Getting Started with GPCP

John Gough

March 29, 2018

This document applies to GPCP version 1.4 for .NET (Microsoft Common Language Runtime)

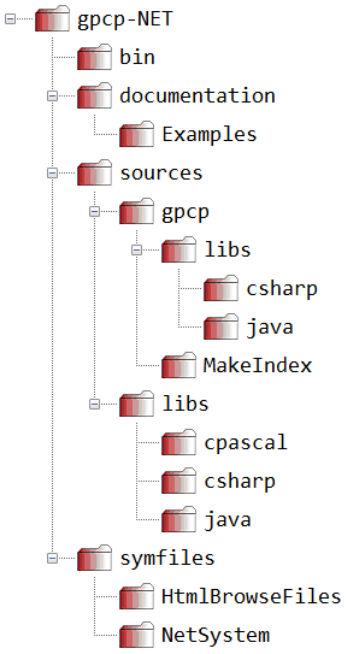


Figure 1: Distribution File Tree

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 bytecodes as output. The compiler can be bootstrapped on either platform. These notes refer to the Microsoft.NET 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 *.NET* runtime system, running on any compatible Windows platform. Previous versions of *gpcp* (1.3.∗) run on versions 2.0 to 3.5 of the framework. The binaries in those distributions are not compatible with version 4.0 of the framework. The new distribution v1.4.0 is compatible with version 4+ of the runtime.

The distribution is a simple zip file, which may be expanded at whatever location the user prefers. A single environment variable needs to be set. See the endnote Section 6 if the command line tools fail to find any of the *.NET* executables.

The distribution includes subdirectories that contain 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-NET` and `\gpcp-JVM` respectively.

On the Windows platform, the archive is expanded into some directory, typically `C:\Users\`*username*`\gpcp-NET`, which is pointed to by the environment variable `CROOT`.

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 four first-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 tool chain relies on just two environment variables —

CROOT — the root of the distribution

CPSYM — the path to the symbolfiles

Each installation will require *CROOT* 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 —

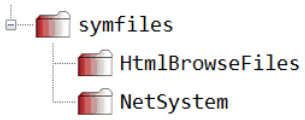
```
set CROOT=%HOMEPATH%\gpcp-NET
set PATH=%CROOT%\bin;%PATH%
set CPSYM=.;%CROOT%\symfiles;%CROOT%\symfiles\NetSystem
```

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 meta-data 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 *CROOT**symfiles*, (that is, a module from the *Component Pascal* standard libraries); finally the compiler checks *CROOT* *symfiles**NetSystem*, (that is, a "module" denoting a namespace of the *.NET* libraries).

The "`symfiles`" distribution sub-directory contains the symbol files for the Component Pascal standard libraries. The "`NetSystem`" sub-sub-directory contains the symbol files which interface to the *.NET* runtime libraries. The separate "`HtmlBrowseFiles`" directory contains html files which display the method signatures of all the 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 the *CROOT* environment variable is set, and that `%CROOT%\bin` is on the executable path.

From the command line, type

```
> gpcp hello.cp
```

the system should respond

```
#gpcp: created Hello.exe
#gpcp: <Hello> no errors
> _
```

Three files will have been created by the compiler: these are `Hello.cps`, `Hello.exe`, and `Hello.pdb`.

The "`.cps`" extension is the symbol file that declares the publicly accessible facilities of the program. The "`.exe`" extension shows that the program executable file that was created is an application, rather than a library. The "`.pdb`" extension is used for the program database file that holds debugging information. No `pdb` file is produced if the `/nodebug` option is given to the compiler.[1]

In order for this program to run, it must have access to the public methods of the CP runtime system. The methods are found in the file "`RTS.dll`", which must be copied to the working directory. This file is in `%CROOT%\bin`

You may now run the program by the command "`Hello`".

3 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 be found in the Release Notes.

4 Using the visual debugger

Update

Since VS2005 distributions of .NET have not included the Dbg-Clr program. It is nevertheless possible to use Visual Studio to debug CP programs. There are probably many methods of doing this, but my particular method is described in my blog entry of March 2011 at softwareautomata.blogspot.com

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 mailgroup for gpcp (see Section 5.1) with details of the event. It would be particularly helpful if you can send code of the shortest program which can illustrate the error.

[1] In normal use the compiler also creates another, temporary file, in this case named "`Hello.il`", containing *.NET* assembly language. Once the il-assember has finished with it, you may safely delete all such.

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 error 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
#gpcp: created Crash.exe
#gpcp: <Crash> No errors
> Crash
Caught Exception:
    System.NullReferenceException:
Object reference not set to an instance of an object.
    at Crash.Crash.Catch() in Crash.cp:line 9
> _
```

If the detailed stack trace is required, the exception is 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 —

```
Unhandled Exception: System.NullReferenceException:
   Object reference not set to an instance of an object.
   at Crash.Crash.Catch() in Crash.cp:line 13
   at Crash.Crash..CPmain(String[] A_0) in Crash.cp:line 17
```

You may care to note that the stack trace produced by the system starts at the line at which the exception was rethrown, rather than the point at which it was first raised. This is why the first line in the back-trace starts at line 13 in the example, rather than the line 9 which first raised the exception.

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 *.NET* Common Language Specification. 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).

5.1 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.

6 Checking your Installation

If you have Visual Studio *.NET* installed on your system, rather than just the Software Development Kit, you may need to manually add some extra components to your *PATH* variable.

The *gpcp* compiler needs to be able to access "`ilasm`" in order to operate correctly. If you want to understand how the *.NET* system works, rather than just using *gpcp-NET*, then it is helpful to have access to "`ildasm`" and "`peverify`". You can check whether these programs are accessible by opening a command window and try —

```
> ilasm dummy.il
Microsoft (R) .NET Framework IL Assembler. Version 4.7.2046.0
Copyright (C) Microsoft Corporation. All rights reserved.
...
```

The command "`ildasm`" should launch an empty window, and calling "`peverify`" should respond —

```
Microsoft (R) .NET Framework PE Verifier. Version 1.1.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.
...
```

On a typical installation, "`ilasm.exe`" is found in the folder —

```
C:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\
```

The *C#* compiler and many other tools are found in the same folder. This folders should be added to the *PATH* if it is not automatically added by the installation process for *.NET*.

"`ildasm`", "`peverify`" and other tools are found in the folder —

```
C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools
```