z/OS

UNIX System Services Programming Tools

SA22-7805-00

z/OS

UNIX System Services Programming Tools

SA22-7805-00

Note

Before using this information and the product it supports, be sure to read the general information under "Appendix C. Notices" on page 279.

First Edition, March 2001

This edition applies to Version 1 Release 1 of z/OS (5694-A01) and to subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405
FAX (Other Countries):
Your International Access Code +1+845+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)
Internet e-mail: mhvrcfs@us.ibm.com
World Wide Web: http://www.ibm.com/servers/eserver/zseries/zos/webqs.html

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Tables

About This Book

This book presents the information you need to use the **lex**, **yacc**, and **make** z/OS UNIX shell utilities.

This book also describes debugging services associated with z/OS UNIX System Services (z/OS UNIX).

Using the book, the people who plan for and write application programs can create, compile, debug, and maintain application programs that adhere to open standards and, optionally, take advantage of traditional OS/390 services.

Who Should Use This Book

This book is for application programmers who need to:

- Port to z/OS UNIX their POSIX-conforming applications that use **lex** and **yacc**
- Develop POSIX-conforming applications for z/OS UNIX that use **lex** and **yacc**
- Manage application development using **make**
- Debug applications

This book assumes that readers are somewhat familiar with z/OS and with the information for z/OS and its accompanying elements and features.

How to Use This Book

This book contains information useful to application programmers who are using the z/OS UNIX shells and utilities to develop applications.

Where to Find More Information

Where necessary, this book references information in other books about the elements and features of z/OS. For complete titles and order numbers for all z/OS books, see *z/OS Information Roadmap*.

Direct your request for copies of any IBM publication to your IBM representative or to the IBM branch office serving your locality.

There is also a toll-free customer support number (1-800-879-2755) available Monday through Friday from 6:30 a.m. through 5:00 p.m. Mountain Time. You can use this number to:

- Order or inquire about IBM publications
- Resolve any software manufacturing or delivery concerns
- Activate the program reorder form to provide faster and more convenient ordering of software updates

Softcopy Publications

The z/OS UNIX library is available on the *z/OS Collection Kit*, SK2T-6700. This softcopy collection contains a set of z/OS and related unlicensed product books. The CD-ROM collection includes the IBM Library Reader, a program that enables customers to read the softcopy books.

Softcopy z/OS publications are also available for web-browsing and PDF versions of the z/OS publications for viewing or printing using Adobe Acrobat Reader at this URL:

`http://www.ibm.com/servers/eserver/zseries/zos/`

Select “Library”.

Accessing licensed books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link Web site at:

`http://www.ibm.com/servers/resourcelink`

Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link Web userid and password, and a key code. With your z/OS order you received a memo that includes this key code.

To obtain your IBM Resource Link Web userid and password log on to:

`http://www.ibm.com/servers/resourcelink`

To register for access to the z/OS licensed books:

1. Log on to Resource Link using your Resource Link userid and password.
2. Click on **User Profiles** located on the left-hand navigation bar.
3. Click on **Access Profile.**
4. Click on **Request Access to Licensed books.**
5. Supply your key code where requested and click on the **Submit** button.

If you supplied the correct key code you will receive confirmation that your request is being processed. After your request is processed you will receive an e-mail confirmation.

Note: You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

To access the licensed books:

1. Log on to Resource Link using your Resource Link userid and password.
2. Click on **Library**.
3. Click on **zSeries**.
4. Click on **Software**.
5. Click on **z/OS**.
6. Access the licensed book by selecting the appropriate element.

Using LookAt to look up message explanations

LookAt is an online facility that allows you to look up explanations for z/OS messages and system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

You can use LookAt on the Internet at:

```
http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html
```

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on **News and Help** or from the *z/OS Collection*, SK3T-4269.

To find a message explanation from a TSO command line, simply enter: **lookat** *message-id* as in the following example:

```
lookat iec192i
```

This results in direct access to the message explanation for message IEC192I.

To find a message explanation from the LookAt Web site, simply enter the message ID. You can select the release if needed.

Note: Some messages have information in more than one book. For example, IEC192I has routing and descriptor codes listed in *z/OS MVS Routing and Descriptor Codes*. For such messages, LookAt prompts you to choose which book to open.

IBM Systems Center Publications

IBM systems centers produce redbooks that can be helpful in setting up and using z/OS UNIX System Services. You can order these publications through normal channels, or you can view them with a web browser from this URL:

http://www.redbooks.ibm.com

These books have not been subjected to any formal review nor have they been checked for technical accuracy, but they represent current product understanding (at the time of their publication) and provide valuable information on a wide range of z/OS UNIX topics. You must order them separately. A selected list of these books follows:

- *Selecting a Server — The Value of S/390,* SG24-4812
- *OS/390 Version 2 Release 6 UNIX System Services Implementation and Customization,* SG24-5178
- *OS/390 TCP/IP OpenEdition Implementation Guide,* SG24-2141. Written for OS/390 TCP/IP OpenEdition, a replacement for TCP/IP for MVS Version 3 Release 2 Application Feature.
- *Accessing OS/390 OpenEdition MVS from the Internet,* SG24-4721. Written for TCP/IP for MVS Version 3 Release 2 Application Feature.
- *MVS/ESA SP 5.2.2 OpenEdition MVS Installation and Customization Starter Kit,* SG24-4529
- *Porting Applications to the OpenEdition MVS Platform,* GG24–4473. This book was written for the OpenEdition MVS feature of MVS/ESA SP 5.1.

z/OS UNIX Porting Information

There is a *Porting Guide* on the z/OS UNIX porting page on the World Wide Web, at this URL:

http://www.ibm.com/s390/unix/bpxa1por.html

You can read the *Porting Guide* from the web or download it as a PDF file that you can view or print using Adobe Acrobat Reader. The *Porting Guide* covers a range of useful topics, including: sizing a port, setting up a porting environment, ASCII-EBCDIC issues, performance, and much more.

The porting page also features a variety of porting tips, and lists porting resources that will help you in your port.

z/OS UNIX Courses

The following classroom courses are available:

- *UNIX System Services for OS/390 Implementation,* ESP25
- *Introduction to OS/390 UNIX Services*, ESP05
- *Exploiting OS/390 Through UNIX Apps,* ES74A
- *TCPIP Implementation for OS/390,* CB694
- *UNIX System Services for OS/390 DCE Implementation,* ESP26
- *Web Server Implementation on OS/390,* ES170
- *1999 OS/390 Expo and Performance Conference,* E5438

The availability of educational offerings changes. For current information on classroom courses and other offerings, see your IBM representative or call 1-800-IBM-TEACH (1-800-426-8322).

z/OS UNIX Home Page

The z/OS UNIX home page on the World Wide Web has the latest technical news, customer stories, tools, and FAQs (frequently asked questions). You can visit it at this URL:

```
http://www.ibm.com/s390/unix/
```

Some of the tools available from the web site are ported tools, and some are home-grown tools designed for z/OS UNIX. All this code works in our environment at the time we make it available, but is not officially supported. Each tool has a README file that describes the tool and any restrictions on its use.

The simplest way to reach these tools is through the z/OS UNIX home page. From the home page, click on **Tools and Toys**.

The code is also available from **www.ibm.com.s390** through **anonymous ftp**.

Restrictions

Because the tools are not officially supported,
- There are no guaranteed enhancements.
- No APARs can be accepted.

z/OS UNIX Customization Assistant

If you'd like help with customizing z/OS UNIX, then check out our Web-based wizard. You can access it at this url:

```
http://www.ibm.com/servers/eserver/zeries/zos/bkserv/wizards.html
```

This wizard builds two BPXPRMxx parmlib members; one with system processing parameters and one with file system statements. It also builds a batch job that does the initial RACF security setup for z/OS UNIX. Whether you are installing z/OS UNIX for the first time or are a current user who wishes to verify settings, you can use this wizard.

Beginning with OS/390 R9, the wizard also allows sysplex users to build a single BPXPRMxx parmlib member to define all the file systems used by systems participating in shared HFS.

An edition of the wizard is available for OS/390 V2R8, as well.

Discussion List

Customers and IBM participants also discuss z/OS UNIX on the **mvs-oe discussion list**. This list is not operated or sponsored by IBM.

To subscribe to the mvs-oe discussion so you can receive postings, send a note to:

`listserv@vm.marist.edu`

Include the following line in the body of the note, substituting your first name and last name as indicated:

```
subscribe mvs-oe first_name last_name
```

After you are subscribed, you will receive further instructions on how to use the mailing list.

Chapter 1. Tutorial on Using lex and yacc

This tutorial introduces the basic concepts of **lex** and **yacc** and describes how you can use the programs to produce a simple desk calculator. New users should work through the tutorial to get a feel for how to use **lex** and **yacc**.

Those who are already familiar with the concepts of input analysis and interpretation may decide to skip this chapter and go directly to Chapter 2 and Chapter 3. These chapters give full details of all aspects of the programs.

All documentation for **lex** and **yacc** assumes that you are familiar with the C programming language. To use the two programs, you need to be able to write C code.

Uses for the lex and yacc Utilities

lex and **yacc** are a pair of programs that help write other programs. Input to **lex** and **yacc** describes how you want your final program to work. The output is source code in the C programming language; you can compile this source code to get a program that works the way that you originally described.

You use **lex** and **yacc** to produce software that analyzes and interprets input. For example, suppose you want to write a simple desk calculator program. Such a desk calculator is easy to create using **lex** and **yacc**, and this tutorial shows how one can be put together.

The C code produced by **lex** analyzes input and breaks it into *tokens*. In the case of a simple desk calculator, math expressions must be divided into tokens. For example:

```
178 + 85
```

would be treated as **178**, **+**, and **85**.

The C code produced by **yacc** *interprets* the tokens that the **lex** code has obtained. For example, the **yacc** code figures out that a number followed by a **+** followed by another number means that you want to add the two numbers together.

lex and **yacc** take care of most of the technical details involved in analyzing and interpreting input. You just describe what the input looks like; the code produced by **lex** and **yacc** then worries about recognizing the input and matching it to your description. Also, the two programs use a format for describing input that is simple and intuitive; **lex** and **yacc** input is much easier to understand than a C program written to do the same work.

You can use the two programs separately if you want. For example, you can use **lex** to break input into tokens and then write your own routines to work with those tokens. Similarly, you can write your own software to break input into tokens and then use **yacc** to analyze the tokens you have obtained. However, the programs work very well together and are often most effective when combined.

Code Produced by lex and yacc

The C code that is directly produced by **lex** and **yacc** is intended to be POSIX-conforming. When user-written code is added, the portability of the resulting program depends on whether the added code conforms to the POSIX standards.

To make it easy for you to add your own code, all identifiers and functions created by **lex** and **yacc** begin with `yy` or `YY`. If you avoid using such identifiers in your own code, your code will not conflict with code generated by the two programs.

lex Output

The goal of **lex** is to generate the code for a C function named **yylex()**. This function is called with no arguments. It returns an **int** value. A value of 0 is returned when end-of-file is reached; otherwise, **yylex()** returns a value indicating what kind of token was found.

lex also creates two important external data objects:

1. A string named **yytext**. This string contains a sequence of characters making up a single input token. The token is read from the stream **yyin**, which, by default, is the standard input (**stdin**).
2. An integer variable named **yyleng**. This gives the number of characters in the **yytext** string.

In most **lex** programs, a token in **yytext** has an associated *value* that must be calculated and passed on to the supporting program. By convention, **yacc** names this data object **yylval**. For example, if **yylex()** reads a token which is an integer, **yytext** contains the string of digits that made up the integer, while **yylval** typically contains the actual value of the integer. By default, **yylval** is declared to be an **int**, but there are ways to change this default.

Usually, a call to **yylex()** obtains a single token from the standard input; however, it is possible to have **yylex()** process the entire input, applying transformations and writing new output.

yacc Output

The goal of **yacc** is to generate the code for a C function named **yyparse()**. The **yyparse()** function calls **yylex()** to read a token from the standard input until end of file. **yyparse()** uses the return values from **yylex()** to figure out the types of each token obtained, and it uses **yylval** in each case as the actual value of that token.

The **yyparse()** function is called without any arguments. The result of the function is 0 if the input that was parsed was valid (that is, if the form of the input matched the descriptions given to **lex** and **yacc**). The result is 1 if the input contained errors of any kind.

Defining Tokens

As noted previously, **yylex()** returns a code value indicating what kind of token has been found, and **yyparse()** bases its actions on this value. Obviously then, the two functions must agree on the values that they assign to different tokens.

One way to do this is by using C *header files*. For example, consider a simple desk calculator program. Its input consists of expressions with simple forms such as:

```
789 + 45
3 * 24
9045 - 723
```

Thus there are two types of tokens: integer operands and mathematical operators.

If **yylex()** reads an operator like **+** or **–**, it can just return the operator itself to indicate the type of token obtained. If it reads an integer operand, it should store the value of the operand in **yylval**, and return a code indicating that an integer has been found. To make sure that both **yylex()** and **yyparse()** agree on this code, you might create a file that contains the C definition:

```
#define INTEGER 257
```

The values of the tokens are started at 257 to distinguish them from characters, and because **yacc** uses 256 internally. After this has been defined, you can include this file, with a C **#include** statement, and then use the **INTEGER** definition any time you want to refer to an integer token.

Suppose now that we expand our desk calculator so that it recognizes variables as well as integer operands. Then we can change our header file to show that there are now two types of operands:

```
#define INTEGER 257
#define VARIABLE 258
```

Again, by using these definitions, we can make sure that **yylex()** and **yyparse()** agree on what stands for what.

yacc has facilities that can automatically generate such definition files; therefore, early chapters speak in terms of header files created by hand; later chapters use header files created by **yacc**.

Calling the Code

The code produced by **lex** and **yacc** only constitutes part of a program. For example, it does not include a **main** routine. At the very minimum, therefore, you need to create a **main** routine of the form:

```
main()
{
    return yyparse();
}
```

This calls **yyparse()**, which then goes on to read and process the input, calling on **yylex()** to break the input into tokens. **yyparse()** terminates when it reaches end-of-file, when it encounters some construct that marks the logical end of input, or when it finds an error that it is not prepared to handle. The value returned by **yyparse()** is returned as the status of the whole program. This **main** routine is available in a **yacc** library, as shown later.

