Notices

Edition

Publication date: November 2013
Book number: UNV-112-GCI-1
Product version: Rocket UniVerse V11.2

Copyright

© Rocket Software, Inc. or its affiliates 1985-2014. All Rights Reserved.

Trademarks

Rocket is a registered trademark of Rocket Software, Inc. For a list of Rocket registered trademarks go to: www.rocketsoftware.com/about/legal. All other products or services mentioned in this document may be covered by the trademarks, service marks, or product names of their respective owners.

Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

License agreement

This software and the associated documentation are proprietary and confidential to Rocket Software, Inc., are furnished under license, and may be used and copied only in accordance with the terms of such license.

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Contact information

Website: www.rocketsoftware.com
Rocket Software, Inc. Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA
Tel: +1 781 577 4321
Fax: +1 617 630 7100
Contacting Global Technical Support

If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Customer Portal to report and track a problem, to submit an enhancement request or question, or to find answers in the U2 Knowledgebase. The Rocket Customer Portal is the primary method of obtaining support.

To log in to the Rocket Customer Portal, go to:

www.rocketsoftware.com/support

If you do not already have a Rocket Customer Portal account, you can request one by clicking Need an account? on the Rocket Customer Portal login page.

Alternatively, you can contact Global Technical Support by email or by telephone:

Email: u2support@rocketsoftware.com

Telephone:

North America +1 800 729 3553
United Kingdom/France +44 (0) 800 773 771 or +44 (0) 20 8867 3691
Europe/Africa +44 (0) 20 8867 3692
Australia +1 800 707 703 or +61 (0) 29412 5450
New Zealand +0800 505 515
## Table of Contents

### Chapter 1: Using the GCI

- What is the GCI? ........................................ 1-3
- What you need to do .................................. 1-4
- Preparing the subroutine ................................ 1-5
- Defining the subroutine ................................. 1-6
  - Adding a subroutine definition record.............. 1-6
  - Adding the subroutine to UniVerse ................. 1-10
  - Building a new run file on UNIX .................. 1-10
  - Making a GCI library on Windows platforms ....... 1-13

### Chapter 2: The calling program

### Chapter 3: GCI subroutines

- Direct calls ............................................. 2-3
- Indirect calls ........................................... 2-4
- Function calls .......................................... 2-5
- Declaring a function ................................... 2-6
- Passing arguments ..................................... 2-7
- Accessing the system `errno` variable .............. 2-8
- UNIX file units ........................................ 3-11
- C data types ............................................. 3-12
  - Data types for Windows system calls .............. 3-13
  - Allocating memory for character strings ......... 3-13
  - Converting data types between C and BASIC .... 3-14
  - C arrays ............................................... 3-14
- FORTRAN data types .................................. 3-15
- FORTRAN arrays ....................................... 3-16
- FORTRAN portability .................................. 3-17
- Data types for multibyte characters ............... 3-18
### Chapter 4: GCI functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UVClosePipe function</td>
<td>4-3</td>
</tr>
<tr>
<td>UVCreatePipe function</td>
<td>4-4</td>
</tr>
<tr>
<td>UVCreateProcess function</td>
<td>4-5</td>
</tr>
<tr>
<td>UVGetExitCodeProcess function</td>
<td>4-7</td>
</tr>
<tr>
<td>UVPeekNamedPipe function</td>
<td>4-8</td>
</tr>
<tr>
<td>UVReadPipe function</td>
<td>4-10</td>
</tr>
<tr>
<td>UVRunCommand function</td>
<td>4-11</td>
</tr>
<tr>
<td>UVWritePipe function</td>
<td>4-12</td>
</tr>
<tr>
<td>Example</td>
<td>4-13</td>
</tr>
</tbody>
</table>

### Appendix A: PI/open GCI definitions

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Importing PI/open GCI definitions</td>
<td>A-2</td>
</tr>
<tr>
<td>GCI differences</td>
<td>A-4</td>
</tr>
</tbody>
</table>

### Appendix B: Example programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supplied GCI programs</td>
<td>B-2</td>
</tr>
<tr>
<td>*hello</td>
<td>B-2</td>
</tr>
<tr>
<td>multiply</td>
<td>B-2</td>
</tr>
<tr>
<td>*gci3</td>
<td>B-3</td>
</tr>
<tr>
<td>gci4</td>
<td>B-3</td>
</tr>
<tr>
<td>System calls</td>
<td>B-4</td>
</tr>
<tr>
<td>Example GCI programs</td>
<td>B-5</td>
</tr>
<tr>
<td>Interluded system call</td>
<td>B-5</td>
</tr>
<tr>
<td>Arrays in C</td>
<td>B-9</td>
</tr>
<tr>
<td>Arrays in FORTRAN</td>
<td>B-11</td>
</tr>
</tbody>
</table>
Chapter 1: Using the GCI

What is the GCI? ........................................... 1-3
What you need to do ..................................... 1-4
Preparing the subroutine ............................. 1-5
Defining the subroutine ................................ 1-6
  Adding a subroutine definition record ............. 1-6
Adding the subroutine to UniVerse ................. 1-10
  Building a new run file on UNIX .................. 1-10
  Making a GCI library on Windows platforms ...... 1-13
This chapter covers the following topics:

- Introduction to the GCI
- An overview of how to use the GCI
- Preparing subroutines
- Creating subroutine definitions
- Adding subroutines to UniVerse
What is the GCI?

The General Calling Interface (GCI) acts as a gateway from UniVerse BASIC to an external subroutine. The GCI passes data to and from subroutines through arguments and arrays.

The GCI allows UniVerse BASIC programs to make:

- Calls to external subroutines written in FORTRAN 77, C, and C++.¹
- System calls to operating system commands. Some examples of these are provided when you first install the GCI. For more information, see Appendix B, “Appendix B: Example programs.”

You can also use the GCI to catalog a subroutine so that it can be accessed from an I-descriptor or run from the UniVerse prompt.

The GCI comes with UniVerse. On Windows platforms, the GCI is ready to use. On UNIX systems, you must install the GCI using the Package option of the System Administration menu before you can use it.

**Note:** Microsoft Windows does not support mixing 32-bit and 64-bit applications. As a result, gci dlls used with 32-bit version of UniVerse must be 32-bit. Dlls used with gci on the 64-bit version of UniVerse must be compiled and linked as 64-bit.

¹. In this manual, all references to C also apply to C++.
What you need to do

This section gives an overview of the steps you take before you can use the GCI to call an external subroutine from a UniVerse BASIC program. You must be logged on as a UniVerse Administrator to perform these steps.

1. Prepare the subroutine: write, compile, and test it, then copy the source into the gcidir directory in the UV account directory.
2. Add to the GCI definition file a record defining the subroutine.
3. Add the subroutine to UniVerse. This process varies according to the operating system you are using:
   - **On UNIX systems**: Build a new UniVerse run file (uvsh) that incorporates the subroutine.
   - **On Windows Platforms**: Create a dynamic link library (DLL) for loading when UniVerse starts up.

The following sections describe these steps in more detail.
Preparing the subroutine

The GCI supports subroutines written in FORTRAN 77, C, and C++. For information about specifying data types, handling arrays, and so forth, see Chapter 3, “Chapter 3: GCI subroutines.” Note that compilers vary on different operating systems.

Take care when naming the subroutine. You should not use UniVerse reserved words or any term that is a record ID in the VOC file. If you want to call the subroutine as a cataloged subroutine or as a function using the DEFFUN statement, you must use $, –, *, or ! as the first character of the subroutine name.

