What Do You Do After You See the Dollar Sign?

A Getting Started Guide to the XLNT® Language

Advanced Systems Concepts, Inc.

1180 Headquarters Plaza
West Tower, Third Floor
Morristown, NJ 07960
Phone: (973)539-2660 Fax: (973)539-3390
www.advsyscon.com

Fourth Printing: November 2012
Third Printing: March 2006
Second Printing: May 2001
First Printing: September, 1997

The information in this document is subject to change without notice and should not be construed as a commitment by Advanced Systems Concepts, Inc. (ASCI). ASCI assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under license, and may only be used or copied in accordance with the terms of such license. The terms and conditions of license are furnished with the product in both hard copy as well as electronic form.

ASCI logo and XLNT are registered trademarks of Advanced Systems Concepts, Inc.

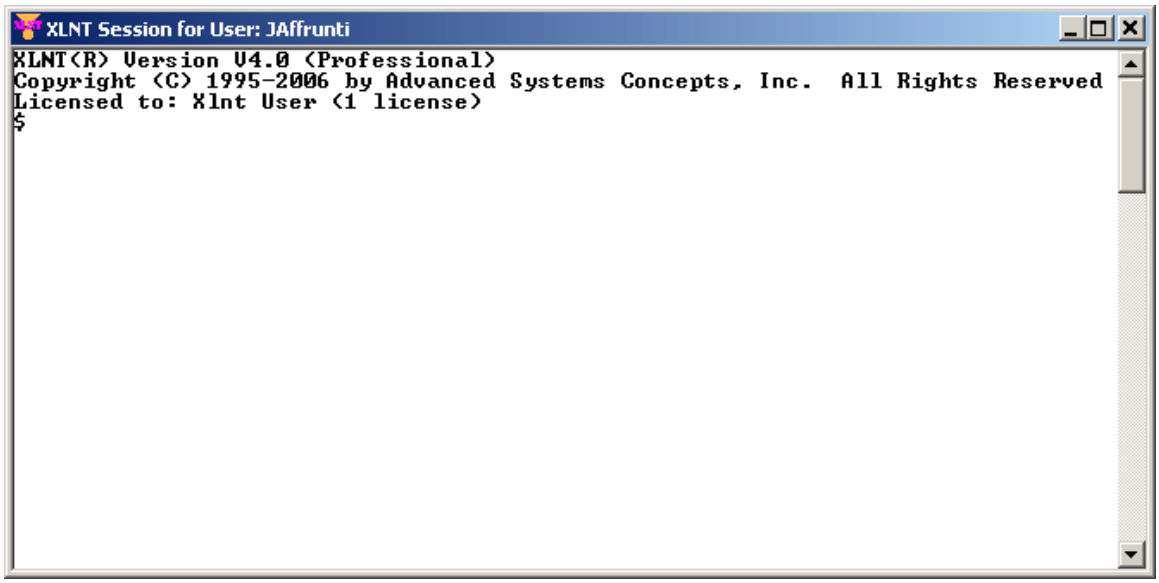
ASCI and XLNT logo are trademarks of Advanced Systems Concepts, Inc.

All other trademarks and registered trademarks are the property of their respective holders.

Welcome

Welcome to XLNT!

If you've followed the instructions to install and start the product, you should now be staring at an empty console window, on which is displayed a dollar sign ($). A question you may have at this point is "What do I do, now?".



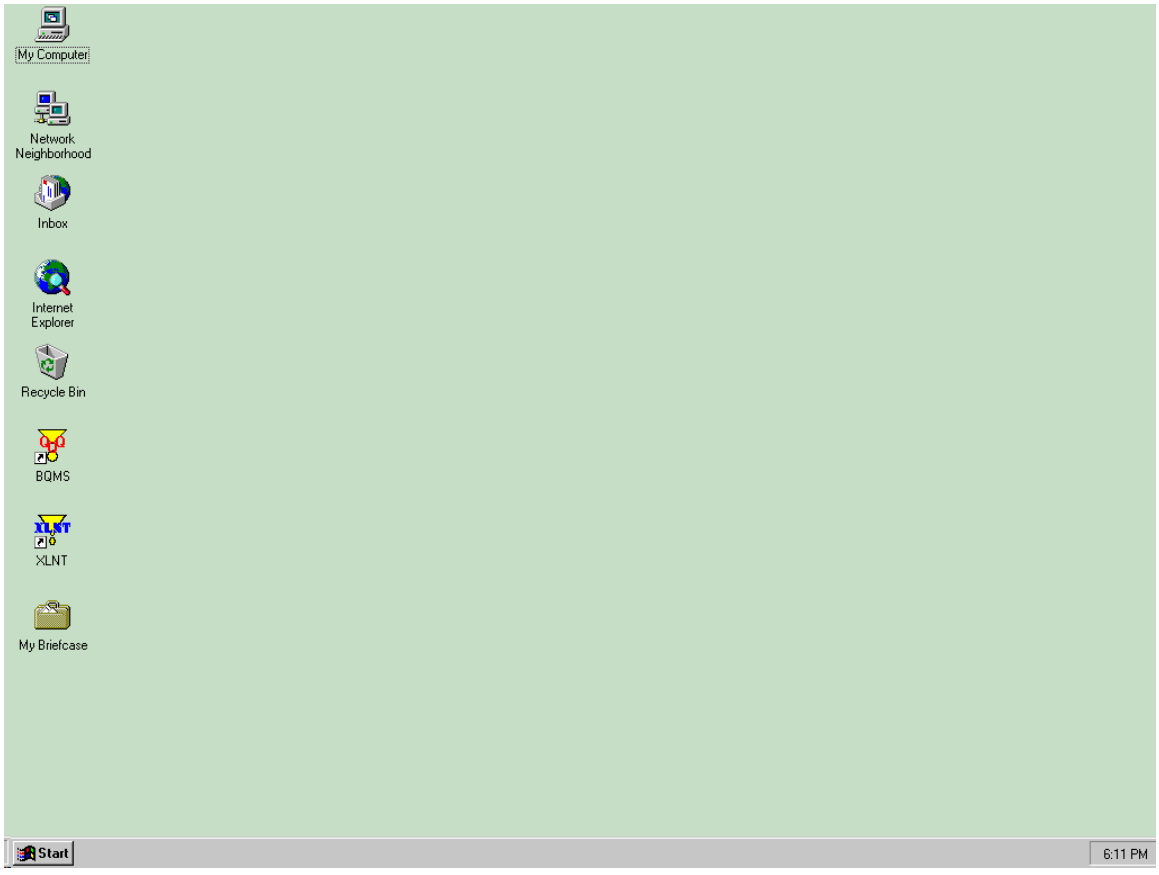
What you do now is the subject of this guide…

A Quick Overview

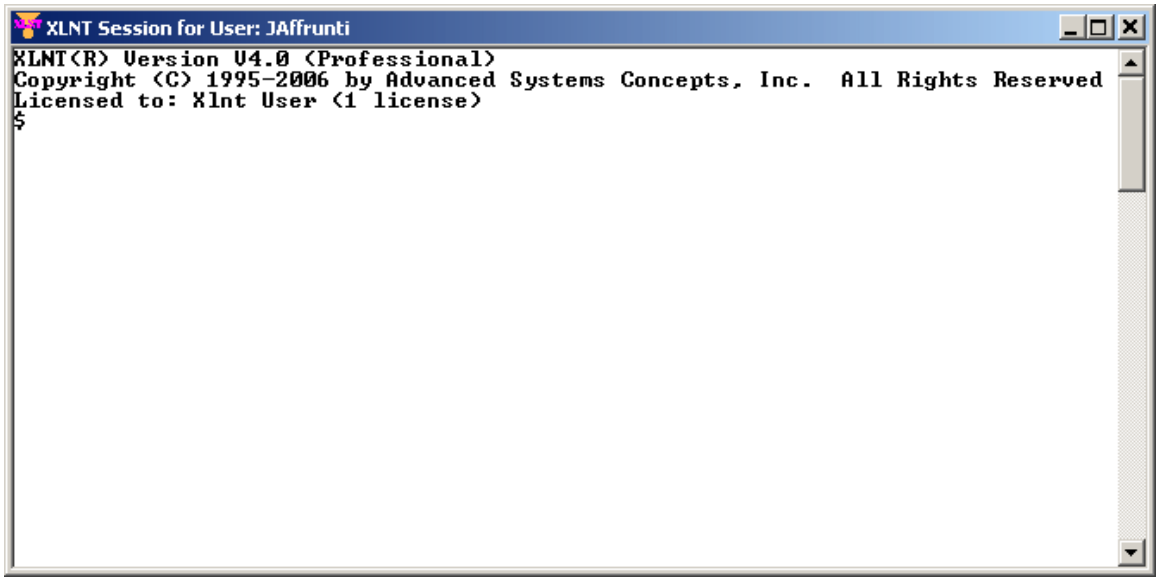
XLNT, the e***X***tended ***L***anguage for *Windows **NT***, is a new software product engineered by Advanced Systems Concepts, Inc. for the Windows marketplace. It provides a powerful, but easy to use scripting language that simplifies the development of command-line and batch interfaces for the users of these operating systems. XLNT eases the pain in tasks such as Remote Systems Administration, File Maintenance, CGI Development, Ad-hoc programming and Event Scheduling.