Obviously, the **main** routine may have to be much more complex It may also be necessary to write a number of functions that are called by **yylex()** to help analyze the input, or by **yyparse()** to help process it.

Using the lex and yacc Commands

Suppose that **file.l** contains **lex** input. Then the command

```
lex file.l
```

uses that input to produce a file named **lex.yy.c**. This file can then be compiled using **c89** to produce the object code for **yylex()**.

Suppose that **file.y** contains **yacc** input. Then the command:

```
yacc file.y
```

uses that input to produce a file named **y.tab.c**. You can then compile this file using **c89** to produce the object code for **yyparse()**.

To produce a complete program, you must link the object code for **yyparse()** and **yylex()** together, along with any other necessary functions.

The z/OS UNIX shells provide a library of useful **lex** routines. It also provides a **yacc** library that contains the entry point for the simple **main** entry point described earlier. These libraries should have been installed or created as part of your installation. When you use any library, be sure to add the library name to the linker commands that you use to build the final program. Chapter 2 and Chapter 3 describe these routines.

Tokenizing with lex

As mentioned earlier, the code produced by **lex** breaks its input into *tokens*, the basic logical pieces of the input. This section discusses how you describe input tokens to **lex** and what **lex** does with your description.

Characters and Regular Expressions

lex assumes that the input is a sequence of characters. The most important of these characters are usually the printable ones: the letters, digits, and assorted punctuation marks.

The input to **lex** indicates the *patterns* of characters that make up various types of tokens. For example, suppose you are using **lex** to help make a desk calculator program. Such a program performs various calculations with numbers, so you must tell **lex** what pattern of characters makes up a number. Of course, a typical number is made up of a sequence of one or more digit characters, so you need a way to describe such a sequence.

In **lex**, patterns of characters are described using *regular expressions*. The sections that follow describe several kinds of regular expressions you can use to describe character patterns.

Character Strings

The simplest way to describe a character pattern is just to list the characters. In **lex** input, enclose the characters in quotation marks:

```
"if"
"while"
"goto"
```

These are called *character strings*. A character string matches the sequence of characters enclosed in the string.

Inside character strings, the standard C escape sequences are recognized:

```
\n — newline
\b — backspace
\t — tab
```

and so on. See Chapter 2 for the complete list. These can be used in regular expressions to stand for otherwise unprintable characters.

Anchoring Patterns

A pattern can be anchored to the start or end of a line. You can use ˆ at the start of a regular expression to force a match to the start of a line, and **$** at the end of an expression to match the end of a line. For example,

```
ˆ"We"
```

matches the string `We` only when it appears at the beginning of a line. The pattern:

```
"end"$
```

matches the string `end` only when it appears at end of a line, whereas the pattern:

```
ˆ"name"$
```

matches the string `name` only when it appears alone on a line.

Character Classes

A *character class* is written as a number of characters inside square brackets, as in:

```
[0123456789]
```

This is a regular expression that stands for any one of the characters inside the brackets. This character class stands for any digit character.

```
[0123456789][0123456789][0123456789]
```

stands for any three digits in a row.

The digit character class can be written more simply as:

```
[0-9]
```

The **–** stands for all the characters that come between the two characters on either side. Thus:

```
[a-z]
```

stands for all characters between *a* and *z,* whereas:

```
[a-zA-Z]
```

stands for all characters in both the range *a* to *z* and the range *A* to *Z*.

Note: **–** is *not* treated as a range indicator when it appears at the beginning or end of a character class.

If the first character after the **[** is a circumflex (ˆ), the character class stands for all characters that are *not* listed in the brackets. For example:

```
[ˆ0-9]
```

stands for all characters that are *not* digits. Similarly:

```
[ˆa-zA-Z0-9]
```

stands for all characters that are not alphabetic or numeric.

There is a special character class—written as **.** —that matches *any* character except newline. The pattern:

```
"p.x"
```

matches any 3-character sequence starting with *p* and ending with *x*.

Note: A newline is never matched except when explicitly specified as **\n**, or in a range. In particular, a **.** never matches newline.

New character class symbols have been introduced by POSIX. These are provided as special sequences that are valid only within character class definitions. The sequences are:

```
[.coll.]" collation of character coll
[=equiv=] collation of the character class equiv
[:char-class:] any of the characters from char-class
```

lex accepts only the POSIX locale for these definitions. In particular, multicharacter collation symbols are not supported. You can still use, for example, the character class:

```
[[.a.]-[.z.]]
```

which is equivalent to:

```
[a-z]
```

for the POSIX locale.

lex accepts the following POSIX-defined character classes:

```
[:alnum:]  [:cntrl:]  [:lower:]  [:space:]
[:alpha:]  [:digit:]  [:print:]  [:upper:]
[:blank:]  [:graph:]  [:punct:]  [:xdigit:]
```

It is more portable (and more obvious) to use the new expressions.

Repetitions

Any regular expression followed by an asterisk (*****) stands for zero or more repetitions of the character pattern that matches the regular expression. For example, consider:

```
[[:digit:]][[:digit:]]*
```

This stands for a pattern of characters beginning with a digit, followed by zero or more additional digits. In other words, this regular expression stands for the pattern of characters that form a typical number. As another example, consider:

```
[[:upper:]][[:lower:]]*
```

This stands for an uppercase letter followed by zero or more lowercase letters.

Take a moment to consider the regular expression that matches any legal variable name in the C programming language. The answer is:

```
[[:alpha:]_][[:alnum:]_]*
```

which stands for a letter or underscore, followed by any number of letters, digits, or underscores.

The ***** stands for zero or more repetitions. You can use the **+** character in the same way to stand for one or more repetitions. For example:

```
[[:digit:]]+
```

stands for a sequence of one or more digit characters. This is another way to represent the pattern of a typical number. It is equivalent to:

```
[[:digit:]][[:digit:]]*
```

You can indicate a specific number of repetitions by putting a number inside brace brackets. For example:

```
[[:digit:]]{3}
```

stands for a sequence of three digits. You can also indicate a possible range of repetitions with a form such as:

```
[[:digit:]]{1,10}
```

This indicates a pattern of one to ten digits. You might use this kind of regular expression if you want to avoid numbers that are too large to handle. As another example:

```
[[:alpha:]_][[:alnum:]_]{0,31}
```

describes a pattern of 1 to 32 characters. You might use this to describe C variable names that can be up to 32 characters long. (Just remember that you must provide an action to discard the extra characters in a longer name.)

Optional Expressions

A regular expression followed by a question mark (**?**) makes that expression optional. For example:

```
A?
```

matches 0 or 1 occurrence of the character *A*.

Alternatives

Two regular expressions separated by an “or” bar (**|**) produces a regular expression that matches either one of the expressions. For example:

```
[[:lower:]]|[[:upper:]]
```

matches either a lowercase letter or an uppercase one.

Grouping

You may use parentheses to group together regular expressions. For example,

```
("high"|"low"|"medium")
```

matches one occurrence of any of the three strings `high`, `low`, or `medium`.

Note: Quotes do *not* group; a common mistake is to write:

```
"is"?
```

This pattern matches the letter *i*, followed by an optional *s*. To make the entire string optional, use parentheses:

```
("is")?
```

Definitions

A **lex** definition associates a name with a character pattern. The format of a definition is:

```
name regular-expression
```

where the *regular-expression* describes the pattern that gets the name. For example:

```
digit [[:digit:]]
lower [[:lower:]]
upper [[:upper:]]
```

are three definitions that give names to various character patterns.

A **lex** definition can refer to a name that has already been defined by putting that name in brace brackets. For example,

```
letter {lower}|{upper}
```

defines the `letter` pattern as one that matches the previously defined `lower` or `upper` patterns. Similarly,

```
variable {letter}({letter}|{digit})*
```

defines a `variable` pattern as a letter followed by zero or more letters or digits.

For POSIX conformance, **lex** now treats the definition, when expanded, as a group. Essentially, the expression is treated as if you had enclosed it in parentheses. Older **lex** processors did not always do this.

Definitions are always the first things that appear in the input to **lex**. They make the rest of the **lex** input more readable, since names are more easily understood than regular expressions. In **lex** input, the last definition is followed by a line containing only:

```
%%
```

This serves to mark the end of the definitions.

Translations

After the **%%** that marks the end of definitions, **lex** input contains a number of *translations*. The translations describe the actual tokens that you expect to see in input, and what is to be done with each token when it is received.

The format of a translation is:

```
token-pattern { actions }
```

The *token-pattern* is given by a regular expression that may contain definitions from the previous section. The *actions* are a set of zero or more C source code statements indicating what is to be done when such a pattern is recognized. Actions are written with the usual C formatting rules, so they can be split over a number of lines.

Also allowed as an action is a single "or" bar (|) which indicates that the action to be used is that of the next translation rule; for example:

```
"if"|
"while"{
/* handle keywords */
}
```

This could have been written as:

```
("if")|("while") { .... }
```

but you will find that using the alteration operator (|) makes your scanner larger and slower. It is always better to have many simple expressions that share one action separated with a single "or" bar.

In general, the actions associated with a token should determine the value to be returned by **yylex()** to indicate the token type. The actions may also assign a value to **yylval** to indicate the *value* of the token.

As a simple example, let's go back to the desk calculator. This might have the translation rule:

```
[[:digit:]]+ {
yylval = atoi(yytext);
return INTEGER;
}
```

Recall that **yytext** holds the text of the token that was found, and **yylval** is supposed to hold the actual value of that token. Thus:

```
yylval = atoi(yytext);
```

uses the C **atoi()** library function to convert this text into an integer value and assigns that integer to **yylval**. After this conversion has taken place, the action returns the defined value **INTEGER** to indicate that an integer has been obtained. ("Defining Tokens" on page 2 talks about this kind of definition.)

As another example of a translation, consider this:

```
[-+*/] {
return *yytext;
}
```

This says each of the four operator characters inside the parentheses is also a separate token. If one of these is found, the action returns the first character of **yytext**, which is the operator character itself; therefore if **yylex()** finds an operator, it returns the operator itself, which is the first character in **yytext**. (Remember that **–** is *not* treated as a range indicator when it appears at the beginning or end of a character class.) If the action in a translation consists of a single C statement, you can omit the brace brackets. For example, you could have written:

```
[-+*/] return *yytext;
```

Declarations

The definition or translation sections of **lex** input may contain *declarations*. These are normal C declarations for any data objects that the actions in translations may need.

If a translation section contains declarations, they must appear at the beginning of the section. The special construct **%{** is used to begin the declarations, and **%}** is used to end them. These constructs must appear alone at the beginning of a line.

As an example, consider the following:

```
%%

%{
int wordcount = 0;
%}

[ˆ \t\n]+ { wordcount++; }
[ \t]     ;
[\n] {
printf("%d\n",wordcount);
    wordcount = 0;
}
```

This generates a simple program that counts the words on each line of input. If **yylex()** finds a token consisting of one or more characters that are not spaces, tabs, or newlines, it increments **wordcount**. For sequences of one or more tabs or spaces, it does nothing (the action is just **;**—a null statement). When it encounters a newline, it displays the current value of **wordcount** and resets the count to zero.

Declarations given in the translations section are local to the **yylex()** function that **lex** produces. Declarations may also appear at the beginning of the definition section; in this case, they are external to the **yylex()** function. As an example, consider the following **lex** input, provided as the file **wc1.l** in the **/samples** directory:

```
%{
        int characters = 0;
        int words = 0;
        int lines = 0;
%}
%%
\n      {
                ++lines;
                ++characters;
        }
[ \t]+          characters += yyleng;
[^ \t\n]+ {
                ++words;
                characters += yyleng;
        }

%%
```

The definition section ends at the **%%**, which means that it consists only of the given declarations. These declare external data objects. After the **%%** come three translations. If a newline character is found, **yylex()** increments the count of lines and characters. If a sequence of spaces or tabs is found, the character count is incremented by the length of the sequence (specified by **yyleng**, which gives the length of a token). If a sequence of one or more other characters is found, **yylex()** increments the word count and again increments the character count by the value of **yyleng**.

You can use the **yylex()** generated by this example with a **main** routine of the form:

```
#include <stdio.h>

int yylex(void);

int main(void)
{
        extern int characters, words, lines;

        yylex();
        printf("%d characters, ", characters);
        printf("%d words, ", words);
        printf("%d lines\n", lines);
        return 0;
}
```

This example is provided as **wc.c** in the **/samples** directory. It calls **yylex()** to tokenize the standard input. Since none of the translation actions tell **yylex()** to return, it keeps reading token after token until it reaches end-of-file. At this point, it returns and the **main** function proceeds to display the accumulated counts of characters, words, and lines in the input.

lex Input for Simple Desk Calculator

This chapter has been discussing the **lex** input for a simple desk calculator. To finish things off, here's the complete input file. (This example, with minor changes, is provided as the file **dc1.l** in the **/samples** directory of the distribution.) Assume that the file **y.tab.h** contains the C definition for **INTEGER**, as given earlier:

```
#define INTEGER 257
```

```
%{
#include "y.tab.h"
extern int yylval;
%}

%%

[[:digit:]]+    {
                yylval = atoi(yytext);
                return INTEGER;
        }

[-+/*\n]        return *yytext;

[ \t]+          ;
```

This is almost the same as the previous presentation, except that it includes the newline as one of the operator characters. Each line of input is a separate calculation, so you have to pay attention to where lines end.

This input creates a **yylex()** that recognizes all the tokens required for the desk calculator. The next section, yacc Grammars, discusses how to use **yacc** to create a **yyparse()** that can use this **yylex()**.

yacc Grammars

By tradition, the input for **yacc** is called a *grammar*. **yacc** was invented to create parsers for compilers of computing languages; the **yacc** input was used to describe the grammar of such a language.

The primary output of **yacc** is a file named **y.tab.c**. This file contains the source code for a function named **yyparse()**. **yacc** can also produce a number of other kinds of output, as later sections describe.

yacc input is divided into three sections: the *declarations* section, the *rule* section, and the *function* section.

