Rocket UniVerse

GCI User Guide

Version 11.3.1

October 2016
UNV-1131-GCI-1

Notices

Edition

Publication date: October 2016
Book number: UNV-1131-GCI-1
Product version: Version 11.3.1

Copyright

Trademarks

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

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.

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country	Toll-free telephone number
United States	1-855-577-4323
Australia	1-800-823-405
Belgium	0800-266-65
Canada	1-855-577-4323
China	800-720-1170
France	08-05-08-05-62
Germany	0800-180-0882
Italy	800-878-295
Japan	0800-170-5464
Netherlands	0-800-022-2961
New Zealand	0800-003210
South Africa	0-800-980-818
United Kingdom	0800-520-0439

Contacting Technical Support

The Rocket Customer Portal is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Customer Portal and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Customer Portal or to request a Rocket Customer Portal account, go to www.rocketsoftware.com/support.

In addition to using the Rocket Customer Portal to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Contents

Chapter 1: GCI overview

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

Note: At 11.3.x, major modifications have been added for GCI. If you are using UniVerse 11.2.x or earlier, visit https://docs.rocketsoftware.com for the documentation specific to your release. Navigate to **Rocket U2** → **UniVerse** → **Previous Versions**.

GCI allows UniVerse BASIC programs to make:

- Calls to external subroutines written in FORTRAN 77, C, and C++. In this manual, all references to C also apply to C++.
- System calls to operating system commands. Some examples of these are provided when you first install GCI. For more information, see Example programs, on page 33.

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

Note: Microsoft Windows does not support mixing 32-bit and 64-bit applications. As a result, GCI DLLs used with 32-bit versions 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.

The dynamically linked shared library

At UniVerse 11.3.1 on UNIX and Linux platforms, the General Calling Interface (GCI) has changed from statically linking the functions inside new executables (such as `uvsh.new`) to a dynamically linked shared library called `libu2gci.so` or `libu2gci.sl`.

Note: On Linux platforms, the shared library extension is `.so` – shared object. This is the same as most UNIX platforms, except on HP-UX, which uses the `.sl` extension – shared library.

Dynamic linking allows the C code routines to be integrated with UniVerse and called by BASIC programs. With dynamic linking, GCI utilities do not have to be statically linked to each new release, allowing easier installation of product updates and reducing system administration.

Chapter 2: Setting up GCI

Before you can use GCI to call an external subroutine, you must set it up properly. You must be logged on as a UniVerse Administrator to perform these steps.

1. Preparing the subroutine
   Write, compile, and test the subroutine, then copy the source into the `gcidir` directory in the UV account directory.
2. Adding a subroutine definition record
   Define the subroutine to UniVerse.
3. Adding the subroutine to UniVerse
   When you have defined the subroutine, you must add it to UniVerse.

Preparing the subroutine

Write, compile, and test the subroutine, then copy the source into the `gcidir` directory in the UV account directory.

GCI supports subroutines written in FORTRAN 77, C, and C++. For information about specifying data types, handling arrays, and so forth, see GCI subroutines, on page 18. 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 *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.

Next topic: Adding a subroutine definition record
Parent topic: Setting up GCI

Adding a subroutine definition record

Define the subroutine to UniVerse.

About this task

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

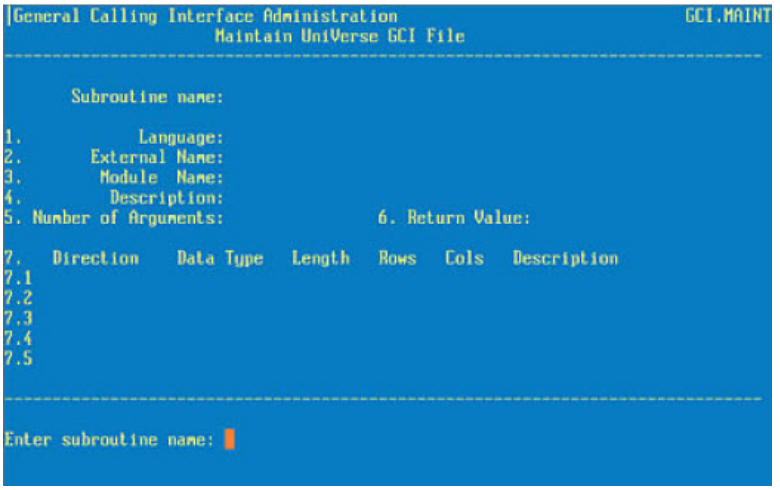
The GCI uses this information to convert any data that is passed into the correct data type for the receiving program.

