

Release Notes — Gardens Point Component Pascal Version 0.96 for .NET (November 2000, Beta-1)

1. Introduction

Gardens Point Component Pascal (*gpcp*) is an implementation of the Component Pascal Language, as defined in the Component Pascal Report 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 three programs, and a number of libraries. The programs are the compiler *gpcp*, the make utility *CPMake*, and a symbol file unparser *ShowSyms*. There will be other utilities added later.

The compiler produces either Microsoft.NET intermediate language (MSIL) or Java byte-codes as output. The compiler can be bootstrapped on either platform. These release notes refer to the Microsoft.NET platform.

There are a number of syntactic extensions to the Component Pascal language accepted by the compiler which are 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 which form part of Component Pascal, but rather to provide syntax for accessing features of other languages, which have no direct counterpart in Component Pascal.

2. Overall Structure

2.1 Input and Output files

In normal usage the compiler creates either three or 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 meta-information that describes the facilities exported from the module. The “*.il” file contains the MSIL intermediate language 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”. All of these files are created in the current directory. If a listing file is created it will have extension “.lst”.

Be aware that the stem name of the output files comes from the *module* name, not from the source-file name. Thus if module “Foo” is in source file “Hello.cp” 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 then invoke the intermediate language assembler manually. The assembler **ilasm** 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

```
$> gpcp [options] files←
```

where options include

/copyright	display the copyright notice
/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
/dostats	emit timing and other statistics
/hsize=NNN	set hashtable size \geq NNN (0 .. 65000)
/nocode	create il output, but do not assemble
/noasm	produce a symbol file, but no il
/nosym	produce no output files, not even a symbol file
/nocheck	produce code without arithmetic overflow checks
/verbose	chatter on about progress during compilation
/version	emit version information
/warn-	suppress warning messages from the console
/nowarn	same as /warn-
/target=xxx	emit assembler output for platform “xxx”
/special	used for creating symbol files for foreign interfaces

Any number of files may be added in a white-space separated list. The compiler also accepts Unix-style comments starting with ‘-’. In the *JVM* version the ‘-’ form is the expected default.

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 (.il) for the *.NET* virtual object system. The recognized options are –

/target=net	this is the default <i>.NET</i> virtual object system format
/target=jvm	this causes Java byte codes to be emitted
/target=dcf	this chooses the Gardens Point “d-code” form

The Java output option produces either jvm class files directly, or produces assembly language files for the “Jasmin” byte code assembler.

The dcf format is not yet available, but is intended to access the Gardens Point native code generators on all the platforms for which Gardens Point Modula-2 is implemented.

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 MSIL 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, an MSIL file and a program executable file (either **.DLL** or **.EXE**) is produced for each input file.

Output files with **/target=jvm**

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, and do not require the installation of Jasmin.

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

If, instead of **/nocode** the **/jasmin** flag is added, Jasmin assembly language files are produced for each input file. Following this, the Jasmin assembler will be automatically invoked to create the corresponding class files. Because a separate process needs to be created for each invocation of Jasmin, this is quite slow.

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 below.

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 following format –

```
1  MODULE BarMod;
2    IMPORT
3      FooMod;
4    TYPE
5      Bar* = POINTER TO ABSTRACT RECORD (FooMod.Foo) END;
****    ^ Only ABSTRACT basetypes can have abstract extensions
```

```

6             i,j,k : INTEGER
7             END;
8 END BarMod.

```

2.6 Statistics output

If the compiler is invoked with option `/dostats` then compile time statistics are produced. Here is an example for the program *ShowSyms*.

```

C:\CPtree\CPgnws> gpcp /dostats ShowSyms.cp
#gpcp: <ShowSyms> No errors
#gpcp: vos version 0.7 of 18 June 2000
#gpcp: 683 source lines
#gpcp: import recursion depth 3
#gpcp: 824 entries in hashtable of size 4099
#gpcp: import time      251mSec
#gpcp: source time     10mSec
#gpcp: parse time      250mSec
#gpcp: analysis time   20mSec
#gpcp: symWrite time   0mSec
#gpcp: asmWrite time   180mSec
#gpcp: assemble time   591mSec
#gpcp: total time      1302mSec
C:\CPtree\CPgnws>

```

The meaning of the values written to the console are as follows.

