
D:\gpcp-CLR\work>_
```

Figure 8: Running peverify over an example *PE*-file

Output might fail to verify if a manually constructed interface to a library does not correspond to the internal metadata of the imported assembly. This potential problem has largely gone away with the use of *PeToCps*.

4.15 Unchecked arithmetic

By default, all arithmetic is overflow-checked, and all narrowing assignments are range checked. Sometimes it is necessary to turn off this behaviour. There are two means to do this. One of these is a custom attribute that is applied on a per-procedure basis. Checks may also be turned off from the command line for all compilations in that invocation.

The syntax of the custom attribute is a context sensitive marker that appears immediately after the keyword *BEGIN* in a procedure or module body. The syntax is —

$$Body \quad ::- \quad \text{"BEGIN" ["[UNCHECKED\_ARITHMETIC]"]}$$
$$StatementSequence \text{ "END" } identifier \text{ .}$$

An example of the use of this construct, from the source of the compiler itself, is the identifier hash function shown in Figure 9. This function performs a rotate-and-add computation, in which bits are carried out of the sign bit back into the least significant bit of the variable "tot". Overflow checking must be turned off, in order to prevent very long identifiers from crashing the compiler.

```
PROCEDURE hashStr(IN str : ARRAY OF CHAR) : INTEGER;
  VAR tot : INTEGER;
      idx : INTEGER;
      len : INTEGER;
BEGIN [UNCHECKED_ARITHMETIC]  (* Turn off overflow checks *)
  len := LEN(str$);
  tot := 0;
  FOR idx := 0 TO len-1 DO
    INC(tot, tot);
    IF tot < 0 THEN INC(tot) END;
    INC(tot, ORD(str[idx]));
  END;
  RETURN tot MOD size;
END hashStr;
```

Figure 9: Code of the hash function

Important note on parameter passing semantics for the *JVM*

The *JVM* version of *gpcp* takes liberties with the precise semantics of parameter passing almost everywhere. Actual parameters of unboxed[a] value type that are passed to reference formals are passed by copying. In the case of formal parameters of *VAR* mode, actual values of unboxed value type are copied in **and** copied out. In the case of formal parameters of *OUT* mode the value is only copied out. The current implementation method is necessary in order to obtain reasonable performance on the *JVM*. The change will not affect the results of your program unless you access the actual of a reference formal along two paths (either by having two reference formals sharing the same actual argument value, or accessing a static variable directly and through a parameter). You should not write programs that do this! You might also care to know that with this change, the performance of code is good if you have only one such copied parameter, but becomes poor if you have more than one in any frequently called procedure.

In contrast, on the *.NET* platform unboxed reference parameters are only passed inexactly if they are non-locally accessed from within a nested procedure, as described on page 15.

---

[a]Unboxed value types on the *JVM* platform are the built-in standard types such as *CHAR* and *INTEGER*, together with the pointer types. Structures and arrays are always boxed at runtime in the *JVM*, and are not affected by this semantic inexactness.

5   Exception Handling

Component Pascal does not define exception handling, but it is necessary to deal with foreign libraries that may throw exceptions. There is one new keyword and one new built-in procedure introduced to facilitate this.

5.1   The RESCUE clause

Procedures, but not modules may include exactly one *RESCUE* clause, at the end of the procedure body. This has syntax —

> *ProcBody*   ::-   "BEGIN" *Statements*
> ["RESCUE" "(" *ident* ")" *Statements*]
> "END" *ident*.

The identifier introduced in the parentheses is of type *RTS.NativeException*, and must have a name that is distinct from every other identifier in the local scope.

If any exception is thrown in the body of the procedure, or if any exception is unhandled in a procedure called from this procedure, then the rescue clause is entered with the exception object in the named local variable. This variable is read-only within the rescue clause, and is not known in the rest of the procedure body.

If the program has imported or defined any extensions of the native exception type, filtering may be performed by using the usual type-test syntaxes. The compiler will check that the rescue clause fulfills any contracts implied by the procedure signature. For example, in the case of function procedures the rescue clause must explicitly return a type-correct value, or explicitly throw another exception.

5.2   The THROW statement

Code may throw an exception by using the built-in procedure THROW. This procedure has two signatures —

```
PROCEDURE THROW(x : RTS.NativeException);
PROCEDURE THROW(x : RTS.NativeString);
```

These may be used anywhere in the program. The first is useful for rethrowing an exception from within a rescue clause. The second of these may be passed a literal string, without requiring a call of *MKSTR*() since the the compiler will automatically coerce literal strings to formals of native string type. This call will create and throw an exception object of the underlying native exception type, *System.Exception*, with the given string as embedded information. If you want to create an exception object

> **Warning**
> Remember, if you use any of these non-standard facilties for exception handling your program source will not be portable to other implementations of *Component Pascal*.

to abort program execution with a meaningful string, you may also use the library function

```
RTS.Throw(msg : ARRAY OF CHAR);
```

Exceptions thrown by this library function can be caught by a *RESCUE* clause.

6   Facilities of the CP Runtime System

6.1   Supplied libraries

This release has a small number of libraries supplied. These are —

- ⋆ *Console* writes strings and numbers to the console

- ⋆ *StdIn* reads characters and whole lines from the console

- ⋆ *Error* this library writes strings and number to the error stream

- ⋆ *ProgArgs* provides access to the command line arguments, if any

- ⋆ *GPText* a basic library for handling text formatting

- ⋆ *GPFiles* defines the supertype of *GPBinfFiles.FILE* and *GPTextFiles.FILE*

- ⋆ *GPBinFiles* reading and writing binary files

- ⋆ *GPTextFiles* reading and writing text files

- ⋆ *RealStr* formatting real numbers: based on the *ISO-Modula-2* library

- ⋆ *RTS* access to the facilities of the runtime system

- ⋆ *StringLib* string library, based on the *ISO-Modula-2* library

For the most part these libraries are the ones that were required to bootstrap the compiler.

6.2   The runtime system (RTS)

The runtime system provides a variety of low-level access facilities. The source file for this module, "RTS.cp", is not really the source. This file is a dummy foreign module, as is denoted by the context-sensitive mark *SYSTEM* appearing before the keyword *MODULE*. All such "modules" are actually implemented in the *C#* file named RTS.cs, and at runtime are found in the assembly RTS.dll. In the /jvm/ version the source of RTS.cp is spread over several *.java files.

The "source" of *RTS* is shown in Figure 10.   This figure [8] is only a fragment of the complete *API* for the runtime system library. The complete definition, RTS.cp, is found in ${*CROOT*}/sources/libs/cpascal.

The four character *defaultTarget* string will hold "net" when running on the .*NET* platform, and "jvm" when running under the Java Runtime Environment.

The word *SYSTEM* in the first line of the definition is a context sensitive mark, rather than a reserved word. This means that the word may be used as an identifier elsewhere in the program. *SYSTEM* and *FOREIGN* have slightly different semantics on the .*NET* platform, but are synonyms on the *JVM* version.

The function procedures *TypeName*, *CharAtIndex* and *Length* are new in version 1.3.12.

6.3   The ProgArgs library

The *ProgArgs* library provides access to the command line argument, if any. From *gpcp* release 1.3 it also provides access to the process environment. This is a system library, with the following public interface —

---

[8] The figure is a fragment of the version 1.4.08 source of RTS.cp. See Section 7.1.1 for a discussion of the change.

```
SYSTEM MODULE RTS;
  TYPE   CharOpen*   = POINTER TO ARRAY OF CHAR;
         CharVector* = VECTOR OF CHAR;

  TYPE
    NativeString*    = POINTER TO RECORD END;
    NativeObject*    =
      POINTER TO EXTENSIBLE RECORD
      STATIC  (* This syntax new for v1.4.08 *)
        PROCEDURE init*() : NativeObject, CONSTRUCTOR;
      END;
    NativeType*      =
      POINTER TO ABSTRACT RECORD
      STATIC
        PROCEDURE init*() : NativeType, CONSTRUCTOR;
      END;
    NativeException* =
      POINTER TO EXTENSIBLE RECORD
      STATIC
        PROCEDURE init*() : NativeException, CONSTRUCTOR;
      END;

  VAR
    defaultTarget- : ARRAY 4 OF CHAR;
    eol- : CharOpen; (* OS-dependent EOL string *)
    fltNegInfinity- : SHORTREAL;
    dblNegInfinity- : REAL;
    fltPosInfinity- : SHORTREAL;
    dblPosInfinity- : REAL;

  PROCEDURE TypeName(x : NativeType) : CharOpen;
  (* Get type name in target conventions *)

  PROCEDURE CharAtIndex(s : NativeString; i : INTEGER) : CHAR;
  (* Get char at index postion i in string s *)

  PROCEDURE Length(s : NativeString) : INTEGER;
  (* Get length of string s *)

  PROCEDURE getStr(x : NativeException) : CharOpen;
  (* Get error message from Exception x *)

  PROCEDURE StrToReal*(IN  s  : ARRAY OF CHAR;
                       OUT r  : REAL;
                       OUT ok : BOOLEAN);
  (* Parse array into an IEEE double REAL *)

                                            RTS continues ...
```

Figure 10: Source of the *RTS* pseudo-module

```
RTS continuation ...
  PROCEDURE StrToInt*(IN  s  : ARRAY OF CHAR;
                      OUT i  : INTEGER;
                      OUT ok : BOOLEAN);
  (* Parse an array into a CP INTEGER *)

