



# UniVerse

## **GCI Guide**

Version 10.2  
September, 2006

IBM Corporation  
555 Bailey Avenue  
San Jose, CA 95141

Licensed Materials – Property of IBM

© Copyright International Business Machines Corporation 2006. All rights reserved.

AIX, DB2, DB2 Universal Database, Distributed Relational Database Architecture, NUMA-Q, OS/2, OS/390, and OS/400, IBM Informix®, C-ISAM®, Foundation.2000™, IBM Informix® 4GL, IBM Informix® DataBlade® module, Client SDK™, Cloudscape™, Cloudsync™, IBM Informix® Connect, IBM Informix® Driver for JDBC, Dynamic Connect™, IBM Informix® Dynamic Scalable Architecture™ (DSA), IBM Informix® Dynamic Server™, IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager), IBM Informix® Extended Parallel Server™, i.Financial Services™, J/Foundation™, MaxConnect™, Object Translator™, Red Brick® Decision Server™, IBM Informix® SE, IBM Informix® SQL, InformiXML™, RedBack®, SystemBuilder™, U2™, UniData®, UniVerse®, wIntegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

This product includes cryptographic software written by Eric Young (eay@cryptosoft.com).

This product includes software written by Tim Hudson (tjh@cryptosoft.com).

Documentation Team: Claire Gustafson, Shelley Thompson

#### US GOVERNMENT USERS RESTRICTED RIGHTS

Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Table of Contents

## Preface

Organization of This Manual . . . . .	vi
Documentation Conventions. . . . .	vii
UniVerse Documentation. . . . .	ix
Related Documentation . . . . .	xi
API Documentation . . . . .	xii

## 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

Direct Calls . . . . .	2-3
Indirect Calls . . . . .	2-4
Function Calls . . . . .	2-5
Declaring a Function . . . . .	2-6
Passing Arguments. . . . .	2-7
Accessing the System <i>errno</i> Variable . . . . .	2-8

## Chapter 3

### GCI Subroutines

UNIX File Units . . . . .	3-3
C Data Types . . . . .	3-4
Data Types for Windows System Calls . . . . .	3-5
Allocating Memory for Character Strings . . . . .	3-5
Converting Data Types Between C and BASIC . . . . .	3-6

	C Arrays . . . . .	3-6
	FORTRAN Data Types . . . . .	3-7
	FORTRAN Arrays . . . . .	3-8
	FORTRAN Portability . . . . .	3-8
	Data Types for Multibyte Characters . . . . .	3-9
<b>Chapter 4</b>	<b>GCI Functions</b>	
	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
<b>Appendix A</b>	<b>PI/open GCI Definitions</b>	
	Importing PI/open GCI Definitions . . . . .	A-2
	GCI Differences . . . . .	A-4
<b>Appendix B</b>	<b>Example Programs</b>	
	Supplied GCI Programs . . . . .	B-2
	*hello . . . . .	B-2
	multiply . . . . .	B-2
	*gci3 . . . . .	B-3
	*gci4 . . . . .	B-3
	System Calls. . . . .	B-4
	Example GCI Programs . . . . .	B-5
	Interluded System Call . . . . .	B-5
	Arrays in C . . . . .	B-9
	Arrays in FORTRAN . . . . .	B-11

---

## Preface

This manual describes how to use the General Calling Interface (GCI) to call external subroutines written in C, C++, or FORTRAN from BASIC programs. The manual assumes that you are an experienced UniVerse user and that you know how to do the following:

- Write, compile, and catalog a BASIC program.
- Write and compile a C, C++, or FORTRAN subroutine.
- Use the operating system *make* or *nmake* commands.
- Use the UniVerse System Administration menus. (For more information about these User Menus, see *Administering UniVerse*.)

For FORTRAN subroutines, you should also be familiar with the FORTRAN compiler and libraries on your system.

---

## Organization of This Manual

This manual contains the following:

Chapter 1, “[Using the GCI](#),” describes how the GCI works and how to use it.

Chapter 2, “[The Calling Program](#),” describes how to call a GCI subroutine from a BASIC program.

Chapter 3, “[GCI Subroutines](#),” describes data types and array handling in C and FORTRAN subroutines.

Chapter 4, “[GCI Functions](#),” describes GCI functions you can use with named pipes.