On NLS-enabled systems: Use the correct map for GCI subroutines. Also, use the correct data type for strings containing multibyte characters. Use the SET.GCI.MAP command to set a map for GCI subroutines. For more information, see the UniVerse NLS Guide.

When the subroutine is complete:

On UNIX systems: Copy the source module to the gcidir directory in the UV account directory.

On Windows Platforms: Use LIB or LINK to create a library file from the source module, then copy the library into the gcidir directory in the UV account directory.
Defining the subroutine

Before you can use the subroutine, you must define it to UniVerse. By default, UniVerse holds external subroutine definitions in the GCI file in the UV account. On Windows platforms, you can create and use other GCI definition files.

The subroutine definition contains the following:

- The name of the subroutine
- The language in which it was written
- The number and type of arguments passed

The GCI uses this information to convert any data that is passed into the correct data type for the receiving program.

When it is first installed, the GCI definition file contains example definitions for some simple C subroutines and system calls. For more information, see Appendix B, “Appendix B: Example programs.”

Adding a subroutine definition record

To add a subroutine definition record, follow these steps:

1. Choose Package -> Gci administration from the UniVerse System Administration menu. or enter GCI.ADMIN at the UniVerse prompt to invoke the GCI Administration menu. You must be a UniVerse Administrator to use this command.
2. Choose **Add, Modify, and Delete GCI subroutines**. On Windows platforms, enter the name of the GCI definition file at the prompt. You see the **GCI Maintenance** menu shown in the following example, and you enter information about the subroutine at the prompt. The following section explains what to enter in each field.

**Note:** The subroutine definition must match its intended use. For example, if you expect to call the subroutine as a function, you should specify a return value other than `void`.

![GCI Maintenance menu](image)

**Subroutine name:**

Enter the name of the subroutine to be used by the calling program. If you want to call the subroutine as a UniVerse BASIC subroutine or by using the DEFFUN statement, you must use $, -, *, or ! as the first character of the subroutine name to ensure it is cataloged. Subroutines to be defined as functions using the DECLARE GCI statement must not have this prefix character, as they should not be cataloged.
1. Language:

Enter the programming language you used to write the subroutine. You should enter either `c` (for C or C++) or `f77` (for FORTRAN). If you do not enter a value, it defaults to C.

2. External name:

Enter the external name of the subroutine, that is, the name that would be used to call the subroutine in C or FORTRAN. If you do not enter a value, it defaults to the value you entered at the Subroutine name: prompt.

3. Module name:

*On UNIX systems:* Enter the name of the module containing the subroutine, that is, the name of the file holding the subroutine, without its suffix. For example, for a subroutine stored in a file called `progs.c`, enter the module name `progs`. If you do not supply a value, it defaults to the value you entered at the Subroutine name: prompt.

*On Windows platforms:* Enter the name of the library file that you created in the `gcidir` directory in “Preparing the subroutine” on page 5, without its `.lib` suffix. For example, if your library name is `gci_subs.lib`, enter the module name `gci_subs`. If you do not supply a value, it defaults to the value you entered at the Subroutine name: prompt. If you want to define a system call to functions defined in the Microsoft Win32 API, you must specify the module name as Win32.

4. Description:

You can enter a short description of the subroutine using up to 50 characters. This field is optional.

5. Number of arguments:

Enter the total number of arguments. Enter 0 (zero) if there are no arguments. If you specify arguments for the subroutine, enter details for each one at prompt 7. If you specify that there are no arguments, you do not see prompts 7.1, 7.2, and so forth.
6. Return value:

Enter the data type of the value returned by the subroutine. Specify **void** if you intend to call the subroutine as a UniVerse BASIC subroutine, or if it is a FORTRAN 77 subroutine. For subroutines written in C that you intend to call as functions, you can declare the return value as any of the C data types listed in Chapter 3, “Chapter 3: GCI subroutines.”

**Note:** If you specify **void**, the GCI assumes that there is no return value. If the subroutine does return a value, it is ignored.

7. Argument details

This prompt appears only if you specified at prompt 5 that the subroutine has arguments. The details you need to supply for each argument are as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Direction</th>
<th>Data Type</th>
<th>Length</th>
<th>Rows</th>
<th>Cols</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specify <strong>I</strong> for input, <strong>O</strong> for output, or <strong>B</strong> for both.</td>
<td>Specify the data type for the argument, for example, <strong>int</strong>. (For a list of data types supported by the GCI, see Chapter 3, “Chapter 3: GCI subroutines”).</td>
<td>If you specified a data type of <strong>lchar</strong>, <strong>charvar</strong>, or <strong>character</strong>, enter the length of the character string at the prompt.</td>
<td>If you are defining an array argument, enter the number of rows here. (You only see this prompt if the data type you specified is one that allows arrays.)</td>
<td>If you are defining an array argument, enter the number of columns here. (You only see this prompt if the data type you specified is one that allows arrays, and you specified a value greater than 0 for the number of rows.)</td>
<td>You can specify a short description of the argument using up to 15 characters. This field is optional.</td>
</tr>
</tbody>
</table>

When you have completed the subroutine definition, remember to save it.
Adding the subroutine to UniVerse

When you have defined the subroutine, as described in the previous section, you must add it to UniVerse. You can do this in two ways:

- Using the GCI Administration menu. This method is appropriate for most C subroutines.
- Manually from the operating system. You must use this method for FORTRAN subroutines or for C routines that have special compiler requirements.

The procedure for adding the subroutine depends on the operating system you are using:

- For UNIX systems, read Building a new run file on UNIX.
- For Windows platforms, see “Making a GCI library on Windows platforms” on page 13.

Building a new run file on UNIX

When you want to add an external subroutine to UniVerse on a UNIX system, you must rebuild the UniVerse run file (uvsh). This means that only a defined set of subroutines can be called through the GCI for any particular version of the uvsh run file. The following sections describe how you can build and install a new UniVerse run file automatically from the GCI Administration menu, or manually from the UNIX shell prompt.

Automatically building the run file

To build a new uvsh run file, choose Make a new UniVerse from the GCI Administration menu. The following message appears:

This procedure will generate a new gci.c program and make a new 'UniVerse' in the file /usr/ibm/uv/uvsh.new.
It will catalog any subroutines which require cataloging.

Are you sure you want to continue? (Y/N)

When you answer Y (yes) to the prompt, the GCI does the following:

- Catalogs the new subroutines if they have a prefix of $, -, *, or !
- Modifies the gci.c module in the gcidir directory to call the added subroutines
- Creates the Makefile
- Runs the Makefile to create a new run file called uvsh.new

You can test the new run file before you install it by running the uvsh.new file. For example:

```
$ /usr/ibm/uv/uvsh.new
```

You can then test the UniVerse BASIC programs that call the added subroutine and debug them if necessary.

To install the run file from the GCI Administration menu, choose Install new UniVerse. When you install the new run file, the following happens:

- The old run file, /usr/uv/bin/uvsh, is copied to /usr/uv/bin/uvsh.save.
- The new run file, /usr/uv/uvsh.new, is copied to /usr/uv/bin/uvsh.

**Note:** If any other users on your system are running UniVerse when you install the new run file, they continue to run the old run file until they log out and then log back on again.

**Manually building the run file**

This section tells you how to build a new uvsh run file from the UNIX shell prompt.