  PROCEDURE StrToLong*(IN  s  : ARRAY OF CHAR;
                       OUT i  : LONGINT;
                       OUT ok : BOOLEAN);
  (* Parse an array into a CP LONGINT *)

  PROCEDURE RealToStr*(r : REAL;
                    OUT s : ARRAY OF CHAR);
  (* Decode a CP REAL into an array *)

  PROCEDURE IntToStr*(i : INTEGER;
                   OUT s : ARRAY OF CHAR);
  (* Decode a CP INTEGER into an array *)

  PROCEDURE LongToStr*(i : LONGINT;
                    OUT s : ARRAY OF CHAR);
  (* Decode a CP INTEGER into an array *)

  PROCEDURE realToLongBits*(r : REAL) : LONGINT;
  (* Convert IEEE double to a longint with same bit pattern *)

  PROCEDURE longBitsToReal*(l : LONGINT) : REAL;
  (* Convert longint to an IEEE double with same bit pattern *)

  PROCEDURE hiInt*(l : LONGINT) : INTEGER;
  (* Get hi-significant word of long integer *)

  PROCEDURE loInt*(l : LONGINT) : INTEGER;
  (* Get lo-significant word of long integer *)

  PROCEDURE Throw*(IN s : ARRAY OF CHAR);(* Abort execution *)

  PROCEDURE GetMillis*() : LONGINT;(* Get time in milliseconds *)

  PROCEDURE ClassMarker*(o : ANYPTR);(* Write class name *)

  PROCEDURE GetDateString*(OUT str : ARRAY OF CHAR);
  (* Get a date string in some native format *)
END RTS.
```

Figure 11: Source of the *RTS* pseudo-module, continued

```
SYSTEM MODULE ProgArgs;
  PROCEDURE ArgNumber*() : INTEGER;
  PROCEDURE GetArg*(num : INTEGER; OUT arg : ARRAY OF CHAR);
  PROCEDURE ExpandWildcards*(argsToSkip : INTEGER);
  PROCEDURE GetEnvVar*(IN str : ARRAY OF CHAR;
                      OUT val : ARRAY OF CHAR);
END ProgArgs.
```

The procedure *ExpandWildcards* is new in *gpcp* v1.4. It is a response to an issue which arises when a *Component Pascal* program wants use exactly the same source code on both *.NET* and *JVM* platforms. The *JVM* implementation natively expands wildcards from command-line arguments, while the *.NET* implementation does not. Previously, if *gpcp* was invoked with a command —

```
> gpcp *.cp
```

the *JVM* version would dutifully compile all the cp files in the current directory, while the *.NET* version would respond —

```
#gpcp:  cannot open local file <*.cp>
```

A call to *ExpandWildcards*(N) will expand any such wildcards, starting with the *Nth*, making the behavior the same on both platforms. The argument *argsToSkip* allows for those rare cases when the first few arguments should *not* be expanded. In any case, arguments starting with "−" or "/" will never be expanded.

ExpandWildcards rewrites the argument-list internally stored in the runtime system, changing both its length and its contents. After wildcard expansion the calls *ArgNumber* and *GetArg* will refer on the rewritten list.

Note carefully that on the *.NET* platform *GetEnvVar* fetches an environment variable, or returns an empty string. On the *JVM* platform the procedure first seeks a *Property String*, and if that fails seeks an environment variable. Property strings are passed to the underlying *Java* process at startup, using options of the form —

−D*name=value*

6.4 The RealStr library

The RealStr library is a port to *Component Pascal* of the *ISO-Modula-2* real number formatting library. The interface to the library is shown in Figure 12.

The library contains procedures to transform real number values into fixed format strings, floating format strings and the so-called "engineering" format in which exponents are always a multiple of three. For the string parser, *StrToReal*, the recognized format is given by the regular expression —

Number   ::-   ["+" | "−"] *dig* {*dig*} ["." {*dig*}] ["E" ["+" | "−"] *dig* {*dig*}] .

where *dig* denotes a decimal digit.

The *RealStr* library will exactly round trip numbers via *RealToFloat* and *StrToReal*, provided a full 17 significant figures are specified for *RealToFloat*. So far as possible the results of using module *RealStr* should be identical on the two platforms.

6.5 The StringLib library

The *StringLib* library reproduces the functionality of the *ISO Modula-2* string library, although the implementation has little similarity. The publicly accessible interface to the library is shown in Figure 13.

The library contains the expected procedures for assigning, extracting, replacing, deleting, concatenating and searching strings. As well, each of the procedures that mutates a string value has a corresponding predicate function that tests if the operation can be carried out exactly. This allows a guarded style of coding.

None of these routines raises program exceptions, but have sensible behaviour in the case that the incoming arguments do not allow correct completion. For example,

```
MODULE RealStr;

(* Ignores any leading spaces in str. If the subsequent characters in str are in the    *)
(* format of a signed real number, assigns a corresponding value to real. Argument       *)
(* res reports whether conversion was successful.                                        *)
  PROCEDURE StrToReal*(str      : ARRAY OF CHAR;
                       OUT real : REAL;
                       OUT res  : BOOLEAN);

(* Converts the value of real to floating-point string form, with sigFigs significant    *)
(* digits and copies the possibly truncated result to str.                               *)
  PROCEDURE RealToFloat*(real    : REAL;
                         sigFigs : INTEGER;
                         OUT str : ARRAY OF CHAR);

(* Converts the value of real to floating-point string form, with sigFigs significant    *)
(* digits, and copies the possibly truncated result to str. The number is scaled with one *)
(* to three whole-number digits and an exponent that is a multiple of three.             *)
  PROCEDURE RealToEng*(real    : REAL;
                       sigFigs : INTEGER;
                       OUT str : ARRAY OF CHAR);

(* Converts the value of real to fixed-point string form, rounded to the given place     *)
(* relative to the decimal point, and copies the result to str.                          *)
  PROCEDURE RealToFixed*(real  : REAL;
                         place : INTEGER; (* num. of frac. places *)
                     OUT str   : ARRAY OF CHAR);

(* Converts the value of real as RealToFixed if the sign and magnitude can be shown      *)
(* within the capacity of str, or otherwise as RealToFloat, and copies the possibly      *)
(* truncated result to str. The format is implementation-defined.                        *)
  PROCEDURE RealToStr*(real: REAL; OUT str: ARRAY OF CHAR);
END RealStr.
```

Figure 12: Interface of the RealStr library

in the case of the *Assign* procedure, if the source string is too long for the supplied destination the result is truncated to fit. Similarly, for the *Extract* procedure the length of the extracted string is the least of: (i) the requested character count, (ii) the number of characters left in the source string, and (iii) the capacity of the destination array.

6.6 The SYSTEM facilities (.NET only)

The *SYSTEM* module consists of three procedures. It must be explicitly imported, and programs that import it will only compile if the command line argument "`/unsafe`" is in effect and the target is *.NET*. Programs which use any of these facilities will be unverifiable. Furthermore, the careless use of these facilities may compromise the correctness of the garbage collector. The module is useful for diagnostic testing, but should never be used in deployed code.

The procedures are —

```
MODULE StringLib; (* from GPM module StdStrings.mod *)

  PROCEDURE CanAssignAll*(sLen : INTEGER;
                         IN dest : ARRAY OF CHAR) : BOOLEAN;
(* Check if an assignment is possible without truncation.                 *)

  PROCEDURE Assign*   (IN  src : ARRAY OF CHAR;
                       OUT dst : ARRAY OF CHAR);
(* Assign as much as possible of src to dst, with terminating nul          *)

  PROCEDURE CanExtractAll*(len : INTEGER;
                           sIx : INTEGER;
                           num : INTEGER;
                       OUT dst : ARRAY OF CHAR) : BOOLEAN;
(* Check if extraction of "num" chars starting at index sIx is possible.   *)

  PROCEDURE Extract*  (IN  src : ARRAY OF CHAR;
                           sIx : INTEGER;
                           num : INTEGER;
                       OUT dst : ARRAY OF CHAR);
(* Extract num characters starting from sIx. Result is truncated if there  *)
(* are fewer characters left, or the destination is too short.             *)

  PROCEDURE CanDeleteAll*(len,sIx,num : INTEGER) : BOOLEAN;
(* Check if num chars may be deleted starting from sIx. len is the source length  *)

  PROCEDURE Delete*(VAR str : ARRAY OF CHAR;
                        sIx : INTEGER;
                        num : INTEGER);
(* Delete num chars starting from sIx. Less are deleted if there are less num after sIx. *)

  PROCEDURE CanInsertAll*(sLen : INTEGER;
                          sIdx : INTEGER;
                      VAR dest : ARRAY OF CHAR) : BOOLEAN;
(* Check if sLen chars may be inserted into dest starting from sIdx.       *)

  PROCEDURE Insert*   (IN  src : ARRAY OF CHAR;
                           sIx : INTEGER;
                       VAR dst : ARRAY OF CHAR);
(* Insert src string into dst starting from sIx. Less chars are inserted if there is  *)
(* insufficient space in dst. dst is unchanged if sIx is beyond the end of dst.       *)

  PROCEDURE CanReplaceAll*(len : INTEGER;
                           sIx : INTEGER;
                       VAR dst : ARRAY OF CHAR) : BOOLEAN;
(* Check if len chars may be replaced in dst starting from sIx.            *)
                                              StringLib continues ...
```

Figure 13: Interface to the *StringLib* library