When it is first installed, the GCI definition file contains example definitions for some simple C subroutines and system calls. For more information, see Example programs, on page 33.

Procedure

1. Select **Package** → **GCI administration** from the UniVerse **System Administration** menu (UNIX and Linux platforms only), 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. Select **Add, Modify, and Delete GCI subroutines**. On Windows platforms, enter the name of the GCI definition file at the prompt.

   The **GCI Maintenance** menu displays as shown in the following example.



3. Enter information about the subroutine at the prompt.

   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.

   a. **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.

   b. **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.

   c. **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.

   d. **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 7, 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`.

e. **Description:** (Optional) Enter a short description of the subroutine using up to 50 characters.

f. **Number of arguments:** Enter the total number of arguments. Enter `0` 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.

g. **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 GCI subroutines, on page 18.

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

h. **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:

Argument	Description
Directions	Specify `I` for input, `O` for output, or `B` for both.
Data Type	Specify the data type for the argument, for example, `int` . (For a list of data types supported by GCI, see GCI subroutines, on page 18,)
Length	If you specified a data type of `lchar*` , `charvar*` , or `character` , enter the length of the character string at the prompt.
Rows	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.)
Cole	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.)
Description	You can specify a short description of the argument using up to 15 characters. This field is optional.

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

Next topic: Adding the subroutine to UniVerse
Previous topic: Preparing the subroutine
Parent topic: Setting up GCI

Adding the subroutine to UniVerse

When you have defined the subroutine, 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:

- Building a new u2gci library on UNIX
  When you want to add an external subroutine to UniVerse on a UNIX system, you must rebuild the u2gci library (`libu2gci.so` or `libu2gci.sl`).
- Making a GCI library on Windows platforms
  On Windows platforms, when you have created a GCI definition record for the subroutine 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.
- Using GCI libraries
  UniVerse accesses GCI libraries in one of two ways:

Previous topic: Adding a subroutine definition record
Parent topic: Setting up GCI

Building a new u2gci library on UNIX

When you want to add an external subroutine to UniVerse on a UNIX system, you must rebuild the u2gci library (`libu2gci.so` or `libu2gci.sl`).

If prior to 11.3.1, you relied on the UniVerse GCI Administration menu to incorporate GCI functions into UniVerse, no major rework is required to generate the `libu2gci.so/sl` file. However if you have made custom build scripts or makefiles, then converting the GCI library from static linking to dynamic linking might require significant changes.

Object modules that comprise a dynamically linked library must be compiled with position-independent code. Each combination of compiler, operating system, and processor architecture might require different compiler and linker switches to accomplish this. For example:

- gcc and g++ on Linux platforms require `-fPIC`
- Solaris Studio cc and CC on Intel require `-Kpic`
- Solaris Studio cc and CC on SPARC require `-xcode=pic32`

When converting from static to dynamic libraries, use the `ldd` command to verify that libraries will be found in the search path.

When calling the `uvsh` command and other UniVerse executables and referencing the rebuilt u2gci library, the custom subroutines can be called. See the information about dynamic linking in *Administering UniVerse* to learn how to properly reference the u2gci library.

You can build and install the u2gci library using the following methods:

- Automatically building the u2gci library
  Build a new u2gci library from the **GCI Administration** menu.
- Manually building the u2gci library
  Build a new u2gci library from the UNIX shell prompt.

Parent topic: Adding the subroutine to UniVerse

Automatically building the u2gci library

Build a new u2gci library from the **GCI Administration** menu.