**Note:** You must use this method to add FORTRAN subroutines.

1. Choose **List GCI Subroutines** from the **GCI Administration** menu to check that you have completed definitions for each of the subroutines you want to add to the GCI, as described earlier in “Adding a subroutine definition record” on page 6.
2. Check that you have a Makefile. Change to the gcidir directory in the UV account directory (for example, /usr/uv/gcidir). List the directory, and look for the gci.c and Makefile files. If there is no Makefile, create one using the following command:

\$ cp Make.gci Makefile

Then make a new gci.c file that includes your new subroutines using the following command:

\$ make gci

3. Edit the file gcidir/Makefile as follows:

- Add the object files for the new subroutines to the GCILIB variable. For example:
  
  GCILIB=gci_mult.o

- Update the GCI Makefile as necessary if your program has any special requirements, for example, nonstandard C libraries or compilers. If you want to use a nonstandard C compiler, add compilation rules for each object file to the end of the Makefile. For example:
  
  routine_1.o:
  c89–croutine_1.c –I/include.file –Ddefine/token

- Add any FORTRAN library-loading options used by your system to the LIBES variable. (See the FORTRAN 77 manual provided with your system.)

- Add any FORTRAN compiler options used by your system to the F77FLAGS variable.

  Note: For Hewlett-Packard systems you must also add the F77FLAGS option as follows:

  F77FLAGS = +E3 –c

4. Make the new run file with the make command. If you change to the /usr/uv directory and list it, you see the file uvsh.new that you just created. You can test the new run file before you install it by running the uvsh.new file. For example:

\$ /usr/uv/uvsh.new

You can then test the UniVerse BASIC programs that call the added subroutines and debug them if necessary.
You can install the new run file automatically by choosing Install New UniVerse from the GCI Administration menu. To install the run file manually, follow these steps:

1. At the UNIX shell prompt, change to the bin directory in the UV account directory. For example:
   $$ cd /usr/uv/bin $$
2. Save the old run file by moving it to another file. For example:
   $$ mv uvsh uvsh.save $$
3. Copy the new run file to the old file. For example:
   $$ cp uvsh.new uvsh $$

This completes the procedure for adding a GCI subroutine to UniVerse on UNIX systems. See Chapter 2, “Chapter 2: The calling program,” for details of how to call a GCI subroutine from a UniVerse BASIC program.

Making a GCI library on Windows platforms

On Windows platforms, when you have created a GCI definition record for the subroutine (as described in “Adding a subroutine definition record” on page 6) you must add the subroutine to UniVerse. On Windows platforms, you add the subroutine to UniVerse by turning the GCI definition file into a DLL (dynamic link library). You then install the DLL into UniVerse and add it to the list of DLLs in the Windows Registry.

Adding a library file

When you have created the library file, you can add the subroutine in two ways:

- Automatically, using the GCI Administration menu. This method is suitable for most C subroutines where you are using the Microsoft C compiler and linker to build the DLL.
- Manually, using MS-DOS and UniVerse commands. You must use this method for FORTRAN 77 subroutines or if you are not using the Microsoft C compiler or linker.
Automatically building the DLL

To build the GCI DLL automatically, choose Make a GCI Library from a GCI Definition File from the GCI Administration menu. Enter the name of the GCI definition file at the prompt, then confirm the action.

Note: This option fails if you do not have an appropriate compiler installed. See “Preparing the subroutine” on page 5.

You can test the DLL before you install it by creating a Windows environment variable called UVCIDLLS containing a list of library names, separated by semicolons. The library names must be either a full path or a path relative to the UV account directory. When UniVerse starts, it searches this local list before looking at the system list of GCI DLLs.

To install the DLL, choose Install a GCI Library from the GCI Administration menu. This option does the following:

- Copies the DLL file from the gcdir directory to the bin directory in the UV account directory
- Adds the name of the copied file to the GCI library list held in the Windows Registry

The DLL is now ready for use.

Manually building the DLL

To build a GCI library using UniVerse commands and MS-DOS commands, follow these steps:

1. Catalog the subroutine (if you want to call it through catalog space) using the following UniVerse command syntax:

   `RUN APP.PROGS CATLG.GCI filename`

   `filename` is the name of the GCI definition file containing the subroutine definition.
2. Create a makefile in the gcidir directory using the following UniVerse command syntax:

   **RUN APP.PROGS GCI_MAKEFILE filename**

   *filename* is the name of the GCI definition file containing the subroutine definition.

   This command generates a makefile to run with the Microsoft *nmake* command and the Microsoft C compiler and linker. If you want to use a different compiler, you must now edit the makefile to specify the utilities you want to use.

3. Generate the conversion module using the following UniVerse command syntax:

   **RUN APP.PROGS GEN.GCI filename**

   *filename* is the name of the GCI definition file containing the subroutine definition.

   This generates a C source file in the gcidir directory with a name in the format *filename*.c.

4. From an MS-DOS window, compile and link the conversion module to generate the library file.

5. Test the DLL by creating a Windows environment variable called UVGCIDLLS containing a list of library names separated by semicolons. The library names must be either a full path or a path relative to the UV account directory. When UniVerse starts, it searches this local list before looking at the system list of GCI DLLs.

6. Install the DLL. You can do this in one of two ways:

   - Use the Install a GCI Library option from the GCI Administration menu, as described in “Automatically building the DLL” on page 14.
   - Install the library manually by copying the DLL file from the gcidir directory to the bin directory in the UV account directory. Use the Edit the Standard GCI Library List option from the GCI Administration menu to add the DLL to the system list of GCI DLLs.

**Using GCI libraries**

UniVerse accesses GCI libraries in one of two ways:
Locally, through the UVGCIDLLS environment variable

Globally, through the Windows Registry

The UVGCIDLLS environment variable is used as described in step 5 under "Manually building the DLL" on page 14.

When a GCI library is installed from the GCI Administration menu, an entry for it is added to the list of GCI DLLs in the Windows Registry. You can modify this list or add further entries by choosing Edit the Standard GCI Library List from the GCI Administration menu.

You can use a GCI library on a Windows system that does not have the GCI installed by following these steps:

1. Copy the DLL file to the bin directory of the UV account directory.
2. Update the Windows Registry using the following UniVerse command syntax:

   **RUN APP.PROGS GCI.NTINST.B filename**

   *filename* is the name of the DLL file.
Chapter 2: The calling program

Direct calls .................................................. 2-3
Indirect calls ................................................. 2-4
Function calls ............................................... 2-5
Declaring a function ....................................... 2-6
Passing arguments ......................................... 2-7
Accessing the system errno variable ................. 2-8
# Chapter 3: GCI subroutines

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX file units</td>
<td>3-3</td>
</tr>
<tr>
<td>C data types</td>
<td>3-4</td>
</tr>
<tr>
<td>Data types for Windows system calls</td>
<td>3-5</td>
</tr>
<tr>
<td>Allocating memory for character strings</td>
<td>3-5</td>
</tr>
<tr>
<td>Converting data types between C and BASIC</td>
<td>3-6</td>
</tr>
<tr>
<td>C arrays</td>
<td>3-6</td>
</tr>
<tr>
<td>FORTRAN data types</td>
<td>3-7</td>
</tr>
<tr>
<td>FORTRAN arrays</td>
<td>3-8</td>
</tr>
<tr>
<td>FORTRAN portability</td>
<td>3-9</td>
</tr>
<tr>
<td>Data types for multibyte characters</td>
<td>3-10</td>
</tr>
</tbody>
</table>
This chapter describes how to call a GCI subroutine from a UniVerse BASIC program by:

- Using the CALL statement to call the subroutine directly
- Assigning the subroutine name to a variable and using the CALL statement to call it indirectly
- Declaring it as a function using the DEFFUN statement
- Declaring it as a GCI function using the DECLARE GCI statement

The following sections show examples of these methods. Note the following general points when you write your calling program:

- You can call the GCI subroutine as many times as required from the same program.
- You cannot call a GCI subroutine directly from another GCI subroutine; you must return to the main program first (but see the next point).
- If you declare a routine in one program, you can call it from other programs linked to the first one through $INCLUDE or $CHAIN without redeclaring it.
Direct calls

To make a direct call to a subroutine, use the CALL statement. You can call one of the example subroutines supplied with the GCI using the following command:

    CALL *hello

For more information about this subroutine, see Appendix B, “Appendix B: Example programs.”