```
StringLib continuation ...
  PROCEDURE Replace*  (IN  src : ARRAY OF CHAR;
                            sIx : INTEGER;
                       VAR dst : ARRAY OF CHAR);
```
(∗ *Insert the characters of* src *into* dst *starting from* sIx*. Less chars are replaced if the* ∗)
(∗ *initial length of* dst *is insufficient. The string length of* dst *is unchanged.*   ∗)

```
  PROCEDURE CanAppendAll*(len : INTEGER;
                     VAR dst : ARRAY OF CHAR) : BOOLEAN;
```
(∗ *Check if* len *characters may be appended to* dst   ∗)

```
  PROCEDURE Append*(src : ARRAY OF CHAR;
               VAR dst : ARRAY OF CHAR);
```
(∗ *Append the chars of* src *string onto* dst*. Less characters are appended if the*   ∗)
(∗ *length of the destination string is insufficient.*   ∗)

```
  PROCEDURE Capitalize*(VAR str : ARRAY OF CHAR);

  PROCEDURE FindNext*    (IN  pat : ARRAY OF CHAR;
                          IN  str : ARRAY OF CHAR;
                              bIx : INTEGER; (∗ Begin index ∗)
                          OUT fnd : BOOLEAN;
                          OUT pos : INTEGER);
```
(∗ *Find the first occurrence of the pattern* pat *in* str *starting the search from* bIx   ∗)
(∗ *If no match is found* fnd *is false and* pos *is* bIx*. Empty patterns match everywhere.*   ∗)

```
  PROCEDURE FindPrev*(IN  pat : ARRAY OF CHAR;
                      IN  str : ARRAY OF CHAR;
                          bIx : INTEGER; (∗ Begin index ∗)
                      OUT fnd : BOOLEAN;
                      OUT pos : INTEGER);
```
(∗ *Find the previous occurrence of the pattern* pat *in* str *starting the search from* bIx.   ∗)
(∗ *If no match is found* fnd *is false and* pos *is* bIx*. Empty patterns match everywhere.*   ∗)

```
  PROCEDURE FindDiff*    (IN  str1 : ARRAY OF CHAR;
                          IN  str2 : ARRAY OF CHAR;
                          OUT diff : BOOLEAN;
                          OUT dPos : INTEGER);
```
(∗ *Find the index of the first char of difference between the two input strings.*   ∗)
(∗ *If the strings are identical* diff *is false, and* dPos *is zero.*   ∗)

```
END StringLib.
```

Figure 14: Interface to the *StringLib* library

```
PROCEDURE ADR(IN obj : any type) : INTEGER;
PROCEDURE GET(IN adr : INTEGER; OUT dst : any basic type);
PROCEDURE PUT(IN adr : INTEGER; IN val : any basic type);
```

There is a demonstration program named \examples\hello\testadr.cp This example demonstrates some of the capabilities of the library. Study the results, you may find them surprising. Note, for example, that *ADR*(*arr*) is not equal to *ADR*(*arr*[0]).

6.7 The StdIn library

In version 1.3 a new library is supplied that provides primitives for reading single characters and whole lines from the standard input stream. This stream is connected by default to the machine console, but may be redirected using the facilities of the underlying platform libraries.

This library has very simple functionality, described by the foreign module shown in Figure 15. In the first release the predicate function *More* always returns the *TRUE*

```
SYSTEM MODULE StdIn;
   (* Read a line of text, discarding new-line *)
   PROCEDURE ReadLn*(OUT arr : ARRAY OF CHAR);
   PROCEDURE SkipLn*();(* Discard remainer of line *)
   PROCEDURE Read*(OUT ch : CHAR);(* Fetch next character *)
   PROCEDURE More*()  : BOOLEAN;(* Return TRUE in gpcp v1.3! *)
END StdIn.
```

Figure 15: Source of the *StdIn* pseudo-module

value. The team will restore the functionality when we figure out a way of making the behaviour the same on the two execution platforms.

7 Foreign Language Interface

7.1 Accessing the underlying native types

As seen in Figure 10 the *RTS* module defines four type aliases. The binding of these types to the native platform types is determined dynamically, at compile time. Thus, the underlying types are accessible without any other import other than *RTS*. At compiler-runtime the compiler queries the target flag, or takes the default target value if there is no target command option.

If the target is "net" then *NativeObject*, *NativeString* and *NativeException* will be the *CLR* types *System.Object*, *System.String* and *System.Exception* respectively.

If the target is "jvm" then *NativeObject*, *NativeString* and *NativeException* will be the *Java* types *java.lang.Object*, *java.lang.String* and *java.lang.Exception* respectively.

In any case, literal strings may be implicitly coerced to either the native string type, or to the native object type. This saves a lot of clutter in code that interfaces to foreign libraries. However, if the value of a character array *variable* needs to be transformed to a native string, the non-standard built-in function —

```
PROCEDURE MKSTR(IN s : ARRAY OF CHAR) : RTS.NativeString;
```

must be used. See the appendix for an extended example of using these facilities for working with native string types.

7.1.1 Calling NEW() on Native Types

An ambiguity arises in the definition of the RTS.Native* types. All record types in *Component Pascal* have a no-arg constructor, which is invoked by the proper procedure "NEW(...)". However, for foreign types NEW is only valid if the foreign class has a

public no-arg constructor. For symbol files created by *PeToCps* the symbol file will record what constructors are defined in the type. For dummy definition modules such as "RTS" the situation was ambiguous.

In version 1.4.08 this ambiguity is fixed by demanding that dummy definition modules must indicate if a no-arg constructor is accessible in the foreign type. In figure 10 it will be seen that (for example) "NativeException" has a no-arg constructor, but "NativeString" does not. The syntax to allow this is new in version 1.4.08.

Note that, in figure 10 only the no-arg constructor is mentioned in RTS.cp. This is so that *gpcp* knows that it is allowed to call NEW(excVar) when *excVar* is a variable declared as *NativeException* type. This much, at least, is common for the *.NET* and *JVM* versions. For a program that is *.NET*-specific the *mscorlib* module could be imported, and a further three constructors, shown in figure 16, will be available.

```
POINTER TO EXTENSIBLE RECORD (Object + mscorlib_System_Runtime_Serialization.ISerializable + mscorlib_System
  (* class | has noArg-ctor | has arg-ctor | no value-assign *)
  (* Typebound Procedures *)
STATIC
  PROCEDURE init*(message         : String;
                  innerException : Exception) : Exception,CONSTRUCTOR;
  PROCEDURE init!(info    : mscorlib_System_Runtime_Serialization.SerializationInfo;
                  context : mscorlib_System_Runtime_Serialization.StreamingContext) : Exception,CONSTRUCTOR;
  PROCEDURE init*() : Exception,CONSTRUCTOR;
  PROCEDURE init*(message : String) : Exception,CONSTRUCTOR;
END;
```

Figure 16: The *System.Exception* API hiding behind *RTS.NativeException*

7.2   Compiling dummy definition modules

As a convenience during bootstrapping, the compiler has been enhanced so as to allow the construction of metainformation files for foreign language libraries. Such modules must be compiled with the "/special" option.

Foreign language interfaces are denoted by the context sensitive marks *FOREIGN* or *SYSTEM* preceding the keyword *MODULE* at the start of the file. Such "dummy" modules do not contain the code of the foreign language facilities, but simply define the interface to those facilities. Such modules must be compiled with the "/special" option. The system marker has special meaning in the *.NET* platform, but has the same semantics as foreign in the *JVM* platform.

When a dummy definition module is compiled there are a small number of syntactic extensions and changes.

- ⋆ Modules can be given an explicit external name

- ⋆ Procedures can be given an explicit external name

- ⋆ Features with "protected" scope may be defined

- ⋆ Static features of classes may be defined

- ⋆ Escaped identifiers may be defined

- ⋆ Interface types may be defined

- ⋆ Overloaded names may be given aliases

⋆ Constructors may be given an alias

A module declaration of the form —

```
MODULE Foo["[blah]namespace"];
```

declares that this module will be found in *.NET* assembly "`blah`" within namespace "`namespace`". It is not necessary to use this mechanism if you write the foreign module so that it has the default name as described in Section 3.4.

A procedure declaration of the form —

```
PROCEDURE (x : T)BarII*["Bar"](i,j : INTEGER);
```

declares that this type-bound procedure has the external name "`Bar`" and the internal (CP) name "`BarII`". This mechanism allows overloaded names in the *CLS* to be given non-overloaded aliases in CP.

The mark "`!`" is used to declare that a foreign name has protected scope. The mark is placed in the same position in a declaration as the standard export markers "`*`" and "`-`".

If a name clashes with a *Component Pascal* keyword, it should be defined using the back-quote escape, as described on page 14.

Here is an example of the syntax that is required to define a foreign interface type.