Appendix A, “[PI/open GCI Definitions](#),” describes how to convert PI/open GCI definition files for use with UniVerse.

Appendix B, “[Example Programs](#),” gives information about the sample programs that are supplied with the GCI and provides additional programming examples in C and FORTRAN.

---

## Documentation Conventions

This manual uses the following conventions:

Convention	Usage
<b>Bold</b>	In syntax, bold indicates commands, function names, and options. In text, bold indicates keys to press, function names, menu selections, and MS-DOS commands.
UPPERCASE	In syntax, uppercase indicates UniVerse commands, keywords, and options; BASIC statements and functions; and SQL statements and keywords. In text, uppercase also indicates UniVerse identifiers such as file names, account names, schema names, and Windows file names and paths.
<i>Italic</i>	In syntax, italic indicates information that you supply. In text, italic also indicates UNIX commands and options, file names, and paths.
Courier	Courier indicates examples of source code and system output.
<b>Courier Bold</b>	In examples, courier bold indicates characters that the user types or keys the user presses (for example, <b>&lt;Return&gt;</b> ).
[ ]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA   itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
...	Three periods indicate that more of the same type of item can optionally follow.
↪	A right arrow between menu options indicates you should choose each option in sequence. For example, “Choose <b>File</b> -> <b>Exit</b> ” means you should choose <b>File</b> from the menu bar, then choose <b>Exit</b> from the File pull-down menu.
⌘	Item mark. For example, the item mark (⌘) in the following string delimits elements 1 and 2, and elements 3 and 4: 1⌘2F3⌘4V5

---

### Documentation Conventions

Convention	Usage
<b>F</b>	Field mark. For example, the field mark ( <b>F</b> ) in the following string delimits elements <b>FLD1</b> and <b>VAL1</b> : FLD1 <b>F</b> VAL1 <b>V</b> SUBV1 <b>S</b> SUBV2
<b>V</b>	Value mark. For example, the value mark ( <b>V</b> ) in the following string delimits elements <b>VAL1</b> and <b>SUBV1</b> : FLD1 <b>F</b> VAL1 <b>V</b> SUBV1 <b>S</b> SUBV2
<b>S</b>	Subvalue mark. For example, the subvalue mark ( <b>S</b> ) in the following string delimits elements <b>SUBV1</b> and <b>SUBV2</b> : FLD1 <b>F</b> VAL1 <b>V</b> SUBV1 <b>S</b> SUBV2
<b>T</b>	Text mark. For example, the text mark ( <b>T</b> ) in the following string delimits elements 4 and 5: 1 <b>F</b> 2 <b>S</b> 3 <b>V</b> 4 <b>T</b> 5

**Documentation Conventions (Continued)**

The following are also used:

- Syntax definitions and examples are indented for ease in reading.
- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.



---

# UniVerse Documentation

UniVerse documentation includes the following:

***UniVerse Installation Guide:*** Contains instructions for installing UniVerse 10.2.

***UniVerse New Features Version 10.2:*** Describes enhancements and changes made in the UniVerse 10.2 release for all UniVerse products.

***UniVerse BASIC:*** Contains comprehensive information about the UniVerse BASIC language. It is for experienced programmers.

***UniVerse BASIC Commands Reference:*** Provides syntax, descriptions, and examples of all UniVerse BASIC commands and functions.

***UniVerse BASIC Extensions:*** Describes the following extensions to UniVerse BASIC: UniVerse BASIC Socket API, Using CallHTTP, and Using WebSphere MQ with UniVerse.

***UniVerse BASIC SQL Client Interface Guide:*** Describes how to use the BASIC SQL Client Interface (BCI), an interface to UniVerse and non-UniVerse databases from UniVerse BASIC. The BASIC SQL Client Interface uses ODBC-like function calls to execute SQL statements on local or remote database servers such as UniVerse, DB2, SYBASE, or INFORMIX. This book is for experienced SQL programmers.

***Administering UniVerse:*** Describes tasks performed by UniVerse administrators, such as starting up and shutting down the system, system configuration and maintenance, system security, maintaining and transferring UniVerse accounts, maintaining peripherals, backing up and restoring files, managing file and record locks, and network services. This book includes descriptions of how to use shell commands on UNIX systems to administer UniVerse.