The following example directly calls a subroutine named $TEST which has three arguments, A, B, and C, and returns void:

    CALL $TEST(A,B,C)
Indirect calls

To call the same subroutine indirectly, use this example:

```
    SUB = "$TEST"
    .
    .
    .
    CALL @SUB(A, B, C)
```
Function calls

The following example calls a subroutine called FUNC which has three arguments and returns an int:

```
DEFFUN TEST.FUNC(A, B, C) CALLING "FUNC"
ANSWER = TEST.FUNC(A, B, C)
```

The $TEST subroutine described earlier can also be called as a function, using the DEFFUN statement, as follows:

```
DEFFUN TEST.FUNCTION(B, C) CALLING "$TEST"

ANSWER = TEST.FUNCTION(B, C)
```

In the last example the value returned by TEST.FUNCTION is the first argument to the subroutine.
Declaring a function

The following example shows the DECLARE GCI statement used to declare one of the C subroutines supplied with the GCI. See also Appendix B, “Appendix B: Example programs.”

```
DECLARE GCI multiply
   ...
   x = multiply(i, j);* call multiply routine to get the answer
```

*Note: DECLARE GCI cannot be used with cataloged subroutines (that is, any subroutines prefixed with $, −, *, or !).
Passing arguments

If your subroutines have arguments, your CALL statement must specify them, as shown in the examples in the previous sections. An argument can be any valid UniVerse BASIC expression that can be converted into a data type that the subroutine recognizes. For lists of valid data types, see Chapter 3, “Chapter 3: GCI subroutines.”

All arguments returned from a GCI subroutine to a UniVerse BASIC program must be variables.
Accessing the system *errno* variable

Most operating system calls return a value indicating success or failure. In the case of a failure, the external variable *errno* holds a further value indicating the reason for failure.

If you want to make system calls directly through the GCI, or if your subroutine makes a system call, you can access this variable by using the UniVerse BASIC !ERRNO subroutine. It has the following syntax:

```
CALL !ERRNO (variable)
```

*variable* is the name of a UniVerse BASIC variable. This returns the value of *errno* that was captured immediately after your GCI subroutine was called and stores it in *variable*. The system include file *errno.h* lists the values of *errno* that apply to your system.
This chapter gives details of the following:

- File units in GCI subroutines
- Data types and array handling in C subroutines
- Data types and array handling in FORTRAN subroutines
- Data types for multibyte characters
UNIX file units

On UNIX systems, the operating system limits the number of file units that can be held open simultaneously by the system and by each user. If your GCI subroutine requires a large number of open file units, you can raise the operating system limit.
### C data types

The following table shows the GCI data types that you must specify in your GCI subroutine, and how they map to the C data types that you use in your program.

<table>
<thead>
<tr>
<th>GCI Data Type</th>
<th>Description</th>
<th>C Data Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Single character</td>
<td>char</td>
<td>I</td>
</tr>
<tr>
<td>char*</td>
<td>Pointer to character string</td>
<td>char*</td>
<td>O or B</td>
</tr>
<tr>
<td>pchar*</td>
<td>Pointer to character string</td>
<td>char*</td>
<td>I</td>
</tr>
<tr>
<td>tchar*</td>
<td>Pointer to character string</td>
<td>char**</td>
<td>O or B</td>
</tr>
<tr>
<td>lchar*</td>
<td>Pointer to character string</td>
<td>char**</td>
<td>I, O, or B</td>
</tr>
<tr>
<td>charvar*</td>
<td>Pointer to character varying string</td>
<td>charvar*</td>
<td>I, O, or B</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>int</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int*</td>
<td>O or B</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
<td>long</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long*</td>
<td>O or B</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
<td>short</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>short*</td>
<td>O or B</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>float</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>float*</td>
<td>O or B</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>double</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double*</td>
<td>O or B</td>
</tr>
<tr>
<td>void</td>
<td>Void</td>
<td>void</td>
<td>return only</td>
</tr>
</tbody>
</table>
**Data types for Windows system calls**

Use the data types in the following table to make system calls from C subroutines to functions defined in the Win32 API. These all have uppercase names to match the values in the standard include file `windows.h`.

<table>
<thead>
<tr>
<th>GCI Data Type</th>
<th>Description</th>
<th>Win32 Data Type</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>Integer value used for true or false: 1 is true; 0 is false</td>
<td>BOOL</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LPBOOL</td>
<td>O or B</td>
</tr>
<tr>
<td>BYTE</td>
<td>Unsigned 8-bit character</td>
<td>BYTE</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LPBYTE</td>
<td>O or B</td>
</tr>
<tr>
<td>WORD</td>
<td>Unsigned 16-bit integer</td>
<td>WORD</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LPWORD</td>
<td>O or B</td>
</tr>
<tr>
<td>DWORD</td>
<td>Unsigned 32-bit integer</td>
<td>DWORD</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LPDWORD</td>
<td>O or B</td>
</tr>
</tbody>
</table>

---

**Allocating memory for character strings**

UniVerse BASIC expects variables to have memory space allocated for them. This is achieved in different ways according to the data type you use for the variable, as shown in the following list:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Memory Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pchar*</td>
<td>The GCI uses the memory used by the string. For example if the string <code>abcde</code> is input to the GCI routine, the maximum size for output is 5.</td>
</tr>
<tr>
<td>tchar*</td>
<td>The GCI assumes <code>malloc</code> allocates the memory within the GCI routine. The example routine <code>gci_malloc.c</code> uses <code>malloc</code> in this way.</td>
</tr>
<tr>
<td>lchar*</td>
<td>The GCI uses the length defined for the subroutine in the GCI file to determine how much memory to allocate.</td>
</tr>
<tr>
<td>charvar*</td>
<td>Memory is allocated based on the length defined for the subroutine in the GCI file. The length of the string is stored in a separate word attached to the beginning of the string.</td>
</tr>
</tbody>
</table>
Converting data types between C and BASIC

Note the following points when converting data types:

- The length specified in the GCI definition determines the amount of space allocated for character strings of type lchar* or charvar*.
- The GCI optionally supports arrays for the following data types. They can be input, output, or input/output.
  - short
  - long
  - int
  - float
  - double

C arrays

An array with a maximum of two dimensions can be passed to a C subroutine as long as it satisfies the following conditions:

- The array elements must be numeric or convertible to numeric.
- For the C subroutine, the GCI supports only arrays of type short integer, long integer, float, or double.
- The UniVerse BASIC array must match that of the expected argument in the GCI template in both size and dimensions, otherwise a conversion error occurs and the call is aborted.
FORTRAN data types

The following table shows the FORTRAN 77 data types supported by the GCI and all the possible conversions of UniVerse BASIC data types to FORTRAN 77 data types. The following sections give more information about each data type.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Numeric</th>
<th>Nonnumeric</th>
<th>Array</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer2</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>I, O, B</td>
</tr>
<tr>
<td>integer4</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>I, O, B</td>
</tr>
<tr>
<td>real4</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>I, O, B</td>
</tr>
<tr>
<td>real8</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>I, O, B</td>
</tr>
<tr>
<td>logical</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>I, O, B</td>
</tr>
<tr>
<td>character</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>I, O, B</td>
</tr>
</tbody>
</table>

**FORTRAN77 Data Types**

*Note:* All FORTRAN 77 data types are pass-by-reference, and as such all arguments can be input/output. The GCI does not support FORTRAN 77 function return values.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>All numeric data and wholly numeric strings can be converted to integer. The data conversion is aborted if it encounters a string containing a nonnumeric character.</td>
</tr>
</tbody>
</table>

**FORTRAN77 Data Types**
FORTRAN arrays

UniVerse BASIC stores two-dimensional arrays in row-major order with the rightmost subscript changing most rapidly. FORTRAN 77 stores arrays in column-major order. For example, consecutive elements in a UniVerse BASIC array are (1,1) and (1,2). If you want to keep the same order when passing a two-dimensional array to a FORTRAN 77 subroutine, you must reverse the dimensions and subscripts.