```
TYPE Foo* = POINTER TO INTERFACE RECORD (* always empty *) END;
```

The keyword *INTERFACE* is reserved. Such types cannot declare any instance fields in the record, nor can they define type-bound procedures which are not declared *ABSTRACT*.

Finally, constructors must be declared with the special name "`.ctor`". Declaring a constructor is not necessary if only the no-arg constructor is required, since *NEW*(*obj*) works in this case as for all other types in *Component Pascal* (see Section 8.4 for more detail). If access to constructors with arguments is required, then these may be given a *Component Pascal* alias, and are marked as constructors by using the magic explicit name. For the "`/target=jvm`" version, the magic name is "`<init>`".

7.3   Accessing Static Features of Foreign Classes

If a class has been imported from a foreign definition, and the class has static members, these may be accessed by means of a semantic extension to the designator grammar.

Normally, the syntactic construct —

QualifiedIdent {*Selector*}

is in error if the qualified identifier resolves to a type-identifier. However there are two exceptional cases where this is legal in *gpcp*. If a designator begins —

TypeIdentifier "`.`" *Identifier* ...

and the following is true —

The type identifier resolves to an imported, foreign type, **and either**
  the identifier is a static field or constant of the type, **or**
  the identifier is a static method of the named type

then this is a legal reference to the named static feature of the type.

In order to define such constructs in the syntax of dummy definitions the following productions are added to the record syntax. Note that these extensions are only recognised if the module is compiled with the "`/special`" command-line option.

Record	::-	"RECORD" ["(" *TypeId* ")"] {*FieldList*}
		["STATIC" {*StaticFeature*}] "END".
StaticFeature	::-	*ProcHeading*";" | *Constructor*";" | *StaticConst*";" | *StaticField*";" .
StaticConst	::-	*identifier* "=" *ConstExpression* .
StaticField	::-	*identifier* ":" *TypeId* .
Constructor	::-	"PROCEDURE" *IdentDef FormalPars* "," "CONSTRUCTOR".

All undefined syntactic categories in the fragment have the same meaning as in the unmodified *Component Pascal* syntax. In particular, procedure headings have the same syntax as elsewhere in the language.

7.4 Accessing Nested Classes

The *CLS* allows for class declarations to be nested within other classes. In *Component Pascal* such classes have names of the form

   EnclosingClassName$*NestedClassName*

This compound name is a single identifier, as far as *gpcp* is concerned, and must not have any embedded spaces. The *Browse tool* uses the same convention. As an example, in the foreign module "System_Windows_Forms_" there is a class that browse displays as —

```
Control$ControlCollection* =
    POINTER TO EXTENSIBLE RECORD (mscorlib_System.Object)
    STATIC
      PROCEDURE init*(p0 :  Control) :
                    Control$ControlCollection,CONSTRUCTOR;
    END;
```

In this example the nested class *ControlCollection* is enclosed by the class *Control*. Figure 17 is an example program that accesses the nested class, and creates an instance of the class. Note that the outer, enclosing object is constructed first, using the no-arg

```
MODULE Nested;
  IMPORT SWF := System_Windows_Forms_;
  VAR ct : SWF.Control;
      cc : SWF.Control$ControlCollection;
BEGIN
  NEW(ct); (* Create outer object and pass to inner constructor *)
  cc := SWF.Control$ControlCollection.init(ct);
 ...
END Nested.
```

Figure 17: Using a nested class

constructor, and then is passed as an argument to the explicit constructor for the nested class object.

8 Creating and Using Foreign Definition Modules

This Section is only of relevance if you plan to write your own foreign definition modules. For most users the information in the previous section on the usage of these facilities will be sufficient.

> Hint:
> This section is included for mainly historical reasons. The need to write foreign definition modules has significantly decreased with the availablity of the *PeToCps* and *J2CPS* tools. It is usually easier to write the foreign language code, use the tool to produce the symbol file, and *Browse* to produce a human-readable version.
> An exception occurs when the same module is required for both platforms. In that case it may still be simpler to write a foreign module, and then separately implement the code in *Java* and *C#* to match the shared definition.

8.1 Syntax of Foreign Definitions

The syntax of foreign definition is shown in Figure 18. The syntax for the declaration of record types has been extended. Unless otherwise defined here, the meanings of other syntactic-category symbols is the same as in the Component Pascal Report.

GPModule	::-	*Module* | *ForeignMod* .
ForeignMod	::-	("FOREIGN" | "SYSTEM") "MODULE" *ident* [*string*] ";"
		ImportList DeclSeq "END" *ident* "." .
DeclSeq	::-	*ConstDecls TypeDecls VarDecls*
		{ *ProcHeading* ";" *MethodHeading* ";" } .
TypeDecls	::-	[*TypeDecl* {";" *TypeDecl* }] .
TypeDecl	::-	*IdentDef* "=" *Type* .
Type	::-	*ForeignRecord*
	|	... *// Other types as for GPModule*.
ForeignRecord	::-	["POINTER" "TO"] *Attributes* "RECORD" [*SuperTypes*]
		FieldList {";" *FieldList* } *// Instance fields*
		["STATIC" *StaticDecl* {";" *StaticDecl*}] "END" .
StaticDecl	::-	*StaticField* | *StaticConst* | *StaticProc* | *Constructor* .
StaticField	::-	*IdentDef* ":" *typeId* .
StaticConst	::-	*IdentDef* "=" *constExpr* .
StaticProc	::-	"PROCEDURE" *IdentDef* [*FormalPars*] .
Constructor	::-	"PROCEDURE" *IdentDef FormalPars* "," "CONSTRUCTOR".
Attributes	::-	"ABSTRACT" | "EXTENSIBLE" | "INTERFACE" .
SuperTypes	::-	" (" [*Qualident*] {"+"*Qualident*}")" .

Figure 18: Syntax of foreign modules

The syntax begins with the context sensitive mark *FOREIGN* or *SYSTEM*. On the *.NET* platform the system marker indicates that the code will be found in the runtime

system assembly. In the *JVM*, where each class file contains a single class, the marker has the same semantic effect as the foreign marker.

8.2 Explicit package or namespace names

The way in which runtime names are generated from module names was described in Section 3.4. In the case of the *JVM* we have the following correspondence —

Component Pascal Name	JVM Name
`MODULE ModNm;`	`CP.ModNm` *// package name*
`  TYPE Cls = RECORD...END;`	`CP.ModNm.ModNm_Cls`
`  VAR varNm : Cls;`	`CP.ModNm.ModNm.varNm`
`  PROCEDURE ProcNm();`	`CP.ModNm.ModNm.ProcNm()`
`  PROCEDURE (t:Cls)MthNm();`	`CP.ModNm.Cls.MthNm()`
`END ModNm.`	

Notice that in the JVM there are no features that are defined outside of classes, so that the static features *varNm* and *ProcNm* are considered at runtime to belong to an implicit static class with the same name as the module name. However, so far as an importing *Component Pascal* program is concerned, these features will be accessed by the familiar *ModuleName.memberName* syntax.

Component Pascal Name	.NET CLS Name
`MODULE ModNm;`	`[ModNm]ModNm` *// namespace name*
`  TYPE Cls = RECORD...END;`	`[ModNm]ModNm.Cls`
`  VAR varNm :  Cls;`	`[ModNm]ModNm.ModNm::varNm`
`  PROCEDURE ProcNm();`	`[ModNm]ModNm.ModNm::ProcNm()`
`  PROCEDURE (t:Cls)MthNm();`	`[ModNm]ModNm.Cls::MthNm()`
`END ModNm.`	

In the virtual object system of *.NET* the situation is similar, with an implicit static class being defined with the same name as the module.

If, as a user, you are writing a foreign definition and plan to implement the library yourself in either *Java* or in *C#* (say), then you may define the foreign module in this way and write the foreign code so as to match the default "name mangling" scheme. In this case you may even use the same foreign definition for both versions of *gpcp*, and implement a foreign module on each underlying platform. If on the other hand you are planning to match a foreign definition to an existing library written in *Java* or *C#*, then you must override this default naming scheme.

The syntax —

"FOREIGN" "MODULE" *ident* "[" *string* "]" ";"

allows an arbitrary package or namespace name to be defined. For example, in order to access the facilities of the package `java.lang.Reflect` a foreign module might begin

```
FOREIGN MODULE java_lang_Reflect["java.lang.Reflect"];
```

Similarly, in order to access the facilities of the namespace *System.Reflect* in the assembly *mscorlib* a foreign module might begin

```
FOREIGN MODULE mscorlib_System_Reflect
                      ["[mscorlib]System.Reflect"];
```

Note that the form of the literal string is different on the two platforms, and thus any such foreign modules will be specific to a particular platform. Notice also that there is no mechanism to explicitly give a name to an implicit static class.

8.3 Dealing with overloaded names

Each of the underlying platforms allows name overloading for methods. This feature is deliberately not permitted in *Component Pascal*. Nevertheless, it is necessary to gain access to library methods that have overloaded names. The option of using explicit external method names facilitates this. Suppose we have two methods, both of which are named *Add*(), one with a single integer parameter, and the other with two. We might define these as follows in a foreign definition.