Using XLNT

The XLNT command interpreter is installed as a standard Windows desktop application:



When you click on its icon, a console window is activated. After a brief period of initialization, XLNT will prompt you for input. By default, the prompt is the dollar sign character, followed by a blank space:



When the prompt string appears, XLNT is waiting for you to tell it to do something. This is accomplished by typing an XLNT *command*. For example, if you wish to know the current date and time, enter the **SHOW TIME** command:

```
$ show time
```

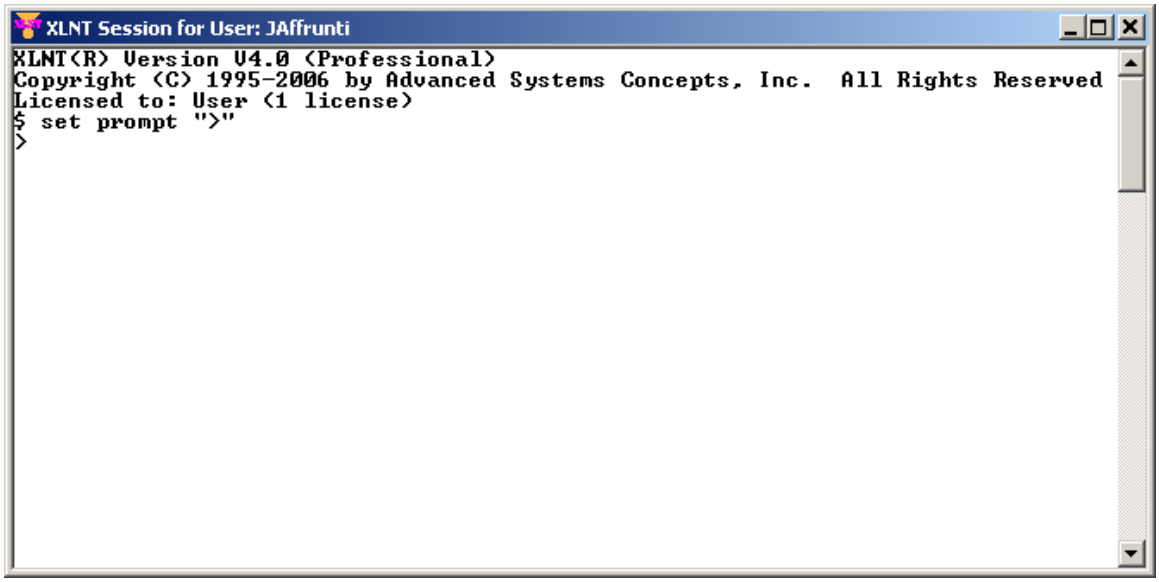
XLNT responds to this command by displaying the current day of the week, date and time in the following format:

```
Fri 06-May-2001 07:59:04.955
```

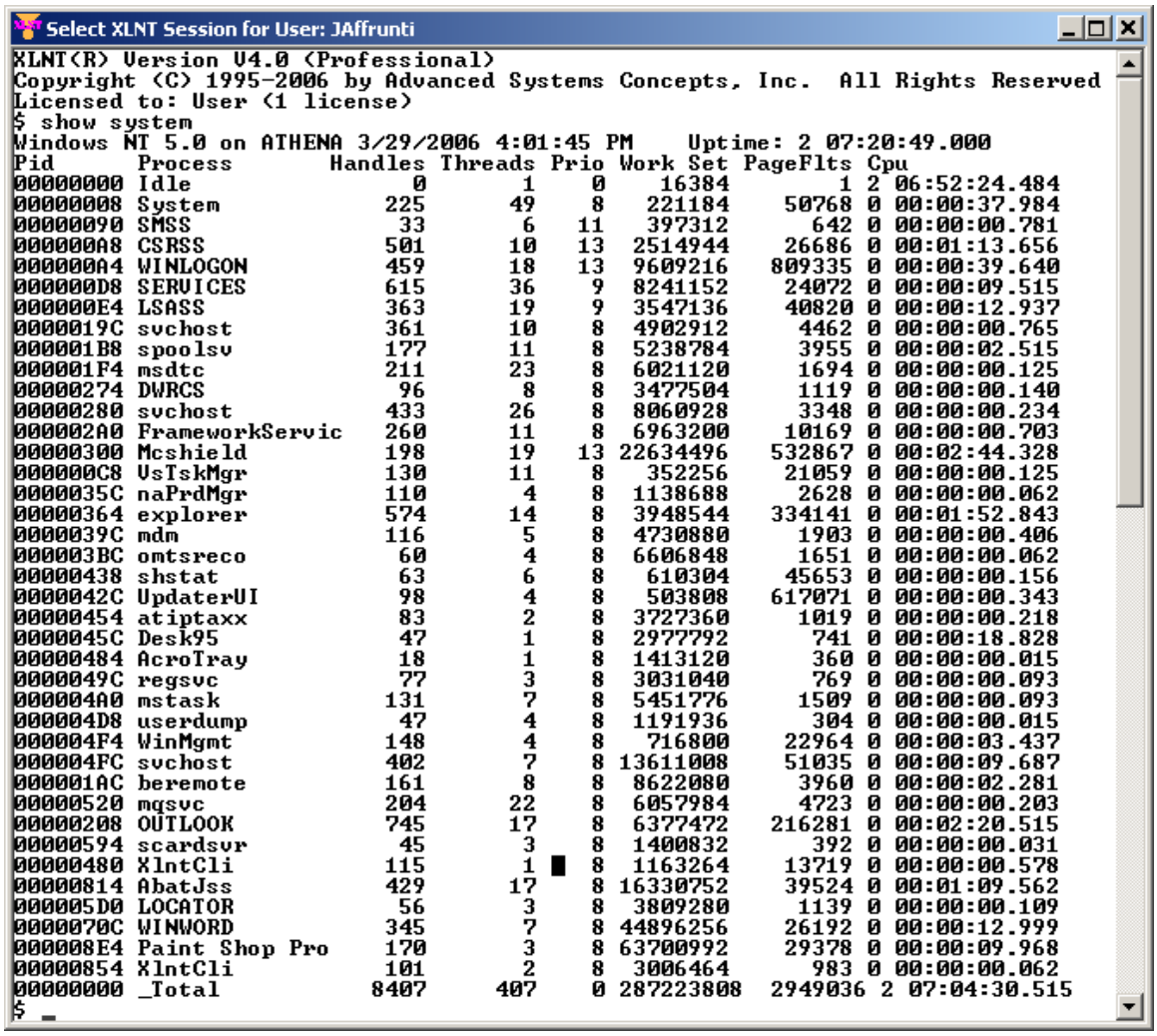
XLNT provides many commands. One of them, **SET PROMPT,** allows you to change the prompt string. If you want to be prompted by the string "XLNT> ", rather than $, enter the following:

```
$ set prompt "XLNT> "
```

Notice that the prompt changed immediately after you issued the command:



When you enter a command, XLNT will accept it, validate it and, based upon the results of these actions, either execute or reject it. If successfully executed, the output of the command will appear directly on your console or to wherever else you have directed it. The **SHOW SYSTEM** command displays the current status of your *Windows* system.


Select XLNT Session for User: JAffrunti
XLNT(R) Version V4.0 (Professional)
Copyright (C) 1995-2006 by Advanced Systems Concepts, Inc. All Rights Reserved
Licensed to: User (1 license)
$ show system
Windows NT 5.0 on ATHENA 3/29/2006 4:01:45 PM Uptime: 2 07:20:49.000
Pid Process Handles Threads Prio Work Set PageFlts Cpu
00000000 Idle 0 1 0 16384 1 2 06:52:24.484
00000008 System 225 49 8 221184 50768 0 00:00:37.984
00000090 SMSS 33 6 11 397312 642 0 00:00:00.781
000000A8 CSRSS 501 10 13 2514944 26686 0 00:01:13.656
000000A4 WINLOGON 459 18 13 9609216 809335 0 00:00:39.640
000000D8 SERVICES 615 36 9 8241152 24072 0 00:00:09.515
000000E4 LSASS 363 19 9 3547136 40820 0 00:00:12.937
0000019C svchost 361 10 8 4902912 4462 0 00:00:00.765
000001B8 spoolsv 177 11 8 5238784 3955 0 00:00:02.515
000001F4 msdtc 211 23 8 6021120 1694 0 00:00:00.125
00000274 DWRCS 96 8 8 3477504 1119 0 00:00:00.140
00000280 svchost 433 26 8 8060928 3348 0 00:00:00.234
000002A0 FrameworkServic 260 11 8 6963200 10169 0 00:00:00.703
00000300 Mcshield 198 19 13 22634496 532867 0 00:02:44.328
000000C8 VsTskMgr 130 11 8 352256 21059 0 00:00:00.125
0000035C naPrdMgr 110 4 8 1138688 2628 0 00:00:00.062
00000364 explorer 574 14 8 3948544 334141 0 00:01:52.843
0000039C mdm 116 5 8 4730880 1903 0 00:00:00.406
000003BC omtsreco 60 4 8 6606848 1651 0 00:00:00.062
00000438 shstat 63 6 8 610304 45653 0 00:00:00.156
0000042C UpdaterUI 98 4 8 503808 617071 0 00:00:00.343
00000454 atiptaxx 83 2 8 3727360 1019 0 00:00:00.218
0000045C Desk95 47 1 8 2977792 741 0 00:00:18.828
00000484 AcroTray 18 1 8 1413120 360 0 00:00:00.015
0000049C regsvc 77 3 8 3031040 769 0 00:00:00.093
000004A0 mstask 131 7 8 5451776 1509 0 00:00:00.093
000004D8 userdump 47 4 8 1191936 304 0 00:00:00.015
000004F4 WinMgmt 148 4 8 716800 22964 0 00:00:03.437
000004FC svchost 402 7 8 13611008 51035 0 00:00:09.687
000001AC beremote 161 8 8 8622080 3960 0 00:00:02.281
00000520 mqsvc 204 22 8 6057984 4723 0 00:00:00.203
00000208 OUTLOOK 745 17 8 6377472 216281 0 00:02:20.515
00000594 scardsvr 45 3 8 1400832 392 0 00:00:00.031
00000480 XlntCli 115 1 8 1163264 13719 0 00:00:00.578
00000814 AbatJss 429 17 8 16330752 39524 0 00:01:09.562
000005D0 LOCATOR 56 3 8 3809280 1139 0 00:00:00.109
0000070C WINWORD 345 7 8 44896256 26192 0 00:00:12.999
000008E4 Paint Shop Pro 170 3 8 63700992 29378 0 00:00:09.968
00000854 XlntCli 101 2 8 3006464 983 0 00:00:00.062
00000000 _Total 8407 407 0 287223808 2949036 2 07:04:30.515
$ _