An array with a maximum of two dimensions can be passed to a FORTRAN 77 subroutine as long as it satisfies the following conditions:

- The array elements must be numeric or convertible to numeric.
- For the FORTRAN 77 subroutine, the GCI supports only arrays of type integer2, integer4, real4, real8, character, or logical.
- The UniVerse BASIC array must match that of the expected argument in the GCI template in both size and dimensions, otherwise a conversion error occurs and the call is aborted.

For an example of passing an array from a UniVerse BASIC program to a FORTRAN subroutine, see Appendix B, “Appendix B: Example programs.”
FORTRAN portability

FORTRAN 77 programs are not as portable as C programs. If you want to use your FORTRAN subroutines on a different system, or if you want to use a different compiler from that for which they were originally written, you should test them before trying to run them through the GCI. Note especially that the LOGICAL data type may have the reverse meaning under a different compiler.
Data types for multibyte characters

If NLS mode is enabled, use the GCI data types in the following table to specify multibyte characters.

<table>
<thead>
<tr>
<th>GCI Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wchar_t*</td>
<td>Pointer to wchar.</td>
</tr>
<tr>
<td>pwchar_t*</td>
<td>Pointer to preallocated string memory.</td>
</tr>
<tr>
<td>twchar_t*</td>
<td>Pointer to character memory allocated by the subroutine.</td>
</tr>
<tr>
<td>lwchar_t*</td>
<td>Pointer to character memory allocated by the GCI.</td>
</tr>
<tr>
<td>wchar_tvar*</td>
<td>Pointer to a string type. Memory is allocated to the character length in the first word of the buffer.</td>
</tr>
</tbody>
</table>

Use these data types to accommodate wide character data when you work with Unicode or an external double-byte character set in C. For more information about writing client programs in NLS mode, see the UniVerse NLS Guide.
Chapter 4: GCI functions

- UVClosePipe function ........................................ 4-3
- UVCreatePipe function ........................................ 4-4
- UVCreateProcess function .................................... 4-5
- UVGetExitCodeProcess function .............................. 4-7
- UVPeekNamedPipe function .................................... 4-8
- UVReadPipe function ........................................... 4-10
- UVRunCommand function ....................................... 4-11
- UVWritePipe function .......................................... 4-12
- Example ......................................................... 4-13
This chapter discusses GCI functions that create, read, write, or manage pipe and or child processes from a UniVerse BASIC program.
UVClosePipe function

Syntax

UVClosePipe(pipe_handle)

Description

The UVClosePipe function closes a pipe previously created by the UVCreatePipe function.

*Note:* This function is supported on UniVerse for Windows platforms only.

Parameter

The following table describes the parameter of the syntax.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipe_handle</td>
<td>pipe_handle can be either the readPipe_handle or the writePipe_handle returned by a previously executed UVCreatePipe function.</td>
</tr>
</tbody>
</table>

UVClosePipe Parameter

Return codes

The following table describes the return codes of the UVClosePipe function.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

UVClosePipe Return Codes
UVCreatePipe function

Syntax

UVCreatePipe(readPipe_handle, writePipe_handle)

Description

The UVCreatePipe function creates an anonymous pipe, and returns the handles used to access the read and write ends of the pipe to your program. You must execute the UVCreatePipe function prior to using any of the other functions.

*Note*: The UVCreatePipe function is supported on Windows platforms only.

Parameters

The following table describes each parameter of the syntax.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPipe_handle</td>
<td>The handle to the READ end of the pipe that was created.</td>
</tr>
<tr>
<td>writePipe_handle</td>
<td>The handle to the WRITE end of the pipe that was created.</td>
</tr>
</tbody>
</table>

UVCreatePipe Parameters

Return codes

The following table describes the return codes of the UVCreatePipe function.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

UVCreatePipe Return Codes
UVCreateProcess function

Syntax

`UVCreateProcess(command, input_handle, output_handle, error_handle, pid, child_handle)`

Description

The UVCreateProcess function creates a new process that executes the `command` you specify.

Parameters

The following table describes each parameter of the syntax.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>The command you want to execute.</td>
</tr>
<tr>
<td>input_handle</td>
<td>Specifies a handle UniVerse uses as the standard input handle to the process. This parameter can be a <code>readPipe_handle</code> to a pipe created by the <code>UVCreatePipe function</code>.</td>
</tr>
<tr>
<td>output_handle</td>
<td>Specifies a handle UniVerse uses as the standard output handle for the process. This parameter can be a <code>writePipe_handle</code> to a pipe created by the UVCreatePipe function.</td>
</tr>
<tr>
<td>error_handle</td>
<td>Specifies a handle UniVerse uses as the standard error handle for the process. This parameter can be a <code>writePipe_handle</code> created by the UVCreatePipe function.</td>
</tr>
<tr>
<td>pid</td>
<td>Specifies a variable to receive the process ID created by this function.</td>
</tr>
<tr>
<td>child_handle</td>
<td>Specifies the variable to receive the handle to the newly created function.</td>
</tr>
</tbody>
</table>
Return codes

The following table describes the return codes of the UVCreateProcess function.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

UVCreateProcess Return Codes
UVGetExitCodeProcess function

Syntax

UVGetExitCodeProcess(child_handle, exit_code)

Description

The UVGetExitCodeProcess function retrieves the termination status of the process ID you specify.

Parameters

The following table describes each parameter of the syntax.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>child_handle</td>
<td>The handle to the process you are interrogating. This parameter can be the child_handle for a process created by the UVCreateProcess function.</td>
</tr>
<tr>
<td>exit_code</td>
<td>Specifies a variable to receive the process termination status.</td>
</tr>
</tbody>
</table>

UVGetExitCodeProcess Parameters

Return codes

The following table describes the return codes of the UVGetExitCodeProcess function.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

UVGetExitCodeProcess Return Codes
UVPeekNamedPipe function

Syntax

UVPeekNamedPipe(pipe_handle, readPipe_buffer, buffer_size, numberOfBytesRead, totalBytesAvailable, bytesLeftThisMessage)

Description

The UVPeekNamedPipe function copies data from a named or anonymous pipe into a buffer, without removing the data from the pipe. This function also returns information about the number of bytes read from the pipe, the number of bytes available to be read from the pipe, and the number of bytes left in the current message in the pipe.

Note: The UVPeekNamedPipe function is supported on Windows platforms only.