```
PROCEDURE (this : Cls)AddI*["Add"](I : INTEGER),NEW;
PROCEDURE (this : Cls)AddII*["Add"](I,J : INTEGER),NEW;
```

Within the importing *Component Pascal* program the two names are distinct, but the program executable will correctly refer to the underlying overloaded methods. This manually specified name-mangling is rather awkward, particularly in the case of parameters of object types.

Since *gpcp* release 1.1 users are able to access the unmangled names of overloaded foreign methods directly. The *PeToCps* and *J2CPS* tools create symbol files that have overloaded names, and the compiler will match calls to the intended method. Because this is a language extension, the compiler is strict about matching calls to methods in the presence of automatic type coercions. If more than one method matches when taking into account all legal, implicit coercions then *gpcp* will proceed as follows —

- ⋆ *gpcp* will check if one of the methods has formal parameter types that *exactly* match the arguments without using type coercion.

- ⋆ if this fails, *gpcp* will check if one of the methods matches using only the conversion of the *Component Pascal* string type (array of char) to the native string type, and the *Component Pascal ANYREC* and *ANYPTR* to the native object type.

- ⋆ if this fails, an error is notified listing the ambiguous candidate methods.

These rules allow most ambiguities to by resolved by the programmer using an explicit conversion of the actual parameters to the exact type of the formal parameters of the desired method.

8.4 Interfacing to constructors

If a foreign class has a "no-arg" constructor, then this will be implicitly called whenever an object is created by the use of the standard procedure *NEW*. However if it is necessary to access constructors with arguments, then it is possible to define an alias for the constructor in a foreign module. In every case the constructor will be accessed by means of a static, value returning function that returns an object of the constructed class. The fact that this is a constructor *must* be made known to *gpcp* since the way in which these methods are called differs from other methods. On each underlying platform there is a "magic" name that is used for calling a constructor. On the *JVM* the name is "<init>", while on .*NET* the name is ".ctor". These two strings are used as the explicit string that defines such a procedure in the foreign definition. An example of an interface to a constructor with arguments, in the syntax used by the *Browse* tool, might be —

```
PROCEDURE Init*(width,height : INTEGER) : Rect,CONSTRUCTOR;
```

The identifier "CONSTRUCTOR" is not a reserved word, but a context sensitive mark that may be used as an ordinary identifier elsewhere in the program.

Note that this declaration would normally appear in the static part of the record defining the class *Rect*. Calls to this procedure in a *Component Pascal* program, such as —

```
rec1 := F.Rect.Init(25,17);
```

would, depending on the target platform, translate into a call to one or the other of —

namespaceName.Rect::.ctor(int32,int32)
packageName.Rect.<init>(II)

Of course, if you extend a foreign class that does not have a public no-arg constructor, then you will not be able to construct values of your own type using *NEW*, since this implicitly calls the no-arg constructor of its super-type. In this case, it is necessary to define a new constructor signature for your extended type. From *gpcp* release 1.2 there are two ways to do this. If the desired constructor has the same signature as the constructor of the supertype, then the first method may be used. In the case of the example above, the required syntax is shown in the following fragment —

```
TYPE MyRect* = POINTER TO RECORD (Mod.Rect) ... END;
   ...
PROCEDURE Init*(w,h : INTEGER) : MyRect,CONSTRUCTOR;
```

The constructor does not define a code body, and simply passes its arguments to the super-type constructor with matching signature.

The new syntax in *gpcp* version 1.2 is considerably more flexible. The *Component Pascal* constructor is not required to have the same signature as the constructor of the super-type. An example of the syntax defining another constructor for the extended type defined above is —

```
PROCEDURE MkMyRect*(Formals) : MyRect,BASE(actuals);
  (* Local-declarations *)
BEGIN
  (* Constructor body code *)
  RETURN SELF;
END MkMyRect;
```

in the code the formal and actual parameter lists have been left un-elaborated.

The identifier "BASE" is a not a reserved word, but is a context sensitive mark. Of all publicly available constructors for the super-type it specifies a call of the one with signature matching the types of the "*actuals*" argument list. This super-type constructor will be called as the first action of the constructor, before the new fields of the derived object are initialized. Within the body of the constructor the object under construction is denoted by the identifier "SELF". The constructor *must* return this object along every terminating path of the body. It is an error if the actual parameter expression types in the *BASE* super-call do not choose a unique super-type constructor.

8.5   Declaring static features of classes

Classes in foreign modules may be declared either as records or as pointers to records. However, it is recommended that on the *JVM* platform the pointer form be always used, as a helpful reminder to the user that at runtime the objects will be dynamically allocated. On the *.NET* platform value classes should be declared as plain records, with no explicit base type. On both platforms array types should be declared as pointers to arrays, again reminding the user that all arrays are dynamically (and explicitly) allocated.

In order to access static features of foreign classes, the syntax extension of records given in Figure 18 must be used. In the optional static section of a record declaration we may define constants, static fields and static (i.e. non type-bound) procedures.

We may consider the following example —

CP Foreign Definition	Component Pascal Usage	
`FOREIGN MODULE ModNm;`		
`  TYPE Cls =`	`ModNm.Cls`	(* *class name* *)
`      POINTER TO RECORD`		
`      STATIC`		
`        statVar* :  CHAR;`	`ModNm.Cls.statVar`	
`        PROCEDURE StatProc();`	`ModNm.Cls.StatProc()`	
`      END;`		
`END ModNm.`		

In this example we select the static member by qualifying the designator by the type-name of the class.

Type-bound methods will be defined lexically outside of the record declaration in the normal *Component Pascal* way, remembering that only the heading is required. On the *.NET* platform the distinction between virtual and instance methods is made automatically. Instance methods are *NEW* but not *EXTENSIBLE*. On the *JVM* platform the possibility of optimizing the calls to such methods is left to the *JIT* to determine.

Note that the foreign modules which arise from *C#* on the *.NET* platform or are written in *Java* can never have static features outside of classes. If you are writing the foreign module yourself you may use the default class naming scheme described in Section 3.4. However if you are matching an existing package, you will need to use the explicit name override described earlier in this Section. This allows you to control the package name, but does not allow you to name an implicit static class for static features. Therefore you will need to use the mechanisms of this sub-section if the package contains any static features.

8.6 Automatic module renaming

Programs written in *C#* that contain a single class definition only are often created in files that take their name from the name of the class. If you try to match this same structure in *Component Pascal*, you run into a small difficulty on the *.NET* platform. Suppose you want to export a class *Rename* from a module named *Rename*. In this case the external class name in *.NET* will be "`[Rename]Rename.Rename`", and this name will clash with the name of the "synthetic static class". In this circumstance *gpcp* will automatically rename the static class, by pre-pending two underscore characters. If the module with the renamed class is imported, *gpcp* will find the renamed symbol file. In both contexts *gpcp* will issue a warning that the renaming is taking place —

```
 C:\gpcp\work> gpcp Rename.cp UseRename.cp
    1 MODULE Rename;
**** -------^ Warning: Default static class has name clash
**** Renaming static class to <__Rename>
#gpcp: <Rename> No errors, and one warning
    2   IMPORT Rename, CPmain;
**** ---------^ Warning: Looking for a auto-renamed module
**** ---------^ Looking for module "Rename" in <__Rename.cps>
#gpcp: <UseRename> No errors, and one warning
```

9 Installing and Trying the Compiler

9.1 Installation

The compiler is packaged in a single zip file "`gpcp-distroXXX.zip`". Complete instructions for installing and trying out the compiler are in the separate document "*Getting Started with GPCP*".

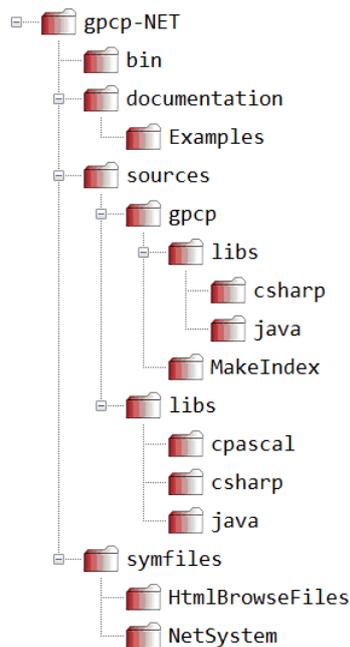Figure 19 is the complete folder hierarchy of the installed compiler. The four first-

```
gpcp-NET
    bin
    documentation
        Examples
    sources
        gpcp
            libs
                csharp
                java
            MakeIndex
        libs
            cpascal
            csharp
            java
    symfiles
        HtmlBrowseFiles
        NetSystem
```

Figure 19: Distribution File Tree

level subdirectories of the distribution are

* **bin** — the binary files of the compiler

* **documentation** — the documentation, including this file

* **sources** — the source files

* **symfiles** — contains the library symbol files

The bin directory needs to be on your *PATH*, and the environment variable *CPSYM* must point to the "`symfiles`" directory. Typical commands to set these variables on Windows systems are —

```
set CPSYM=.;%CROOT%\symfiles;%CROOT%\symfiles\NetSystem
set PATH=%PATH%;%CROOT%\bin
```

Where *CROOT* is an environment variable pointing to the root of the expanded distribution, typically —
`C:\Users\`*user*`\gpcp-NET`

10   Rebuilding from Sources

This section describes a number of alternative ways of rebuilding *gpcp* from the sources. Since almost all the code of *gpcp* is implemented in *Component Pascal* you must have a working *gpcp* executable.

10.1   Rebuilding Using the MkNetDistro Script

The script for rebuilding all the *gpcp* tools from source is `MkNetDistro.bat`. The script is located in the `sources\gpcp` directory. The script relies on `%CROOT%` being defined, `%CROOT%\bin` being on the executable path and the script being executed from the `sources\gpcp` directory.

   The script compiles the source code in the various subdirectories of the `sources` directory using the current binaries in *CROOT*`\bin`. The documentation, program examples and the *.NET* symbol files are copied from the existing distribution rooted at *CROOT*.

10.2   Manual Recompilation

Compiling the Libraries

The library sources are found in *CROOT*`\sources\libs`. There are three subdirectories containing the *Component Pascal*, *C#* and *Java* files.

   To compile the *CP* sources, execute —