The Declarations Section

The declarations section of a **yacc** grammar describes the tokens that make up the grammar.

The simplest way to describe a token is with a line of the form:

```
%token name
```

where *name* is a name that stands for some kind of token. For example, you might have:

```
%token INTEGER
```

to state that **INTEGER** represents an integer token.

Creating Token Definition Files

When you run a grammar through **yacc** using the **-d** option, **yacc** produces a C definition file containing C definitions for all the names declared with **%token** lines in the declarations section of the grammar. The name of this file is **y.tab.h**. Each definition created in this way is given a unique number.

You can use a definition file created by **yacc** to provide definitions for **lex**, or other parts of the program. For example, suppose that **file.l** contains **lex** input for a program and that **file.y** contains **yacc** input:

```
yacc −d file.y
```

creates a **y.tab.h** file as well as a **y.tab.c** file containing **yyparse()**. In the declarations part of the definitions section of the **lex** input in **file.l**, you can have:

```
%{
#include "y.tab.h"
%}
```

to get the C definitions from the generated file. The rest of **file.l** can make use of these definitions.

Precedence Rules

The declarations section of **yacc** input can also contain *precedence rules*. These describe the *precedence* and *binding* of operator tokens.

To understand precedence and binding, it is best to start with an example. In conventional mathematics, multiplication and division are supposed to take place before addition and subtraction (unless parentheses are used to change the order of operation). Multiplication has the *same precedence* as division, but multiplication and division have *higher precedence* than addition and subtraction have.

To understand binding, consider the C expressions:

```
A - B - C
A = B + 8 * 9
```

To evaluate the first expression, you usually picture the operation proceeding from left to right:

```
(A - B) - C
```

To evaluate the second however, you perform the multiplication first, because it has higher precedence than addition:

```
A = ( B + (8 * 9) )
```

The multiplication takes place first, the value is added to **B**, and then the result is assigned to **A**.

Operations that operate from left to right are called *left associative*; operations that operate from right to left are called *right associative*. For example:

```
a/b/c
```

can be parsed as:

```
(a/b)/c left associative
```

or:

```
a/(b/c) right associative
```

Inside the declarations section of a **yacc** grammar, you can specify the precedence and binding of operators with lines of the form:

```
%left operator operator ...
%right operator operator ...
```

Operators listed on the same line have the same precedence. For example, you might say:

```
%left '+' '-'
```

to indicate that the **+** and **–** operations have the same precedence and left associativity. The operators are expressed as single characters inside apostrophes. Literal characters in **yacc** input are always shown in this format.

When you are listing precedence classes in this way, list them in order of precedence, from lowest to highest. For example:

```
%left '+' '-'
%left '*' '/'
```

says that addition and subtraction have a lower precedence than multiplication and division have.

As an example, C generally evaluates expressions from left to right (that is, left associative) while FORTRAN evaluates them from right to left (that is, right associative).

Code Declarations

The declarations section of a **yacc** grammar can contain explicit C source code declarations. These are external to the **yyparse()** function that **yacc** produces. As in **lex**, explicit source code is introduced with the **%{** construct and ends with **%}**; thus:

```
%{
    /* source code */
%}
```

The Grammar Rules Section

The end of the declarations section is marked by a line consisting only of:

```
%%
```

After this comes the rules section, the heart of the grammar.

A rule describes a valid *grammatical construct,* which can be made out of the recognized input tokens and other grammatical constructs. To understand this, here are some sample rules that make sense for a desk calculator program:

```
expression : INTEGER;
expression : expression '+' expression;
expression : expression '-' expression;
expression : expression '*' expression;
expression : expression '/' expression;
```

These rules describe various forms of a grammatical construct called an *expression*. The simplest expression is just an integer token. More complex expressions may be created by adding, subtracting, multiplying, or dividing simpler expressions. In a rule like:

```
expression : expression '+' expression
```

the definition has three *components*: an expression, a **+** token, and another expression.

If a program uses this grammar to analyze the input:

```
1 + 2 + 3
```

what does it do? First, it sees the number 1. This is an **INTEGER** token, so it can be interpreted as an expression. The input thus has the form:

```
expression + 2 + 3
```

Of course, the 2 is also an **INTEGER** and therefore an expression. This gives the form:

```
expression + expression + 3
```

But the program recognizes the first part of this input as one valid form of an expression. Thus, it boils down to:

```
expression + 3
```

In a similar way, this is interpreted as a valid form for an expression.

Actions

The rules section of a **yacc** grammar does not just describe grammatical constructs; it also tells what to do when each construct is recognized. In other words, it lets you associate *action*s with rules. The general form of a rule is:

```
name : definition { action } ;
```

where *name* is the name of the construct being defined, *definition* is the definition of the construct in terms of tokens and other nonterminal symbols, and *action* is a sequence of zero or more instructions that are to be carried out when the program finds input of a form that matches the given *definition*. For compatibility with older **yacc** processors, a single **=** can be placed before the opening **{** of the action.

The instructions in the *action* part of the rule can be thought of as C source code; however, they can also contain notations that are not valid in C. The notation **$1** stands for the *value* of the first component of the definition; if the component is a token, this is the **yylval** value associated with the token. Similarly, **$2** stands for the value of the second component, **$3** stands for the value of the third component, and so on. The notation **$$** is used to represent the value of the construct being defined.

As an example, consider the following rule:

```
expression: expression '+' expression { $$ = $1 + $3; } ;
```

This action adds the value of the first component (the first subexpression) to the value of the third component (the second subexpression) and uses this as the *result* of the whole expression. Similarly, you can write:

```
expression: expression '-' expression { $$ = $1 - $3; };
expression: expression '*' expression { $$ = $1 * $3; };
expression: expression '/' expression { $$ = $1 / $3; };
expression: INTEGER { $$ = $1; };
```

The last rule says that if the form of an expression is just an integer token, the value of the expression is just the value of the token.

If no action is specified in a rule, the default action is:

```
{ $$ = $1 }
```

This says that the default value of a construct is the value of its first component. Thus you can just write:

```
expression: INTEGER ;
```

Compressing Rules

If several rules give different forms of the same grammatical construct, they can be compressed into the form:

```
name : definition1 { action1}
| definition2 { action2 }
| definition3 { action3 }
...
;
```

There must be a semicolon to mark the end of the rule. Also, each definition has its own associated action. If a particular definition does not have an explicit action, the default action **$$=$1** is assumed.

Using this form, you can write:

```
expression:
  INTEGER
| expression '+' expression { $$ = $1 + $3;}
| expression '−' expression { $$ = $1 − $3; }
| expression '*' expression { $$ = $1 * $3; }
| expression '/' expression { $$ = $1 / $3; }
;
```

Start Symbols

The first grammatical construct defined in the rules section must be the most *all-inclusive* construct in the grammar. For example, if **yacc** input describes the grammar of a programming language, the first rule defined should be a complete program. The name of this first rule is called the *start symbol*.

The goal of the **yyparse()** routine is to gather input that fits the description of the start symbol. If your grammar defines a programming language and the start symbol represents a complete program, **yyparse()** stops when it finds a complete program according to this rule.

Obviously, you should define the starting symbol in such a way that it takes in all the valid streams of input that you expect. For example, consider our desk calculator. You might define a program with the rule:

```
program :
   program expression ’\n'
|  /* NOTHING */
;
```

This gives two definitions for a program: it can consist of an expression followed by a newline character (a line to be calculated) followed by more such lines; or it can be nothing at all. The nothing definition comes into play at the start of input.

Interior Actions

We need to associate an action with the program rule of the previous section. We want this action to display the result of the expression on the input line as soon as the entire line has been read; therefore, we write:

```
program:
  expression ’\n’ { printf("%d\n",$1); } program
| /* NOTHING */
;
```

This rule contains an *interior action*. The instruction in the brace brackets is run as soon as **yyparse()** reaches the part of the rule where the instruction appears (that is, as soon as it has read the newline token). This call to the **printf()** function of the C library immediately displays the value of the first component **$1** as a decimal integer. Then **yyparse()** goes on to gather the rest of the definition of **program** (more input lines).

Explicit Internal Source Code Declarations

The rules section of a **yacc** grammar can contain explicit source code declarations. As before, these begin with **%{** and end with **%}**. They are internal to the **yyparse()** function that **yacc** produces.

The Functions Section

The functions section of a **yacc** grammar does not always appear. When it does, it must begin with another **%%** construct (thus there is one **%%** between the declarations and the rules section, and another between the rules and the functions).

The functions section consists entirely of C source code. This source code typically contains definitions of functions that actions in the rules section call.

It is usually better to compile all such functions separately, rather than include them as part of the **yacc** input.

The Simple Desk Calculator

We are now ready to present the **yacc** input for our simple desk calculator program. This input corresponds to the **lex** input given in the previous section. (This example is provided as the file **dc1.y**.)

```
%{
#include <stdio.h>
%}

%token INTEGER
%left '+' '-'
%left '*' '/'

%%

program:
                program expression '\n'         = { printf("%d\n", $2); }
|               /* NOTHING */
;

expression:
                INTEGER
|               expression '+' expression       = { $$ = $1 + $3; }
|               expression '-' expression       = { $$ = $1 - $3; }
|               expression '*' expression       = { $$ = $1 * $3; }
|               expression '/' expression       = { $$ = $1 / $3; }
;
```

When this is run through **yacc**, the result is source code for a function named **yyparse()** that reads and interprets line after line of input. Linking this program with the **yacc** and **lex** libraries, you get a simple **main** function that calls **yyparse()** and exits. The exit status is 1 if the input was not in the correct format (for example, if you mistyped a calculation); it is 0 if the input was correct. (Be sure to link the **yacc** library before the **lex** library, to get the **main** routine in the **yacc** library that calls **yyparse()**.)

Error Handling

Errors are possible in any input. Dealing with errors always tends to be difficult, because there is no way to predict the forms that errors may take. Dealing with errors in highly structured forms of input (for example, program source code) is especially difficult, because you want to get back on track as soon as possible. Usually, you want to discard erroneous input and then resume processing good input as normal. The trick lies in figuring out where erroneous input ends and where good input begins.

This section looks at some of the error handling abilities of **lex** and **yacc**. To make things more concrete, the examples give the simple desk calculator program the ability to handle errors. They also give it a few more sophisticated features:

- Users can store integer values in variables (using assignment statements). Variables have names that are only one letter long. Uppercase letters are equivalent to lowercase ones, so there are a maximum of 26 possible variables.
- You can use parentheses in the usual way, to change the order of arithmetic evaluation.
- You can express integers in octal or hexadecimal as well as decimal forms, using the C conventions for octal numbers (leading zero) and hexadecimal numbers (leading 0x).

It may be useful to think about how you might go about writing **lex** and **yacc** descriptions of these new features before you read the rest of this chapter.

Error Handling in lex

From the point of view of **lex**, the most common sort of error is an input that does not have the form of any of the recognized tokens. A translation rule of the form:

```
.    { action }
```

(a dot followed by an *action*) can be placed at the end of all the other translation rules to take care of unrecognized input. For example, you can write:

```
.    { printf("Invalid input: %s\n",yytext); }
```

to issue an error message for any input that is not one of the recognized tokens. Since the *action* is a single C statement, you can omit the brace brackets, as in:

```
.    printf("Invalid input: %s\n",yytext);
```

Instead of using **printf()**, you can make use of a **lex** library function named **yyerror()**. **yyerror** simply displays a text string, followed by a newline, to **stderr** using **fprintf** and returns the integer value returned from **fprintf**. It is better to use **yyerror()** than making your own call to **printf()**, since **lex** also uses **yyerror()** for issuing error messages. If your code uses **yyerror()**, all the error messages are issued in the same way. Thus, you might write:

```
. { yyerror("Unrecognized input"); }
```

You can replace the standard **yyerror()** function with a version of your own if there is some standard error message format that you want to use.

Other Errors in lex

It is possible for other errors to be detected in the **yylex()** function that **lex** produces, but these have to be *expected* errors. In other words, you must write a translation rule that says, “If you see a token with *this* format, it is an error and here is how it is to be handled.” This sort of behavior is different for each application;

however, trying to detect such errors is a useful exercise if there are some types of erroneous input that you can predict and handle in some special useful way.

lex Input for the Improved Desk Calculator

The following is the **lex** input to produce our improved version of the desk calculator program. (This example is provided as **dc2.l**.)

```
%{
#include "y.tab.h"
extern int yylval;
char upper[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char lower[] = "abcdefghijklmnopqrstuvwxyz";
%}

%%

[[:upper:]]    {
          int i;
          for (i = 0; *yytext != upper[i]; ++i)
               ;
          yylval = i;
          return VARIABLE;
     }

[[:lower:]]    {
          int i;
          for (i = 0; *yytext != lower[i]; ++i)
               ;
          yylval = i;
          return VARIABLE;
     }

[[:digit:]]+   {
          yylval = strtol(yytext, (char **)NULL, 0);
          return INTEGER;
     }

0x[[:xdigit:]]+     {
          yylval = strtol(yytext, (char **)NULL, 16);
          return INTEGER;
     }

[-()=+/*\n]     return *yytext;

[ \t]+    ;

.         yyerror("Unknown character");
```

Note: This code looks complex because it handles character sets where characters are not continuous. If you're familiar with ASCII, it's tempting to get the value of **yylval** with a statement like `yylval = *yytext - 'A';` but this depends upon character ordering in ASCII. The loops in the example are slower but portable. If an input token is a single letter (uppercase or lowercase), the program returns a token value named **VARIABLE**. This is defined in the **yacc** input; it is obtained with the **#include** directive that gets the **y.tab.h** file that **yacc** generates.

To indicate which **VARIABLE** is being referred to, the **yylval** variable is set to an integer that indicates the letter: 0 for **A**, 1 for **B**, and so on. The same integer is used for both the uppercase and lowercase version of the letter. This corresponds to the 26 different variables that the desk calculator recognizes.

There are two types of integer tokens. Ones that begin with **0x** are interpreted as hexadecimal integers using the C **strtol()** function (which takes an integer string and produces the corresponding integer). Other integer tokens are also interpreted by **strtol()**, which determines the correct base to use (base 8 or base 10).