Procedure

1. From the **GCI Administration** menu, select **Make a new u2gci library**.

The following message appears:

```
This procedure will generate a new "gci.c" program and make a new
/usr/uv/libu2gci.so file.
It will also catalog any subroutines which require cataloging.
Are you sure you want to continue? (Y/N,A)
```

2. Answer the prompt as `Y` or `A`.

   If you answer `Y` (Yes) to the prompt, 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 u2gci library

   If you answer `A` (All) to the prompt, GCI answers "Yes" to all of the prompts on the screen.
3. Test the new u2gci library before you install it by performing the following steps:
   a. Start a new session or exit to the operating-system level.
   b. Set the environment variable LD_LIBRARY_PATH or LIPATH (on AIX) to include the UniVerse home directory first.
   c. Execute the `uvsh` command in UniVerse `bin` directory.
4. Test the UniVerse BASIC programs that call the added subroutine and debug them if necessary.
5. Install the u2gci library from the **GCI Administration** menu by selecting **Install new u2gci library**.

   When you install the new u2gci library, the following happens:

   - The old u2gci library `/usr/uv/bin/libu2gci.so` is copied to `/usr/uv/bin/libu2gci.save`.
   - The new u2gci library `/usr/uv/libu2gci.so` is copied to `/usr/uv/bin/libu2gci.so`.
   - If the `/usr/uv/lib.d` directory exists, the new u2gci library is also copied into this directory.

   Note: If any other users on your system are running UniVerse when you install the new u2gci library, they continue to run the old u2gci library until they log out and then log back on again.

Parent topic: Building a new u2gci library on UNIX

Manually building the u2gci library

Build a new u2gci library from the UNIX shell prompt.

Prerequisites

Check that you have completed definitions for each of the subroutines you want to add to the GCI (as described in Adding a subroutine definition record, on page 7) by selecting **List GCI Subroutines** from the **GCI Administration** menu.

Note: You must use the following method to add FORTRAN subroutines.

Procedure

1. Check that you have a makefile:
   a. Change to the `gcidir` directory in the UV account directory (for example, `/usr/uv/gcidir`).
   b. 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
      ```
   c. Make a new `gci.c` file that includes your new subroutines using the following command:
      ```
      $ make gci
      ```
2. Edit the file `gcidir/Makefile` as follows:
   a. Add the object files for the new subroutines to the *GCILIB* variable. For example:
      ```
      GCILIB=gci_mult.o
      ```
   b. 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 -Iinclude.file -Ddefine.token
      ```
   c. Add any FORTRAN library-loading options used by your system to the *LIBES* variable. (See the FORTRAN 77 manual provided with your system.)
   d. 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`
3. Make the new u2gci library with the `make` command. If you change to the `/usr/uv` directory and list it, you see the `libu2gci.so/sl` file that you just created.
4. Test the new u2gci library before you install it by performing the following steps:
   a. Start a new session or exit to the operating-system level.
   b. Set the environment variable LD_LIBRARY_PATH or LIPATH (on AIX) to include the UniVerse home directory first.
   c. Execute the `uvsh` command in UniVerse `bin` directory.
5. Test the UniVerse BASIC programs that call the added subroutine and debug them if necessary.
6. Install the u2gci library from the **GCI Administration** menu by selecting **Install new u2gci library**.

   Alternatively, install the u2gci library manually by performing the following steps:
   a. At the UNIX shell prompt, change to the `bin` directory in the **UV account directory**. For example:
      ```
      $ cd /usr/uv/bin
      ```
   b. Save the old library file by moving it to another file. For example:
      ```
      $ cp -f libu2gci.so libu2gci.so.save
      ```
   c. Copy the new u2gci library file to the old file. For example:
      ```
      $ cp -f /usr/uv/libu2gci.so libu2gci.so
      ```
   d. If the `/usr/uv/lib.d` directory exists, update the file in this path as well. This can help if calling the debug versions of the UniVerse default libraries (`libuniverse.so/sl`). For example:
      ```
      $ cp -f /usr/uv/libu2gci.so /usr/uv/lib.d/libu2gci.so
      ```

Next step

See The calling program, on page 16 for details about how to call a GCI subroutine from a UniVerse BASIC program.