```
gpcp /special ASCII.cp Console.cp CPmain.cp\
     Error.cp GPBinFiles.cp GPFiles.cp\
      GPTextFiles.cp ProgArgs.cp RTS.cp\
       StdIn.cp WinMain.cp STA.cp
```

(where the command line has been folded to fit on the page!).

   This command compiles the dummy definition files of those libraries that have separate *C#* and *Java* implementations. There are two additional libraries which have native *Component Pascal* implementations —

```
gpcp RealStr.cp StringLib.cp
```

In this case, as well as producing symbol files, there are two *PE*-files created. If you wish to re-create the *html* browser files simply execute —

```
Browse /html /sort /dst:HtmlBrowseFiles *.cps
```

   To compile the *C#* files, execute —

```
csc /t:library /debug RTS.cs
csc /t:library /debug GPFiles.cs
csc /t:library /debug /r:GPFiles.dll GPBinFiles.cs
csc /t:library /debug /r:GPFiles.dll GPTextFiles.cs
```

Compiling the Compiler Tools

There are two versions of the *JavaTarget* module, namely *JavaTargetForCLR* and *JavaTargetForJVM*. If you try to compile the wrong version *CPMake* will try to import the *ASM5* libraries that are not part of this distribution and fail. So, to be sure, before going any further execute —

```
copy JavaTargetForCLR.cp JavaTarget.cp
```

From the `sources\gpcp` directory first create a target directory for the *PE*-files, say `dest`. Then execute —

```
CPMake /all /bindir=dest gpcp
gpcp /bindir=dest ModuleHandler.cp SymbolFile.cp\
     CPMake.cp
gpcp /bindir=dest Browse.cp
```

The first of these compiles, in dependence order, all of the modules of *gpcp*. The second compiles all the extra modules that *CPMake* requires, while the last compiles the entry point module of *Browse*.

To compile *PeToCps* execute —

```
CPMake /all /bindir=dest PeToCps
```

To compile the new browse-file indexer, change directory to `MakeIndex` and execute —

```
CPMake /all /bindir=..\dest MakeIndex
```

Both the *.NET*-hosted and *JVM*-hosted versions of *gpcp* can produce *.NET PE*-files. This is achieved by writing a textual intermediate language file, then forking and executing Microsoft's *IL*-assembler `ilasm`. The fork-and-execute is performed by module *MsilAsm*. The file "`MsilAsm.cp`" in the `gpcp` directory is a dummy definition module with two implementations, each specialized for one of the host virtual machine environments —

```
sources\gpcp\libs\csharp\MsilAsm.cs
sources\gpcp\libs\java\MsilAsm.java
```

We must compile the first of these for `gpcp-NET` (the other is used by the `gpcp-JVM`). So, change directory to `sources\gpcp\libs\csharp` and execute —