***Using UniAdmin:*** Describes the UniAdmin tool, which enables you to configure UniVerse, configure and manage servers and databases, and monitor UniVerse performance and locks.

***UniVerse Transaction Logging and Recovery:*** Describes the UniVerse transaction logging subsystem, including both transaction and warmstart logging and recovery. This book is for system administrators.

***UniVerse System Description:*** Provides detailed and advanced information about UniVerse features and capabilities for experienced users. This book describes how to use UniVerse commands, work in a UniVerse environment, create a UniVerse database, and maintain UniVerse files.

***UniVerse User Reference:*** Contains reference pages for all UniVerse commands, keywords, and user records, allowing experienced users to refer to syntax details quickly.

***Guide to Retrieve:*** Describes Retrieve, the UniVerse query language that lets users select, sort, process, and display data in UniVerse files. This book is for users who are familiar with UniVerse.

***Guide to ProVerb:*** Describes ProVerb, a UniVerse processor used by application developers to execute prestored procedures called procs. This book describes tasks such as relational data testing, arithmetic processing, and transfers to subroutines. It also includes reference pages for all ProVerb commands.

***Guide to the UniVerse Editor:*** Describes in detail how to use the Editor, allowing users to modify UniVerse files or programs. This book also includes reference pages for all UniVerse Editor commands.

***UniVerse NLS Guide:*** Describes how to use and manage UniVerse's National Language Support (NLS). This book is for users, programmers, and administrators.

***UniVerse Security Features:*** Describes security features in UniVerse, including configuring SSL through UniAdmin, using SSL with the CallHttp and Socket interfaces, using SSL with UniObjects for Java, and automatic date encryption.

***UniVerse SQL Administration for DBAs:*** Describes administrative tasks typically performed by DBAs, such as maintaining database integrity and security, and creating and modifying databases. This book is for database administrators (DBAs) who are familiar with UniVerse.

***UniVerse SQL User Guide:*** Describes how to use SQL functionality in UniVerse applications. This book is for application developers who are familiar with UniVerse.

***UniVerse SQL Reference:*** Contains reference pages for all SQL statements and keywords, allowing experienced SQL users to refer to syntax details quickly. It includes the complete UniVerse SQL grammar in Backus Naur Form (BNF).

---

## Related Documentation

The following documentation is also available:

***UniVerse GCI Guide:*** Describes how to use the General Calling Interface (GCI) to call subroutines written in C, C++, or FORTRAN from BASIC programs. This book is for experienced programmers who are familiar with UniVerse.

***UniVerse ODBC Guide:*** Describes how to install and configure a UniVerse ODBC server on a UniVerse host system. It also describes how to use UniVerse ODBC Config and how to install, configure, and use UniVerse ODBC drivers on client systems. This book is for experienced UniVerse developers who are familiar with SQL and ODBC.

***UV/Net II Guide:*** Describes UV/Net II, the UniVerse transparent database networking facility that lets users access UniVerse files on remote systems. This book is for experienced UniVerse administrators.

***UniVerse Guide for Pick Users:*** Describes UniVerse for new UniVerse users familiar with Pick-based systems.

***Moving to UniVerse from PI/open:*** Describes how to prepare the PI/open environment before converting PI/open applications to run under UniVerse. This book includes step-by-step procedures for converting INFO/BASIC programs, accounts, and files. This book is for experienced PI/open users and does not assume detailed knowledge of UniVerse.

---

## API Documentation

The following books document application programming interfaces (APIs) used for developing client applications that connect to UniVerse and UniData servers.

***Administrative Supplement for Client APIs:*** Introduces IBM's seven common APIs for UniData and UniVerse, and provides important information that developers using any of the common APIs will need. It includes information about UniRPC, the UCI Config Editor, the *ud\_database* file, and device licensing.

***UCI Developer's Guide:*** Describes how to use UCI (UniCall Interface), an interface to UniVerse and UniData databases from C-based client programs. UCI uses ODBC-like function calls to execute SQL statements on local or remote UniVerse and UniData servers. This book is for experienced SQL programmers.

***IBM JDBC Driver for UniData and UniVerse:*** Describes UniJDBC, an interface to UniData and UniVerse databases from JDBC applications. This book is for experienced programmers and application developers who are familiar with UniData and UniVerse, Java, JDBC, and who want to write JDBC applications that access these databases.