- The compiler imports symbol files in dependency order, if necessary. The maximum recursion depth for this example turned out to be 3.
- The compiler will in future allow command line choice of hashtable size. The number of entries used is shown
- Import time is the time to read and process meta-information for all imports. In the example *ShowSyms* imports most of the compiler meta-information for *gpcp*.
- 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.
- 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 (il) output.
- Assemble time is the time taken to spawn a new process and run `ilasm`. (or `jasmin` if `/target=jvm` has been set)

2.7 Setting the hash table size

The compiler uses closed hashing internally, with a default number of identifiers of 4099 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 the default table, but succeeds with `/hsize=5000`.

2.8 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 which 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, but must be the name of a module which imports *CPMain*, that is, it must be a base module.

When source files of a program have been modified in general 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.

The options accepted by the program are exactly the options accepted by *gpcp*, except that the option */all* forces compilation of all modules in the local directory irrespective of date stamps and magic numbers.

2.9 Symbol file reader

The program *ShowSyms* reads the symbol file of a module and displays the information in a human readable form. At this stage the program sends its output to the console, but later versions will produce hyperlinked, html text. The program is invoked by the command –

```
$> ShowSyms moduleName↵
```

any filename extension given to *moduleName* will be ignored.

Release 1.0 of *gpcp* will have a much more fully-functional class interface browser tool.

3. Lexical Issues

3.1 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)
ENUM	- used in dummy foreign modules in the <i>.NET</i> system

INTERFACE	- used in dummy foreign modules for defining interfaces
REMO	- an additional arithmetic operator (C integer remainder)
RESCUE	- used to mark a procedure-level exception catch block
STATIC	- used to declare static features in dummy foreign modules.

Only **DIV0**, **REMO** and **RESCUE** may be used in normal programs.

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

MKSTR	- function to convert a CP “string” to the native string type
THROW	- procedure that (re)throws a native exception object

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.

Remember, if you use any of these non-standard keywords or procedures, your program source will not be portable to other implementations of Component Pascal.

3.2 Common Language Specification names

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

Assembly name	- this specifies the dll in which the class will be found
Namespace name	- his 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 variable “x” in module **Foo** will have the somewhat boring CLS name

[Foo]Foo.Foo::x

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.

3.3 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 as *Main()*, optionally taking an array of strings as parameters.

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

If the source file does not import **CPmain**, 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 run with the `/nocode` option, then only the assembler (**.il**) 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 Unimplemented constructs

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

- Procedure variables
- Non-local variable access

Both of these features were implemented in a prototype version of the compiler, but have been removed from this release. Each is somewhat inefficient, particularly on the jvm platform, and both are being considered for alternative implementation strategies.

Procedure variables are deprecated in the CP report, so that the lack of the feature may simply anticipate complete removal from the language.

4.3 Additional Arithmetic Operators

The usual arithmetic operators **DIV** and **MOD** in Pascal-family languages have well defined semantics which 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 –

$$(i \text{ DIV } j) \times j + (i \text{ MOD } j) = i$$
$$i \text{ DIV } j = \lfloor i/j \rfloor; \text{ where } i, j \text{ are integers, and } i/j \text{ denotes real division.}$$

Notice that **DIV** always rounds toward negative infinity unlike most C-language implementations (which normally round towards 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 **REMO** denotes the corresponding remainder operation.

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

4.4 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* the selected variable is treated as a constant, and neither the type nor the value can be modified either directly or indirectly. Any attempt to do so attracts a compile-time error message.

4.5 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 from the *CLS*. The syntax extension to access this feature is –

```
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.

The semantics of type casts are also relaxed whenever a reference is cast to an imported interface type. For non-interface types many erroneous casts can be detected at compile time, but for interfaces no cast of an object of a foreign type can be rejected at compile time.

It is not possible to *define* interface types in Component Pascal.

4.6 Additional built-in functions

There are two additional built-in functions added to the language. 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.*

The other new built-in function allows programs to throw exceptions, and is described below.

4.7 Deprecated features and warnings

The use of procedure variables or 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 within their defining module. This situation is usually an error arising from failure to mark the procedure for export.

4.8 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 *pverify* over the file. Here is an example –

```
C:\CPtree\CPngws> peverify /IL ShowSyms.exe
```

```
Microsoft © COM+ PE Verifier. Version 2000.14.1812.10  
Copyright © Microsoft Corp. 1998-2000
```

```
All Classes and Methods in ShowSyms.exe verified
```

```
C:\CPtree\CPngws>
```

Output might fail to verify if a manually constructed interface to a library does not correspond to the internal metainformation of the imported assembly. This potential problem will go away when the tool for automatic construction of interfaces is available.

4.9 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 → BEGIN [ “[ “UNCHECKED_ARITHMETIC” ]” ]  
StatementSequence END identifier .
```

An example of the use of this construct, from the source of the compiler itself, is the identifier hash function –

```
PROCEDURE hashStr(IN str : ARRAY OF CHAR) : INTEGER;  
  VAR tot : INTEGER;  
      idx : INTEGER;  
      len : INTEGER;  
BEGIN [UNCHECKED_ARITHMETIC]  
  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;
```

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.