```
csc /t:library /debug MsilAsm.cs
```

copy the resulting `dll` and `pdb` files to your `dest` directory.

10.3   Cross-Compiling using *gpcp-JVM*

In order to use the *JVM* version of *gpcp* to compile a *.NET*-hosted version of the compiler, it is necessary to first do the following —

1. Copy `JavaTargetForCLR.cp` to `JavaTarget.cp`.

2. Make sure that the *.NET* symbol files are on the *CPSYM* path. That is, you must use the *.NET* version of *CPSYM* so that the compiler can find symbol files like `mscorlib_System.cps`.

Now, if we suppose that the *gpcp-JVM* tool-launching scripts are in some directory *gpcp-bin* then, in the `sources/gpcp` directory —

1. Run the *gpcp-JVM* tools as follows —
   gpcp-bin\cpmake -target:jvm -all CPMake
   gpcp-bin\cpmake -target:jvm -all gpcp
   gpcp-bin\gpcp -target:jvm Browse.cp

2. Change directory to `sources/gpcp/libs/csharp` and rebuild module `MsilAsm` from the *C#* source —
```
csc /t:library /debug MsilAsm.cs
```

3. Change directory to `sources/libs/csharp` and rebuild the *gpcp* runtime system libraries —
```
csc /t:library /debug RTS.cs
csc /t:library /debug GPFiles.cs
csc /t:library /debug /r:GPFiles.dll GPBinFiles.cs
csc /t:library /debug /r:GPFiles.dll GPTextFiles.cs
```

Finally, you will need to run the `MakeAllCLR.bat` script in the `sources/libs/cpascal` directory. This compiles the foreign language interfaces to the runtime system, and the two libraries `StringLib, RealStr`. The html files are for human consumption, so you may move them anywhere you please. The symbol files `*.cps` need to be copied into a directory on the *CPSYM* path, and the *PE*-files copied to your target *CROOT* `\bin` directory.

11   GPCP Community

The definitive version of *gpcp* is maintained on github, at `https://github.com/k-john-gough/gpcp/`.

Posting to the Mail Group

There is a discussion group for users of *gpcp*. You may subscribe by sending an email to GPCP-subscribe@yahoogroups.com. The development team monitor traffic on the group, and will post update messages to the group.

12 Change summary – v1.4.08

Changes from 1.4.07

Changes in version 1.0.08 relate to two bug-fix areas for the *.NET* version. The changes introduce a syntax change for dummy definition modules, requiring such modules to indicate whether or not each *Pointer to Record* type in the underlying foreign module has a no-args constructor.

Because the new version of RTS.cp uses this new syntax the library sources can only be compiled with the new v1.4.08 binaries.

⋆ An error in the processing of dummy definition modules, such as RTS.cp resulted in all foreign reference types being defaulted as having the *has no no-arg constructor* attribute. The opposite default would have also been an error, leading to method not found errors at runtime.

In the new version the default is unchanged, but a syntax extension allows the default to be overwritten with an explicit constructor declaration. (See Section 7.1.1 in the v1.4.08 release notes.)

⋆ The *System.Windows.Forms* symbol files distributed with v1.4.07 referenced a non-existent module. *PeToCps* had pruned the module, since it had no non-generic features, but did not consistently remove references to that module.

PeToCps has been fixed to remove this error.

Changes from 1.4.06

Changes in these version 1.4.07 releases apply to both targets. They provide better facilities for accessing functionality from the *Java* and *.NET* standard libraries. The previous limitations were discovered during the development of the forthcoming *PE*-file emitter based on the *System.Reflection.Emit* module.

There is a significant change to the *Browse* tool, with additional information on the limitations of the interfaces to foreign libraries. See the *Module Interface Browser* section of the release notes.

There is one change which may be a breaking change for some users.

⋆ Prior to version 1.4.07 different implementations of *GPBinFiles.readByte* behaved inconsistently. If an attempt was made to read past the end of file the *JVM* version threw an exception, while the *.NET* version returned -1. To have programs portable between implementations this difference has to be removed even though this will be a breaking change for some users. From version 1.4.07 both versions will return -1 if an attempt is made to read past the end of file.

⋆ Http output from the *Browse* tool now includes comments which, for foreign language symbol-files, spell out any limitation on the semantics of each defined type. These comments have pop-up explanations when clicked with a pointing device.

⋆ A bug in the implementation of the /jasmin option which caused an assert trap on modules with *WITH* statements has been removed.

⋆ Previous versions of *gpcp* were incorrectly rejecting value-assignments of *.NET* structs. In fact such assignments are permitted by the *.NET* runtime, and are now available for v1.4.08.

⋆ A new option /rflemit has been introduced. This will be activated when the Reflection-Emit filewriter is finished.

Changes from 1.4.05

Changes in these version 1.4.06 releases apply mainly to the .*NET* target, but there are also some bug fixes to the *JVM* version.

The major change is the removal of the dependency on *PERWAPI* which will now be removed from the distribution. *PeToCps* now depends on *System.Reflection* to create symbol files from *PE*-files. The program should now be robust against changes in executable file format.

⋆ A new version of *PeToCps* is included. This has made a change to the options for this program. Most significantly the processing of mscorlib requires use of a new option. It is also only able to create a symbol file for the currently executing version of mscorlib. Behaviour for other libraries is unchanged.

⋆ The new version of *PeToCps* filters the list of features declared on abstract types, such as System.Char, and their companion primitive types, such as char. The symbol file includes all the static fields and static methods but excludes the instance features. Instance features on primitive types are, in any case, inaccessible to *Component Pascal*.

⋆ The *Browse* program now sorts the import list when the /html option is in force. For record types with long implements-lists the link to typebound procedures is reformatted to avoid being pushed off-screen.

⋆ There was an error in the encoding of the record-type-specific value copy function, __copy__, if the record contained a fixed-length array field that was not the final field of the record. This error had been there since version 0.5. Now fixed.

⋆ In the *JVM* version, for SE 7+, if the end of a module body was not reachable in the control flow graph the emission of a "return" byte-code caused a verification error. Now fixed.

⋆ In the *JVM* version, if a procedure body was empty, but enclosed one or more nested procedures *gpcp* crashed while trying to compute the data flow in the (inaccessible) nested procedure(s). This bug is common to both the -legacy and -jasmin code emitters. *gpcp* now processes the parameter list of the enclosing procedure, fixing the bug.

Changes from 1.4.04

All of the host-independent bug-fixes of version 1.4.04 have been applied to v1.4.05 of *gpcp-NET*. The following additional changes are applied in *gpcp-NET*.

⋆ There is a new utility *MakeIndex* that creates an index file "index.html" in a symbol-browse directory.

⋆ There is a new procedure in *ProgArgs*, ExpandWildcards. This is applied in both gpcp and Browse. Previously only the *JVM* host allowed commands like "gpcp *.cp".

* The compiler no longer offers the `perwapi` option. `perwapi.dll` is still in-cluded in the distribution because *PeToCps* still requires it. The next version will instead use `system.reflection.emit`.

* All the program examples have been put back into the distribution under the *documentation* directory. A readme file distinguishes files that are *.NET*-only or *JVM*-only, or are platform-agnostic.

The following *gpcp-JVM*-related bug fixes have been pushed to the source tree, but are not in v1.4.04 *gpcp-JVM*.

* Fixing unimplemented method trap in *CallGetClass* of `AsmUtil.cp`

* Fix option parsing of "/hsize=" vs "-hsize:" that broke on MS command line.

* Fixed import-order dependent failure when importing `RTS` and `java_lang`.

Changes from 1.3.16

The following major changes have been made to the v1.4.04 JVM version of *gpcp*.

* The packaging of the compiler has been modified so that the compiler, and the other tools, may be invoked from the *jar* files included in the distribution. How-ever, the default compiler output is unchanged, as a class-file hierarchy.

* The *JVM* version now uses the *ASM v5* library to emit its class files. This enables the compiler to emit files that are compatible with recent versions of the *Java SE* platform.

* New command line flags allow the compiler to produce class files for various *JVM* class-file standards. Output targeting *SE 6+* uses the new, preferred ver-ification framework. This slows down compilation slightly but results in faster class loading.

* The previous direct class file writer is still included in the distribution, invoked by using the `-legacy` option. This file-writer will be retired at some stage in the future, but is included in v1.4, so that *gpcp-JVM* may still be cross-compiled using *gpcp-NET*.

* A very substantial rewrite of the jave-to-symbolfile utility is now part of the distribution. This new version no longer fails when dealing with recent library jar-files. Furthermore this program deals directly with "`jar`" files as well as class-file file-trees.

* A number of errors, have been fixed, removing a number of crashes inside the compiler for invalid source files.

* Use of pointer-type local variables that are not *definitely assigned* previously raised a warning – "local variable may have its default NIL value". This is now a fatal error, as it leads to generation of *Java* bytecodes that fail verification.

* *Browse* has also had a significant rewrite. It now works corretly when given multiple symbol files, including the common case of "`browse *.cps`".

The corresponding changes to the *.NET* version will arrive later this year.

Changes from 1.3.15

The following corrections and changes are included in the 1.3.16 release.

⋆ Fixed a bug with builtin arithmetic shift function *ASH* when applied to 64-bit operands.

⋆ Added new builtin logical shift function *LSH*. This function applies to 32 and 64-bit integers. As for the standard *ASH* function, positive shifts are leftward.

⋆ The semantics of both shift operations have been changed. Shift amount is now range checked, and shifts of greater than or equal to data-word width return zero or minus one as required.

⋆ Added new builtin rotate function *ROT*. This generic function can rotate any integer-typed value from 8 to 64-bits. As with the shift functions, positive shifts are leftward.

⋆ Fixed a bug with anonymous return types of procedures. Thus, public procedures may return (pointers to) anonymous arrays of public types without error.

⋆ Corrected an error with anonymous procedure types on the *JVM* version.

⋆ The IL emitter of the *.NET* version now uses the invariant culture to write *REAL* literal values. This fixes an issue for host machines with non-Anglocentric localization settings.

Changes from 1.3.14

The following change is included in the 1.3.15 release.

⋆ The prohibition on writing to the guarded variable within a *WITH* statement has been varied to make it compatible with the behavior of *BlackBox Component Builder*. If the guarded variable is of *record* type it is now allowed to write to the fields. However any attempt to change the type of the guarded variable is a semantic error.

⋆ The behavior with *pointer* types is unchanged. The fields of the object may be written to, but the pointer itself is read-only.

Changes from 1.3.13

The following corrections and changes are included in the 1.3.14 release.

⋆ Procedure Types and variables are now supported for the *JVM* target, with the same limitation as for the *.NET* target. Specifically, values of procedure type are compatible if the types have the same *name*. The *Report* requires that values with the same *signature* be compatible.

⋆ A error in the generation of the value copy runtime support methods for the *JVM* target has been corrected. The error was rather obscure, but caused some permitted entire assignments to fail to copy some base-class fields under certain specific circumstances.

Changes from 1.3.12

The following corrections and changes are included in the 1.3.13 release.

* A significant rewrite of *J2CPS* has corrected a bug. The bug caused a module import to be missed under certain very specific, rare circumstances.

* *gpcp* now populates the definition of *RTS.NativeObject* with the appropriate methods from the underlying platform base type, *java.lang.Object* or or *System.Object*, depending on the target platform setting. This means that, for example, a type derived from *RTS.NativeObject* may override these methods without an explicit import of the whole of the system module.

Changes from 1.3.11

The following corrections and changes are included in the 1.3.12 release.

* Symbol file reading and writing have been modified so that string literals may include arbitrary Unicode, and be of unbounded size.

* Literal handling throughout the compiler has been rewritten to allow for the possibility that strings might contain embedded NUL characters.

* Reading and writing of Unicode character sequences in symbol files now uses *modified* UTF-8.

* A new pseudo-module import *STA* causes the compiler to emit a code wrapper that runs the module body in a new thread with the STA property set to true.

* Some significant errors in the implementation of vectors of *CHAR* element type have been corrected, as has some inconsistency in the implementation of entire assignment for the vector types.

* Some programs that imported both *RTS* and mscorlib were were finding that native string receivers were being denied access to the inherited methods of *System.Object*. This is now fixed.

* Programs using the *TYPEOF* extension function now work correctly when compiled with *PERWAPI*.

One consequence of these changes is that conversions between character arrays, string literals and open arrays of characters have been made consistent with the language standard. This might constitute a breaking change for programs that were relying on an implicit "stringification" of an argument array. Within the code of the compiler there was exactly one case where a call that passed an argument array "`arr`" had to be changed to the correct, "`arr$`", form.

Changes from 1.3.10

The following corrections and changes are included in the 1.3.11 release.

* A new diagnostic message is added for unresolved opaque types when using the /perwapi option.

* When an opaque type is unresolved due to a missing import the PEFile Writer attempts to correct the situation by generating a dummy import and a corresponding type-ref descriptor.

Changes from 1.3.9

The following corrections and changes are included in the 1.3.10 release.

* The resolution of calls to overloaded methods from foreign language libraries now takes place in two steps. First an exact match of the argument types to the method formals is attempted, followed, if necessary, by a match which uses no type conversions other than between the *Component Pascal* character array types and the platform string type, and between the *Component Pascal ANYREC* and *ANYPTR* types and the platform object type.

* The dummy symbol files produced by *PeToCps* from *PE*-files now ignores non-CLS compliant methods that *Component Pascal* cannot call.

* Some corrections to the encoding of the "vector types" extension avoid verifier objections.

* Some corrections to code of the separate *PERWAPI* project avoid certain failures of *PeToCps*.

* *PeToCps* does not create version information in symbol files for *PE*-files that are versioned but not strongly named.

* *gpcp* now produces code for procedures with covariant return types that is verifiable.

* The *RealStr* library now uses the invariant culture methods from the runtime system for *RealToStr* and *StrToReal*. If you need the localized methods you may directly access the methods in the runtime system library "RTS.dll".

* The *gpcp* scanner also now uses the invariant culture methods for real literals uniformly in all cases.

* The symbol file for the *ProgArgs* library now reveals the previously undocumented method *GetEnvVar* which (only in the *.NET* version) returns environment variable strings.

The changes to overload resolution do not constitute a breaking change, since all previously working cases will still work. However, a useful set of extra cases are handled. See also the comments in the new example program *Params.cp* in the *NETexamples* directory.

Changes from 1.3.8

The following corrections are included in the 1.3.9 release.

* *PeToCps* extracts public key tokens from *PE*-files using new methods of *PERWAPI*. This avoids an issue with compact framework libraries.

* *BOX* once again works correctly on *.NET* framework structs.

* Constructors with arguments for *Component Pascal* types that extend foreign classes now work as documented.

Changes from 1.3.6

The following changes and corrections are included in the 1.3.8 release.

* *PeToCps* has been extended to correctly deal with foreign *PE*-files from the compact framework.

* Limited records may be extended, but only in the defining module. New error messages are attached to the new semantic checks.

* New switch */quiet* makes *gpcp* run silently whenever possible.

* New switch */cpsym=XXX* allows the symbol file lookup path to be varied from the command line.

* *CPMake* may be started on a module which is not a "main" module. If a non-main module is used as a starting point a warning is issued to ensure that the choice was deliberate.

* Uninitialized local variables of pointer type now attract only a warning.

* Empty *CASE* and *WITH* statements no longer cause the compiler to trap, but attract a warning in the absence of an *ELSE* branch.

* *Browse* now emits import statements in v1.3.6 extended syntax.

* The new import syntax is disallowed when */strict* is in force.

Changes from 1.3.4

The following changes and corrections are included in the 1.3.6 release.

* The import declaration syntax is extended to allow foreign imports to be declared using their *.NET* syntax rather than by using the canonicalized names generated by *PeToCps*.

* Latin-8 characters are permitted in identifiers and strings.

* Much improved error reporting based on text-spans rather than (line, column) pairs. This feature also upgrades the stepping behavior in the *GuiDebug* debugger.

* New /perwapi option forces use of *PERWAPI* even when producing debuggable *PE*-files. This depends on the new version of *PERWAPI*, which can read and write *.pdb files.

* A bug in the parsing of numeric tokens ending in H and L is fixed.

* New errors are reported for numbers too large for H format, and for numbers even too large for L format.

* A bug in the *BITS* function on integers larger than max-int has been fixed.

Changes from 1.3.3

The following changes and corrections are included in the 1.3.4 release.

* ⋆ A more flexible canonicalization of assembly names has been introduced, to allow access to assemblies with filenames containing characters illegal in *Component Pascal* identifiers

* ⋆ Fixed some incorrect cases of coercion of character arrays to native strings

* ⋆ Fixed some incorrect cases of usage for *MIN, MAX* and *INC* for short integral types

* ⋆ Fixed an error in some usages of *arrays* of procedure types

Changes from 1.3.0

The following changes and corrections are included in the 1.3.1 release.

* ⋆ A new symbol file generator *PeToCps* replaces *N2CPS*. As a result, static methods, fields and constants are available for the system value types that map into the built-in types of *Component Pascal*.

* ⋆ *Browse* displays the names of formal parameters if these are available in the symbol file. Browse has a new "*/hex*" option so as to output integer literals in hexadecimal notation. Browse has a new "*/sort*" option so as to output types and static features in sorted order.

* ⋆ *LEN* now allows an argument that is an array typename, as well as the traditional case of a variable designator.

* ⋆ New Built-in constants *INF, NEGINF* have been implemented. These may be used either as REAL or SHORTREAL values.

* ⋆ The treatment of foreign modules that overload member names with fields as well as methods are now correctly handled. This is permissible behaviour in *Java*, but not *C#*.

* ⋆ Calls of *NEW* on open arrays with multiple dimensions now correctly handle arbitrary expressions in the length arguments.

* ⋆ Extremely long method signature strings in the *JVM* emitter now no longer cause a compiler panic.

Changes from 1.2.0

The following changes and corrections are included in the 1.2.x release.

* ⋆ Support for boxing and unboxing of *CLS* value types is included.

* ⋆ The vector types have been included.

* ⋆ The parser now allows return types and formal parameters to be anonymous constructed types. The compiler gives a warning when the type so defined will be inaccessible and hence useless.

⋆ A string library *StringLib* has been included.

⋆ Some corrections have been made to the *RealStr* library.

⋆ The "`WinMain`" pseudo-module introduced to mark base modules for windows executables that do not start a console when launched.

⋆ Unsafe facilities in module "`SYSTEM`" introduced.

⋆ Enhanced compatability between native strings, string literals and character literals.

⋆ Correction to the semantics of subset inclusion tests, both versions.

Changes from 1.1.6

The following changes and corrections are included in the 1.2.0 release.

⋆ The semantics of "super-calls" were incorrect in the case that the immediate super-type did not define the method being overridden. In version 1.2 the notation "`Foo^()`" denotes the overridden method no matter how distant it is in the inheritance hierarchy.

⋆ New options have been implemented for output directories.

⋆ The default behavior for the "`/nodebug`" option is to use the direct *PE*-file writer. This is significantly faster than going through `ilasm`. Unfortunately, this new file-writer does not produce debug symbols at this stage. There is separate documentation for the *PERWAPI* component included with this release.

⋆ The permitted semantics for constructors with arguments is significantly enhanced. This is of some importance when deriving from types that do not have public no-arg constructors.

Changes from 1.1.4

The following changes and corrections are included in the 1.1.6 release.

⋆ Uplevel addressing of reference parameters is now permitted in the *.NET* release, although this has inexact semantics in some cases.

⋆ A number of corrections to the *JVM* code-emitter have been added.

⋆ The new built-in function *BOX* has been added.

⋆ Trapping of types that attempt to indirectly include themselves is improved.

⋆ An automatic renaming scheme is implemented for modules that attempt to export types with the same name as the module on the *.NET* platform.

Changes from 1.1.3

The following changes and corrections are included in the 1.1.4 release.

- ⋆ The copyright notice has been revised. *gpcp* is still open source, but now has a "FreeBSD-like" licence agreement.

- ⋆ A correction to the *Java* class-file emitter now puts correct visibility markers on package-public members. Appletviewer didn't care, but most browsers objected!

- ⋆ It is now permitted to export type-bound procedures of non-exported types, provided the procedure overrides an exported method of a super-type.

- ⋆ More line-markers are emitted to *IL* in *.NET*. This makes it possible to place a breakpoint on the predicate of a conditional statement, and have the debugger stop on the predicate rather than the next executable statement.

- ⋆ The type-resolution code of "`SymFileRW.cp`" has been radically revised. It is believed that the code is now immune to certain problems caused by importing foreign libraries with circular dependencies.

13 Appendix: Working with Native Strings

There are some subtleties in converting to native strings. The following example demonstrates several strategies. The example tries to call the *Equals*() method of *System.String* to compare with a *Component Pascal* literal string.