All the operators that the desk calculator recognizes are simply returned directly from **yylex()**. Blanks and horizontal tabs are skipped.

Any other character produces the error message associated with the **.** translation rule. The **yylex()** function then tries to get another token; if this also finds erroneous input, **yylex()** keeps looping until it finds something it recognizes. The result is that erroneous input is skipped—**yylex()** never returns any indication that it found such input.

Error Handling in yacc

Error handling in **yacc** must be much more sophisticated than in **lex**. The **yylex()** function that **lex** produces only has to detect erroneous input that can *never* have a recognized meaning; the **yyparse()** function that **yacc** produces has to figure out what to do with tokens that can be valid in some contexts but are not valid in the current context. For example, **yyparse()** has to figure out what to do with:

```
A = + * 5
```

All the tokens in this input are valid tokens, but put together in this way, they have no meaning. **yyparse()** has to figure out what to do when things do not make sense.

The Error Construct

To handle errors, **yacc** introduces a symbol named **error**. This stands for any *ungrammatical* construct: any sequence of one or more tokens that do not fit into the grammar anywhere else.

The **yacc** input for the new desk calculator shows how this is used. This example is provided as **dc2.y**.

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'

%{
static int variables[26];
%}

%%

program:
program statement '\n'
| program error '\n' { yyerrok; }
| /* NULL */
;

statement:
expression { printf("%d\n", $1); }
| VARIABLE '=' expression  { variables[$1] = $3; }
;

expression:
INTEGER
| VARIABLE { $$ = variables[$1]; }
| expression '+' expression  { $$ = $1 + $3; }
| expression '-' expression  { $$ = $1 - $3; }
| expression '*' expression  { $$ = $1 * $3; }
| expression '/' expression  { $$ = $1 / $3; }
| '(' expression ')' { $$ = $2; }
;
```

The rules for **expression** are almost the same as before. To evaluate an operand that consists of a variable, you obtain the value of the variable from the **variables** array. To evaluate a parenthesized expression, just take the value of the expression inside the parentheses.

The rules for **statement** are new, but simple. If a statement just consists of an expression, it displays the value of the expression; otherwise, it assigns the result of an expression to a variable, so you can store the value of the expression in the array element associated with the variable.

A program is either a null input, a valid program followed by a statement, or a valid program followed by an error. You do not have to do anything for null inputs. You do not have to do anything for valid programs followed by statements either, since the definition of **statement** does the work associated with each statement.

Now, consider what **yyparse()** does when it reads a line that contains an error.

1. Up to the point when it begins reading the line, it has collected a valid program construct.
2. It begins reading the erroneous line. Since it has already gathered a valid program construct, there are two rules that can apply to the situation:
   ```
   program : program statement '\n'
   program : program error '\n'
   ```
3. Partway through the line, it comes across an erroneous construct. This rules out the possibility that the input has the form:
   ```
   program statement '\n'
   ```
   Therefore the form of the input must be:
   ```
   program error '\n'
   ```
4. **yyparse()** keeps reading. Any sequence of tokens matches the **error** construct, so **yyparse()** is happy.
5. When it finally gets to the end of the line, **yyparse()** has successfully read the sequence:
   ```
   program error '\n'
   ```

This is one definition for a *valid* **program** construct. It performs the action associated with this rule; a later section discusses the action.

When **yyparse()** finishes performing the action, it has successfully dealt with the rule:

```
program : program error '\n'
```

In essence, **yyparse()** has found one of the expected forms of a valid **program** construct. **yyparse()** therefore proceeds to process the next line as if it has just finished reading a valid **program**.

Using yyerror()

As soon as **yyparse()** encounters input that does not match any known grammatical construction, it calls the **yyerror()** function. In this case, the argument that it passes to **yyerror()** is:

```
"Syntax error"
```

If you are using the default version of **yyerror()**, it simply displays this message to **stderr**; however, you can supply your own **yyerror()** function if you want to do other processing. See “The yyerror Function” on page 74 for more details.

The yyerrok Function

When **yyparse()** discovers ungrammatical input, it calls **yyerror()**. It also sets a flag saying that it is now in an *error state*. **yyparse()** stays in this error state until it sees three consecutive tokens that make sense (that is, are not part of the error).

It is possible for **yyparse()** to leave the error state as soon as it finds one or two tokens that make sense; however, experience has shown that this is not enough to be sure that the error has really passed; one or two tokens being correct may just be a coincidence. If **yyparse()** leaves its error state quickly and then finds more erroneous input, it raises another error, calls **yyerror()** again to issue a new error message, and so on. In other words, it behaves as if it had found a brand new error, even though it is likely just a continuation of the old error. Waiting for three good tokens prevents a lot of error messages arising from a single error.

There are, however, times when you want **yyparse()** to leave the error state before it finds the three good tokens. To do this, invoke the macro **yyerrok**, as in:

```
yyerrok;
```

In effect, **yyerrok** says, “The old error is finished. If something else goes wrong, it is to be regarded as a new error.”

This should help you understand the rule:

```
program : program error ’\n' { yyerrok; }
```

in the desk calculator program. Once **yyparse()** has found the newline that ends an erroneous input line, you want to leave the error state. Any errors on the line should be regarded as *closed*. If the next line also contains errors, you want to see a new error message produced.

Other Error Handling Facilities

The error handling facilities in **yacc** offer a much greater level of sophistication than the simple features discussed here. For further details, see Chapter 3.

A Sophisticated Example

This section examines a sophisticated desk calculator program. This is similar to the example in the previous section, but has several new features:

- **while** loops (similar to C **while** loops).
- **if** and **if**-**else** constructs.
- The introduction of C comparison operations (**>**, **>=**, **<**, **<=**, **==**, **!=**) to support condition testing.
- An explicit **print** command that displays the result of an expression.
- Statements can now extend over more than one line, using a semicolon to mark the end of a statement.
- Blocks of statements can now be enclosed in brace brackets, as in C.

Here is an example of the sort of input that the new program accepts:

```
a = 100;
while (a > 0) {
    print a;
    b = 50;
    while (b > 0) {
        print b;
        b = b - 10;
    }
    a = a - 20;
}
```

These new features introduce an interesting amount of complexity to the problem. For example, with the introduction of loops and **if**-**else** statements, you can no longer evaluate a statement as soon as you come to the end of the statement; you must save the input and run it when you reach the end of each construct. Because you can nest constructs, you need a way to record a lot of information.

Multiple Values for yylval

By default, the **yylval** variable has the **int** type. Up until now, this has been satisfactory; however, **yylval** should be able to represent the value of any token you find, which means that in some programs it should be able to represent more than just the **int** type. This means giving **yylval** a **union** type, the different interpretations of which match the various types of value that tokens may have. This is done in the **yacc** input using a construct of the form:

```
%union {
/* union declaration */
};
```

For example, suppose that you want the **yylex** routine to be able to return either integers or floating point numbers. Then you write:

```
%union {
int i;
float f;
};
```

to show that **yylval** can have either type.

In the case of the desk calculator, you want to represent variables and integers. You can therefore define:

```
%union {
char variable;
int ivalue;
};
```

lex Input

Here is the **lex** input for the new desk calculator program. This example is provided as **dc3.l**.

```
%{
#include "header.h"
#include "y.tab.h"
char upper[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char lower[] = "abcdefghijklmnopqrstuvwxyz";
%}

%%

[[:upper:]]     {
                int i;
                for (i = 0; *yytext != upper[i]; ++i)
                        ;
                yylval.variable = i;
                return VARIABLE;
        }
```

```
[[:lower:]]     {
                int i;
                for (i = 0; *yytext != lower[i]; ++i)
                        ;
                yylval.variable = i;
                return VARIABLE;
        }

[[:digit:]]+    {
                yylval.ivalue = strtol(yytext, (char **)NULL, 0);
                return INTEGER;
        }

0x[[:xdigit:]]+ {
                yylval.ivalue = strtol(yytext, (char **)NULL, 16);
                return INTEGER;
        }

[-{}()<>=+/*;]  return yylval.ivalue = *yytext;

">="            return yylval.ivalue = GE;
"<="            return yylval.ivalue = LE;
"=="            return yylval.ivalue = EQ;
"!="            return yylval.ivalue = NE;

"while"         return WHILE;
"if"            return IF;
"else"          return ELSE;
"print"         return PRINT;

[ \t\n]         ;

.       yyerror("Unknown character");
```

The new definitions are:

```
">="       return GE;
"<="       return LE;
"=="       return EQ;
"!="       return NE;
"while"    return WHILE;
"if"       return IF;
"else"     return ELSE;
"print"    return PRINT;
```

The symbols **GE**, **LE**, and so on are all C definitions. They represent new kinds of tokens that can be found in the input. If **yylex()** finds one of these new tokens, it returns the corresponding defined value.

These definitions, as given, recognize only lowercase keywords. The translation rule:

```
"while"|"WHILE"   return WHILE;
```

recognizes either all uppercase or all lowercase. To accept mixed case, you can write:

```
[wW][hH][iI][lL][eE]  return WHILE;
```

The Bare Grammar

The following is the bare grammar without actions attached to the rules in the rules section. It also leaves out a bit of explicit code in the declarations section.

```
%union {
        int ivalue;
        char variable;
        struct nnode *np;  /* discussed later */
};
.*.5v
%token <variable>  VARIABLE
%token <ivalue>    INTEGER '+' '-' '*' '/' '<' '>' GE LE NE EQ

%token WHILE IF PRINT ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'

%type <np> statement expression statementlist simplestatement

%%

program: program statement
       | error ';'
       | /* NOTHING */
       ;

statement:  simplestatement ';'
         |  WHILE '(' expression ')' statement
         |  IF '(' expression ')' statement ELSE statement
         |  IF '(' expression ')' statement
         |  '{' statementlist '}'
         ;

statementlist:   statement
              |   statementlist statement
             ;

simplestatement: expression
               | PRINT expression
               | VARIABLE '=' expression
               ;

expression: INTEGER
          | VARIABLE
          | expression '+' expression
          | expression '-' expression
          | expression '*' expression
          | expression '/' expression
          | expression '<' expression
          | expression '>' expression
          | expression GE expression
          | expression LE expression
          | expression EQ expression
          | expression NE expression
          | '(' expression ')'
          ;
```

As you can see, the definition of the grammar is quite straightforward. You may notice that the format of the **%token** lines has changed.

```
%token <ivalue>  INTEGER
```

states that when the return value of **yylex()** is **INTEGER**, the **yyparse()** routine is to use the **ivalue** interpretation of **yylval**. The same sort of thing applies to:

```
%token <variable> VARIABLE
```

You may also notice that this example introduces:

```
%type <np> statement expression statementlist simplestatement
```

as a new statement. This tells how to interpret the **$$** construct in definitions of **statement**, **expression**, **statementlist**, and **simplestatement**. In those constructs, **$$** (the *value* of the constructs) should have the **np** type. Since **program** does not have an assignment to **$$**, it is not given a type.

np is given as another possible interpretation in the **%union** directive. The **%union** gives possible interpretations of both **yylval** and **$$**, so we had to add the extra interpretation to the **%union**.

In general, **%type** lines can indicate the type of **$$** in any construct. The form of the directive is:

```
%type <interp>
construct construct ...
```

where *interp* is one of the interpretation names given in the **%union** directive.

The next section discusses what the **np** type does.

Expression Trees

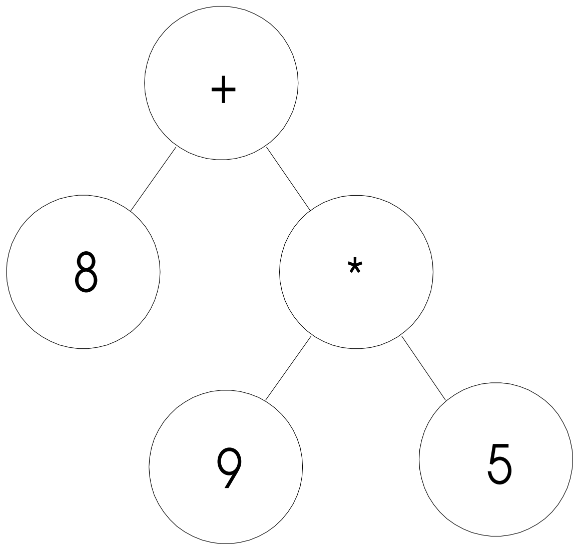
Earlier this chapter discussed the need to *record* expression while reading them for future evaluation. The best way to do this is by using a *tree*. To understand how a tree works, consider an expression such as:

```
8 + 9 * 5
```

which is evaluated as:

```
8 + (9 * 5)
```

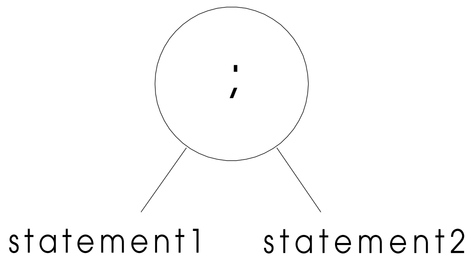
Each operation has three components: the operator, and the two operands.



The operators are called the *nodes* of the tree. At each node, there are two *branches*, representing the two operands of the operator. The end of each branch is a simple operand that is not an expression; such an operand is called a *leaf*.

Tree structures are a good way to represent expressions. They record all the information needed to evaluate the expression.

Tree structures can also represent a list of statements. In this case, think of the operator as the semicolon that separates the two.



A **while** loop is represented similarly, with one branch giving the condition expression and the other giving the statement list. Finally, an **if**-**else** statement can be represented as a tree with *three* branches: one for the condition expression, one for the **if** statements, and one for the **else** statements. An **if** without an **else** is just a special case where the third branch is empty.

To represent these trees, the desk calculator example creates the following data types. These are defined in the header file **header.h**, which you include (with the **#include** directive) into the appropriate C source code files.