If your command is rejected, a specific error message will appear, detailing the reasons for the rejection. XLNT messages follow a standard pattern:

FACILITY-S-MSGID, text

where:

FACILITY indicates the component of the product that is generating the error; in most cases, the facility name will be XLNT.

S is a single character indicating the message's severity level. It will be one of the following:

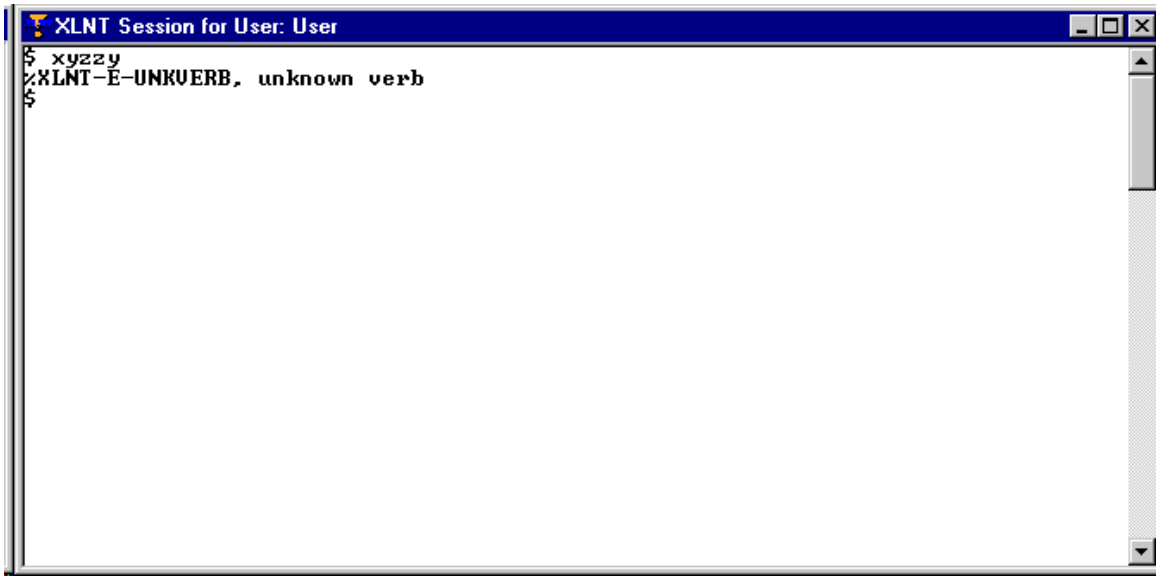
E - Error (most severe status)
W - Warning
I - Informational
S - Success

Generally, success and informational messages won't be displayed.

MSGID is a short string that uniquely identifies the message.

Text is the full explanatory text of the message.

Typing an invalid command, for example, will result in the following:



XLNT will continue to prompt you for commands and execute them until you enter the **LOGOUT** command. When you do, the command interpreter process will exit and the console window will be removed from your desktop.

When operating in this manner, you are at the XLNT interactive prompt level. After using the product for a while, you will probably find yourself executing the same sequence of commands over and over. Since this can quickly become tedious, XLNT allows you to store command sequences within a text file and "play" them back at a later time, just as if you were entering them at the command prompt. This text file is called an XLNT *command procedure*. Command procedures provide a great deal of power and flexibility.

Use your favorite text editor to create a command procedure. By convention, the file's extension should be *.xcp*, for XLNT Command Procedure. When running a command procedure, XLNT will assume the .xcp extension if it is not provided. The actual mechanics of command procedures are discussed in a later section, however, we will now present some typical uses of procedures.

An area of concern to system administrators is the management of disk space. Gathering information on the use of your hard disks can be extremely time consuming. XLNT provides functions that can ease this task. A command procedure can be developed to scan a hard disk and display the name of each file, its owner, and its size, providing a gross look at how the disk is being consumed. If you were to name such a procedure *showuse.xcp* and you want to use it to look at your "C" drive, the command to run it would be:

```
$ @showuse c:
```

A portion of the output generated by this command follows:

```
File: C:\ASCI Owner: Administrators Size: 0
File: C:\AUTOEXEC.BAT Owner: Administrators Size: 0
File: C:\BOOT.BAK Owner: Administrators Size: 485
File: C:\BOOT.INI Owner: Administrators Size: 480
File: C:\BOOTSECT.DOS Owner: Administrators Size: 512
File: C:\FONTS.DOC Owner: Administrators Size: 11776
                                .
                                .
                                .
```

Now, let's take a look at the contents of the command procedure:

```
$!
$! Command Procedure to display disk usage
$!
$ loop:
```

```
$       fname = f$search ("''P1'\...\*.*", ctx)
$       if  fname .eqs. "" then goto finish
$       owner = f$file_attributes (fname,  "MBM")
$       size     = f$file_attributes (fname,  "ALQ")
$       write $stdout "File:  ''fname'  Owner: ''owner'  Size:  ''size'"
$       goto loop
$ finish:
$       exit
```

The procedure makes use of several features that are available within XLNT. It accepts an argument (the device name) so that you will not need a separate procedure for each disk you want to scan. Since it is the first parameter on the command line, the argument is referred to by the symbol P1. The procedure then enters a simple loop, looking at each file on the disk. The *f$search* lexical function then returns a string containing the full name of the next file in the specified directory. (Lexical functions are built-in XLNT routines that can be invoked by command procedures to perform many useful functions - see section on *Lexical Functions*.) The procedure uses a wild-card specification to look at every directory on the disk. The *f$file_attributes* lexical is used to retrieve various items of interest about a specific file. In this case, the owner name (as defined by the "MBM" item code) and the file's size or allocation quantity (as defined by the "ALQ" item code) are retrieved. When this information is obtained, it is displayed on the standard output device and the procedure loops back for the next file. When *f$search* returns a null string, all the files on the disk have been scanned.

To use the *showuse.xcp* procedure on another drive, just change the device letter:

```
$ @showuse  d:
```

Now that you have the gross usage of the device, you might want to narrow your focus to individual users. To do that, we can modify the preceding command procedure to look for files owned by a specific user. Let's say we call this new procedure *showowner.xcp*. It will accept two arguments: the name of the device you wish to search, and the name of the user you want to scan for. If we want to search the C disk for all files owned by user *jjones*, the command would be:

```
$ @showowner  c:  jjones
```

A portion of the output produced by this procedure follows:

```
Disk Usage on C: by  jjones
File:  C:\TESTAPP\TESTAPP.CPP  Size:  10648
```

```
File   C:\TESTAPP\TESTAPP.RC  Size:  3068
File:  C:\TESTAPP\TESTAPP.RES  Size:
File:  C:\TESTAPP\README.TXT  Size
                    .
                    .
                    .
```

The *showowner.xcp* command procedure looks like this:

```
$!
$!  Command Procedure to Display Disk Usage by Owner
$!
$       requested_owner = P2
$       file_count = 0
$       total_size = 0
$       write $stdout "Disk Usage on ''P1' by  ''requested_owner'"
$       fname = f$search ("''P1'\...\*.*", ctx)
$       while  fname  .nes.  ""
$               owner = f$file_attributes (fname, "MBM")
$               If owner .eqs. requested_owner
$               then
$                       size = f$file_attributes (fname, "ALQ")
$                       write $stdout "File: ''fname'  Size: ''size'"
$                       file_count = file_count + 1
$                       total_size  = total_size + size
$               endif
$               fname = f$search ("''P1'\...\*.*",  ctx)
$       endwhile
$       if  file_count .gt. 0
$       then
$               write $stdout "Number of Files:  ''file_count'"
$               write $stdout "Total Size        :  ''total_size'"
$       endif
$       exit
```

This procedure will accept two arguments: the device letter (P1) and the user name (P2). As each file name is retrieved, its owner's name is compared to that requested. If equal, the file's name and size are displayed. Note that in this command procedure, the looping mechanism is performed by the **WHILE/ENDWHILE** statements. The WHILE command will repetitively execute the commands within its block as long as the specified condition is true. In this case, the loop will be performed as long as the file name is not a null string. As an added feature, the number of files owned by this user, and their total disk size, are computed. These values are displayed when the procedure completes.