***InterCall Developer's Guide:*** Describes how to use the InterCall API to access data on UniVerse and UniData systems from external programs. This book is for experienced programmers who are familiar with UniVerse or UniData.

***UniObjects Developer's Guide:*** Describes UniObjects, an interface to UniVerse and UniData systems from Visual Basic. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Visual Basic, and who want to write Visual Basic programs that access these databases.

***UniObjects for Java Developer's Guide:*** Describes UniObjects for Java, an interface to UniVerse and UniData systems from Java. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with Java, and who want to write Java programs that access these databases.

***UniObjects for .NET Developer's Guide:*** Describes UniObjects, an interface to UniVerse and UniData systems from .NET. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with .NET, and who want to write .NET programs that access these databases.

***Using UniOLEDB:*** Describes how to use UniOLEDB, an interface to UniVerse and UniData systems for OLE DB consumers. This book is for experienced programmers and application developers who are familiar with UniVerse or UniData, and with OLE DB, and who want to write OLE DB programs that access these databases.

---

# Using the GCI

What Is the GCI? . . . . .	1-4
What You Need to Do . . . . .	1-5
Preparing the Subroutine . . . . .	1-6
Defining the Subroutine . . . . .	1-7
Adding a Subroutine Definition Record . . . . .	1-7
Adding the Subroutine to UniVerse . . . . .	1-11
Building a New Run File on UNIX. . . . .	1-11
Making a GCI Library on Windows Platforms . . . . .	1-14

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 BASIC programs to make:

- Calls to external subroutines written in FORTRAN 77, C, and C++.<sup>1</sup>
- System calls to operating system commands. Some examples of these are provided when you first install the GCI. For more information, see 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.

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 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, “[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, [“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.*

```
|General Calling Interface Administration                                GCI.MAINT
|Maintain UniVerse GCI File
|-----
|
|Subroutine name:
|
|1.      Language:
|2.      External Name:
|3.      Module Name:
|4.      Description:
|5. Number of Arguments:           6. Return Value:
|
|7.  Direction  Data Type  Length  Rows  Cols  Description
|7.1
|7.2
|7.3
|7.4
|7.5
|-----
|Enter subroutine name: |
```

### ***Subroutine name:***

Enter the name of the subroutine to be used by the calling program. If you want to call the subroutine as a 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 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, “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:

Argument	Description
Direction	Specify <b>I</b> for input, <b>O</b> for output, or <b>B</b> for both.
Data Type	Specify the data type for the argument, for example, <b>int</b> . (For a list of data types supported by the GCI, see Chapter 3, “GCI Subroutines”).
Length	If you specified a data type of <b>lchar*</b> , <b>charvar*</b> , or <b>character</b> , 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.)
Cols	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.