Parent topic: Building a new u2gci library on UNIX

Making a GCI library on Windows platforms

On Windows platforms, when you have created a GCI definition record for the subroutine 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.

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

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

Parent topic: Adding the subroutine to UniVerse

Automatically building the DLL

Use the **GCI Administration** menu to automatically build the DLL. This method is suitable for most C subroutines where you are using the Microsoft C compiler and linker to build the DLL.

Prerequisites

Install the appropriate compiler, as described in Preparing the subroutine, on page 7.

Procedure

1. From the **GCI Administration** menu, select **Make a GCI Library from a GCI Definition file**.
2. Enter the name of the GCI definition file at the prompt, then confirm the action.
3. Test the DLL before you install it 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.
4. Install the DLL by selecting **Install a GCI Library** from the **GCI Administration** menu.

Results

When you install the DLL, the following happens:

- The DLL file is copied from the `gcidir` directory to the `bin` directory in the UV account directory.
- The name of the copied file is added to the GCI library list held in the Windows Registry.

Parent topic: Making a GCI library on Windows platforms

Manually building the DLL

Using MS-DOS and UniVerse commands to manually build the DLL. You must use this method for FORTRAN 77 subroutines or if you are not using the Microsoft C compiler or linker.

Procedure

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 before you install it 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 by selecting **Install a GCI Library** from the **GCI Administration** menu.

   Alternatively, 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.

Results

When you install the DLL, the following happens:

- The DLL file is copied from the `gcidir` directory to the `bin` directory in the UV account directory.
- The name of the copied file is added to the GCI library list held in the Windows Registry.

Parent topic: Making a GCI library on Windows platforms

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 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.

Parent topic: Adding the subroutine to UniVerse

Chapter 3: The calling program

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 Example programs, on page 33.

