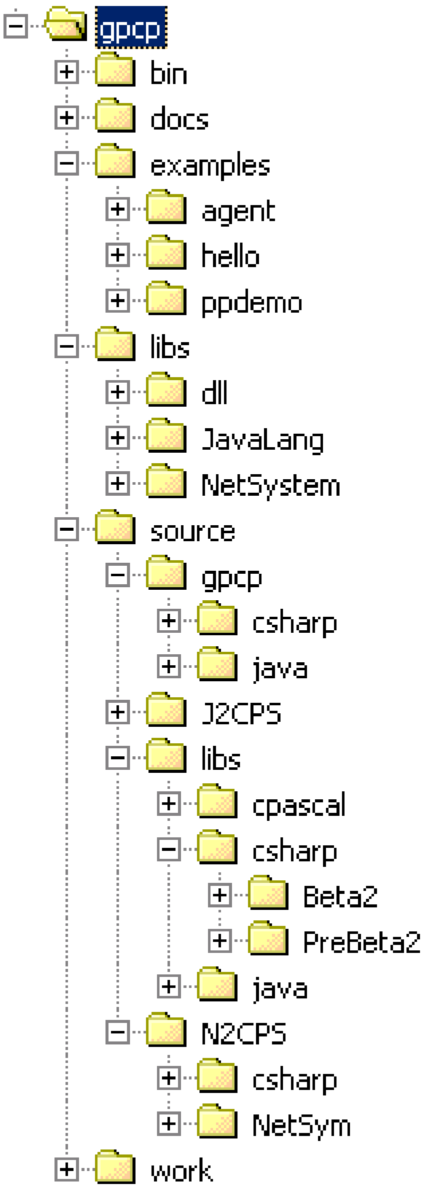
Getting Started with Gardens Point Component Pascal
Version 1.2.0 .NET (*September 2002*)

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. 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 Windows platform. The released version of the compiler has been tested against build 3705 of the .NET framework, under Windows-2000. This is the “*release to manufacture*” version of the .NET runtime. The compiler is distributed as a single zip file, which has the structure shown on the left.

From version 1.1.4 the compiler has been available as an installer file. Download whichever `setup.exe` file you require and run the installer.

The archive is typically expanded into a root directory named **`\gpcp`** and has six or seven subdirectories. These include the binary files of the compiler, the documentation, the program examples, the library symbol files, and the source of the compiler.

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

The distribution

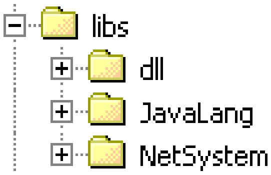
The six subdirectories of the distribution are –

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
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 to set these variables are –

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

Preferably they should be set in the system window of the control panel. If you use the installer version, the paths should be set automatically during installation.



The **libs** directory contains the symbol files for the Component Pascal libraries. There are three subdirectories under **libs**. The first of these contains the library dynamic link libraries for the runtime system **RTS.dll**. This file may need to be copied into your working directory, in order to run your programs. The **JavaLang** directory is for the symbols files to interface to the Java runtime. This directory is empty in this version. The **NetSystem** directory contains the symbol files that allow Component Pascal programs to access the base classes of the **.NET** system.

Running your first program

Go to the work 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 `CPSYM` environment variable includes the `gpcp\libs` directory, and that `gpcp\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
> ▯
```

The files **`Hello.il, Hello.cps`** and **`Hello.exe`** should have been created in the working directory.

In order for this program to run, it must have access to the facilities of the CP runtime system. These facilities are found in the file **`RTS.dll,`** which must be copied to the working directory. It may be found in **`gpcp\libs\dll.`**

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

The examples

The example programs are in three sub-directories under the examples directory. The folder **hello** holds some simple command line programs. *HelloWorld.cp* is an elaborate version of the "hello world" canonical program. *Nqueens.cp* is a recursive backtracking version of the N-Queens problem solved for all board sizes from 8 to 13. *Hennessy.cp* is a version of the Hennessy integer benchmarks.

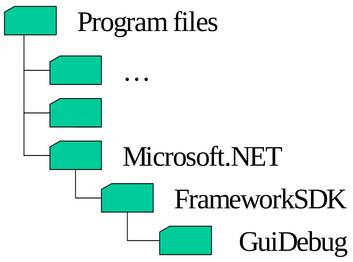
A file *README.txt* gives instructions for compiling and running the programs.

The folder **agent** has two simple Microsoft agent demonstrations. You must have the agent system installed on your machine to use these. The folder contains a README file with further details.

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



The .NET system comes with a very capable visual debugger. This debugger has many of the facilities of the debugger in MS Visual Studio.

The debugger executable is in the **`GuiDebug`** directory in the .NET distribution, typically under **`\Program files\Microsoft.NET\FrameworksSDK,`** as in the diagram above. The name of the program is **`DbgCLR.exe`**. It may be useful to drag and drop the icon onto your task bar, so that you can start it with a single mouse click.

The first time that you run the program you may simply specify the program that you wish to debug. If you save the session as a "solution", you will be able to restart the session from the "open session" menu.

Start up the program, and go to the *debug … program to debug* pull-down menu. In the dialog box enter the simple name of the program, "**`Hello.exe`**" say. In the other boxes, fill in any arguments that you wish to send to your program, and the path name of the working directory that you wish to execute the program from. Be aware that if your program needs any environment variables you must set these outside of the debugger. Dismiss the dialog box with "ok" and begin debugging.

Most debugging operations can be controlled from the icon bar. You may execute the program stepwise using either the icons or the function keys.

If you want to save your session, go to the File menu and choose "Save solution". You will now be able to restart debugging the same program by using the *File … open solution* pull-down menu.

It is the default of ***gpcp*** to assemble programs with debug information included, so that the system should be able to display the source text of your program as you step through it. If you have mixed language programs, provided the other files have been compiled with debugging information included you should be able to automatically step from language to language in the source window.

5. Reporting Bugs

If you find a bug

If you find what you believe is a bug, please send a report to gpcp@qut.edu.au 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 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 Ptr = POINTER TO ARRAY OF CHAR;
  VAR  p : Ptr;
  PROCEDURE Catch;
  BEGIN
    P[0] := “a”;
  RESCUE (exc)   (* exc is of type RTS.NativeException *)
    Console.WriteString(“Caught Exception: ”); Console.WriteLn;
    Console.WriteString(RTS.getStr(exc)); Console.WriteLn;
    (* THROW(exc) *)
  END Catch;
BEGIN
  Catch
END Crash.
```

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

```
➢ gpcp Crash.cp
➢ #gpcp: CP/Crash/Crash.class
➢ #gpcp: <Crash> No errors
➢ cprun Crash
Caught Exception:
System.NullReferenceException
  at Crash.Crash.Catch()
➢ ▯
```

If the detailed stack trace is required, the exception is re-raised by calling the non-standard built-in procedure **`THROW(ex).`** The comment in the source shows where to place the call.

Read the Release Notes!

There are a number of extensions to the language, 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!

Posting to the Mail Group

The Gardens Point Modula-2 mail group can be used to discuss issues concerning the evolution of ***gpcp***. In order to join this mail group, send email to majordomo@dstc.qut.edu.au with a blank subject line and the words **subscribe gpm** in the body. The team will post notices regarding updates to that mail group.