Argument Details

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 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/ibm/uv/bin/uvsh*, is copied to */usr/ibm/uv/bin/uvsh.save*.
- The new run file, */usr/ibm/uv/uvsh.new*, is copied to */usr/ibm/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/ibm/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 -Iinclude.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/ibm/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/ibm/uv/uvsh.new
```

You can then test the 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/ibm/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, [“The Calling Program,”](#) for details of how to call a GCI subroutine from a 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 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.

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 *gcidir* 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 See “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.

---

# 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 <i>errno</i> Variable . . . . .	2-8

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, “[Supplied GCI 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, [“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, [“GCI Subroutines.”](#)

All arguments returned from a GCI subroutine to a 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.

---

# GCI Subroutines

UNIX File Units . . . . .	3-4
C Data Types . . . . .	3-5
Data Types for Windows System Calls . . . . .	3-6
Allocating Memory for Character Strings. . . . .	3-6
Converting Data Types Between C and BASIC . . . . .	3-7
C Arrays . . . . .	3-7
FORTTRAN Data Types . . . . .	3-8
FORTTRAN Arrays . . . . .	3-9
FORTTRAN Portability . . . . .	3-9
Data Types for Multibyte Characters . . . . .	3-10

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*	Pointer to character string	char*	I
pchar*		char**	O or B
tchar*		char**	O or B
lchar*		char*	I, O, or B
charvar*	Pointer to character varying string	charvar*	I, O, or B
int	Integer	int int*	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

---

### C Data Types

## 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

### Data Types for Windows NT System Calls

## 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 <i>abcde</i> is input to the GCI routine, the maximum size for output is 5.
tchar*	The GCI assumes <i>malloc</i> allocates the memory within the GCI routine. The example routine <i>gci_malloc.c</i> uses <i>malloc</i> 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.

### Memory Allocation

## 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

### 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.

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.
Logical	When you pass a logical argument to a FORTRAN 77 routine, <i>in general</i> , 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.

### 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, “[FORTRAN Subroutine](#).”

## 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.

---

### Data Types for Multibyte Characters

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*.

# 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 [UVClosePipe Function](#).

## Parameter

The following table describes the parameter of the syntax.

Parameter	Description
<i>pipe_handle</i>	<i>pipe_handle</i> can be either the <i>readPipe_handle</i> or the <i>writePipe_handle</i> returned by a previously executed UVCreatePipe function.

UVClosePipe Parameter

## Return Codes

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

Return Code	Description
0	Success
-1	Failure

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.

## Parameters

The following table describes each parameter of the syntax.

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

**UVCreatePipe Parameters**

## Return Codes

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

Return Code	Description
0	Success
-1	Failure

**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.

Parameter	Description
<i>command</i>	The command you want to execute.
<i>input_handle</i>	Specifies a handle UniVerse uses as the standard input handle to the process. This parameter can be a <i>readPipe_handle</i> to a pipe created by the <a href="#">UVCreatePipe Function</a> .
<i>output_handle</i>	Specifies a handle UniVerse uses as the standard output handle for the process. This parameter can be a <i>writePipe_handle</i> to a pipe created by the UVCreatePipe function.
<i>error_handle</i>	Specifies a handle UniVerse uses as the standard error handle for the process. This parameter can be a <i>writePipe_handle</i> created by the UVCreatePipe function.
<i>pid</i>	Specifies a variable the receive the process ID created by this function.
<i>child_handle</i>	Specifies the variable to receive the handle to the newly created function.

**UVCreateProcess Parameters**

# Return Codes

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

Return Code	Description
0	Success
-1	Failure

UVCreateProcess Return Codes

---

# UVGetExitCodeProcess Function

## Syntax

**UVGetExitCodeProcess**(*child\_handle*, *exit\_code*)

## Description

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

## Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>child_handle</i>	The handle to the process you are interrogating. This parameter can be the <i>child_handle</i> for a process created by the <a href="#">UVCreateProcess Function</a> .
<i>exit_code</i>	Specifies a variable to receive the process termination status.

**UVGetExitCodeProcess Parameters**

## Return Codes

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

Return Code	Description
0	Success
-1	Failure

**UVGetExitCodeProcess Return Codes**

---

# UVPeekNamedPipe Function

## Syntax

**UVPeekNamedPipe**(*pipe\_handle*, *readPipe\_buffer*, *buffer\_size*,  
*number\_bytes\_read*, *total\_bytes\_available*, *bytes\_left\_this\_message*)

## Description

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.

## Parameters

The following table describes each parameter of the syntax.

Parameter	Description
<i>pipe_handle</i>	The <i>readpipe_handle</i> to a pipe created by the <a href="#">UVCreatePipe Function</a> .
<i>readPipe_buffer</i>	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.
<i>buffer_size</i>	The size of the <i>readPipe_buffer</i> , in bytes, to be read. If the value of <i>readPipe_buffer</i> is 0, UniVerse ignores this parameter.

---

### UVPeekNamedPipe Parameters

Parameter	Description
<i>number_bytes_read</i>	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.
<i>total_bytes_available</i>	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.
<i>bytes_left_this_message</i>	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 <a href="#">UVCreatePipe Function</a> , or when no data is to be read.

**UVPeekNamedPipe Parameters (Continued)**

## Return Codes

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

Return Code	Description
0	Success
-1	Failure

**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](#).

## Parameters

The following table describes each parameter of the syntax.

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

**UVReadPipe Parameters**

## Return Codes

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

Return Code	Description
0	Success
-1	Failure

**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](#).

## Parameters

The following table describes each parameter of the syntax.

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

**UVWritePipe Parameters**

## Return Codes

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

Return Code	Description
0	Success
-1	Failure

**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 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