If you are charged with the administration of your Windows network, you are probably intimately familiar with the *User Manager for Domains* facility. This graphical application allows you to maintain your domains' security by managing user accounts and groups and domain security policies. It provides all of the functions to do the job, coupled with the convenience of a graphical user interface. However, if you must add large numbers of users to your domain, using a GUI can be a daunting task. Only one user can be added at a time and you must be physically present to add each one.

XLNT provides a group of commands that fulfill the functions required for domain and workstation security management. These commands allow you to create, modify, delete, and display users, groups and security policies.

The **SECURITY CREATE** command is used to create users and groups for a domain or workstation within the Windows security system. Its format is:

```
SECURITY CREATE/qualifiers type entity
```

where *type* is either USER or GROUP and *entity* is either a group or user name. Many XLNT commands accept *qualifiers*, which are simply items of information supplied to the command, which can alter its operation. For example, the */DOMAIN* qualifier directs the SECURITY CREATE command to operate on the current domain, rather than the local workstation.

If you enter the following command in interactive mode:

```
$ security create user testuser /domain /password=test
```

a user named *testuser* will be created on the current domain. Testuser will have a password of *test*. (Note, there are many other security qualifiers that can be specified on the command line, but for illustrative purposes we have limited them to the required ones.) Entering new users in this manner is easy, but it doesn't give you any great advantage over the standard GUI. However, if you have to create large numbers of new users, a different method may be used.

XLNT solves this problem with a simple command procedure, rather than you having to sit at your keyboard for hours or resort to writing programs. A text file containing user names and passwords could be created. If the names and passwords are separated by a blank space, a command procedure could be developed that would read the file and add each user name found to the security database. Assuming that the name of the file containing the names and passwords was *c:\secure\users.dat*, the following procedure would do the job:

```
$!
$!  Command Procedure to Add New Users
$!
$       open user_file  "c:\secure\users.dat"
$ loop:
$       read user_file/end_of_file=done user_record
$       usernam = f$element (0, "   ",  user_record)
$       passwrd = f$element (1, "    ",  user_record)
$       security create user 'usernam'/domain/password='passwrd
$       goto loop
$ done:
$       close user_file
$       exit
```

This procedure illustrates some of the file commands supplied by XLNT. The **OPEN** command opens the file and assigns a symbol, in this case *user_file*. This symbol will be used by subsequent XLNT file commands to refer to the file. The **READ** command accesses a record in the file and assigns its value to the symbol defined as *user_record*. If an end-of-file condition occurs, control is transferred to the label defined as *done*. The *f$element* string lexical is then used to extract the user name and password from the record. These are then input to the **SECURITY CREATE** command, to add the new user. The procedure will continue to loop in this manner until an end-of-file is reached, at which point the file will be closed and the procedure will exit. While this procedure is running, you can be performing other useful tasks.

(Note that this procedure is somewhat simplistic. In a real environment, you would probably implement some error checking to make sure everything worked. XLNT provides extensive error checking facilities.)

Windows provides a facility that allows system administrators to execute a *login script* to implement common as well as user-specific commands. Typically these commands, issued when the user logs into the system, help set up the operating environment for the user. XLNT provides the perfect language and features for implementing the login script. To do so, create a small batch file (e.g., *login.bat*) with the following lines:

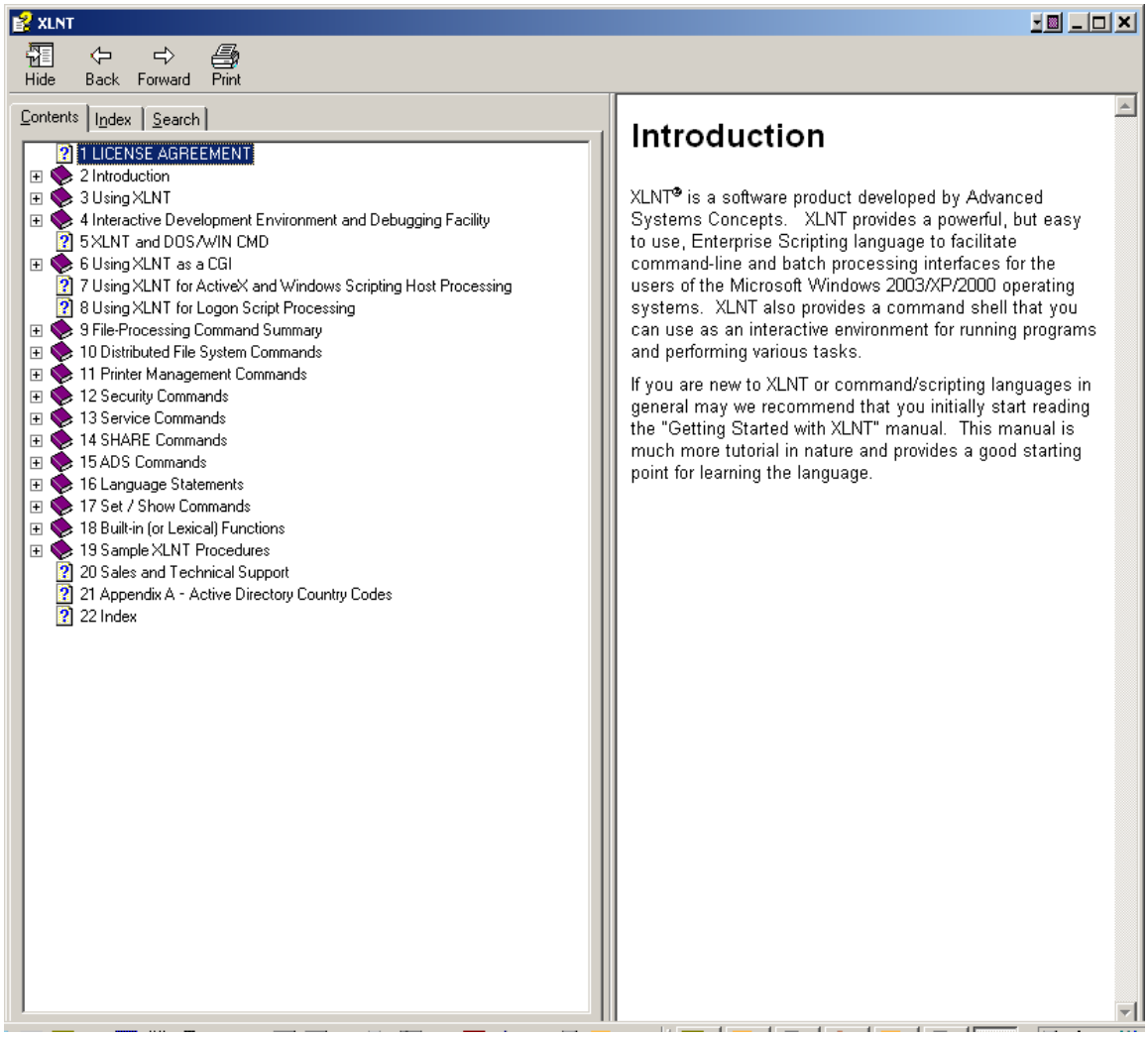
```
cd  c:\asci\xlnt
xlnt @\\ntserver\login_script_dir\xllogin.xcp
```

where *ntserver* is the machine name of the NT Server, *login_script_dir* is the directory on the server where the command procedure is kept, and *xllogin.xcp* is the name of the command procedure to execute. The name of

the batch file that executes the command procedure should be entered into the Login Script Name field of each user environment profile.

A common requirement for login scripts is to check the groups to which the user belongs and set up the operational environment. Certain users may need access to network drives, based on the applications they intend to run. This information is usually inferred from the user's group membership. The XLNT **SECURITY SHOW USER** command can display complete information concerning the user's security profile and provide a programmatic approach to solving what should be an easy task.

We have barely skimmed the surface of the functionality offered by XLNT. In subsequent sections, we will go a little deeper into the concepts and facilities of this powerful language. For a complete and thorough description of the XLNT language, refer to the *XLNT Reference Manual* and the online help. The online help supplied with XLNT is quite comprehensive. Not only can you use it to look up specific items of interest, but it was designed to be read sequentially, giving a complete tutorial on the product. To use the help facility, type **HELP** at the XLNT command prompt.


XLNT
Hide
Back
Forward
Print
Contents
Index
Search
1 LICENSE AGREEMENT
2 Introduction
3 Using XLNT
4 Interactive Development Environment and Debugging Facility
5 XLNT and DOS/WIN CMD
6 Using XLNT as a CGI
7 Using XLNT for ActiveX and Windows Scripting Host Processing
8 Using XLNT for Logon Script Processing
9 File-Processing Command Summary
10 Distributed File System Commands
11 Printer Management Commands
12 Security Commands
13 Service Commands
14 SHARE Commands
15 ADS Commands
16 Language Statements
17 Set / Show Commands
18 Built-in (or Lexical) Functions
19 Sample XLNT Procedures
20 Sales and Technical Support
21 Appendix A - Active Directory Country Codes
22 Index
Introduction
XLNT® is a software product developed by Advanced Systems Concepts. XLNT provides a powerful, but easy to use, Enterprise Scripting language to facilitate command-line and batch processing interfaces for the users of the Microsoft Windows 2003/XP/2000 operating systems. XLNT also provides a command shell that you can use as an interactive environment for running programs and performing various tasks.
If you are new to XLNT or command/scripting languages in general may we recommend that you initially start reading the "Getting Started with XLNT" manual. This manual is much more tutorial in nature and provides a good starting point for learning the language.