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 globally cataloged $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 Example programs, on page 33.

```
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 GCI subroutines, on page 18.

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.

Chapter 4: GCI subroutines

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.

GCI data type	Description	C data type	Direction
`char`	Single character	`char char*`	I O or B

`char* pchar*`
`tchar* lchar*`
	Pointer to character string	`char*`
		`char**`
		`char**`
		`char*`
			I
			O or B
			O or B
			I, O, or B

`charvar*`	Pointer to character varying string	`charvar*`	I, O, or B
`int`	Integer	`intint*`	I O or B
`long`	Long	`long` `long*`	I O or B
`short`	Short	`short` `short*`	I O or B
`float`	Float	`float` `float*`	I O or B
`double`	Double	`double` `double*`	I O or B
`void`	Void	`void`	return only

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`.

GCI data type	Description	Win32 data type	Direction
BOOL	Integer value used for true or false: 1 is true; 0 is false	BOOL LPBOOL	I O or B
BYTE	Unsigned 8-bit character	BYTE LPBYTE	I O or B
WORD	Unsigned 16-bit integer	WORD LPWORD	I O or B
DWORD	Unsigned 32-bit integer	DWORD LPDWORD	I O or B

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:

Data type	Memory allocation
`pchar*`	The GCI uses the memory used by the string. For example if the string *abcde* is input to the GCI routine, the maximum size for output is 5.
`tchar*`	The GCI assumes malloc allocates the memory within the GCI routine. The example routine *gci_malloc.c* uses *malloc* in this way.
`lchar*`	The GCI uses the length defined for the subroutine in the GCI file to determine how much memory to allocate.
`charvar*`	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.

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.

Data type	Numeric	Nonnumeric	Array	Direction
`integer2`	Yes	No	Yes	I, O, B
`integer4`	Yes	No	Yes	I, O, B
`real4`	Yes	No	Yes	I, O, B
`real8`	Yes	No	Yes	I, O, B
`logical`	Yes	Yes	Yes	I, O, B
`character`	Yes	Yes	No	I, O, B

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.

Data type	Description
Integers	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.
Floating points	All numeric data and wholly numeric strings can be converted to floating point. The data conversion is aborted if it encounters a string containing a nonnumeric character.

Data type	Description
Logical	When you pass a logical argument to a FORTRAN 77 routine, in general, `0` or an empty string represents false, while any other value is true. This varies according to the operating system and compiler you use.
Character	This is a fixed-length string. The GCI subroutine definition includes the length definition. The string is padded to the right with blanks if it is shorter than the specified length. A conversion error occurs if it is longer.

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 Example programs, on page 33.

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.

GCI data type	Description
`wchar_t*`	Pointer to `wchar`.
`pwchar_t*`	Pointer to preallocated string memory.
`twchar_t*`	Pointer to character memory allocated by the subroutine.
`lwchar_t*`	Pointer to character memory allocated by the GCI.
`wchar_tvar*`	Pointer to a string type. Memory is allocated to the character length in the first word of the buffer.

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 *NLS Guide*.

Chapter 5: GCI functions

This chapter discusses GCI functions that create, read, write, or manage pipe and or child processes from a UniVerse BASIC program.

UVClosePipe function

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

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

Syntax

`UVClosePipe``(`*`pipe_handle`*`)`

Parameter

The following table describes the parameter of the syntax.

Parameter	Description
pipe_handle	*pipe_handle* can be either the *readPipe_handle* or the *writePipe_handle* returned by a previously executed `UVCreatePipe` function.

Return codes

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

Return code	Description
0	Success
-1	Failure

UVCreatePipe function

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.

Syntax

`UVCreatePipe``(`*`readPipe_handle, writePipe_handle`*`)`

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

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
readPipe_handle	The handle to the READ end of the pipe that was created.
writePipe_handle	The handle to the WRITE end of the pipe that was created.

Return codes

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

Return code	Description
0	Success
-1	Failure

UVCreateProcess function

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

Syntax

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

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
command	The command you want to execute.
input_handle	Specifies a handle UniVerse uses as the standard input handle to the process. This parameter can be a *readPipe_handle* to a pipe created by the UVCreatePipe function, on page 23.
output_handle	Specifies a handle UniVerse uses as the standard output handle for the process. This parameter can be a *writePipe_handle* to a pipe created by the `UVCreatePipe` function.
error_handle	Specifies a handle UniVerse uses as the standard error handle for the process. This parameter can be a *writePipe_handle* created by the `UVCreatePipe` function.
pid	Specifies a variable to receive the process ID created by this function.
child_handle	Specifies the variable to receive the handle to the newly created function.

Return codes

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

Return code	Description
0	Success
-1	Failure

UVGetExitCodeProcess function

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

Syntax

`UVGetExitCodeProcess``(`*`child_handle, exit_code`*`)`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
child_handle	The handle to the process you are interrogating. This parameter can be the *child_handle* for a process created by the UVCreateProcess function, on page 24.
exit_code	Specifies a variable to receive the process termination status.

Return codes

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

Return code	Description
0	Success
-1	Failure

UVPeekNamedPipe function

The `UVPeekNamedPipe` function copies data from a named or anonymous pipe into a buffer, without removing the data from the pipe. This functions 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.