```

---

# PI/open GCI Definitions

# A

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, “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, “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:

PI/open	UniVerse
SHORT-INT	short
LONG-INT	long
DOUBLE*	double
FLOAT*	float
INT	int
SHORT-INT*	short
LONG-INT*	long
INT*	int
CHAR-VAR	charvar*
CHAR*	char* (input only)
	lchar* (output or input/output)
CHAR[n]	char* (input only)

---

**GCI Data Type Mappings**



PI/open	UniVerse
	lchar* (output or input/output)
INTEGER*2	integer2
INTEGER*4	integer4
REAL*4	real4
REAL*8	real8
LOGICAL	logical
CHAR[n]	character (FORTRAN 77)
CHARACTER	character

**GCI Data Type Mappings (Continued)**

**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*`.

---

# Example Programs

# B

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.

<b>Calling program:</b>	<i>uv/BP/GCI1</i>	
<b>C subroutine:</b>	<i>gcidir/gci_hello.c</i>	
<b>GCI definition:</b>	<b>Subroutine Name:</b>	<b>*hello</b>
	<b>Language:</b>	<b>c</b>
	<b>External Name:</b>	<b>hello</b>
	<b>Module Name:</b>	<b>gci_hello</b>
	<b>Description:</b>	
	<b>Number of Arguments:</b>	<b>0</b>
	<b>Return Value:</b>	<b>void</b>

### **multiply**

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

<b>Calling program:</b>	<i>uv/BP/GCI2</i>
<b>C subroutine:</b>	<i>gcidir/gci_mult.c</i>

<b>GCI definition:</b>	<b>Subroutine Name:</b>	<b>multiply</b>
	<b>Language:</b>	<b>c</b>
	<b>External Name:</b>	<b>multiply</b>
	<b>Module Name:</b>	<b>gci_mult</b>
	<b>Description:</b>	
	<b>Number of Arguments:</b>	<b>2</b>
	<b>Return Value:</b>	<b>int</b>
	<b>Argument Types:</b>	<b>Data Types:</b>
	<b>I</b>	<b>int</b>
	<b>I</b>	<b>int</b>

### **\*gci3**

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

**Calling program:** *uv/BP/GCI3*

**C subroutine:** *gcidir/gci\_args.c*

<b>GCI definition:</b>	<b>Subroutine Name:</b>	<b>*gci3</b>
	<b>Language:</b>	<b>c</b>
	<b>External Name:</b>	<b>passing</b>
	<b>Module Name:</b>	<b>gci_args</b>
	<b>Description:</b>	
	<b>Number of Arguments:</b>	<b>2</b>
	<b>Return Value:</b>	<b>void</b>
	<b>Argument Types:</b>	<b>Data Types:</b>
	<b>B</b>	<b>pchar*</b>
	<b>B</b>	<b>int</b>

### **\*gci4**

This program demonstrates memory allocation.

**Calling program:** *uv/BP/GCI4*

**C subroutine:** *gcidir/gci\_malloc.c*

<b>GCI definition:</b>	<b>Subroutine Name:</b>	<b>*gci4</b>
	<b>Language:</b>	<b>c</b>

<b>External Name:</b>	<b>gci_c4</b>
<b>Module Name:</b>	<b>gci_malloc</b>
<b>Description:</b>	
<b>Number of Arguments:</b>	<b>3</b>
<b>Return Value:</b>	<b>int</b>
<b>Argument Types:</b>	<b>Data Types:</b>
<b>I</b>	<b>char*</b>
<b>B</b>	<b>pchar*</b>
<b>O</b>	<b>tchar*</b>

## 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:

```

/*****
 *   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",
            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_buf = '\0';
    } /* end if */

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

```

### ***GCI Definition***

This is the correct GCI definition for the previous program.