Back to the Basics

Since its name states that XLNT is an extended language, you might be wondering what it is extended from. Therefore, a little history may prove useful at this point.

XLNT was patterned on DCL, the command language developed by Digital Equipment Corporation for its family of computers. In the early 1970's, DEC's PDP-11 family was the most popular of the 16-bit minicomputers then on the market. PDP-11 users spanned the spectrum of the computing industry, from scientific to commercial. Multiple operating systems were developed for the PDP-11. Among these were *RSTS/E* (a commercially-oriented time sharing system); *RSX-11* (a family of real-time operating systems); *RT-11*; *MUMPS*; in fact, Bell Laboratories originally designed *UNIX* for the PDP-11. Each of these operating systems had its own user interface, based on unique command languages.

In an effort to reduce the learning curve required by each operating system, Digital proposed a common command language. They produced a definition for this new language, called the *Digital Command Language Standard*, or *DCLS*. DCLS interpreters were implemented on several PDP-11 operating systems. However, the language really hit its stride when Digital introduced the VAX computer, in 1979.

The VAX was an immediate success. It was a 32-bit processor that gave its users a virtual address range of 4 gigabytes. Digital provided a single operating system for the VAX: VMS. The primary user interface to VMS was the standard command language that DEC had proposed for its previous generation of computers, although, by now, the language's name had lost its trailing 'S' and was known simply as DCL.

Over the years, DCL has grown into a robust, mature language familiar to thousands of world-wide users. After VMS was ported onto the 64-bit ALPHA machine in the late 1980's, DCL still remained its primary interface.

With the advent of Microsoft's cross-platform operating system, *Windows NT/2000/2003*, Advanced Systems Concepts saw the need for a powerful command line interface as an alternative to the graphical one provided by Windows. XLNT capitalizes on the strengths of DCL and extends them to fit the unique demands of a modern, twenty-first century operating system environment.

XLNT Command Format

The format of XLNT commands and the rules that govern their format are called **command syntax**. Since XLNT is based on DCL, much of its syntax is derived from that of DCL. XLNT commands generally follow this pattern:

$[label:] verb [parameter…] [/qualifier…]

A *label* is an optional string of up to 255 characters. It is used only within command procedures and identifies the targets of control flow statements. A label may consist of the characters A-Z, a-z, the underscore (_), dollar sign ($), and the digits 0-9. It may not begin with a digit, however. A label is terminated by a colon (:).

The *verb* is a word that identifies the command to be executed. It is required. As long as there is no ambiguity, the verb may be abbreviated.

A command may accept one or more *parameters*, which are items of data that you want the command to act on. For example, the **COPY** command copies files from one location to another. The name of the file or files to be copied is a parameter and the target location is another parameter.

$ copy c:\prodfiles*.dat d:\testfiles

For the COPY command, both of these parameters are required. Other commands allow for optional parameters and still other commands need no parameters at all. Parameters are separated from each other by spaces. If the parameter contains spaces, slashes, or other special characters, it must be enclosed in double quotes (“). Some parameters may be specified as a list of items. If so, the items in the list must be separated by commas.

Qualifiers allow you to alter the standard operation of the command. In the **SHOW SYSTEM** command presented above, the information displayed was for the local machine. If you are suitably privileged, the **SHOW SYSTEM** command will allow you to view the current status of other computers in your network. You tell the command which system to look at by specifying the **/ON** qualifier with the name of the target computer:

$ show system/on=gamma

This command will display the status of the computer named “Gamma”.

Qualifiers are specified by typing a slash (/) followed by the qualifier name. The name may be abbreviated as long as it is unambiguous. Some qualifiers may take a value. If so, the value must be separated from the qualifier name by an equal sign (=). If the qualifier value contains spaces, commas, slashes, or other special characters, it must be enclosed in double quotes (“).

Entering XLNT Commands

An XLNT command may be entered in upper or lower case, or both. Internally, the XLNT interpreter will set all elements of the command line to upper case before it operates on them. Exceptions to this rule are data contained within double quotes and the arguments to *foreign commands*, which will be explained in a subsequent section. To continue a command on additional lines, end each line (except the last) with a hyphen (-). You can split the command at any point where a space or comma would appear.

Command Procedures

While you most certainly will continue to use the interactive functions of XLNT, the real power of the language lies in its *command procedure* facility. A command procedure is a sequential text file that contains one or more XLNT commands. You may create a command procedure file with any text editor. When you invoke a command procedure, XLNT will attempt to open the file and, if successful, will begin to execute its commands. It will continue to do so until an **EXIT** command or the end of file is encountered.

You run a command procedure by entering the @ command, immediately followed by the command file name:

```
@c:\myfiles\mycommands.xcp
```

This command executes the commands contained in the file *mycommands.xcp*, located in the *myfiles* directory of the "c:" device. Note the file extension of ".xcp". This extension, which stands for "XLNT Command Procedure", is the default file extension for command procedure files. It will be assumed if you omit the extension when you invoke the procedure.

A command procedure can be simply a sequence of XLNT commands that you want to execute repeatedly. If you want to know the current date and time, the directory you are currently in, and the status of your process, you might create a command procedure with the following lines:

```
$ ! A Simple Command Procedure
$   show time
$   show default
$   show process
$   exit
```

Note that the first command begins with an exclamation point (!). This indicates that the line contains a *comment*. Comments are for documentation purposes only and have no effect on the procedure's execution. An entire line need not be dedicated to a comment. Anything following the exclamation point is considered part of the comment:

```
$   show  time          ! Display the current time
```

Command procedures can be quite complex and perform many functions. A command procedure can invoke another command procedure. XLNT will

allow up to 32 levels of command procedure nesting. Level zero is the interactive prompt level. As each command procedure is started, the nesting level will be incremented. When the procedure completes, the level number is decremented and control is returned to the next higher command level – either an invoking procedure or interactive level. A **LOGOUT** command, executed at any nesting level, will end the XLNT session and cause the command interpreter process to exit.

All of the commands that you use interactively (and quite a few that you cannot) can be executed within a command procedure. Command procedures do not require the physical presence of a human being, as does the use of a GUI. For example, a command procedure that backs up your disk drives can be scheduled to run in the off-hours, when no users are around.

Flow of Control

The commands within a procedure are normally executed sequentially, one line after the other, until the end of the file is reached. Many times, however, you will find it necessary to alter the flow of control through the procedure, based on the results of condition testing. XLNT provides a comprehensive set of commands to accomplish this.

The **IF** command is the method used by command procedures to make decisions. It has two forms:

```
$ if expression then command
```

and

```
$ if expression
$ then
$       command
$             .
$             .
$             .
$ else
$       command
$             .
$             .
$             .
$ endif
```

The first form of the command tests an expression and, if the result is true, executes the single command on the same line. If the result is false, the command is skipped. With the second form, the expression is tested and, if

true, the sequence of commands following the **THEN** command is executed. The **ELSE** command is optional but, if specified and the result of the expression is false, the commands following the **ELSE** are executed.

The **GOTO** command causes a direct branch to be taken to a label within a command procedure. It can be used to jump around large sections of code and to implement simple loops. The following command sequence increments a counter until it exceeds 10, then exits the command procedure:

```
$       count = 0
$ loop:
$       if count .gt. 10 then goto break_out
$       count = count + 1
$       goto loop
$ break_out:
$       exit
```

The **GOSUB** command invokes a subroutine, executes the commands within it, then returns to the command following the **GOSUB.**

```
$       msg = "Displaying process information"
$       gosub  show_status
$                   .
$                   .
$                   .
$       exit
$ show_status:
$       write $stdout msg
$       show time
$       show process
$       return
```

The **CALL** command is similar to the **GOSUB** command, except that it creates a new command level to execute the specified subroutine, just as though it was a separate command procedure. It is also possible to pass from one to eight arguments, or parameters, to a called subroutine. The subroutine refers to these parameters by the symbols *P1* through *P8*.

```
$       msg = "Displaying process information"
$       call  show_status msg
$                   .
$                   .
$                   .
$ show_status:  subroutine
$       write $stdout p1
```

```
$       show time
$       show process
$       endsubroutine
```

The **WHILE, UNTIL,** and **FOR** commands can be used to implement complex loops.

The **WHILE** commands tests an expression and performs the body of the loop as long as the expression is true:

```
$ while expression
$         command
$             .
$             .
$             .
$ endwhile
```

The **UNTIL** command tests an expression and executes the commands within the body of the loop until the expression becomes true:

```
$ until expression
$         command
$             .
$             .
$             .
$ enduntil
```

The **FOR** command executes the commands within a loop a given number of times, until the specified expression is evaluated as true. Execution of a **FOR** command proceeds as follows:

1. The *init-command* is executed.
2. The *expression* is evaluated. If false, the commands within the loop are executed. If true, control is transferred to the first command following the terminating **ENDFOR** command.
3. The *iterate* command is executed and step 2 is repeated.

```
$ for (init-command, expression, iterate-command)
$       command
$             .
$             .
$             .
$ endfor
```

The **LEAVE** command immediately exits the body of a loop, regardless of the result of the expression. If encountered out of the body of a loop, the command is ignored.