Syntax

`UVPeekNamedPipe``(`*`pipe_handle, readPipe_buffer, buffer_size, number_bytes_read, total_bytes_available, bytes_left_this_message`*`)`

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

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
pipe_handle	The *readpipe_handle* to a pipe created by the UVCreatePipe function, on page 23.
readPipe_buffer	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.

Parameter	Description
buffer_size	The size of the *readPipe_buffer*, in bytes, to be read. If the value of *readPipe_buffer* is 0, UniVerse ignores this parameter.
number_bytes_read	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.
total_bytes_available	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.
bytes_left_this_message	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 UVCreatePipe function, on page 23, or when no data is to be read.

Return codes

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

Return Code	Description
0	Success
-1	Failure

UVReadPipe function

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

Syntax

`UVReadPipe``(`*`readPipe_handle, readPipe_buffer, readPipe_buffer_size`*`)`

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

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
readPipe_handle	The handle to the READ end of the pipe.
readPipe_buffer	Specifies the variable where UniVerse will store the data read from the pipe.
readPipe_buffer_size	The number of bytes to read from the pipe.

Return codes

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

Return code	Description
0	Success
-1	Failure

UVRunCommand function

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

Syntax

`UVRunCommand`(*`command`*)

The following example shows how to execute the command:

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

Note: You must use single or double quotation marks around the string argument.

UVWritePipe function

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

Syntax

`UVWritePipe`(*`writePipe_handle, writePipe_buffer`*)

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

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
writePipe_handle	The handle to the WRITE end of the pipe.
writePipe_buffer	The expression specifying the data to write to the pipe.

Return codes

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

Return code	Description
0	Success
-1	Failure

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 UVPeekNamedPipe
DECLARE GCI UVCreateProcess
DECLARE GCI UVGetExitCodeProcess
```

```
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 = UVCreateProcess(cmd,StdInPipeRead,StdOutPipeWrite,0,PID,Child)

      If return_value < 0 then
    crt 'Error creating child process'
    stop
End

crt "PID = ": PID : " Child = ":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.

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.

Importing PI/open GCI definitions

If you want to use PI/open GCI definitions, you can import them into UniVerse.

Prerequisites

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

Procedure

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. Import the PI/open definition file.

   On UNIX systems:

   a. From the **GCI Administration** menu, select **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.

   Note: 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:

   a. From the **GCI Administration** menu, select **Create a GCI Definition file** to create UniVerse GCI definition files for all the PI/open definition files that you want to import.
   b. Select **Import a PI/open definition file**.
   c. 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.
5. If on a UNIX system, continue with the following steps:
   a. From the **GCI Administration** menu, choose **Make a new u2gci library** to link the GCI subroutines to the u2gci library.

      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 or installing UniVerse. Similarly, if you use any nonstandard libraries in a C or C++ subroutine you should include them in the GCI makefile. For more information, see Manually building the u2gci library, on page 11.

   b. Select **Install a new u2gci library** from the **GCI Administration** menu to install the newly created library.
6. If on a Windows platform, continue with the following steps:
   a. Select **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 rebuilding or installing UniVerse. Similarly, if you use any nonstandard libraries in a C or C++ subroutine you should include them in the GCI makefile. For more information, see Manually building the u2gci library, on page 11.

   b. Select **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:

P/Iopen	UniVerse
SHORT-INT	short
LONG-INT	long
DOUBLE*	double
FLOAT*	float
INT	int
SHORT-INT*	short
LONG-INT*	long
INT*	int

P/Iopen	UniVerse
CHAR-VAR	charvar*
CHAR*	char* (input only)
	lchar* (output or input/output) GCI Data
INTEGER*2	integers2
INTEGER*4	integer4
REAL*4	real4
REAL*8	real8
LOGICAL	logical
CHAR [n]	character (FORTRAN 77)
CHARACTER	character

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*`.

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, and then add them to UniVerse. For more information, see GCI overview, on page 6.

*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 definitions	
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 definitions	
Subroutine name	`multiply`
Language	`c`
External name	`multiply`

GCI definitions	
Module name	gci_mult
Description	
Number of arguments	2
Return value	int

Argument types	Data types
I	int
I	int

*gci3

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

Calling program: uv/BP/GCI3

C subroutine: gcidir/gci_args.c

GCI definitions	
Subroutine name:	*gci3
Language:	Language:
External name	passing
Module name:	gci_args
Description:	
Number of arguments:	2
Return value:	void

Argument types	Data types
B	pchar*
B	int

gci4

This program demonstrates memory allocation.

Calling program: uv/BP/GCI4

C subroutine: gcidir/gci_malloc.c

GCI definitions	
Subroutine name	gci4
Language	c
External name	gci_c4
Module name	gci_malloc
Description	
Number of arguments	3
Return value	int

Argument types	Data types
I	char*
B	pchar*
O	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.

Note: When compiling with 'cc', you may need to use the '-c' option to skip linking after the compile.