<b>Subroutine Name:</b>	<b>\$FILE.STAT</b>
<b>Language:</b>	<b>c</b>
<b>External Name:</b>	<b>file_stat</b>
<b>Module Name:</b>	<b>file_stat</b>
<b>Description:</b>	<b>Interlude to stat system call</b>
<b>Number of Arguments:</b>	<b>2</b>
<b>Return Value:</b>	<b>int</b>

**Argument Types:****I****O****Data Types:****char\*****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
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*. 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)
      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:

<b>Subroutine Name:</b>	<b>\$ERFARRAY</b>
<b>Language:</b>	<b>c</b>
<b>External Name:</b>	<b>erfarray</b>
<b>Module Name:</b>	<b>erfarray</b>
<b>Description:</b>	<b>Call erf function with array arguments</b>
<b>Number of Arguments:</b>	<b>2</b>
<b>Return Value:</b>	<b>void</b>
<b>Argument Types:</b>	<b>Data Types:   Rows:   Columns:</b>

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 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).
*
*****
*

      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:

<b>Subroutine Name:</b>	<b>\$ARRAYR</b>
<b>Language:</b>	<b>F77</b>
<b>External Name:</b>	<b>arrayr</b>
<b>Module Name:</b>	<b>arrayr</b>
<b>Description:</b>	<b>Pass BASIC array to FORTRAN 77</b>
<b>Number of Arguments:</b>	<b>1</b>
<b>Return Value:</b>	<b>void</b>
<b>Argument Types:</b>	<b>Data Types: Rows: Columns:</b>
<b>I</b>	<b>real14 3 2</b>