The **REPEAT** command passes control to the next iteration of the **WHILE, UNTIL,** or **FOR** block in which it appears.

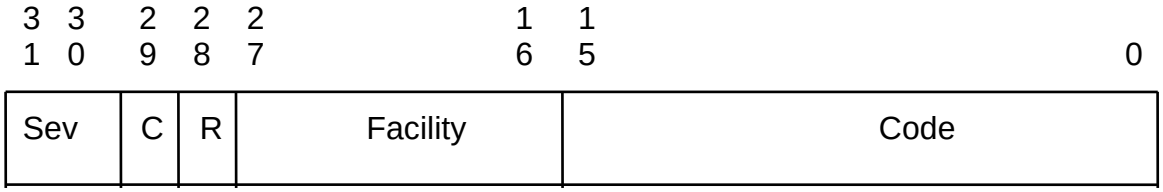
"Login" Command Procedures

XLNT gives you the ability to supply "login" command procedures. These are command procedures that are executed automatically when you start up, or "log in" to, the XLNT command interpreter. Two types of login interfaces are provided: a system-wide login procedure, which is executed whenever *anyone* uses XLNT; and a user-specific login procedure, which is executed only when that specific user starts up XLNT.

The **SET PREFERENCES** command allows you to control various elements of your XLNT environment. Among other things, you may use it to specify the path names of your login command procedures. In order to supply the path for the system-wide login procedure, you must be a member of the *Administrators* group.

Error Handling

When an XLNT command completes processing, it returns a status code indicating its success or failure. The status code is a 32-bit value, formatted as follows:



where:

Sev is the severity code:
- 00 - Success
- 01 - Informational
- 10 - Warning
- 11 - Error

C is the Customer Code Flag
R is a Reserved bit
Facility is the facility code
Code is the status code

The status code is used to set the values of two reserved global symbols: **$STATUS** and **$SEVERITY**[1]. As the names of these symbols imply, the value of the status code is inserted into $STATUS and the severity code is inserted into $SEVERITY.

Each status code corresponds to an XLNT message. If you are operating at the interactive prompt level, and the command completes in error, the message will be displayed on your console. You can then take appropriate action to correct the problem. When a command procedure is running, however, you are not there to take appropriate action. It is therefore up to the procedure itself to handle any error conditions that may occur.

An XLNT command procedure has three methods it can employ to handle errors: it can rely on XLNT's default error handler; it can allow XLNT to detect an error, but the procedure can supply its own error handler; it can disable XLNT's error checking mechanism and perform this function itself.

XLNT normally checks the status of each command that has completed. If the status contains an *error* severity and no other error handler has been supplied, XLNT will invoke its *default error handler*. The default error handler will terminate the command procedure and return the error status. The default error handler will not be executed if the status code contains a *success*, *informational*, or *warning* severity code. The following command procedure excerpt will use the **DELETE** command to attempt to delete a file. If the file does not exist, the DELETE command will return a File Not Found status. This status has a warning severity. Since default error handling is in place, the command procedure will continue to execute even though the DELETE command completed unsuccessfully.

```
$     delete blink.dat
$     show time
$     exit
```

A command procedure can supply its own error handlers via the **ON** command. The format of the ON command is:

```
ON condition THEN command
```

[1] See the section on SYMBOLS for a discussion of global symbols.

where *condition* is the severity level to be handled (i.e., WARNING or ERROR) and *command* is the action to be taken when the condition is raised. If the command specifies **ON ERROR** then only error severity will be handled. However, if the command specifies **ON WARNING**, then both warning and error conditions will be handled. The following command procedure is similar to the first, except that the procedure has provided its own error handler. In this case, if the file to be deleted does not exist, the File Not Found status will cause the command following the THEN clause of the ON command to be executed. This is a **GOTO** command that causes a direct branch to the label *error_handler*. This causes the command procedure to exit with the File Not Found status.

```
$       on warning then goto error_handler
$       delete  blink.dat
$       show time
$       exit
$ error_handler:
$       exit_status  =  $status
$       exit  exit_status
```

The **SET NOON** command disables all XLNT error handling. The following command procedure issues a SET NOON, then attempts to delete a file. The procedure then explicitly checks the severity of the DELETE command. If the severity code is 2 or greater (WARNING is binary 10, or decimal 2, ERROR is binary 11, or decimal 3), the procedure branches off to its error handler. Otherwise, it continues to execute.

```
$       set  noon
$       delete blink.dat
$       if  $severity  .ge.  2  then goto error_handler
$       show time
$       exit
$ error_handler:
$       exit_status = $status
$       exit  exit_status
```

Symbols

Symbols are the means by which you supply data to an XLNT session or command procedure. Symbols correspond to variables in other programming languages. Each symbol has four properties: name, datatype, level, and value.

The symbol's *name* identifies it and allows you to refer to it. The name can consist of any combination of letters, digits, dollar sign ($), and underscore character (_), however, the name cannot begin with a digit. Symbol names must be unique within a level.

The symbol's *datatype* defines the type of data you can store in the symbol. XLNT supports several datatypes (see below).

Each symbol has a command *level* associated with it. The level determines the symbol's scope within the XLNT session. Symbols may be defined globally, in which case they are available to the command level at which they were created and to all command levels below that. Or a symbol may be defined as local, in which case it is available only to the current command level.

The symbol's *value* is the actual data that is associated with it.

A symbol can be defined explicitly or implicitly. An explicit definition is the result of issuing a **DECLARE SYMBOL** command. An implicit definition occurs when you specify a non-declared symbol name as the target of an *assignment* command.

```
$ test_symbol = 125
```

This command creates a new symbol, at the current command level, named **test_symbol**. Because the value being assigned is numeric, test_symbol will be created as an integer symbol. An integer is a signed 32-bit numeric value. Once a symbol is defined, it can be used in subsequent commands.

```
$ new_symbol = test_symbol + 10
```

The above command combines an assignment statement with a numeric expression. It adds 10 to the value of test_symbol (125), creates a new symbol called **new_symbol**, and stores the result of the add operation into new_symbol. New_symbol now contains an integer value of 135, but the value of test_symbol remains unchanged.

A symbol's value can be modified by a new assignment command. If it is not an explicitly declared symbol, its datatype can also be changed.

```
$ test_symbol = "This is a string"
```

This command redefines test_symbol as a string symbol. The previous value of test_symbol is no longer available and its datatype has been changed.

The following command creates an implicit global symbol:

```
$ test_symbol == "This is a global string"
```

The double equal sign (= =) is the notation for global assignment. This symbol does not replace the previous version of test_symbol, since they both exist at different command levels.

Integer and *string* are the only datatypes that can be assigned to implicitly defined symbols.

To *explicitly* define a symbol, use the **DECLARE SYMBOL** command. The format of this command is as follows:

DECLARE SYMBOL ***datatype name[=initial-value] [options…]***

Many more datatypes can be assigned to explicitly declared symbols. The following table presents each datatype's keyword and description.

Datatype	Description
INTEGER	32-bit signed binary value
LONG	32-bit signed binary value
SLONG	32-bit signed binary value
ULONG	32-bit unsigned binary value
DWORD	32-bit unsigned binary value
WORD	16-bit signed binary value
SWORD	16-bit signed binary value
UWORD	16-bit unsigned binary value
BYTE	8-bit signed binary value
SBYTE	8-bit signed binary value
UBYTE	8-bit unsigned binary value
STRING	Character String value
HANDLE	Handle
VOID (functions only)	Void Function Argument

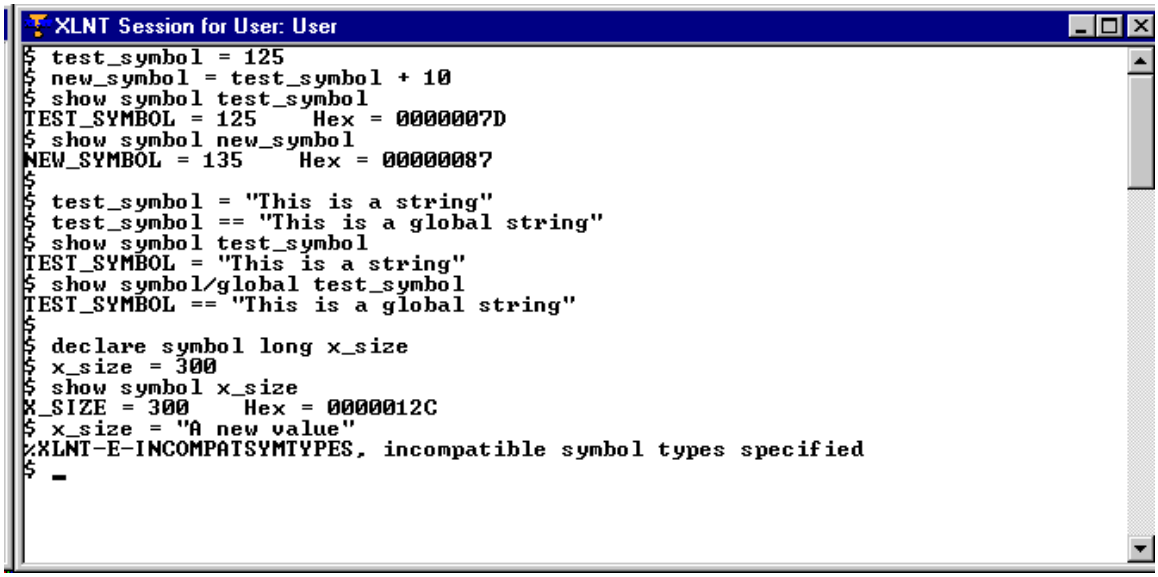
The next two commands define, in turn, a local, 32-bit unsigned symbol and a global 16-bit signed symbol containing an initial value of 100:

```
$ declare symbol dword  BigValue
$ declare symbol word   Counter=100  global
```

The datatypes of explicit symbols cannot be changed by subsequent assignment statements:

```
$ declare symbol long x_size
                .
                .
                .
$ x_size = 300
$ x_size = "A new value"
```

In the previous command sequence, the last command would fail because it attempted to assign a string value to an explicitly-declared numeric symbol.