Important note on parameter passing semantics if you use /target=jvm.

- ✓ *The semantics of parameter passing on the .NET version are precise. They are also precise in the D-Code version.*
- ✗ *The JVM version of **gpcp** takes liberties with the precise semantics of parameter passing. Actual parameters of unboxed value type that are passed to reference formals are passed by copying. (Unboxed value types 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 modification.) 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 copied **out**. This change is necessary in order to obtain reasonable performance on the JVM. This change will not affect the results of your program unless you access the actual of an a reference formal along two paths (either by having two reference formals sharing the same actual, 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.*

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 builtin 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);

This may be used anywhere in the program, but is most useful for rethrowing an exception from within a rescue clause.

☑ ***Remember, if you use these non-standard facilities for exception handling your program source will not be portable to other implementations of Component Pascal. Of course it will still be portable between different implementations of Gardens Point Component Pascal.***

If you want to create an exception object to abort program execution with a meaningful string, the library function **RTS.Throw(msg : ARRAY OF CHAR)** may be used. 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** this library writes strings and numbers to the console
- **Error** this library writes strings and number to the error stream
- **ProgArgs** this library provides access to the command line arguments, if any
- **GPText** a basic library for handling text formatting
- **GPFiles** defines the supertype of *GPBinFiles.FILE* and *GPTextFiles.FILE*
- **GPBinFiles** reading and writing binary files
- **GPTextFiles** reading and writing text files
- **RTS** access to the facilities of the runtime system

For the most part these libraries are the ones that were required to bootstrap the compiler. More will come later ...

6.2 The runtime system RTS.cp

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, 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*.

ProgArgs, *Console*, and *Error* are also system modules, and have their real source in the same C# file.

Here is the “source” of RTS.

```
(**  These are the user accessible static methods of the CP runtime system.  
*   These are the environment-independent ones. Others are in CP*.cp  
*   Note: the bodies of these procedures are dummies, this module is  
*   compiled with /special. The real code is in RTS.cs or other. *)
```

```

SYSTEM MODULE RTS;
  VAR defaultTarget- : ARRAY 4 OF CHAR;

  TYPE
    CharOpen*          = POINTER TO ARRAY OF CHAR;

  TYPE
    NativeObject*     = POINTER TO RECORD END;
    NativeString*     = POINTER TO RECORD END;
    NativeException*  = POINTER TO RECORD END;

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

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

  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 longint with same bit pattern *)

  PROCEDURE longBitsToReal*(l : LONGINT) : REAL;
    (** Convert ieee double to a longint 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 with an error *)

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

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

PROCEDURE ClassMarker*(o : ANYPTR); (* write class name *)
END RTS.

```

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. The mark simply indicates that the resources of this module are actually found in the assembly *RTS.dll*. *Console*, *Error* and *ProgArgs* are also *SYSTEM* modules.

7. Foreign Language Interface

7.1 Accessing the basic underlying types

The underlying types are accessible without any other import other than *RTS*. At runtime the compiler queries the target flag, or takes the default value if there is no target command option.

If the target is “net” then *NativeObject*, *NativeString* and *NativeException* will be the CLS 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 which interfaces to foreign libraries. However, if a CP-style, non-literal string, i.e. a nul-terminated array of char needs to be transformed to a native string, the non-standard built-in function –

```

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

```

may be used.

7.2 Compiling dummy definition modules

As an interim measure, 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

The syntax - `MODULE Foo["[blah]space"];` declares that this module will be found in assembly “blah” in namespace “space”. 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, subsection “*Common Language Specification Names*”.

The syntax - `PROCEDURE (x : T)BarII*["Bar"](i, j : INTEGER);` declares that this 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.

If a name clashes with a CP keyword, it should be defined using the back-quote escape.

