Getting Started with GPCP-JVM

John Gough

March 29, 2018

This document applies to GPCP version 1.4 for JVM (Java Virtual Machine)

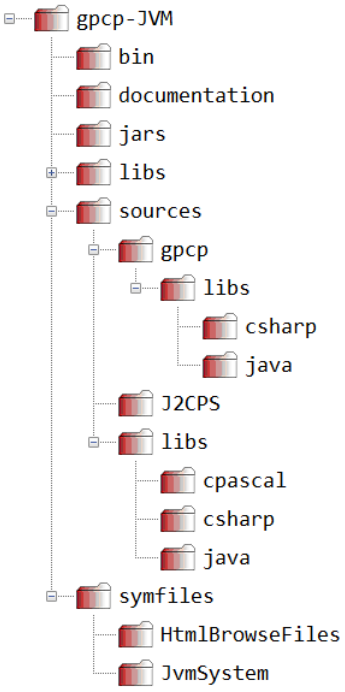


Figure 1: Distribution File Tree

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 in the release notes. Any other differences in detail should be reported as potential bugs.

The compiler produces either Microsoft.NET intermediate language or Java byte-codes as output. Java byte codes are executed by a Java Virtual Machine (*JVM*). The compiler can be bootstrapped on either platform. These notes refer to the JVM platform. Details on the specifics of this implementation of Component Pascal are found in the release notes that come with the distribution.

2 Installing and Testing the Compiler

Environment

The compiler requires the Java Runtime Environment, running on any compatible platform. The release version of the compiler has been tested against Java SE version 1.8.0_161, but works with earlier versions also. *gpcp* is distributed as a zipped archive.

The archive contains several subdirectories. These hold the binary files of the compiler, the documentation, the library symbol files, and the source code of the compiler.

If you plan to install both the *.NET* and the *JVM* version you may wish to place the two distribution trees in distinguished folders such as `\gpcp-JVM` and `\gpcp-NET`.

On the Windows platform, the archive is expanded into some directory, typically `C:\Users\`*username*`\gpcp-JVM`. On *UNIX* platforms the system is typically installed in a shared directory, such as `/usr/local/gpcp-JVM`. In either case an environment variable *JROOT* is defined to point to this directory.

This section describes the steps required to install and try out the compiler

The distribution

The complete distribution tree is shown in Figure 1. The six first-level subdirectories of the distribution are

- **bin** — the wrapper scripts to invoke the gpcp tools
- **documentation** — the documentation, including this file
- **jars** — the executable jar files of all the tools
- **libs** — contains the library class files
- **sources** — the source files of the tools
- **symfiles** — the library symbol files

The tool chain relies on just two environment variables —

JROOT — the root of the distribution

CPSYM — the path to the symbolfiles

Each installation will require *JROOT* to be set to match the choice made for the location of the distribution. In addition, the *bin* sub-directory must be added to the execution path.

On Windows systems, for a typical installation, this will require commands such as —

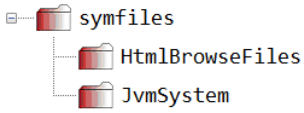
```
set JROOT=%HOMEPATH%\gpcp-JVM
set PATH=%JROOT%\bin;%PATH%
set CPSYM=.;%JROOT%\symfiles;%JROOT%\symfiles\JvmSystem
```

About Symbol Files

Separate compilation in *Component Pascal* depends on the compiler reading the *API* metadata for each module on which a particular module depends. These metadata files are called *symbol files* and have the *cps* file extension. The `CPSYM` environment variable is the search path that the compiler and other tools use to find symbol files.

Except for some special cases, such as recompiling the compiler itself, it should not ever be necessary to vary the `CPSYM` setting. The path given in the example above determines that the compiler first seeks a symbol file in the current directory, (that is, a module in the same project); next it checks *JROOT**symfiles*, (that is, a module from the *Component Pascal* standard libraries); finally the compiler checks *JROOT* *symfiles**JvmSystem*, (that is, a "module" denoting a package of the *Java* libraries).

The "`symfiles`" distribution sub-directory contains the symbol files for the Component Pascal standard libraries. The "`JvmSystem`" sub-sub-directory contains the symbol files which interface to the Java runtime libraries. The separate "`HtmlBrowseFiles`" directory contains html files which display the method signatures of the java libraries in *Component Pascal* format.



Running your first program

Create or go to some working directory. With your favorite editor create the file (say) "`hello.cp`".

```
MODULE Hello;
  IMPORT CPmain, Console;
BEGIN
  Console.WriteString("Hello CP World");
  Console.WriteLn;
END Hello.
```

Make sure that *JROOT* is set, and that `\gpcp\bin` is on the executable path. Then, from the command line, type

```
> gpcp hello.cp
```

the system should respond

```
Created file CP\Hello\Hello.class
#gpcp: <Hello> no errors
> _
```

The file "`Hello.cps`" will have been created in the working directory. The "`.cps`" extension is the symbol file that declares the publicly accessible facilities of the program. By default *gpcp* will create a subdirectory in the working directory named "`CP`", with subdirectories for each module compiled. One or more class files will be created and placed in the subdirectories. In our example there is only one class file produced.

You may now run the program by the command "`cprun Hello`". Note carefully that the base class file has name "`Hello`", and filename arguments to the "`cprun`" command are case sensitive. The shell file *cprun* (a batch file in Windows) passes the *CPSYM* property to the java runtime, and sets the classpath. It also knows, for example, that the entry point class generated from module *Hello* is `CP.Hello.Hello`.