Parameters

The following table describes each parameter of the syntax.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipe_handle</td>
<td>The \texttt{readpipe_handle} to a pipe created by the \texttt{UVCreatePipe} function.</td>
</tr>
<tr>
<td>readPipe_buffer</td>
<td>Specifies the variable to receive the data read from the pipe. If you do not want to read data from the pipe, set this value to 0.</td>
</tr>
<tr>
<td>buffer_size</td>
<td>The size of the \texttt{readPipe_buffer}, in bytes, to be read. If the value of \texttt{readPipe_buffer} is 0, UniVerse ignores this parameter.</td>
</tr>
</tbody>
</table>

UVPeekNamedPipe Parameters
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>number_bytes_read</code></td>
<td>Specifies the variable to receive the number of bytes read from the pipe. If you do not want to read data from the pipe, set this value to 0.</td>
</tr>
<tr>
<td><code>total_bytes_available</code></td>
<td>Specifies the variable to receive the total number of bytes available to be read from the pipe. If you do not want to read data from the pipe, set this value to 0.</td>
</tr>
<tr>
<td><code>bytes_left_this_message</code></td>
<td>Specifies the variable to receive, the number of bytes remaining in this message. UniVerse sets this value to 0 for the pipe created using the <code>UVCreatePipe</code> function, or when no data is to be read.</td>
</tr>
</tbody>
</table>

#### UVPeekNamedPipe Parameters (Continued)

## Return codes

The following table describes the return codes of the UVPeekNamedPipe function.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

### UVPeekNamedPipe Return Codes
UVReadPipe function

Syntax

UVReadPipe(readPipe_handle, readPipe_buffer, readPipe_buffer_size)

Description

The UVReadPipe function reads data from a pipe previously created by the UVClosePipe function.

Note: The UVReadPipe function is supported on Windows platforms only.

Parameters

The following table describes each parameter of the syntax.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPipe_handle</td>
<td>The handle to the READ end of the pipe.</td>
</tr>
<tr>
<td>readPipe_buffer</td>
<td>Specifies the variable where Universe will store the data read from the pipe.</td>
</tr>
<tr>
<td>readPipe_buffer_size</td>
<td>The number of bytes to read from the pipe.</td>
</tr>
</tbody>
</table>

UVReadPipe Parameters

Return codes

The following table describes the return codes of the UVReadPipe function.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

UVReadPipe Return Codes
UVRunCommand function

Syntax

UVRunCommand(command)

Description

The UVRunCommand function executes a Windows executable. You can specify the executable name and its argument as a string.

The following example shows how to execute the command:

    UVRunCommand("c:\WINDOWS\system32\cmd.exe /c dir")

You must use single or double quotation marks around the string argument.
UVWritePipe function

Syntax

UVWritePipe(writePipe_handle, writePipe_buffer)

Description

The UVWritePipe function writes data to a pipe previously created by the UVClosePipe function.

*Note: The UVWritePipe function is supported on Windows platforms only.*

Parameters

The following table describes each parameter of the syntax.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>writePipe_handle</td>
<td>The handle to the WRITE end of the pipe.</td>
</tr>
<tr>
<td>writePipe_buffer</td>
<td>The expression specifying the data to write to the pipe.</td>
</tr>
</tbody>
</table>

UVWritePipe Parameters

Return codes

The following table describes the return codes of the UVWritePipe function.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>-1</td>
<td>Failure</td>
</tr>
</tbody>
</table>

UVWritePipe Return Codes
Example

The following example illustrates the use of each GCI function described in this chapter.

```
DECLARE GCI UVCreatePipe
DECLARE GCI UVWritePipe
DECLARE GCI UVReadPipe
DECLARE GCI UVClosePipe
DECLARE GCI UVSeekNamedPipe
DECLARE GCI UVCaptureProcess
DECLARE GCI UVCaptureExitCodeProcess

StdInPipeRead = 0
StdInPipeWrite = 0

* Open an input/output pipe and obtain the handles

return_value = UVCreatePipe(StdInPipeRead,StdInPipeWrite)
If return_value < 0 then
    crt 'Error creating StdIn pipe'
    stop
End

crt "StdInPipeRead handle = ": StdInPipeRead

crt "StdInPipeWrite handle = ": StdInPipeWrite

StdOutPipeRead = 0
StdOutPipeWrite = 0

* Open an input/output pipe and obtain the handles

return_value = UVCreatePipe(StdOutPipeRead,StdOutPipeWrite)
If return_value < 0 then
    crt 'Error creating StdOut pipe'
    stop
End

crt "StdOutPipeRead handle = ": StdOutPipeRead

crt "StdOutPipeWrite handle = ": StdOutPipeWrite

* Attempt to create an external process

PID = 0
Child = 0

cmd = "c:\winnt\system32\cmd.exe"

return_value = UVCaptureProcess(cmd,StdInPipeRead,StdOutPipeWrite,0,PID,Child)
```
If return_value < 0 then
    crt 'Error creating child process'
    stop
End

crt 'PID = ': PID : ': Child
return_value = UVGetExitCodeProcess(Child, ExitCode);
If return_value < 0 then
    crt 'Error getting child process status'
    stop
End

crt 'ExitCode = ': ExitCode

* Write test message to pipe
return_value = UVWritePipe(StdInPipeWrite, 'DIR':CHAR(13):CHAR(10));

Sleep 1

* Read Output from pipe
TotalBytesAvail = 0

LOOP
    return_value = UVPeekNamedPipe(StdOutPipeRead, 0, 0, 0, TotalBytesAvail, 0);
    If return_value < 0 then
        crt 'Error peeking StdOut read pipe'
        stop
    End

crt "TotalBytesAvail = ": TotalBytesAvail
UNTIL TotalBytesAvail EQ 0

return_value = UVReadPipe(StdOutPipeRead, buffer, 1024);
if return_value < 0 then
    crt 'Error reading StdOut pipe'
    stop
End

crt buffer

REPEAT

* Write test message to pipe
return_value = UVWritePipe(StdInPipeWrite, 'exit':CHAR(13):CHAR(10));

Sleep 1

return_value = UVGetExitCodeProcess(Child, ExitCode);
If return_value < 0 then
   crt 'Error getting child process status'
   stop
End

crt "ExitCode = " : ExitCode

* Close the pipe we just used
return_value = UVClosePipe(StdInPipeRead)
If return_value < 0 then
   crt 'Error closing StdIn read pipe'
   stop
End

return_value = UVClosePipe(StdInPipeWrite)
If return_value < 0 then
   crt 'Error closing StdIn write pipe'
   stop
End

return_value = UVClosePipe(StdOutPipeRead)
If return_value < 0 then
   crt 'Error closing StdOut read pipe'
   stop
End

return_value = UVClosePipe(StdOutPipeWrite)
If return_value < 0 then
   crt 'Error closing StdOut write pipe'
   stop
End
End
Appendix A: PI/open GCI definitions

This appendix tells you how to import and use PI/open GCI definition files in UniVerse and explains the differences between PI/open and UniVerse GCI definitions.
Importing PI/open GCI definitions

If you want to use PI/open GCI definitions, you can import them into UniVerse, as described in the following sections.

**Warning:** As a precaution, copy your PI/open GCI definition files before you start, as this procedure is irreversible.

1. Convert your PI/open GCI definition files into UniVerse files using the following command syntax from the operating system:
   
   `pi.t30conv definition.file.pathname`

   For more information about the `pi.t30conv` command, see Moving to UniVerse from PI/open.