Symbols have many uses in XLNT. Among the most common are:

- as a synonym for a command or command line
- as a variable in a command procedure
- as a value for a declared function or lexical
- to represent file and data record objects in the I/O related commands
- as input arguments to command procedures

In addition to the symbols you define, XLNT provides the following permanent symbols (which are available at any command level):

$JOBID - current batch job number
$LOCAL_MACHINE - local computer name string
$LOGIN - default login directory string
$PID - current process identifier
$REMOTE - XLNT remote login flag
$SEVERITY - severity level of last completed operation
$STATUS - status of last completed operation
$STDERR - handle of STD_ERROR device
$STDIN - handle of STD_INPUT device
$STDOUT - handle of STD_OUTPUT device
$USERNAME - current username string

The following three permanent symbols are present to provide VMS compatibility:

**SYS$ERROR -** handle of STD_ERROR device ($STDERR)
**SYS$INPUT -** handle of STD_INPUT device ($STDIN)
**SYS$OUTPUT -** handle of STD_OUTPUT device ($STDOUT)

Command Shortcuts

A common use for symbols is as abbreviations for long XLNT command names. For example, you can equate the symbol **ST** to the XLNT command **SHOW TIME:**

```
$ ST = "SHOW TIME"
$ ST
$ Sat 07-Jul-1997 11:47:27.416
```

Convenient names can be assigned to frequently used commands. The **SET DEFAULT** command is the XLNT method of changing directories. As described above, the $LOGIN symbol is always set to your default login

directory. You may then define a symbol named **HOME** to always return you to your home directory:

```
$ home = "set default $login"
```

The use of symbols in this manner is limited only by your imagination.

Abbreviating Symbols

Symbol names may be abbreviated through use of the asterisk (*) character. The following example creates the symbol **WHAT** to invoke the SHOW SYSTEM command. The symbol can be abbreviated as WH or WHA:

```
$ wh*at = "show system"
```

Foreign Commands

If you equate the file name of a non-XLNT executable to a symbol, you can run the executable image by typing the symbol name. A symbol that runs an executable image is referred to as a *foreign command*. A foreign command is an executable image that is not recognized by the command interpreter as an XLNT command.

The formats for defining a foreign command are as follows:

```
symbol-name :=[=] $image-file-name
symbol-name =[=] "$image-file-name"
```

Note that the dollar sign ($) is required for foreign command definition. There must be no space between the dollar sign and the file name.

If you have an executable named *myprogram.exe* that resides in the *myfiles* directory of the C: device, you can define a foreign command for it as follows:

```
$ doit := $c:\myfiles\myprogram.exe
```

Thereafter, whenever you type *doit* at an XLNT command prompt, the *myprogram.exe* executable will be run.

If a foreign command requires arguments, they can be entered on the same line as the command. Contrary to other XLNT command lines, these arguments strings will not be changed to upper case. They will remain as

you entered them. It is up to the foreign command's program to obtain these arguments and perform any parsing of the command line, itself.

XLNT also supports Automatic Foreign Command detection. This means that if you enter a command that would otherwise be invalid under XLNT, the XLNT interpreter will scan for "command.EXE" or "command.XCP" files to execute as though they were foreign commands. The following directories are searched in the following order:

- the XLNT installation directory
- the current directory
- the 32-bit Windows system directory
- the 16-bit Windows system directory
- the Windows directory
- the directories listed in the PATH environment variable

If you wish to run the NOTEPAD text editor, and Automatic Foreign Command detection is enabled, you may do so by simply typing:

```
$ notepad
```

Automatic Foreign Command detection can be enabled or disabled on an individual user basis. The **SET PREFERENCE** command is available to customize the XLNT environment to your individual preferences. To enable Automatic Foreign Command detection, type

```
$ set preference/autoforeign
```

To disable it, type:

```
$ set preference/noautoforeign
```

By default, Automatic Foreign Command detection is enabled.

Structures

A *structure* is a collection of fields, of possibly diverse data types, that are related in some manner. For example, an employee record consisting of the employee's name, Social Security number, address, salary, etc., could be defined as a structure. Most programming languages provide a mechanism for defining data structures. In XLNT, this is accomplished by the **STRUCTURE / ENDSTRUCTURE** commands.

Consider the following XLNT command sequence:

```
structure Xyz
     word    XyzType
     word    XyzSize
     dword   XyzIdent
     string  XyzName  32
endstructure
```

These commands define a structure named `Xyz` that consists of four fields. Each field is also known as a member of the structure. The four members are `XyzType` and `XyzSize`, both of which are defined as words (16-bit signed values); `XyzIdent`, a dword (32-bit unsigned value); and `XyzName`, a 32-byte character string. As specified above, the `Xyz` structure definition is local to the command level in which it has been defined. To make the definition global to the entire XLNT session, add the *global* parameter, following the structure name:

```
structure Xyz global
     word    XyzType
     word    XyzSize
     dword   XyzIdent
     string  XyzName  32
endstructure
```

Once a structure has been defined, it becomes an implicit XLNT datatype. To use this structure type, you must create an *instance* of the structure. To do so, use the standard XLNT **DECLARE SYMBOL** command:

```
declare symbol Xyz X1 initialize,global
```

The above command creates an XLNT symbol named `X1` of datatype `Xyz`. Notice the inclusion of the *initialize* and *global* keywords. The latter simply creates `X1` as a global symbol. The *initialize* keyword sets the members of

the structure to initial values: numeric fields are set to zero, string fields are set to spaces. Both keywords are optional.

To reference the individual members of a structure, specify the instance name, followed by the field reference character (|), followed by the field name. The following commands assign values to the members of the X1 structure instance:

```
X1|XyzType = 5
X1|XyzSize = 40
X1|XyzIdent = 100
X1|XyzName = "John Smith"
```

Because they are implicit datatypes, structure definitions may be included as the datatypes of members of other structures:

```
structure Abc
     word   AbcType
     word   AbcSize
     Xyz    AbcXyz
     string AbcAddress1 20
     string AbcAddress2 20
endstructure
```

Note that one of the members of structure Abc is a field described as a datatype Xyz. This includes all of the members of structure Xyz as part of structure Abc. The definition of structure Xyz must have been provided before its inclusion in the second structure. To reference the members of a "structure within a structure", it is necessary to fully qualifiy their names:

```
declare symbol Abc A1
A1|AbcType = 10
A1|AbcSize = 84
A1|AbcXyz|XyzType = 5
A1|AbcXyz|XyzSize = 40
A1|AbcXyz|XyzIdent = 100
A1|AbcXyz|XyzName = "John Smith"
A1|AbcAddress1 = "102030 4th Street"
A1|AbcAddress2 = "Hoboken, NJ"
```

To remove a structure instance, simply delete the symbol:

```
delete/symbol A1
```

Lexical Functions

Lexical functions are built-in XLNT routines that can be invoked by command procedures to perform many useful functions. Lexical functions are available to manipulate strings, perform various date and time conversions, obtain system information, and many other services.