```
typedef union {
        int     value;
        struct nnode *np;
} ITEM;
typedef struct nnode {
        int     operator;
        ITEM    left, right, third;
} NODE;
#define LEFT    left.np
#define RIGHT   right.np

#define NNULL           ((NODE *) 0)
#define node(a,b,c)     triple(a, b, c, NNULL)

extern int variables[26];

int execute(NODE *np);
```

To record an expression, use **malloc()** to allocate an **nnode** structure. The operator is set to the operator of the expression; the tokens **INTEGER**, **VARIABLE**, **WHILE**, and **IF** are also used as appropriate. For leaves of the tree (simple operands), call a function named **leaf()** to fill in the left field and put null pointers in the other two. For operations that have two operands, call a function named **node()** to fill in the left and right fields with pointers to trees for the operands; the third field is given a null pointer value. For operations with three operands, call a function named **triple()** to fill in all three pointers.

As input is collected, tree structures are allocated and organized. When a complete statement has been collected, you can then call a function named **execute()** to *walk* through the tree and run the statement appropriately.

When the statement has been run, the tree is no longer needed. At that point, call a function named **freeall()** to free the memory used for all the structures that make up the tree.

Putting all this together produces the following grammar for the desk calculator program. Note that the functions part of the input contains everything you need except the **execute()** function. This example is provided in **dc3.y**.

```
%{
#include <stdio.h>
#include <stdlib.h>

#include "header.h"

static NODE *nalloc(void);
static NODE *leaf(int type, int value);
static NODE *triple(int op, NODE *left, NODE *right, NODE *third);
static void freeall(NODE *np);

int variables[26];
%}

%union {
        int ivalue;
        char variable;
        NODE *np;
};

%token <variable>  VARIABLE
%token <ivalue>    INTEGER '+' '-' '*' '/' '<' '>' GE LE NE EQ

%token WHILE IF PRINT ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'

%type <np>      statement expression statementlist simplestatement

%%

program:
          program statement     { execute($2); freeall($2); }
        | program error ';'     { yyerrok; }
        | /* NULL */
        ;

statement:
          simplestatement ';'
        | WHILE '(' expression ')' statement
                { $$ = node(WHILE, $3, $5); }
        | IF '(' expression ')' statement ELSE statement
                { $$ = triple(IF,$3,$5,$7); }
        | IF '(' expression ')' statement
                { $$ = triple(IF,$3,$5,NNULL); }
        | '{' statementlist '}'
                { $$ = $2; }
        ;

statementlist:
          statement
        | statementlist statement       { $$ = node(';', $1, $2); }
        ;

simplestatement:
          expression
        | PRINT expression              { $$ = node(PRINT,$2,NNULL); }
        | VARIABLE '=' expression
                        { $$ = node('=', leaf(VARIABLE, $1), $3); }
        ;

expression:
```

```
          INTEGER               { $$ = leaf(INTEGER, $1); }
        | VARIABLE              { $$ = leaf(VARIABLE, $1); }
        | expression '+' expression
                { binary: $$ = node($2, $1, $3); }
        | expression '-' expression     { goto binary; }
        | expression '*' expression     { goto binary; }
        | expression '/' expression     { goto binary; }
        | expression '<' expression     { goto binary; }
        | expression '>' expression     { goto binary; }
        | expression GE expression      { goto binary; }
        | expression LE expression      { goto binary; }
        | expression NE expression      { goto binary; }
        | expression EQ expression      { goto binary; }
        | '(' expression ')'            { $$ = $2; }
        ;

%%

static NODE *
nalloc()
{
        NODE *np;

        np = (NODE *) malloc(sizeof(NODE));
        if (np == NNULL) {
                printf("Out of Memory\n");
                exit(1);
        }
        return np;
}

static NODE *
leaf(type, value)
int type, value;
{
        NODE *np = nalloc();

        np->operator = type;
        np->left.value = value;
        return np;
}

static NODE *
triple(op, left, right, third)
int op;
NODE *left, *right, *third;
{
        NODE *np = nalloc();

        np->operator = op;
        np->left.np = left;
        np->right.np = right;
        np->third.np = third;
        return np;
}

static void
freeall(np)
NODE *np;
{
        if (np == NNULL)
                return;
        switch(np->operator) {
        case IF:                /* Triple */
                freeall(np->third.np);
        /* FALLTHROUGH */
                                /* Binary */
        case '+': case '-': case '*': case '/':
        case ';': case '<': case '>':
        case GE: case LE: case NE: case EQ:
        case WHILE:
        case '=':
                freeall(np->RIGHT);
        /* FALLTHROUGH */
        case PRINT:             /* Unary */
                freeall(np->LEFT);
                break;
        }
        free(np);
}
```

Note that there is a shift-reduce conflict in this grammar. This is due to the rules:

```
statement: IF '(' expression ')' statement ELSE statement ;
statement: IF '(' expression ')' statement ;
```

The default rules for resolving this conflict favor the shift action, which is what is desired in this case. An **else** that follows an **if** statement matches with the closest preceding **if**. (See Chapter 3 for more details.)

The source code for the **execute()** function can be compiled separately. It walks through the tree node by node, calling itself recursively to run the branches at each node. The **execute()** function is basically a big **switch** statement, which looks at the node operator and takes appropriate action. It is quite straightforward. In the examples provided, this is file **execute.c**.

```
#include <stdio.h>
#include <stdlib.h>

#include "header.h"
#include "y.tab.h"

int
execute(np)
struct nnode *np;
{
        switch(np->operator) {
        case INTEGER:   return np->left.value;
        case VARIABLE:  return variables[np->left.value];
        case '+':       return execute(np->LEFT) + execute(np->RIGHT);
        case '-':       return execute(np->LEFT) - execute(np->RIGHT);
        case '*':       return execute(np->LEFT) * execute(np->RIGHT);
        case '/':       return execute(np->LEFT) / execute(np->RIGHT);
        case '<':       return execute(np->LEFT) < execute(np->RIGHT);
        case '>':       return execute(np->LEFT) > execute(np->RIGHT);
        case GE:        return execute(np->LEFT) >= execute(np->RIGHT);
        case LE:        return execute(np->LEFT) <= execute(np->RIGHT);
        case NE:        return execute(np->LEFT) != execute(np->RIGHT);
        case EQ:        return execute(np->LEFT) == execute(np->RIGHT);
        case PRINT:     printf("%d\n", execute(np->LEFT)); return 0;
        case ';':       execute(np->LEFT); return execute(np->RIGHT);
        case '=':
                        return
                          variables[np->LEFT->left.value] = execute(np->RIGHT);
        case WHILE:
                        while (execute(np->LEFT))
                                execute(np->RIGHT);
                        return 0;
        case IF:
                        if (execute(np->LEFT))
                                execute(np->RIGHT);
                        else if (np->third.np != NNULL)
                                execute(np->third.np);
                        return 0;
        }
        printf("Internal error! Bad node type!");
        exit(1);
}
```

Note that **execute()** calls the **yyerror()** function to issue error messages.

Compilation

By changing the **execute** function, you can *compile* the input program instead of just running it. The output of the function is the sequence of hardware commands required to run the program. Doing this for a real machine is too complicated for the purposes of this tutorial; however, this section shows how to do it for a simple hypothetical machine.

Note: This section assumes that you have a basic knowledge of computer architecture.

Consider a hypothetical machine with the following characteristics:

- The machine works with a hardware stack.
- It has 26 registers, numbered 0 through 25.
- It has a *push register* command that pushes the value of a register onto the stack.
- It has a *push constant* command (**push**) that pushes the value of a constant onto the stack.
- It has a *pop register* command (**pop**) that pops the top value off the stack and stores it in a specified register.
- It has the following binary operators:

  ```
  add   sub   /*  + and -  */
  mul   div   /*  * and /  */
  cmpl  cmpg  /*  < and >  */
  cmple cmpge /* <= and >= */
  cmpeq cmpne /* == and != */
  ```

  Each of these instructions pops the top two values from the stack, performs the indicated operation, and then pushes the result. The result of a comparison is 1 if true, and 0 if false.
- There is a **print** operation that pops the top value from the stack and displays it.
- There is a **jmp** command that transfers control to a different location.
- There is a **jfalse** command that pops a value off the stack and transfers to a different location if the value is zero.

Given this setup, here is the *compiling* version of **execute**. Store this in a file so that you can run the compiled program anytime. In the examples, this is the file **compile.c**.

```
#include <stdio.h>
#include <stdlib.h>

#include "header.h"
#include "y.tab.h"

int
execute(np)
struct nnode *np;
{
        int toplab, botlab, falselab;
```

```
        static int labno;

        switch(np->operator) {
        case INTEGER:
                printf("\tpush\t$%d\n", np->left.value);
                break;
        case VARIABLE:
                printf("\tpush\tr%d\n", np->left.value);
                break;
        case '=':
                execute(np->RIGHT);
                printf("\tpop\tr%d\n", np->LEFT->left.value);
                return 0;
        case '+': case '*': case '-': case '/':
        case '<': case '>': case GE: case LE: case NEcreate_option
                execute(np->LEFT); execute(np->RIGHT);
                switch(np->operator) {
                case '+':       printf("\tadd\n"); break;
                case '-':       printf("\tsub\n"); break;
                case '*':       printf("\tmul\n"); break;
                case '/':       printf("\tdiv\n"); break;
                case '<':       printf("\tcmpl\n"); break;
                case '>':       printf("\tcmpg\n"); break;
                case GE:        printf("\tcmpge\n"); break;
                case LE:        printf("\tcmple\n"); break;
                case NE:        printf("\tcmpne\n"); break;
                case EQ:        printf("\tcmpeq\n"); break;
                }
                break;
        case PRINT:
                execute(np->LEFT);
                printf("\tprint\n");
                break;
        case ';':
                execute(np->LEFT); execute(np->RIGHT);
                break;
        case WHILE:
                printf("L%d:", toplab = labno++);
                execute(np->LEFT);
                printf("\tjfalse\tL%d\n", botlab = labno++);
                execute(np->RIGHT);
                printf("\tjmp\tL%d\n", toplab);
                printf("L%d:", botlab);
                break;
        case IF:
                execute(np->LEFT);
                printf("\tjz\tL%d\n", falselab = labno++);
                execute(np->RIGHT);
                printf("\tjmp\tL%d\n", botlab = labno++);
                printf("L%d:", falselab);
                if(np->third.np != NNULL)
                        execute(np->third.np);
                printf("L%d:", botlab);
                break;
        default:
                printf("Internal error! Bad node type!");
                exit(1);
        }
}
```

Chapter 2. Generating a Lexical Analyzer Using lex

A computer program often has an input stream of characters that are easier to process as larger elements, such as tokens or names. A compiler is a common example of such a program: It reads a stream of characters forming a program, and converts this stream into a sequence of items (for example, identifiers and operators) for parsing. In a compiler, the procedures that do this are collectively called the *lexical analyzer* or *scanner*.

Expressing the scanning task in a general-purpose procedural programming language is usually difficult. The scanning transformations are usually easy enough to describe; however, it is hard to express them concisely in these languages.

Introduction to the lex Utility

The z/OS UNIX **lex** utility is a program that writes large parts of a lexical analyzer automatically, based on a description supplied by the programmer. The items or tokens to be recognized are described as regular expressions in a special-purpose language for writing lexical analyzers. **lex** translates this language, which is easy to write, into an analyzer that is both fast and compact.

The purpose of a **lex** program is to read an input stream and recognize *tokens.* As the lexical analyzer usually exists as a subroutine in a larger set of programs, it is usually written to return a *token number*, indicating the token that was found, and possibly a *token value*, providing more detailed information about the token (for example, a copy of the token itself, or an index into a symbol table). This need not be the only possibility; by itself, a **lex** program is often a good description of the structure of a computation.

lex is based on a similar program written by Charles Forsyth at the University of Waterloo (Ontario, Canada), and described in an unpublished paper entitled "A Lexical Analyzer Generator" (1978). The implementation is loosely based on the description and suggestions in the book *Compilers, Principles, Techniques, and Tools* , by A. V. Aho, Ravi Sethi, and J. D. Ullman (Addison-Wesley, 1986).

This **lex** was inspired by a processor of the same name at Bell Labs, which also runs under UNIX systems, and, more distantly, on AED-0. UNIX **lex** is described in the paper "Lex — A Lexical Analyser Generator," by M. E. Lesk, *Computer Science Technical Report* 39 (Bell Labs, October 1975). AED-0 is described in "Automatic Generation of Efficient Lexical Analysers Using Finite State Techniques," by W. L. Johnson, appearing in *Communications of the ACM* 11 (no. 12, 1968): 805-13.

The lex Input Language

In this section we discuss the **lex** input language. This includes the following topics:

- Fundamentals of the language, including characters, strings, and character classes
- Putting together the fundamentals to form regular expressions
- **lex** programs and their basic form
- Using definitions for regular expressions
- Translations, which associate regular expressions with actions
- C declarations that can be included in **lex** programs

Language Fundamentals

lex expressions (also known as *regular expressions* or *patterns*) are basic to its operation. The nature and construction of these expressions is described first.

Characters, strings, and sets of characters called character classes are the fundamental elements of **lex** expressions. These stand for, or match, characters in the input stream; characters and character classes match single characters of the input, whereas strings match a fixed-length sequence of input characters.

Characters

A *character* is any character. The letters *a* through *z, A* through *Z,* the underscore _, and the digits `0` to `9` stand for single occurrences of themselves in the input. Most other characters are treated specially by **lex**. The escape character (\) written in front of a special character has no special significance; it can match an occurrence of itself in the input stream.

The escape can also be used to create an escape sequence standing for a different character. **lex** understands the following C language escape sequences. The value in parentheses is the EBCDIC value for that escape sequence. With these, you can represent any 8-bit character, including the escape character, quotes, and newlines:

```
\a BEL (0X2F)
\b BS (0X16)
\f FF (0X0C)
\n NL (0X15)
\r CR (0X0D)
\t TAB (0X05)
\v VTAB (0X0B)
\nnn (nnn)
\xhh (hh)
\" "
\' '
\c c
\\  \
```

where *nnn* is a number in octal, *hh* is a number in hexadecimal, and *c* is any printable character.

Strings

A *string* is a sequence of characters, not including newline, enclosed in double quotes. For example, "+" is a *string* that matches a single **+** in the input. Within a string, only the escape character (\) has any special significance. The escape sequences given earlier are recognized within a string. You can continue long strings across a line by placing an escape before the end of the line. The escape and the newline are not incorporated into the string.

Character Classes

A sequence of characters enclosed by brackets—[and]—forms a character class, which matches a single instance of any character within the brackets. If a circumflex (ˆ) follows the opening bracket, the class matches any characters except those inside the brackets.

Within a character class the character **–** is treated specially, unless it occurs at the start (after any ˆ) or end of the character class. If two characters are written separated by **–** the sequence is taken to include all characters in the character set from the first to the second (using the numeric values of characters in the character set). Thus [a–z} stands for all characters between *a* and *z*. You can use the escapes used in strings in character classes as well.

The POSIX locale is supported in **lex**. These are provided as special sequences that are valid only within character class definitions. The sequences are:

```
[.coll.]        collation of character coll
[=equiv=]       collation of the character class equiv
[:char-class:]   any of the characters from char-class
```

lex accepts the POSIX locale only for these definitions. In particular, multicharacter collation symbols are not supported. You can still use, for example, the character class:

```
[[.a.]-[.z."]
```

which is equivalent to:

```
[a-z]
```

for the POSIX locale.

lex accepts the POSIX-defined character classes shown in Table 1.

It is more portable (and more obvious) to use the new expressions; for example, the character class:

```
[[:alnum:]]
```

is the same as:

```
[a-zA-Z0-9]
```

in the POSIX locale, but is portable to other locales.

There is a special character class, written as—which matches *any* character but newline. Newline must always be matched explicitly.

Table 1. POSIX-Defined Character Classes in lex

Name	Definition
[:alpha:]	Any letter
[:lower:]	A lowercase letter
[:upper:]	An uppercase letter
[:digit:]	Any digit
[:xdigit:]	Any digit, or the letters *a–fA–F*
[:alnum:]	Any letter or digit
[:cntrl:]	Any control (nonprinting) character
[:space:]	Any spacing character, including blank, tab, and carriage return
[:print:]	Any printable character
[:blank:]	A blank or tab character
[:graph:]	Any printable character other than space
[:punct:]	A punctuation mark

Putting Things Together

Various operators are available to construct regular expressions or patterns from strings, characters, and character classes. A reference to an *occurrence* of a regular expression is generally taken to mean an occurrence of any string matched by that regular expression.

The operators are presented in order of decreasing priority. In all cases, operators work on characters, character classes, strings, or regular expressions.

1. Any character, string, or character class forms a regular expression that matches whatever the character, string, or character class stands for (as described earlier).
2. The operator ***** following a regular expression forms a new regular expression, which matches an arbitrary number of (that is, zero or more) adjacent occurrences of the first regular expression. The operation is often referred to as (Kleene) *closure*. For example, the expression:

   ```
   ab*
   ```

   matches **a** followed by zero or more **b**'s; that is **a**, **ab**, **abb**, and so on.
3. The operator **+** is used like ***** but forms a regular expression that matches one or more adjacent occurrences of a given regular expression. For example:

   ```
   ab+
   ```

   matches **a** followed by one or more **b**'s. This is equivalent to **abb***.
4. A repetition count can follow a regular expression, enclosed in **{}**. This is analogous to simply writing the same regular expression as many times as indicated. A range of repetitions can be provided, separated by a comma. For example:

   ```
   ab{4}
   ```

   matches **a** followed by exactly four **b**'s. That is, **abbbb**.

   ```
   ab{2,4}
   ```

   matches **a** followed by from 2 to 4 **b**'s.
5. The operator **?** written after a regular expression indicates that the expression is optional: the resulting regular expression matches either the first regular expression, or the empty string. For example:

   ```
   [[:lower:]]?
   ```

   matches a lowercase letter or nothing (an optional letter).
6. The operation of *concatenation* of two regular expressions is expressed simply by writing the regular expressions adjacent to each other. The resulting regular expression matches any occurrence of the first regular expression followed directly by an occurrence of the second regular expression. For example:

   ```
   a*b*
   ```

   matches any number of **a**'s followed immediately by any number of **b**'s.
7. The operator **|**, *alternation,* written between two regular expressions forms a regular expression that matches an occurrence of the first regular expression *or* an occurrence of the second regular expression. For example:

   ```
   [[:lower:]]|[[:digit:]]
   ```

   matches a lowercase letter or a digit. This is equivalent to:

   ```
   [[:lower:][:digit:]]
   ```
8. You can enclose any regular expression in parentheses to cause the priority of operators to be overridden. For example, the expression:

   ```
   [[:lower:]]([[:digit:]]|[[:lower:]])*
   ```

matches a name starting with a lowercase letter, followed by any number of lowercase letters or digits.

9. Operators lose special meaning when escaped by **** or quoted as in a string "*...*". The characters also stand for themselves within brackets.

lex Programs

A **lex** program consists of three sections: a section containing *definitions*, a section containing *translations*, and a section containing *functions*. The style of this layout is similar to that of **yacc**.

Throughout a **lex** program, you can freely use newlines and C-style comments; they are treated as white space. Lines starting with a blank or tab are copied through to the **lex** output file. Blanks and tabs are usually ignored, except when you use them to separate names from definitions, or expressions from actions.

The definition section is separated from the following section by a line consisting only of **%%**. In this section, named regular expressions can be defined, which means you can use names of regular expressions in the translation section, in place of common subexpressions, to make that section more readable. The definition section can be empty, but the **%%** separator is required.

The translation section follows the definition section, and contains regular expressions paired with *action*s, which describe what the lexical analyzer is to do when a match of a given regular expression is found. The first nonescaped space or tab on a line in the translation section signals the start of the action. Actions are further described in later sections of this chapter.

You can omit the function section; if it is present, it is separated from the translation section by a line containing only **%%**. This section can contain anything, because it is simply attached to the end of the **lex** output file.

Definitions

You can define regular expressions once, and then refer to them by name in any subsequent regular expression. Definition must precede use. A definition has the form:

```
name expression
```

where a *name* is composed of a letter or underscore, followed by a sequence of letters, underscores, or digits. Within an expression, you can refer to another defined name by enclosing that name in braces, as in {*name*}. For example:

```
digit   [[:digit:]]
letter  [[:alpha:]]
name    {letter}({letter}|{digit})
```

which defines an expression called *name* that matches a variable name. A definition must completely fit onto one line.

As well as definitions, the definition section can also contain declarations and directives. Declarations are described in “Declarations” on page 39. Directives are used to define *start conditions* and to change the size of internal **lex** tables.

New directives are provided to define the type of **yytext**. The **%array** directive causes **yytext** to be defined as an array of **char**; this is also the default. The **%pointer** directive causes **yytext** to be defined as a pointer to an array of **char**.

Internal **lex** tables include NFA and DFA tables, and a move table. (For an explanation of these terms, see the book *Compilers, Principles, Techniques, and Tools* mentioned in the beginning of this chapter.) The default sizes of these tables may not be sufficient for large scanners. You can change table sizes by the following directives, with the number *size* giving the number of entries to use:

Table 2. lex Table Size Specifications

Line	Table Size Affected	Default Size
%e *size*	Number of NFA entries	1000
%n *size*	Number of DFA entries	500
%p *size*	Number of move entries	2500

Often, you can reduce the NFA and DFA space to make room for more move entries. UNIX **lex** allows additional table size specifications, as follows:

Table 3. Additional UNIX lex Table Size Specifications

Line	Table Size Affected
%a*size*	Number of transitions
%k*size*	Packed character classes
%o*size*	Output array size

As these sizes are unnecessary in **lex**, a warning is issued, and the specification is ignored.

Translations

An action can be associated with a regular expression in the translation section. The resulting translation has the following form:

```
expression action
```

or

```
expression  {
action
}
```

The action is given as either a single C statement on the rest of the line, or a C statement within braces, possibly spread out over a number of lines, and starting after the first blank or tab on the line. (Remember not to use blanks or tabs inside an expression unless they are escaped with **** or within strings.)

A compiler typically enters an identifier into a symbol table, reads and remembers a string, or returns a particular token to the parser. In text processing, you might want to reproduce most of the input stream on an output stream unchanged, but make substitutions when a particular sequence of characters is found.

Allowing a translation action to be in C provides a great deal of power to the scanner, as shown in later sections. A library of C functions and macros is provided to allow controlled access to some of the data structures used by the scanner.

Token String and Length

A **lex** expression typically matches a number of input strings. For example:

```
%%
[[:alpha:]_][[:alnum:]_]*
```

matches any C identifiers in the input. It is useful to be able to obtain the portion of the input matched by such expressions, for use by the action code.

In **lex**, the current token is found in the character array **yytext**. The end of the token is marked by a null byte, so that it has the usual form of a string in C. The following **lex** program displays all the identifiers in a C program (including keywords), one per line.

```
%%
[[:alpha:]_][[:alnum:]_]* printf("%s\n", yytext);
\n|. ; /* discard other input */
```

In some applications, the null byte might itself be a valid input character, and it may be useful to know the true length of the token. The value **yyleng** holds the length of the token in **yytext** and also may save a call to **strlen()** to determine the length of a token.

Numbers and Values

Typically, a lexical analyzer returns a value to its caller indicating which token has been found. Within an action, this is done by writing a C **return** statement, which returns the appropriate value:

```
digit [[:digit:]]
letter [[:lower:]]
integer {digit}+
name {letter}({letter}|{digit})*
%%
"goto"    { return GOTO; }
{integer} { return INTEGER; }
{name}    { lookup(yytext); return NAME; }
```

In many cases, the lexical analyzer must supply other information to its caller. Within a compiler, for example, when an identifier is recognized, both a pointer to a symbol table entry and the token number **NAME** must be returned; however, the C **return** statement can return only a single value. **yacc** solves this problem by having the lexical analyzer set an external **yylval** to the *token value*, and return the *token number*. This mechanism can be used by **lex** programs when used with **yacc**; otherwise, you can define another interface. For example:

```
{name} { yylval = lookup(yytext); return(NAME); }
```

In the absence of a **return** statement, the lexical analyzer does not return to its caller, but looks instead for another token. This is typically used when a comment sequence has been discovered and discarded, or when the purpose of the **lex** program is to change a set of tokens into some other set of strings.

To summarize, the token number is set by the action with a **return** statement, and the token value is set by assigning this value to the external value **yylval**. An action need not return.

Declarations

C declarations can be included in both the definition and translation sections. C code in the declarations section should be bracketed by the sequence **%{** and **%}** on lines by themselves, as in **yacc**. Such declarations are external to the function **yylex()**. The characters within these brackets are copied unchanged into the appropriate spots in the lexical analyzer program that **lex** writes.

An action enclosed in braces forms a local block, and declarations therein are local to the particular action, as determined by C scope rules.

To declare variables that are local within **yylex()**, you can use the same **%{ .. %}** syntax at the beginning of the translation section. Names declared in this way do not conflict with other external variables.

Using lex

This section discusses how to use **lex** in practice, with attention to the following aspects:
- Using the lexical analyzer, **yylex()**, in conjunction with **yacc**
- Generating a table file from the **lex** program
- Compiling the table file
- An overview of the **lex** library routines fully usable with **yylex()**

Using yylex()

The structure of **lex** programs is influenced by what **yacc** requires of its lexical analyzer.

To begin with, the lexical analyzer is named **yylex()** and has no parameters. It is expected to return a token number (of type **int**), where that number is determined by **yacc**. The token number for a character is its value as a C character constant. **yacc** can also be used to define token names, using the **token** statement, where C definitions of these tokens can be written on the file **y.tab.h** with the **-d** option to **yacc**. This file defines each token name as its token number.

yacc also allows **yylex()** to pass a value to the **yacc** action routines, by assigning that value to the external **yylval**. The type of **yylval** is by default **int**, but this may be changed by the use of the **yacc %union** statement. **lex** assumes that the programmer defines **yylval** correctly; **yacc** writes a definition for **yylval** to the file **y.tab.h** if the **%union** statement is used.

For compatibility with **yacc**, **lex** provides a lexical analyzer named **yylex()**, which interprets tables formed from the **lex** program, and which returns token numbers from the actions it performs. The actions may include assignments to **yylval** (or its components, if it is a union of types), so that use with **yacc** is straightforward.

In the absence of a **return** statement in an action, **yylex()** does not return but continues to look for further matches. If some computation is performed entirely by the lexical analyzer with no normal return from any action, a suitable main program is:

```
#include <stdio.h>

main()
{
return yylex();
}
```

The value `0` (zero) is returned by **yylex()** at end-of-file; this program allows for an error return to the program's caller. You can find such a main program in the **lex** library.

Generating a Table File

In the absence of instructions to the contrary, **lex** reads a given **lex** language file, and produces a C program file **lex.yy.c**, which contains a set of tables, and a **yylex()** program to interpret them. The actions you supply in each translation are combined with a **switch** statement into a single function, which the table interpreter calls when a particular token is found. The contents of the program section of the

lex file are added at the end of the C program file. Declarations and macro definitions required by **lex** are inserted at the top of the file. You can modify some of these, as described in the following sections. **lex** uses the standard I/O library, and automatically generates the directive:

```
#include <stdio.h>
```

required to use that library.

A set of C macros is provided that allows the user to access values maintained by **lex**, or to control the operation of the lexical analyzer in various ways.

The values maintained by **lex** are:

yytext The characters forming the current token, terminated by a null byte.

yyleng
The length of the token; this is useful if the token may contain a null byte.

yylineno
The current line number of the input.

Some other defined constants are also special to **lex**:

YYLEX
Provides the name of the lexical analyzer function. By default, this is **yylex**, but a user may use **#undef** and then redefine **YYLEX** to obtain another name.

YYLMAX
Specifies the maximum length of the token buffer **yytext**. The default length is 100 characters. This value is checked when pushing characters back into the input (see **unput** in "The lex Library Routines" on page 42). During the scan, an error message is produced if insufficient space remains.

Compiling the Table File

lex is called by the command line:

```
lex source.l
```

where **source.l** is the name of a file containing a **lex** source program. **lex** reads the given file, and (in the absence of any unrecoverable errors) produces the file **lex.yy.c**, described earlier.

Compile this file in the usual way. Using the **c89** command, you can type something like this:

```
c89 −c lex.yy.c
```

When linking, the **lex** library is usually required. This library, described in "The lex Library Routines" on page 42, can be in a number of different places. The usual library is:

```
/usr/lib/libl.a
```

which can be abbreviated on the **c89** command line to **-l1**.

As **lex** writes its output, it prepends the contents of the **/etc/yylex.c** file. The **yylex.c** file contains the prototype scanner.

The following example shows the use of a program with **lex** and **yacc**, with the **lex** source in **scanner.l** and the **yacc** source in **grammar.y**. The user code is in the file **code.c**, and the code uses components of the **lex** library and the **main()** routine from the **yacc** library.

Note: The **yacc** library is specified first. (There is a **main()** routine in the **lex** library as well; if the **lex** library is specified first, that **main()** is used, calling the lexical analyzer once and exiting.) The user code and the scanner make use of tokens defined by **yacc**; so the **-D** option is given to **yacc** to create the **gram.h** file:

```
lex scanner.l
yacc -D gram.h grammar.y
c89 code.c lex.yy.c y.tab.c -ly -ll
```

The **gram.h** file has to be included by the **scanner.l** file, with:

```
%{
#include "gram.h"
%}
```

in the definition section of the **scanner lex** file.

The lex Library Routines

The **lex** library contains routines that are either essential or generally useful to **lex** programs. These routines have an intimate knowledge of **yylex()**, and can correctly manipulate the input stream.

Those functions that produce diagnostics do so by calling **yyerror()**, which is called as:

```
external int yyerror(const char * format)
```

and is expected to write its arguments using **fprintf**, followed by a newline, on some output stream, typically **stderr**. A **yyerror()** function is included in the **lex** library but can be redefined by the programmer.

A description of the typedefs, constants, variables, macros, functions, and library routines currently available follows:

Typedefs

YY_SAVED
A typedef that is an internal data structure used to save the current state of the scanner. See the description of **yySaveScan** in the functions subsection.

yy_state_t
A typedef defined by **lex** to be the appropriate unsigned integral for indexing state tables. It will be either "**unsigned char**" or "**unsigned int**", depending on the size of your scanner.

Constants

YYLMAX
A constant that defines the maximum length of tokens the **lex** scanner can recognize. Its default value is 100 characters, and can be changed with the C preprocessor **#undef** and **#define** directives in the input declarations section.

Variables

yyleng
A variable that defines the length of the input token in **yytext**.

yylineno
A variable that defines the current input line number, maintained by **input** and **yycomment**.

yyin A variable that determines the input stream for the **yylex()** and **input** functions.

yyout A variable that determines the output stream for the **output** macro, which processes input that does not match any rules. The values of **yyin** and **yyout** can be changed by assignment.

yytext A variable that defines the current input token recognized by the **lex** scanner. It is accessible both within a **lex** action and on return of the **yylex()** function. It is terminated with a null (zero) byte. If **%pointer** is specified in the definitions section, **yytext** is defined as a pointer to a preallocated array of **char**.

Macros

BEGIN
A macro that can be used as an action to cause **lex** to enter a new start condition.

ECHO A macro that can be used as an action to copy the matched input token **yytext** to the **lex** output stream **yyout**.

NLSTATE
A macro that resets **yylex()** as though a newline had been seen on the input.

REJECT
A macro that causes **yylex()** to discard the current match and examine the next possible match, if any.

YY_FATAL
A macro that can be called with a string message upon an error. The message is printed to **stderr**, and **yylex()** exits with an error code of 1.

yygetc()
A macro that is called by **yylex()** to obtain characters. Currently, this is defined as:

```
#define yygetc() getc(yyin)
```

A new version can be defined for special purposes, by first using **#undef** to remove the current macro definition.

YY_INIT
A macro that reinitializes **yylex()** from an unknown state. This macro can be used only in a **lex** action; otherwise, use the function **yy_reset**.

YY_INTERACTIVE
A macro that is normally defined in the code as being equal to 1. If defined as 1, **yylex()** attempts to satisfy its input requirements without looking ahead past newlines, which is useful for interactive input. If **YY_INTERACTIVE** is defined as `0`, **yylex()** does look past newlines; it is also slightly faster.

YY_PRESERVE
A macro that is normally not defined. If defined, when an expression is matched, **lex** saves any pushback in **yytext** before calling any user action and restores this pushback after the action. This may be needed for older **lex** programs that change **yytext**. It is not recommended, because the state saves are fairly expensive.

Functions

input A function that returns the next character from the **lex** input stream. (This means that **lex** does not see it.) This function properly accounts for any lookahead that **lex** may require.

unput(int *c***)**
A function that may be called by a translation when **lex** recognizes the sequence of characters that marks the start of a comment in the given syntax.

yycomment
A function that takes a sequence of characters marking the end of a comment, and skips over characters in the input stream until this sequence is found. Newlines found while skipping characters increment the external **yylineno**. An unexpected end-of-file produces a suitable diagnostic (using **yyerror)**. The following **lex** rules match C and shell-style comments:

```
"/*"    yycomment("*/");
#.*\n   ;
```

A **lex** pattern is more efficient at recognizing a newline-terminated comment, whereas the function can handle comments longer than **YYLMAX**.

yyerror
A function that is used by routines that generate diagnostics. A version of **yyerror()** is provided in the library, which simply passes its arguments to **fprintf** with output to the error stream **stderr**. A newline is written following the message. **yyerror()** returns an integer value which is the value returned from **fprintf**. You can provide a replacement. The definition of **yyerror** must agree with the prototype of **yyerror()** defined in **yylex.c**:

```
external int yyerror(const char * format, ...)
```

yylex The scanner that **lex** produces. It returns a token if it has located in the input. A negative or zero value indicates error or end of input.

yymapch(int*delim***,int***esc***)**
A function that can be used to process C-style character constants or strings. It returns the next string character from the input, or –1 when the character *delim* is reached. The usual C escapes are recognized: *esc* is the escape character to use; for C it is backslash.

yymore
A function that causes the next token to be concatenated to the current token in **yytext**. The current token is not rescanned.

yy_reset
A function that can be called from outside a **lex** action to reset the **lex** scanner. This is useful when starting a scan of new input.

yyRestoreScan
A function that restores the state of scanner after a **yySaveScan** call, and frees the allocated save block. The **yySaveScan** and **yyRestoreScan** functions allow an **include** facility to be safely defined for **lex**. Here is how the save functions can be used:

```
include(FILE * newfp)
{
void * saved;
saved = (void *) yySaveScan(newfp);
/*
 * scan new file
 * using yylex() or yyparse()
 */
yyRestoreScan(saved);
}
```

yySaveScan

A function that can be called to save the current state of **yylex()** and initialize the scanner to read from the given file pointer. The scanner state is saved in a newly allocated **YY_SAVED** record; this record is then returned. The contents of the save block are not of interest to the caller. Instead, the save block is intended to be passed to **yyRestoreScan** to reset the scanner.

Library Routines

yywrap

A library routine called by **yylex()** when it gets EOF from **yygetc**. The default version of **yywrap** returns 1, which indicates that no more input is available. **yylex()** then returns `0`, indicating end of file. If the user wishes to supply more input, a **yywrap** should be provided, which sets up the new input (possibly by assigning a new file stream to **yyin**), then returns `0` to indicate that more input is available.

Error Detection and Recovery

A character that is detected in the input stream that cannot be added to the last-matched string, and that cannot start a string, is considered illegal by **lex**. **lex** might be instructed to write the character to an output stream, write a diagnostic and discard the character, ignore the character, or return an **error** token. The default action is to write the character to the output stream **yyout**. **lex** does this by invoking the macro:

```
#define output(c) putc((c),yyout)
```

By replacing the **output** macro, the user may change the default action to any C statement. Some possible definitions are:

```
/* type a diagnostic */
#define output(x) \
error("Illegal character %c (%o)", (x),(x))

/* ignore the character */
#define output(c)
```

The file **yyout** is the standard output, by default.

When **lex** encounters input that cannot be handled, such as an overflow of the buffer, it calls the macro **YY_FATAL**:

```
YY_FATAL("message");
```

This macro displays the indicated message on **stderr** and then exits the program.

To change this behavior, you can redefine **YY_FATAL** in the definition section. For example, if **lex** is scanning an input file, but error recovery requires that other operations be carried out, you can redefine **YY_FATAL** to return a special value to flag that error.

For debugging a complex scanner, you can invoke **lex** with the **-T** option. This causes a description of the various states of the scanner to be left in the text file **l.output**. You can then compile the scanner in **lex.yy.c** with the preprocessor flag **YY_DEBUG** defined, to get a scanner that displays, on **stderr**, the intermediate transitions and states of the scanner as it reads input. With the **l.output** information as a guide, these states can be related back to the input scanner description.

Ambiguity and Lookahead

A **lex** program may be ambiguous, in the sense that a particular input string may match more than one translation expression. Consider this example:

```
%%
[[:lower:]] { putchar(*yytext); }
aaa*  { printf("abc"); }
```

in which the string `aa` matches by both regular expressions (twice by the first, and once by the second). Also, the string `aaaaaa` may be matched in many different ways.

If the input matches more than one expression, **lex** uses the following rules to determine which action to take:

1. The rule that matches the longest possible input stream is preferred.
2. If more than one rule matches an input of the same length, the rule that appears first in the translations section is preferred.

In the previous example, rule 1 causes both `aa` and `aaaaaa` to match the second action, while a single `a` matches the first action.

As another example, the following program works as expected:

```
"<"  { return(LESS); }
"=&"; { return(EQUAL); }
"<=" { return(LESSEQ); }
```

Here, the sequence **<=** is taken to be an instance of a less-than-or-equal symbol, rather than an instance of a less-than symbol followed by an equals symbol.

Consider yet another example:

```
letter [[:lower:]]
%%
a({letter})* { return('A'); }
ab({letter})* { return('B'); }
```

which attempts to distinguish sequences of letters that begin with `a` from similar sequences that begin with `ab`. In this example, rule 1 is not sufficient, as, for example, the string `abb9` applies to either action; therefore, by rule 2, the first matching action should apply.

As written, the second action is never performed. To achieve the effect indicated, reverse the rules as follows:

```
letter [[:lower:]]
%%
ab{letter}* { return('B'); }
a{letter}* { return('A'); }
```

There is a danger in the lookahead that is done in trying to find the longest match. For example, an expression such as:

```
[.\n]+
```

causes the entire input to be read for a match! Another example is reading a quoted expression; for example:

```
'.*'
```

matches the string:

```
'quote one' followed by 'quote two'
```

because **lex** attempts to read too much of the input. The correct definition of this string is:

```
'[^'\n]*'
```

which stops after reading `'quote one'`.

Lookahead

A facility for looking ahead in the input stream is sometimes required. You can also use this facility to control the default ambiguity resolution process.

A traditional example is from FORTRAN, which does not have reserved words. Further scanning is required to determine whether the sequence **if(** is in fact an **if** statement, and not the subscripting of an array named **if**. In this case, a rather large amount of lookahead is required, to see what character follows the closing **)**; if the character is a letter, or a digit, then an **if** statement has indeed been found; otherwise, the array reference (or a syntax error) is indicated.

Another example is from C, where a name followed by **(** is to be contextually declared as an external function if it is otherwise undefined. In Pascal, lookahead is required to determine that:

```
123..1234
```

is an integer 123, followed by the subrange symbol **..**—which is followed by the integer 1234, and not simply two real numbers run together.

In all these cases, the desire is to look ahead in the input stream far enough to be able to make a decision, but without losing tokens in the process.

A special form of regular expression is used to indicate lookahead:

```
re1/ re2
```

where *re1* and *re2* are regular expressions that do not themselves contain lookahead. The slash is treated as concatenation for the purposes of matching incoming characters: Both *re1* and *re2* must match adjacently for an action to be performed. *re1* indicates that part of the input string which is the token to be returned in **yytext**, whereas *re2* indicates the context. The characters matched by *re2* are reread at the next call to **yylex()** and broken into tokens.

For the C external function example, the lookahead operator is used in the following manner:

```
digit [[:digit:]]
letter [[:lower:]]\
name {letter}({digit}|{letter})*

%%

{name}/"(" {
if (name undefined)
declare name a global function;
}
{name} { usual processing for identifiers }
```

To handle the (not reserved) **if** identifier in FORTRAN, the following is used:

```
space [ \t]*
digit [[:digit:]]
letter [[:lower:]]

%%

if/{space}"(".*")"{space}({letter}|{digit}) {
                                   /* if statement */
                               }
{name} { /* any other use of if */ }
```

If a **lex** expression is a prefix of some other expression, it has a hidden 1-character lookahead at the end, whether the lookahead operator is used or not. This enables **lex** to implement the longest-string rule correctly.

Left Context Sensitivity and Start Conditions

Even a fairly simple syntax may be difficult or impossible to describe with a single set of translations. For example, in the C programming language, literal strings have a different structure, and must be read and parsed separately from the rest of the input.

lex provides a facility called *start conditions*, which allow the input to be processed by different sets of rules. Start conditions are declared in the definitions section, with lines of the form:

```
%Start name1 name2 ....
```

(You can abbreviate **%Start** to **%S** or **%s**.) When a start condition name is placed at the beginning of a rule within **<>**, that rule can match only when **lex** is in that start condition. To enter a start condition, you can code the action:

```
BEGIN name
```

To revert to the normal state, use:

```
BEGIN 0
```

To make a rule active in several start conditions, use the prefix:

```
<name1,name2,...>
```

at the beginning of the expression. All rules without a start condition prefix are always active.

Here is a simple example of the use of start conditions. When **lex** sees a line containing only a 1, it switches to the **OTHER** start condition, until a line containing only a `0` is seen. While in the **OTHER** start condition, input is echoed with the text **OTHER** prefixed to each line.

```
%s OTHER
%%

"0"\n         BEGIN 0;
"1"\n         BEGIN OTHER;
<OTHER>.*     printf("OTHER %s",yytext);
```

A more realistic example follows. This parses a C string.