2. **On UNIX systems:** From the GCI Administration menu, choose **Import a PI/open definition file**, and specify the path of a PI/open GCI definition file that you converted in step 1. Repeat this for every definition file that you want to import. On UNIX systems UniVerse uses a single definition file, so it merges all your PI/open definitions into one file held in the UV account directory. The imported PI/open definitions all have a $ prefix.

   **On Windows platforms:** From the GCI Administration menu, choose **Create a GCI Definition File** to create UniVerse GCI definition files for all the PI/open definition files that you want to import. Then choose **Import a PI/open definition file**. At the prompt, enter the path of a PI/open GCI definition file that you converted in step 1, followed by the name of the target UniVerse GCI definition file that will hold the definitions.

3. Check that the subroutines are correct, and that there are no name clashes with existing subroutine definitions. Check especially that the module name field is correct, as this field does not exist in the PI/open definition, and is generated automatically from the external name field during the import process.

4. Copy the subroutines into the directory called `gcidir` in the UV account directory.

   For UNIX systems, continue with steps 5 and 6. For Windows platforms, skip to steps 7 and 8.
5. **On UNIX systems**: From the **GCI Administration** menu, choose **Make a new UniVerse** to link the GCI subroutines to the UniVerse run file.

   **Note**: If you use FORTRAN subroutines with the GCI, you must add the relevant FORTRAN libraries and compiler options for your system to the GCI Makefile before rebuilding UniVerse. Similarly, if you use any non-standard libraries in a C or C++ subroutine you should include them in the GCI Makefile. For more information about this, see **Manually building the run file** in Chapter 1, “Chapter 1: Using the GCI.”

6. Choose **Install new UniVerse** from the GCI Administration menu to install the newly created run file.

7. **On Windows platforms**: Choose **Make a GCI Library from a GCI Definition File** from the GCI Administration menu.

   **Note**: If you use FORTRAN subroutines with the GCI, you must add the relevant FORTRAN libraries and compiler options for your system to the GCI Makefile before installing the library. Similarly, if you use any non-standard libraries in a C or C++ subroutine you should include them in the GCI Makefile. For more information about this, see **Manually building the DLL** in Chapter 1, “Chapter 1: Using the GCI.”

8. Choose **Install a GCI Library** from the GCI Administration menu.
GCI differences

PI/open GCI subroutine definitions differ slightly from those described in this manual. The conversion process (described earlier) changes the PI/open definitions to match the UniVerse GCI definition format, as follows:

- The Security and ECS fields are not used in UniVerse. If your subroutine uses Extended Character Set conversions, you must make the conversions in the BASIC program using the ICONV or OCONV function before calling the GCI subroutine.
- Each subroutine is prefixed with a $ to ensure that it is cataloged automatically.
- If no return value type was defined, it is assumed to be void.
- Any numeric pointers that are input only are changed to input/output.
- PI/open GCI data types are mapped to UniVerse data types as follows:

<table>
<thead>
<tr>
<th>PI/open</th>
<th>UniVerse</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHORT-INT</td>
<td>short</td>
</tr>
<tr>
<td>LONG-INT</td>
<td>long</td>
</tr>
<tr>
<td>DOUBLE*</td>
<td>double</td>
</tr>
<tr>
<td>FLOAT*</td>
<td>float</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
</tr>
<tr>
<td>SHORT-INT*</td>
<td>short</td>
</tr>
<tr>
<td>LONG-INT*</td>
<td>long</td>
</tr>
<tr>
<td>INT*</td>
<td>int</td>
</tr>
<tr>
<td>CHAR-VAR</td>
<td>charvar*</td>
</tr>
<tr>
<td>CHAR*</td>
<td>char* (input only)</td>
</tr>
<tr>
<td></td>
<td>lchar* (output or input/output)</td>
</tr>
</tbody>
</table>

GCI Data Type Mappings
Note: Both the data type specified in the GCI definition and the argument direction define the actual GCI data type used. For example, if you define `int` as an output argument, the actual subroutine handles it as `int*`.

<table>
<thead>
<tr>
<th>PI/open</th>
<th>UniVerse</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR[(n)]</td>
<td>char* (input only)</td>
</tr>
<tr>
<td></td>
<td>lchar* (output or input/output)</td>
</tr>
<tr>
<td>INTEGER*2</td>
<td>integer2</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>integer4</td>
</tr>
<tr>
<td>REAL*4</td>
<td>real4</td>
</tr>
<tr>
<td>REAL*8</td>
<td>real8</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>logical</td>
</tr>
<tr>
<td>CHAR[(n)]</td>
<td>character (FORTRAN 77)</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>character</td>
</tr>
</tbody>
</table>
Appendix B: Example programs

This appendix describes the UniVerse BASIC programs and C subroutines supplied with the GCI, and programming examples in C and FORTRAN.
Supplied GCI programs

The following sections describe the programs that come with the GCI, including:

- *hello (print “hello world”)
- multiply (multiply two numbers)
- *gci3 (pass argument)
- gci4 (allocate memory)

To use these programs you must first add the subroutines to the GCI definition file using the suggested subroutine definitions (as described on page 6), and then add them to UniVerse (as described on page 10).

*hello

This is the classic “hello world” C program called from UniVerse BASIC as a subroutine.

Calling program: uv/BP/GCI1

C subroutine: gcidir/gci_hello.c

GCI definition: Subroutine Name: *hello
Language: c
External Name: hello
Module Name: gci_hello
Description: 
Number of Arguments: 0
Return Value: void

multiply

This is a simple program to multiply two numbers and return the result, called from UniVerse BASIC as a function.

Calling program: uv/BP/GCI2

C subroutine: gcidir/gci_mult.c
| GCI definition: | Subroutine Name: | multiply |
|               | Language: | c |
|               | External Name: | multiply |
|               | Module Name: | gci_mult |
|               | Description: | |
|               | Number of Arguments: | 2 |
|               | Return Value: | int |

**Argument Types:**

<table>
<thead>
<tr>
<th>Data Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

**Language:**

c

**External Name:**

multiply

**Module Name:**

gci_mult

**Description:**

This program demonstrates argument passing from UniVerse BASIC to C and back.

**Calling program:**

uv/BP/GCI3

**C subroutine:**

gcidir/gci_args.c

| GCI definition: | Subroutine Name: | *gci3 |
|               | Language: | c |
|               | External Name: | passing |
|               | Module Name: | gci_args |
|               | Description: | |
|               | Number of Arguments: | 2 |
|               | Return Value: | void |

**Argument Types:**

<table>
<thead>
<tr>
<th>Data Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>pchar*</td>
</tr>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

**gci4**

This program demonstrates memory allocation.

**Calling program:**

uv/BP/GCI4

**C subroutine:**

gcidir/gci_malloc.c
| GCI definition: | Subroutine Name: | c
| Language: | gci4 |
| External Name: | gci_c4 |
| Module Name: | gci_malloc |
| Description: | |
| Number of Arguments: | 3 |
| Return Value: | int |
| Argument Types: | Data Types: |
| I | pchar* |
| B | tchar* |

### System calls

The GCI definition file in the UV account includes definitions for the following UNIX system calls:

- `access(2)`
- `chmod(2)`
- `chown(2)` (not available on Windows Platforms)
- `getpid(2)`
- `link(2)` (not available on Windows Platforms)

You can use these system calls from a UniVerse BASIC program. A UniVerse BASIC program called `uv/BP/GCI5` comes with the GCI to demonstrate a call to `getpid`.
Example GCI programs

The following sections contain examples of subroutines in C and FORTRAN including:

- An interluded system call
- Subroutines that demonstrate array handling in C and FORTRAN

The examples include the correct GCI definitions for the subroutines and examples of UniVerse BASIC calling programs. These examples do not come with the GCI.