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

```
TYPE Foo* = POINTER TO INTERFACE RECORD END;
```

The keyword *INTERFACE* is reserved, and such a type cannot declare any fields in the record, nor can it define type-bound procedure 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 other types in CP (see section 8.4 for more detail). If access to constructors with arguments is required, then these are given a CP 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

where

- the type identifier resolves to an imported, foreign type, and
- the identifier is a static field or constant of the type, or
- the identifier is a static method of the type

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

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

Record → **RECORD** [(“ TypeId “)] { FieldList } [**STATIC** { StatFeature }] **END**.
StatFeature → ProcHeading | StatConst | StatField .
StatConst → identifier “=” ConstExpression .
StatField → identifier “:” TypeId .

Procedure headings have the same syntax as elsewhere in the language.

8.0 Creating 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.

8.1 Syntax of Foreign Definitions

The syntax of foreign definition is as follows. Unless otherwise defined here, the meanings of non-terminal symbols is the same as in the Component Pascal Report.

GPMModule → Module | ForeignMod .
ForeignMod → (**FOREIGN** | **SYSTEM**) **MODULE** *ident* [*string*] “;”
ImportList DeclSeq **END** *ident* “.” .
DeclSeq → { **CONST** { ConstDecl “;” } | **TYPE** { TypeDecl “:” } | **VAR** { VarDecl “;” } }
{ ProcHeading “;” | MethodHeading “;” }
ProcHeading → **PROCEDURE** IdentDef [“[“ *string* “]”] [FormalPars] .
MethodHeading → **PROCEDURE** Receiver IdentDef [“[“ *string* “]”] [FormalPars]
[“;” **NEW**] [“;” **ABSTRACT** | **EMPTY** | **EXTENSIBLE**] .
TypeDecl → IdentDef “=” Type .
Type → [**POINTER TO**] [Attributes] **RECORD** [(“ *Qualident* “)]
FieldList { “;” FieldList }
[**STATIC** StaticDecl { “;” StaticDecl }] **END**
| *Other types as in the Report* .
StaticDecl → IdentList “;” Type | IdentDef “=” ConstExpr | ProcHeading .
Attributes → **ABSTRACT** | **EXTENSIBLE** | **INTERFACE** .

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 “foreign”.

8.2 Explicit package or namespace names

The way in which runtime names are generated from module names was described in Section 3.2. In the case of the *JVM* we have the following correspondence.

Component Pascal Name	JVM Name
MODULE ModNm;	CP.ModNm // <i>package name</i>
TYPE Cls = RECORD ... END ;	CP.ModNm.Cls // <i>class name</i>
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 entities *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 // <i>namespace</i>
TYPE Cls = RECORD ... END ;	[ModNm]ModNm.Cls // <i>class name</i>
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_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 another 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 CP program the two names are distinct, but the program executable will correctly refer to the underlying overloaded methods.

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 which is used for calling a constructor. On *JVM* the name is “<init>”, while on *.NET* the name is “.ctor”. These two strings are used as the explicit string which defines such a procedure in the foreign definition. An example of an interface to a constructor with arguments might be –

```
PROCEDURE NewRectangle*[".ctor"](width,height : INTEGER) : Rect;  
PROCEDURE NewRectangle*["<init>"](width,height : INTEGER) : Rect;
```

Note that this declaration would normally be declared 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.NewRectangle(25,17);
```

would translate into a call to the appropriate one of –

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

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 Section 8.1 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 –

Component Pascal Foreign Definition	Component Pascal Usage
FOREIGN MODULE ModNm;	
TYPE Cls =	ModNm.Cls <i>// class name</i>
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 CP 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 are 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. 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. Installing and Trying the Compiler

8.1 Installation

The compiler is packaged in a single zip file which is usually unzipped into a directory with a name such as **\gpcp**. There are six sub-directories. These are –

- **bin** the binary files of the compiler
- **docs** the documentation, including this file

- **examples** some example programs
- **libs** contains the simple library files
- **source** the source files (will have compiler source later)
- **work** a working directory to play around with

The bin directory needs to be on your PATH, and the environment variable CPSYM must point to the libs directory. Typical commands are –

```
set CPSYM=.;C:\gpcp\libs
set PATH=%PATH%;C:\gpcp\bin
```

9. Future Releases

Release 0.96 has a very limited range of libraries packaged with it, essentially only those needed to bootstrap the compiler. There are also limitations caused by the absence of a tool for generating foreign language interfaces directly from the metadata.

The distribution is sufficient to try out the compiler, and is being updated on a frequent basis. We expect new releases to include –

- A tool for producing foreign interfaces automatically
- A “make” tool for compiling modules in dependency order
- More libraries

Updates are announced and available from www.plasrc.qut.edu.au/ComponentPascal

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.