The Binary files

The "`jars`" distribution sub-directory contains the executable code of the *gpcp* tools. These are —

* `asm-5.1.jar` The library jar-file of the ASM library.
* `asm-5.1.pom` Copyright notice for the ASM library.
* `browse.jar` Executable jar-file of the sym-file browser.
* `cpmake.jar` Executable jar-file of the CPMake tool.
* `cprts.jar` Jar of all the *gpcp* runtime libraries.
* `gpcp.jar` Executable jar-file of the *gpcp* compiler.
* `J2cps.jar` Executable jar-file of the J2cps tool.

The "`bin`" distribution sub-directory contains a small number of shell or batch files to invoke the *Java* runtime on *Component Pascal* executables. For windows systems these are —

* `browse.bat` Invokes browse.jar from *JROOT*`\jars`
* `cpmake.bat` Invokes cpmake.jar from *JROOT*`\jars`
* `cprun.bat` Invokes a *Component Pascal* program from the classpath
* `gpcp.bat` Invokes gpcp.jar from *JROOT*`\jars`
* `j2cps.bat` Invokes J2cps.jar from *JROOT*`\jars`

Apart from the standard tools in the distribution *Component Pascal* programs are usually invoked from the class-file hierarchy using the "`cprun`" wrapper. However, if the tools are unpacked into a class-file tree they too may be invoked by *cprun*.

3 Using CPMake

When *gpcp* compiles a module it must have access to the symbolfiles of every module on which that module depends. This implies that every project with more than one module must be compiled in an order that respects those dependencies. For complicated projects this is hard to do by hand. However, every source module explicitly declares all of its dependencies in the first few lines of its source, allowing a simple programmed solution.

The *CPMake* program is given the name of the entry-point module of a program. It reads the first few lines of each module on the program and builds a representation of the dependencies of the entire program. *CPMake* then invokes the compiler on any out-of-date modules in the correct dependency order. It may also be invoked with an `-all` switch to force recompilation of the complete program in dependency order. Further details on the use of this tool can be found in the Release Notes.

4 Browsing Modules

The *Browse* tool has been included with this release. This tool can show the exported interface for modules in either text or html format. Details on the use of this tool can also be found in the Release Notes.

5 Reporting Bugs

If you find a bug

If you find what you believe is a bug, please post it as an issue to the GitHub page for gpcp, with the detail of the event. It would be particularly helpful if you can send the code of the shortest program which can illustrate the error.

If the compiler crashes

The compiler has an outer-level exception rescue clause (you can see this in the body of procedure "`CPascal.Compile()`") which catches any exceptions raised during any per-file compilation attempt. The rescue code displays a "`<<compiler panic>>`" message on the console, and attempts to create a listing in the usual way. In most cases the rescue clause will be able to build an error message from the exception call chain, and will send this both to the screen and to the listing file.

In almost all cases, the compiler panic will be caused by failed error recovery in the compiler, so that the other error messages in the listing will point to the means of programming around the compiler bug. Nevertheless, it is important to us to remove such bugs from the compiler, so we encourage users who turn up errors of this kind to send us a listing of a (hopefully minimal) program displaying the phenomenon.

In order to see how such a rescue clause works, here is an example of a program that deliberately causes a runtime error. When the program is run, the error is caught at the outer level and an error message is generated. After generating the error message, there is still the option of aborting the program with the standard error diagnostics. This is done by re-raising the same exception, and this time allowing the exception to propagate outwards to the invoking command line processor.

```
MODULE Crash;
  IMPORT CPmain, Console, RTS;

  TYPE OpenChar = POINTER TO ARRAY OF CHAR;
  VAR  p : OpenChar;

  PROCEDURE Catch;
  BEGIN
    p[0] := "a";(* line 9 *)
  RESCUE (exc)   (* exc is of type RTS.NativeException *)
    Console.WriteString("Caught Exception: "); Console.WriteLn;
    Console.WriteString(RTS.getStr(exc)); Console.WriteLn;
   (* THROW(exc) *)(* line 13 *)
  END Catch;

BEGIN
  Catch()(* line 17 *)
END Crash.
```

When this program is compiled and run, the following is the result —

```
> gpcp Crash.cp
Created file CP\Crash\Crash.class
#gpcp: <Crash> No errors
> cprun Crash
Caught Exception:
    java.lang.NullPointerException > _
```

If the detailed stack trace is required, the exception may be re-raised by calling the non-standard built-in procedure *THROW*(), with the incoming exception as argument. The comment in the source shows where to place the call. If this is done then a full stack trace may be produced. For this example it is just —

```
Caught Exception:
    java.lang.NullPointerException
Exception in thread "main" java.lang.NullPointerException
   at CP.Crash.Crash.Catch(Crash.cp:9)
   at CP.Crash.Crash.main(Crash.cp:17)
```

You may care to note that the stack trace produced by the system starts at the line at which the exception was originally thrown, rather than the point at which it was explicitly re-thrown. This behaviour is different on the *.NET* platform.

Read the Release Notes!

There are a number of extensions to the language, many of these have been introduced so that Component Pascal programs will to be able to access all of the facilities of the *Java Runtime Environment*. Read the release notes to find out about all of these! In particular, from version 1.2.4 there is built-in support for extensible arrays (vectors).

Future Releases

The definitive version of *gpcp* is maintained on the CodePlex repository. `http://gpcp.codeplex.com/`. The site has discussion, issue and download pages, and also has the source code revision control system repository.

As of this writing, the distributions and code are in the process of migrating to GitHub. The location is `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.