Interluded system call

If a system call needs arguments that are pointers to data structures, these cannot be mapped directly through a GCI subroutine definition, but can be accessed through an interlude. For example, the `stat(2)` system call returns the following:

- Information about an operating system file in a structure
- A success or failure indicator as a function value
- The value of the system variable `errno` to indicate what went wrong
The following C program is an interlude that returns the data returned by `stat` in a UniVerse BASIC dynamic array, and leaves the calling program to extract the information required:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int stat();

#define FM '\376'

int file_stat(path, ib_buf)
unsigned char * path;
unsigned char * ib_buf;
{
    struct stat stat_buf;
    int stat_value;

    stat_value = stat(path, &stat_buf);
    if (stat_value == 0) {
        sprintf(ib_buf, "%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%...
stat_buf.st_mtime, FM, /* Time of last data change */
stat_buf.st_ctime); /* Time of last file status */
/* change. Times measured in */
/* seconds since 00:00:00 GMT, */
/* Jan. 1, 1970 */

) else {

/************************************************************
* Some error occurred - ensure null string returned.       *
************************************************************/

*ib_buf = '\0';
) /* end if */

return(stat_value);
) /* end of file_stat */

**GCI definition**

This is the correct GCI definition for the previous program.

<table>
<thead>
<tr>
<th>Subroutine Name:</th>
<th>$FILE.STAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language:</td>
<td>c</td>
</tr>
<tr>
<td>External Name:</td>
<td>file_stat</td>
</tr>
<tr>
<td>Module Name:</td>
<td>file_stat</td>
</tr>
<tr>
<td>Description:</td>
<td>Interlude to stat system call</td>
</tr>
<tr>
<td>Number of Arguments:</td>
<td>2</td>
</tr>
<tr>
<td>Return Value:</td>
<td>int</td>
</tr>
</tbody>
</table>
Argument Types: Data Types:
I char*
O char*

Calling program

The following listing shows a simple BASIC program that calls the previous interluded system call:

```
************************************************************
** Example program using an interluded system call. **
** This program uses the 'file_stat' GCI subroutine **
** which is an interlude to the UNIX 'stat' system call. **
*******************************************************************************
DEFFUN FILE.STAT(A,B) CALLING "$FILE.STAT"
DEFFUN ERRNO() CALLING "$ERRNO"

PRINT "Enter pathname":
PROMPT ":' INPUT PATH.NAME
IF LEN(PATH.NAME) = 0 THEN RETURN

STAT.BUF = ''
STAT.RESULT = FILE.STAT(PATH.NAME, STAT.BUF)
IF STAT.RESULT THEN

*******************************************************************************
** Note the use of the !ERRNO subroutine if FILE.STAT **
** returns a non-zero value. **
*******************************************************************************
PRINT "Error in stat call for file ":PATH.NAME
PRINT "Error code is ":ERRNO()
RETURN
END

*******************************************************************************
** Print out details of the file as returned from 'stat'. **
*******************************************************************************
PRINT "File name: ":PATH.NAME
PRINT "File size: ":STAT.BUF" bytes"
PRINT "Owner's UID: ":STAT.BUF>
* ... plus whatever you want.
RETURN

*******************************************************************************
** Take care in using time values returned by FILE.STAT **
** [fields 9, 10, and 11]. If you want to see valid local **
** times and dates, you will need to apply a time zone **
```
** correction. This is most easily done with the **
** localtime() library call. See under CTIME(3C) in **
** your operating system manuals.                         **
*******************************************************************************

END

Arrays in C

This example shows a BASIC array being passed to the C function erf. For
information about C array handling, see C Arrays.

Calling program

Note that the dimensions of the arrays the UniVerse BASIC code defines
match those specified in the GCI definition shown after the program listing.

* First define our two matrices
dim inarray(3, 2)
dim outarray(3, 2)
* Snap to the cataloged subroutine
erf = "$ERFARRAY"
  for i = 1 to 3
    for j = 1 to 2
      inarray(i, j) = (i * j) / 100
    next j
  next i
  call @erf(mat inarray, mat outarray)

  for i = 1 to 3
    for j = 1 to 2
      print "inarray(",i:",";j:") = ":inarray(i,j)
    next j
  next i
  return
end
**GCI subroutine**

This is the C subroutine called by UniVerse BASIC through the GCI:

```c
#include <math.h>

void erfarray(array1, array2)
/
* Subroutine as interlude to erf error function call for a 3 by 2 matrix. *
**************************************************************/

double array1[3][2];
double array2[3][2];
{
  int i;
  int j;
  for (i = 0; i < 3; i++)
    for (j = 0; j < 2; j++)
      array2[i][j] = erf(array1[i][j]);
}
```

**GCI definition**

This is the correct GCI definition for the previous subroutine:

- **Subroutine Name:** $ERFARRAY
- **Language:** c
- **External Name:** erfarray
- **Module Name:** erfarray
- **Description:** Call erf function with array arguments
- **Number of Arguments:** 2
- **Return Value:** void
- **Argument Types:** Data Types: Rows: Columns:
Arrays in FORTRAN

In this example, the GCI passes an array defined in UniVerse BASIC to a FORTRAN 77 subroutine. For information about FORTRAN array handling, see FORTRAN arrays.

Calling program

Note that the dimensions of the arrays defined by the BASIC code match those specified in the GCI definition:

```
* This example shows how BASIC array-handling (which is row-major
* differs from FORTRAN 77 array-handling (which is column-major).
******************************************************************************
*                              ************ In BASIC program *****
* dim a(3, 2)                  for i = 1 to 3
* set up routine name.         for j = 1 to 2
arrayr = "$ARRAYR"             a(i, j) = i + (j / 10)
* assign each element the value i.j eg element a(2, 1) has value
print "array("":i:","":j:"") has value ":a(i, j)
next j
next i
* call the F77 routine, which will print out the array again.
call @arrayr(mat a)
return
end
```
**FORTRAN subroutine**

The FORTRAN 77 program called by the UniVerse BASIC code is as follows:

```fortran
subroutine arrayr(array)
  C     Note that array dimensions and subscripts are reversed.
  C
  real*4 array(2, 3)
  integer i, j

  write (6, 600)
  do 20 j = 1, 3
      do 10 i = 1, 2
          write(6, 601) i, j, array(i, j)
    10     continue
  20   continue

  return

600   format('**** In F77 Subroutine Arrayr ****')
601   format('array(',i1,',',i1,') has value ',f6.4)
end
```

**GCI definition**

The GCI definition for the subroutine is as follows:

- **Subroutine Name:** $ARRAYR
- **Language:** F77
- **External Name:** arrayr
- **Module Name:** arrayr
- **Description:** Pass BASIC array to FORTRAN 77
- **Number of Arguments:** 1
- **Return Value:** void
- **Argument Types:**
  - I: Data Types: real4, Rows: 3, Columns: 2
**Program output**

This is the output produced when the UniVerse BASIC program runs:

```
**** In BASIC program ****
array(1,1) has value 1.1000
array(1,2) has value 1.2000
array(2,1) has value 2.1000
array(2,2) has value 2.2000
array(3,1) has value 3.1000
array(3,2) has value 3.2000

**** In F77 Subroutine Arrayr ****
array(1,1) has value 1.1000
array(2,1) has value 1.2000
array(1,2) has value 2.1000
array(2,2) has value 2.2000
array(1,3) has value 3.1000
array(2,3) has value 3.2000
```