## 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
```

# Index

## A

*access(2)* system call B-4  
 adding  
   library files 1-13  
   subroutine definitions 1-6  
 Add, Modify, and Delete GCI  
   subroutines option 1-7  
 allocating memory 3-5  
 Argument details prompt in GCI  
   Maintenance menu 1-9  
 arguments  
   describing 1-8  
   logical 3-7  
   passing 2-7  
 arrays  
   in C 3-6, B-9  
   in FORTRAN 3-8, B-11  
   passed to C function B-9

## B

BASIC  
   arrays B-9  
   CALL statement 2-3  
   calling a subroutine from 2-2  
   compiler options 2-2  
   declaring a function 2-6  
   ICONV function A-4  
   OCONV function A-4  
   sample programs B-1  
   !ERRNO subroutine 2-8

## C

C language

  arrays in 3-6, B-9  
   compilers 1-12  
   data types 3-4  
   examples B-1  
 CALL statement 2-2  
 char data type 3-4, 3-9  
 character data type 3-7  
 CHARACTER data type in PI/open A-5  
 CHAR*n* data type in PI/open A-4, A-5  
 CHAR-VAR data type in PI/open A-4  
 charvar\* data type 3-4, 3-9  
 char\* data type 3-4, 3-9  
 CHAR\* data type in PI/open A-4  
*chmod(2)* system call B-4  
*chown(2)* system call B-4  
 creating  
   new process 4-5  
 creating makefiles 1-15

## D

data structures B-5  
 data types  
   C 3-4  
   converting 3-6  
   for multibyte characters 3-9  
   FORTRAN 3-7  
   used in calls to Win32 API 3-5  
 DECLARE GCI statement 1-7, 2-2, 2-6  
 declaring a function 2-6  
 DEFFUN statement 1-5, 1-7  
 defining subroutines 1-6  
 Description field in GCI Maintenance  
   menu 1-8



direct calls 2-3  
double data type 3-4  
DOUBLE\* data type in PI/open A-4

## E

ECS field in PI/open A-4  
!ERRNO subroutine 2-8  
*errno* system variable 2-8  
example GCI programs B-1  
Extended Character Set conversion A-4  
External Name field in GCI Maintenance menu 1-8

## F

F77FLAGS variable 1-12  
file units (operating system limit) 3-3  
files  
    GCI definition 1-4  
    *Makefile* 1-12  
    *uvsh* 1-10  
    *uvsh.new* 1-11  
    *windows.h* 3-5  
fixed-length strings 3-7  
float data type 3-4  
floating-point arguments 3-7  
FLOAT\* data type in PI/open A-4  
FORTRAN  
    arrays in B-11  
    compiler compatibility 3-8  
    data types 3-7  
    library-loading options 1-12  
    subroutine examples B-1  
    subroutine portability 3-8  
function calls 2-5  
function declaration 2-6

## G

GCI Administration menu  
    Add, Modify, and Delete GCI subroutines option 1-7  
    Edit the Standard GCI Library List option 1-15

Install a GCI Library option 1-14, 1-15  
GCI definition file 1-4  
    converting to a DLL 1-13  
GCI DLL  
    building automatically 1-14  
    building manually 1-14  
    on systems without GCI installed 1-16  
    using 1-15  
GCI (General Calling Interface)  
    definition 1-3  
    GCI file 1-6  
    installing 1-3  
    Maintenance menu 1-7  
    programs supplied with B-1  
    sample programs B-2  
    subroutine definitions 1-6  
*gclidir* directory 1-11, 1-12  
GCILIB variable 1-12  
GCI.ADMIN command 1-6  
*gci.c* module 1-11  
*getpid(2)* system call B-4

## H

Hewlett-Packard systems 1-12

## I

Import a PI/open definition file  
    option A-2  
indirect calls 2-4  
Install new UniVerse option 1-11  
int data type 3-4  
INT data type in PI/open A-4  
integer2 data type 3-7  
integer4 data type 3-7  
INTEGER\*2 data type in PI/open A-5  
INTEGER\*4 data type in PI/open A-5  
interluded system call B-5  
int\* data type 3-4  
INT\* data type in PI/open A-4

## L

Language field in GCI Maintenance menu 1-8  
LIBES variable 1-12  
library files 1-5  
    adding 1-13  
*link(2)* system call B-4  
logical arguments 3-7  
logical data type 3-7  
LOGICAL data type in PI/open A-5  
long data type 3-4  
LONG-INT data type in PI/open A-4  
LONG-INT\* data type in PI/open A-4  
long\* data type 3-4

## M

Make a new UniVerse option 1-10, A-3  
*Makefile* file 1-12  
makefiles, creating 1-15  
maps in NLS mode 1-5  
memory allocation 3-5  
menus  
    GCI Maintenance 1-7  
    System Administration 1-3  
Module Name field in GCI Maintenance menu 1-8  
multibyte characters 1-5, 3-9

## N

NLS mode 1-5, 3-9  
Number of Arguments field in GCI Maintenance menu 1-8  
numeric strings 3-7

## P

Package option in System Administration menu 1-3  
passing arguments 2-7  
pchar\* data type 3-4  
pipes  
    closing 4-3  
    creating 4-4

reading data from 4-10  
 reviewing contents of 4-8  
 termination status of 4-7  
 writing data to 4-12  
*pi.t30conv* command A-2  
 PI/open GCI definition files  
   data types A-4  
   differences in UniVerse A-4  
   importing A-1

---

## R

real4 data type 3-7  
 real8 data type 3-7  
 REAL\*4 data type in PI/open A-5  
 REAL\*8 data type in PI/open A-5  
 reserved words 1-5  
 Return Value field in GCI Maintenance  
   menu 1-9

---

## S

sample GCI programs B-1  
 Security field in PI/open A-4  
 short data type 3-4  
 SHORT-INT data type in PI/open A-4  
 SHORT-INT\* data type in PI/open A-4  
 short\* data type 3-4  
*stat(2)* system call B-6  
 subroutine arguments  
   defining number of columns in  
     array 1-9  
   passing 2-7  
 Subroutine name field in GCI  
   Maintenance menu 1-7  
 subroutines  
   adding definitions 1-6  
   defining 1-6  
   preparing 1-5  
 system calls 3-5, B-4

---

## T

tchar\* data type 3-4

---

## U

UniVerse run file (*uvsh*) 1-10  
   building 1-9  
 UNIX run file  
   building automatically 1-10  
   building manually 1-11  
 UV account directory 1-6, 1-12  
 UVClosePipe function 4-3  
 UVCreatePipe function 4-4  
 UVCreateProcess function 4-5  
 UVGCIDLLS environment variable 1-14, 1-15  
 UVGetExitCodeProcess function 4-7  
 UVPeekNamedPipe function 4-8  
 UVReadPipe function 4-10  
*uvsh* (UniVerse run file) 1-10  
*uvsh.new* (new run file) 1-11, 1-12  
 UVWritePipe function 4-12

---

## V

void data type 3-4

---

## W

Win32 API 3-5  
 Windows NT  
   making a GCI library 1-13  
   Registry 1-14  
   UVGCIDLLS environment  
     variable 1-14, 1-15  
*windows.h* include file 3-5

---

## Symbols

!ERRNO subroutine 2-8  
 \$INCLUDE statement 2-2  
 \$TEST subroutine 2-3  
*.lib* files 1-5