```
MODULE StringCompare;
  IMPORT Sys := "[mscorlib]System", RTS, CPmain;

  VAR name :  RTS.NativeString;
      ltNm :  RTS.NativeString;
      sObj :  RTS.NativeObject;
BEGIN
  name := TYPEOF(Sys.String).get_Name();
 (*
  * This following does work, but with warnings, since literal strings have several
  * automatic coercions that match different overloads of the Equals( ) method
  *)
  IF name.Equals("Blah") THEN END;
 (*
  * Bad syntax, this looks like a C# cast
  *)
  IF name.Equals((Sys.String)"Blah") THEN END;
 (*
  * The cast construct is a type-check, not a conversion
  * So you cannot "cast" a literal value
  *)
  IF name.Equals("Blah"(Sys.String)) THEN END;
 (*
  * Built-in functions perform conversion. Here is a non-standard one that
  * converts char-arrays to native strings. This works ...
  *)
  IF name.Equals(MKSTR("Blah")) THEN END;
 (*
  * In the case of assigments (or non-overloaded method calls), the compiler can
  * work it out by itself without the MKSTR. Literal char arrays can be assigned to
  * objects or strings. This works.
  *)
  ltNm := "Blah"; (* gpcp automatically converts the string to System.String *)
  IF name.Equals(ltNm) THEN END;
 (*
  * In the case of reference variables the type-assertion / cast syntax does work –
  * the following two calls bind to different overloads.
  *)
  sObj := "Blah"; (* gpcp automatically converts the string to System.Object *)
  IF name.Equals(sObj) THEN END;
  IF name.Equals(sObj(Sys.String)) THEN END;
END StringCompare.
```

Curiously, these problems do not arise for the *JVM* version, since in the *Java* libraries the "equal" predicate for the *string* type overrides the predicate from *object*. In the *JVM* case there is no overloading.

14   Appendix: Overriding the Default Naming

The default naming scheme for the *.NET* version of gpcp uses the module name as the stem name for the output files, the *CLR* assembly name, the namespace name and the dummy static class name. All of these defaults may be overridden as described here. This may be necessary if another component expects a particular naming pattern.

Consider the following short program —

```
MODULE ModId; (* default naming will be used *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output file will be "`ModId.dll`", the name of the dummy static class will be "`[ModId]ModId.ModId`", and the name of the class that represents the record type will be "`[ModId]ModId.ClsId`".

It is allowed to follow the module name with a bracketed string that specifies either or both of the assembly name and the namespace name. A typical string would be —

```
MODULE ModId ["[AsmNm]SpcNm"]; (* both *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output file will be "`AsmNm.dll`", the name of the dummy static class will be "`[AsmNm]SpcNm.ModId`", and the name of the class that represents the record type will be "`[AsmNm]SpcNm.ClsId`".

In the case that only the assembly name is specified, there is no namespace defined.

```
MODULE ModId ["[AsmNm]"]; (* assembly name only *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output file will be "`AsmNm.dll`", the name of the dummy static class will be "`[AsmNm]ModId`", and the name of the class that represents the record type will be "`[AsmNm]ClsId`".

Conversely, if the namespace name is specified, but no assembly name, then the assembly name is taken from the module identifier, as in the default case.

```
MODULE ModId ["SpcNm"]; (* namespace only *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output file will be "`ModId.dll`", the name of the dummy static class will be "`[ModId]SpcNm.ModId`", and the name of the class that represents the record type will be "`[ModId]SpcNm.ClsId`".

There is just one special case remaining. In all of the previous cases the name of the dummy static class is taken from the module identifier, with the symbol (metadata) file using the same stem name. If the default name of the static dummy class clashes with the name of an explicit class then the dummy static class will be renamed, as described in Section 8.6.

```
MODULE ClsId; (* module name clashes with class id *)
  TYPE ClsId* = RECORD ...  END;
END ClsId;
```

In this example the name of the output file will be still be "ClsId.dll" but name of the dummy static class will be renamed to "[ClsId]ClsId.__ClsId", and the name of the class that represents the record type will be "[ClsId]ClsId.ClsId". The symbol file will have the name "__ClsId.cps" and, as noted earlier, will be automatically found by the compiler if the module name appears in an import list.