```
/*********************************************************
 * file_stat.c                                           *
 * Interlude for the UNIX 'stat' system call             *
 *********************************************************/
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>

int stat ();
/*************************************************************
 * The program assumes that a field mark is octal 376.       *
 *************************************************************/
#define FM '\376'
/*************************************************************
 * Arguments to file_stat():                                 *
 * 1: input, string containing name of file system object    *
 * 2: output, string in dynamic array format describing      *
 *   Various attributes of the object.                       *
 * Returns: integer, 0 = OK else error code from stat()      *
 *************************************************************/
int file_stat(path, ib_buf)
unsigned char * path;
unsigned char * ib_buf;
{
struct stat stat_buf;
int stat_value;
/*************************************************************
 * stat() takes a pathname as its first argument, and        *
 * returns a buffer structure as defined in the file         *
 * /usr/include/stat.h as its second argument.              *
 *************************************************************/
stat_value = stat(path, &stat_buf);
if (stat_value == 0) {
/*************************************************************
 * All OK - no error occurred.                               *
 * Convert all elements of the structure stat_buf to fields *
 * of a dynamic array for the BASIC caller.                  *
 *************************************************************/
sprintf(ib_buf, "%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d%c%d",
stat_buf.st_dev, FM,/* ID of device containing a */
/* directory entry for the file */
stat_buf.st_ino, FM,/* Inode number */
stat_buf.st_mode, FM,/* File mode [see mknod(2)] */
stat_buf.st_nlink, FM,/* Number of links */
stat_buf.st_uid, FM, /* User ID of the file's owner */
stat_buf.st_gid, FM, /* Group ID of the file's group */
stat_buf.st_rdev, FM,/* Only valid for special files */
stat_buf.st_size, FM,/* File size in bytes */
stat_buf.st_atime, FM,/* Time of last access */
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_but = '\0';
} /* end if */

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

GCI definition

This is the correct GCI definition for the previous program.

Subroutine name	$FILE.STAT
Language	c
External name	file_stat
Module name	file_stat
Description:	Interlude to stat system call
Number of argument	2
Return value	int

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"
 ERR.VAL = ""
 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
 CALL !ERRNO(ERR.VAL)
 PRINT "Error code is ":ERR.VAL
 RETURN
 END
 ************************************************************
 ** Print out details of the file as returned from 'stat'. **
 ************************************************************
PRINT "File name: ":PATH.NAME
 PRINT
 PRINT "File size: ":STAT.BUF<8>:" bytes"
 PRINT "Owner's UID: ":STAT.BUF<5>
 * ... 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*.

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)
             print "outarray(":i:",":j:") = ":outarray(i,j)
         next j
     next i

     return
 end
```

GCI subroutine

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

```
/************************************************************
 * Subroutine as interlude to erf error function call for a   *
 * 3 by 2 matrix.                                             *
  ************************************************************/

 #include <math.h>

 void erfarray(array1, array2)

 /************************************************************
 * Note that arrays are sized here. They need not be for C,    *
 * but the GCI definition must have knowledge of this to know  *
 * how many elements to pass and that definition must match    *
 * the BASIC arrays passed.                                    *
  ************************************************************/

 double array1[3][2];
 double array2[3][2];
```

```
{
                        int i;
                        int j;
/************************************************************
 * C arrays are indexed from 0.                             *
 ************************************************************/

        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
I	double	3	2
O	double	3	2

Arrays in FORTRAN

In this example, the GCI passes an array defined in UniVerse BASIC to a FORTRAN 77 subroutine.

For more information about FORTRAN array handling, see FORTRAN arrays, on page 21.

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). *
*****************************************************************

       dim a(3, 2)

*   Set up routine name.

    arrayr = "$ARRAYR"

*   Assign each element the value i.j eg element a(2, 1) has value
```

```
    2.1.

        print "**** In BASIC program ****"
        for i = 1 to 3
            for j = 1 to 2
                a(i, j) = i + (j / 10)
                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:

```
                    subroutine arrayr(array)
C
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 names	$ARRAYR
Language	F77
External names	arrayr
Module names	arrayr
Description	Pass BASIC array to FORTRAN 77
Number of arguments	1
Return value	void

Argument types	Data types	Rows	Columns
I	real4	3	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
```