The results of a lexical function can be assigned to a symbol or used directly in many XLNT commands. In this example, the F$GETSYI lexical is used to obtain the operating system version and build number:

```
$ write $stdout "Operating System: ''f$getsyi("VERSION") Build:  ''f$getsyi("BUILD_NUMBER")"
```

result:

```
Operating System: Windows NT 5.0 Build: 2195
```

The following list details the name and purpose of some of the XLNT lexicals:

F$ADDREGISTRY - adds a subkey and value to the registry
F$CHANGEREGISTRY - modifies a value in the registry
F$CHECKLIBRARY - determines if a specific DLL has been loaded
F$CVSI - converts specified bits of a character string to a signed integer
F$CVTIME - performs time and date conversions
F$CVUI - converts specified bits of a character string to an unsigned integer
F$DELETEREGISTRY - deletes values in the registry
F$DIRECTORY - returns the current directory string
F$EDIT - performs string editing
F$ELEMENT - extracts one element from a string of element
F$ENUMDOMAIN - enumerates domain information
F$ENUMMACHINE - enumerates machine server information
F$ENUMSHAREPOINT - enumerates network sharepoint information
F$ENVIRONMENT - returns information on command procedure environment
F$EXTRACT - extracts characters from a string
F$FILE_ATTRIBUTES - returns information on specified file
F$FORMAT - formats a character string
F$FORMATDATE - returns a date string in various formats
F$FORMATTIME - returns a time string in various formats
F$FREELIBRARY - unloads a DLL
F$GETVARIABLE - retrieves an environment variable
F$GETDVI - retrieves device information

F$GETJPI - retrieves process information
F$GETSYI - retrieves system information
F$INTEGER - returns the integer equivalent of the specified expression
F$LENGTH - returns the length of a string
F$LOADLIBRARY - loads a DLL
F$LOCATE - locates a substring within a string
F$LOOKUPREGISTRY - looks up a key and value in the registry
F$MESSAGE - converts a status code to a formatted message string
F$MODE - returns current command mode
F$MSGBOX - creates and displays a message box
F$PARSE - parses file specifications
F$PID - obtain a process identifier
F$READEVENT - reads entries from the Windows NT Event Log
F$REPLACE - searches for and replaces a substring within a string
F$REPORTEVENT - enters an event into the Windows NT Event Log
F$SEARCH - searches directories
F$SERVICE_STATUS - determines the status of a Windows NT service
F$STRING - returns the string equivalent of the specified expression
F$TIME - returns current date and time as a string
F$TYPE - returns the datatype of a symbol
F$VERIFY - indicates the command procedure verification level

User Functions

Besides the built-in functions provided by the lexical commands, XLNT allows you to invoke your own functions from within an XLNT session. This feature is enabled through the use of the **DECLARE FUNCTION** command.

DECLARE FUNCTION ***datatype name library arguments,... [options]***

The *datatype* parameter indicates the datatype that the function will return. You may supply one of the supported XLNT datatypes (see the section on symbols for a list of these datatypes).

The *name* parameter is the name of the function within the specified library. It must not conflict with another symbol name. If the name is case sensitive, it must be enclosed in double quotes (“).

The *library* parameter is the ***filename*** portion of the DLL that contains the function. For example, kernel32 for kernel32.dll.

The *arguments* parameter contains a list of the datatypes for each argument to the function. This parameter is optional.

The *options* parameter lists all options which further modify the handling or declaration of the function. Currently, the options available are **GLOBAL, CDECL,** and **STDCALL.** By default, function declarations are available to their own command level. The GLOBAL parameter makes them available throughout the XLNT session. CDECL is the default calling mechanism used by XLNT when invoking user functions. STDCALL is used when calling WIN32 API functions, and must be specified for them.

The following sequence of XLNT commands uses the Win32 *SetConsoleTitle* api function to change the name of the console window to "My Console". The *SetConsoleTitle* function resides in the "kernel32.dll" dynamic link library, so the first thing the command procedure does is to load the library. There are two forms of the *SetConsoleTitle* api: one for setting straight ASCII characters and one for wide characters. We will use the ASCII for, hence the "SetConsoleTitleA" function name. This function returns an unsigned long status and takes a character string as input.

```
$ k32 = f$loadlibrary("c:\winnt34\system32\kernel32.dll")
$ declare function ulong "SetConsoleTitleA" "kernel32" string stdcall
$ declare symbol ulong stat
$ stat = SetConsoleTitleA ("My Console")
```

Product Support

This guide has presented a brief introduction to the XLNT command and scripting environment. For more detailed and thorough information, consult the following :

> ***XLNT Reference Manual*, available from Advanced Systems Concepts, Inc.**
> ***XLNT Online Help* - type HELP at the XLNT prompt**

XLNT is a fully-supported product of Advanced Systems Concepts, Inc. ASCI provides many support programs to assist XLNT evaluators and customers. You can contact us in any of the following methods:

Telephone: **(973) 539-2660**
Fax: **(973) 539-3390**

Internet:
Sales: **sales@advsyscon.com**
This E-mail destination should be used for sales, pricing and other sales-related questions.

Technical: **support@advsyscon.com**
This E-mail destination should be used for technical product assistance.

Web Site: **http://www.advsyscon.com**
This destination represents Advanced Systems Concepts' world-wide web site. Product information, downloads and other requests can be satisfied by visiting this site.

Appendix A - Login Script

This command procedure is designed to be executed during Windows login. It will check the user account to determine which group(s) he or she belongs to. Based on this information, appropriate drive shares can be established. (Note, this command procedure is included for historical purposes, only. As of XLNT V4, Windows NT V4/95/98/Me are no longer supported. Only Windows 2000 and above.)

```
$! XLLOGIN.XCP
$! Version 1.5
$! Copyright 1997-2006, Advanced Systems Concepts
$! All Rights Reserved
$!
$! Author:   ASCI
$! Created:  07-22-97
$! Modified: 07-25-97
$!
$!
$! Instructions:
$!   In the variables section, change the ntmachine variable value
$!   NT-MACH, to the name of the NT-MACHINE that is running
$!   XLNTSERV. For ease this should be the Domain Controller.
$!   See notes for more details
$!   Right now the script checks if the user is in the Adminstrators
$!   Group. If so, it will do the code that is in the if statement
$!   Change this commented code to anything that you want to
$!   do once the group is checked. Also see the notes on this
$!   section for what to do for checking for multiple groups or
$!   also adding common code for all groups.
$!
$! Description:
$!   This procedure uses RPC Services Found in Service Pack 2
$!   of XLNT to lookup the account information of a person that
$!   is logged into an NT domain and determine if they are in
$!   a certain group or not. Based on that group the script can
$!   map a drive to the login name of the user or any other
$!   drive that the adminstrator wishes.
$!   This script works on Logging in from a Windows NT or '95 machine.
$!
$! Advanced Features:
$!   This script will first determine if the Machine is Windows NT
$!   or Windows '95. It will run the appropriate SECURITY command
```

```
$!   based on the operating system.
$!
$! Notes:
$!   There is a problem when the machine that XLNTSERV is a workstation
$!   that has a local user account of the same name that the user
$!   is logging into the domain. This is a problem only with Windows '95
$!   machines and the work around for it is to make just simply pick the
$!   primary domain controller for the /on= parameter in the security
$!   command
$!
$!
$! ****************************************************************
$! ****************************************************************
$! VARIABLES
$!
$  ntmachine = "\\NT-MACH"
$!
$! ****************************************************************
$! ****************************************************************
$!
$ echo = "write $stdout"
$ set noverify
$ set message /nomessage
$ on error then goto END_OF_SCRIPT
$ cls
$!
$! Here we are going to check what type of operating system we have and
$! determine which SECURITY command to use
$!
$ if f$getsyi("platform") .nes. "Windows NT"
$ then
$       echo "Invalid "
$       sec/out=sec.xt/noformat show user /win95 /on=""'ntmachine'"
$ else
$       echo "Logging in from a Windows NT Workstation"
$       sec/out=sec.txt/noformat show user /domain
$ endif
$!
$! This next line looks up the current user on the domain to obtain user info
$!
$y=1
$!
$! Now we are going to obtain the user name, local groups, and global groups
$! If you would like more information than this, just add the appropriate
$! label with a variable name. Also, tell the user we are doing it.
$!
```

```
$echo "Verifying User Information, Please Wait..."
$ open secfile sec.txt
$!
$ while y .eq. 1
$ read/end_of_file=loop_exit1 secfile infile
$ if F$LOCATE("User Name",infile) .ne. F$LENGTH(infile) then userline = infile
$ if F$LOCATE("Local Groups",infile) .ne. F$LENGTH(infile) then localgroups =
infile
$ if F$LOCATE("Global Groups",infile) .ne. F$LENGTH(infile) then globalgroups
= infile
$ endwhile
$!
$ loop_exit1:
$!
$! We are going to close the file and delete it so that the information is not saved
$! You can also save this information to a log file or other means of server
logging
$!
$ close secfile
$ del sec.txt
$!
$!
$! We have to parse the user name from the information so that we can use
$! it as a variable for drive mappings or any other comparisons
$!
$usertemp = ""f$element(1,"=",userline)"
$userlen = f$length(usertemp) - 1
$usersname= f$extract(1,userlen,usertemp)
$!
$! Code to look for a certain local or global group
$! if you only want it to search for local or global groups just
$! comment out or delete one of the if statements. Also, if you would like
$! to do things for multiple groups, just copy and paste the block below
$! labeled GROUP SEARCH AND PROCESS and change the name of SGROUP
to a group
$! of your choice
$!
$! ********************************************************************************
$! ************************** GROUP SEARCH AND PROCESS
**************************
$! ********************************************************************************
$!
$ SGROUP = "Administrators"
$found = 0
$ if F$LOCATE(""SGROUP"",localgroups) .ne. F$LENGTH(localgroups)
$       then
```

```
$               found = 1
$       endif
$ if F$LOCATE(""SGROUP"",globalgroups) .ne. F$LENGTH(globalgroups)
$       then
$               found = 1
$       endif
$
$!
$! Do something here if the name was in a user group
$!
$ if found .eq. 1
$   then
$!      Add code here to do something if the person is in an adminstators group
$ endif
$!
$! ********************************************************************************
$! ************************** GROUP SEARCH AND PROCESS
**************************
$! ********************************************************************************
$!
$! COMMON TO ALL LOGON USERS
$!
$! Add code here to do something that is generic for all users like
$! mapping drives for showing the user specfic computer or network information
$!
$END_OF_SCRIPT:
$ exit
$ lo
```