```
%{
#include <stdio.h>
      static char buf[200];
      char *s;
      char *strchr();
      long strtol();
      char *yylval;
#define     STRING      1
%}

%s string

%%

<0>\"                 { BEGIN string; s = buf; }
<string>\\[0-7]{1,3}  {
                            *s++ = strtol(yytext+1,
                                                    (char **)0, 8);
                      }
<string>\\\"         *s++ = '"';
<string>\\[rbfntv]    {
                            *s++ = *(strchr("\rr\bb\ff\nn\tt\vv",
                                   yytext[1])-1);
                      }
<string>\\\n         /* Escaped newline ignored */;
<string>\n            {
                             yyerror("Unterminated string");
                             BEGIN 0;
                      }
<string>\"            {
                             *s = '\0';
                             BEGIN 0;
                             yylval = buf;
                             return STRING;
                      }
<string>.             *s++ = *yytext;

%%

main()
{
     while(yylex() == STRING)  {
           printf(">>>"),
           fputs(yylval, stdout),
           printf("<<<\n);
     }
}
```

Sometimes the input is so structured that you require several completely different and conflicting sets of rules. You need a mechanism for defining *minianalyzers* that are enabled for some specific task.

To handle this need, you can define *exclusive* start conditions. When an exclusive start condition is active, no other rules are active; thus, a set of rules with the same (prefix) exclusive start condition effectively describes a minianalyzer that is independent of the normal rules. Exclusive start conditions are entered and exited in the usual way, with the **BEGIN** action. To define exclusive start conditions, use **%x** instead of **%s** in the definition section.

The main feature of exclusive start conditions is that rules without a start condition prefix are *not* automatically applied to all start conditions. This allows a better structuring of the rules in some situations.

Tracing a lex Program

With the **-T** option, **lex** produces a description of the scanner that it is generating in the file **l.output**. This description consists of two parts: a description of the initial state table, specified as an NFA, followed by a description of the minimized DFA for the final scanner. Usually only the latter is of interest. Here is the complete output for the previous example using start conditions. The actions are *not* represented.

```
NFA for complete syntax

state 0
  3: rule 0, start set 0 1 2 3
        epsilon 1
  4: rule 1, start set 0 1 2 3
        epsilon 5
  5: rule 2, start set 2 3
        epsilon 11
  6: rule 3, start set 0 1 2 3
        epsilon 15

state 1
        0  2

state 2
        \n  4

state 4
        final state

state 5
        1  6

state 6
        \n  8

state 8
        final state

state 11
        epsilon  9
        epsilon  12

state 9
        [\0-\t\13-\177]  10

state 10
        epsilon  9
        epsilon  12

state 12
        final state

state 15
        epsilon  13
        epsilon  16

state 13
        [\0-\t\13-\177]  14

state 14
        epsilon  13
        epsilon  16

state 16
        final state


Minimized DFA for complete syntax

state 0, rule 3, lookahead
        [\0-\t]       4
        [\13-/]       4
        0       7
        1       5
        [2-\177]      4

state 1, rule 3, lookahead
        .       same as 0

state 2, rule 2, rule 3, lookahead
        [\0-\t]       9
        [\13-/]       9
        0       11
        1       10
        [2-\177]      9

state 3, rule 2, rule 3, lookahead
        .       same as 2

state 4, rule 3, lookahead
        [01]  4
        .       same as 0


state 5, rule 3, lookahead
        \n      6
        [01]  4
        .       same as 0

state 6, rule 1, lookahead

state 7, rule 3, lookahead
        \n      8
        [01]  4
        .       same as 0

state 8, rule 0, lookahead

state 9, rule 2, rule 3, lookahead
        [01]  9
        .       same as 2

state 10, rule 2, rule 3, lookahead
        \n      6
        [01]  9
        .       same as 2

state 11, rule 2, rule 3, lookahead
        \n      8
        [01]  9
        .       same as 2
```

Looking at the minimal DFA reported, the table transitions are easy to trace. Starting at state 0, the rules are:

```
state 0, rule 3, lookahead
        [\0-\t]       4
        [\13-/]       4
        0       7
        1       5
        [2-\177]      4
```

The meaning of this description is as follows: while in state 0 (which is based on rule 3), on reading the letter **0**, switch to state 7; for the letter **1**, switch to state 5; and on any other letter, switch to state 4.

Assume that the letter **1** is read. The scanner checks the rules for state 0, and transfers to state 5. In states 5 and 6, the following rules apply:

```
state 5, rule 3, lookahead
        \n      6
        [01]  4
        .       same as 0

state 6, rule 1, lookahead
```

The rules in state 5 describe a transition to state 6 upon reading a newline (\n), and a return to state 4 if anything else is read. (An optimization in the state tables allows state 5 to reuse state 0's transitions.) State 6 has no rules; it corresponds to the action that triggers the **OTHER** start condition.

The REJECT Action

To remember the results of a previous scan for the purpose of finding another possible match, the action **REJECT** can be used in the translation section. This action causes **lex** to do the next alternative. For example, the following program counts instances of the words **he** and **she**:

```
she s++;
he h++;
\n |
. ;
```

Anything not matching **he** or **she** is ignored, because of the bottom two rules.

This program, however, does not count instances of **he** embedded inside instances of **she**. To obtain this behavior, a **REJECT** action is required to force **lex** to consider any other rules that might match, adjusting the input accordingly. The program then becomes:

```
she { s++; REJECT; }
he { h++; REJECT; }
\n |
. ;
```

After counting each **he** or **she**, the expression is rejected and the other expression is examined. As **he** cannot include **she**, the second **REJECT** is actually not required in this case.

Character Set

lex handles characters internally as small integer values, as given by the bit pattern on the host computer's character set. To change the interpretation of input characters, you can provide a translation table in the definition section that associates an integer value with a character or group of characters. The translation table should be bracketed by lines containing **%T**.

```
%T
1 Aa
2 Bb
...
26 Zz
27 \n
28 +
29 -
30 0
31 1
...
39 9
%T
```

This table maps lowercase and uppercase letters together into the range 1–26, newline into 27, **+** into 28, **–** into 29, and the digits into 30–39. The character values range from `0` to the highest possible value in the host computer's character set. Every possible input character must be enumerated in the table.

To work properly, the user must then redefine **yygetc** to translate input characters, so that **A** or **a** is given to **lex** as 1, **B** or **b** is given as 2, and so on.

Chapter 3. Generating a Parser Using yacc

The z/OS UNIX **yacc** utility is a tool for writing compilers and other programs that parse input according to strict *grammar* rules. The z/OS UNIX **yacc** utility can produce anything from a simple parser for a desk calculator program to a very elaborate parser for a programming language. Those who are using **yacc** for complex tasks have to know all the idiosyncrasies of the program, including a good deal about the internal workings of **yacc**. On the other hand, the internal workings are mostly irrelevant to someone who is making an easy straightforward parser.

For this reason, novices may want to concentrate on the information in Chapter 1 for an overview of how to use **yacc**. This tutorial also shows how you can use **lex** and **yacc** together in the construction of a simple desk calculator.

How yacc Works

The input to **yacc** describes the rules of a grammar. **yacc** uses these rules to produce the source code for a program that parses the grammar. You can then compile this source code to obtain a program that reads input, parses it according to the grammar, and takes action based on the result.

The source code produced by **yacc** is written in the C programming language. It consists of a number of data tables that represent the grammar, plus a C function named **yyparse()**. By default, **yacc** symbol names used begin with **yy**. This is an historical convention, dating back to **yacc**'s predecessor, UNIX **yacc**. You can avoid conflicts with **yacc** names by avoiding symbols that start with **yy**.

If you want to use a different prefix, indicate this with a line of the form:

```
%prefix prefix
```

at the beginning of the **yacc** input. For example:

```
%prefix ww
```

asks for a prefix of **ww** instead of **yy**. Alternatively, you could specify **–p ww** on the **lex** command line. The prefix chosen should be 1 or 2 characters long; longer prefixes lead to name conflicts on systems that truncate external names to 6 characters during the loading process. In addition, at least 1 of the characters in the prefix should be a lowercase letter (because **yacc** uses an all-uppercase version of the prefix for some special names, and this has to be different from the specified prefix).

Note: Different prefixes are useful when two **yacc**-produced parsers are to be merged into a single program. For the sake of convenience, however, the **yy** convention is used throughout this manual.

yyparse() and yylex()

yyparse() returns a value of `0` if the input it parses is valid according to the given grammar rules. It returns a 1 if the input is incorrect and error recovery is impossible.

yyparse() does not do its own lexical analysis. In other words, it does not pull the input apart into tokens ready for parsing. Instead, it calls a routine called **yylex()** everytime it wants to obtain a token from the input.

yylex() returns a value indicating the *type* of token that has been obtained. If the token has an actual *value*, this value (or some representation of the value, for example, a pointer to a string containing the value) is returned in an external variable named **yylval**.

It is up to the user to write a **yylex()** routine that breaks the input into tokens and returns the tokens one by one to **yyparse()**. See "Function Section" on page 65 for more information on the lexical analyzer.

Grammar Rules

The grammar rules given to **yacc** not only describe what inputs are valid according to the grammar, but also specify what action is to be taken when a given input is encountered. For example, if the parser recognizes a statement that assigns a value to a variable, the parser should either perform the assignment itself or take some action to ensure that the assignment eventually takes place.

If the parser is part of an interactive desk calculator, it can carry out arithmetic calculations as soon as the instructions are recognized; however, if the parser is the first pass in a compiler, it may simply encode the input in a way that is used in a later code-generation pass.

In summary, you must provide a number of things when using **yacc** to produce a parser:

- Grammar rules indicating what input is and is not valid.
- A lexical analyzer—**yylex()**—that breaks raw input into tokens for the parsing routine **yyparse()**.
- Any source code or functions that may be needed to perform appropriate actions once particular inputs are recognized.
- A mainline routine that performs any necessary initializations, calls **yyparse()**, and then performs possible cleanup actions. The simplest kind of mainline is just a function **main** that calls **yyparse()** and then returns.

Input to yacc

This section describes the input to **yacc** when you are defining an LALR(1) grammar.

The input to **yacc** is broken into three sections:
- Declarations section
- Grammar rules section
- Functions section

The contents of each section are described shortly, but first, here are some overall rules for **yacc** input.

Sections of **yacc** input are separated by the symbol **%%**.

The general layout of **yacc** input is therefore:

```
declarations
%%
grammar rules
%%
functions
```

You can omit the declarations section if no declarations are necessary. In this case, the input starts with the first **%%**. You can also omit the function section, from the second **%%** on. The simplest input for **yacc** is therefore:

```
%%
grammar rules
```

Blanks, tabs, and newlines are used to separate items in **yacc** input. These are called *white-space* characters. Wherever a white-space character is valid, any number of blanks, tabs, or newlines can be used. This means, for example, that the **%%** to separate sections does not have to be on a line all by itself; however, giving it a line of its own makes the **yacc** input easier to read.

Comments may appear anywhere a blank is valid. As in C, comments begin with **/*** and end with ***/**.

Identifiers used in **yacc** input can be of arbitrary length, and can consist of all letters (uppercase and lowercase), all digits, and the characters dot (**.**) and underscore (_). The first character of an identifier cannot be a digit. **yacc** distinguishes between uppercase and lowercase letters; **this**, **THIS**, and **This** are all different identifiers.

Literals in **yacc** input consist of a single character enclosed in single quotes—for example, `'c'`. The standard C escape sequences are recognized:

```
\b – backspace
\n – newline
\r – carriage return
\t – tab
\v – vertical tab
\' – single quote
\\ – backslash
\nnn – any character
(nnn is octal representation)
```

For technical reasons, the null character (`\000`) should never appear in **yacc** input.

Declarations Section

The declarations section describes many of the identifiers that are used in the rest of the **yacc** input. There are two types of declarations:

- Token declarations
- Declarations of functions and variables used in the actions that the parser takes when a particular input is recognized

The declarations section can also specify rules for the precedence and binding of operators used in the grammar. For example, you normally define the standard order of arithmetic operations in the declarations section.

Token Declarations

All characters are automatically recognized as tokens. For example, `'a'` stands for a token that is the literal character *a*.

Other tokens are declared with statements of the form:

```
%token name1 name2 name3 ...
```

This tells **yacc** that the given names refer to tokens. For example:

```
%token INTEGER
```

indicates that the identifier **INTEGER** refers to a particular type of token returned by the lexical analyzer **yylex()**. If **INTEGER** stands for any integer number token, you might have the following code in a handcoded **yylex()**:

```
c = getchar();
if ((c >= '0') && (c <= '9')) {
      yylval = 0;
      do {
          yylval = (yylval * 10) + (c - '0');
          c = getchar();
      } while (c >= '0' && c <= '9');
      ungetc(c, stdin);
      return(INTEGER);
}
```

yylex() returns **INTEGER** to indicate that a certain kind of token (an integer number) has been returned. The actual value of this number is returned in **yylval**. The grammar rules in the **yacc** input dictate where an **INTEGER** token is valid.

In the C source code produced by **yacc**, the identifiers named in a **%token** statement appear as constants set up with **#define**. The first named token has a defined value of 257, the next is defined as 258, and so on. Token values start at 257, so they do not conflict with characters that have values in the `0`-to-255 range or with character 256, which is used internally by **yacc**.

Because token identifiers are set up as defined constants, they must not conflict with reserved words or other identifiers that are used by the parser. For example:

```
%token if yyparse ...
```

almost certainly leads to errors when you try to compile the source code output of **yacc**. To avoid this, this manual uses the convention of creating token names in uppercase, and you should follow the same practice.

Precedence and Associativity

Parsers that evaluate expressions usually have to establish the order in which various operations are carried out. For example, parsers for arithmetic expressions usually carry out multiplications before additions. Two factors affect order of operation: precedence and associativity.

Precedence dictates which of two *different* operations is to be carried out first. For example, in:

```
A + B * C
```

the standard arithmetic rules of precedence dictate that the multiplication is to take place before the addition. Operations that are to be carried out first are said to have a *higher* precedence than operations that are to be performed later.

Different operators can sometimes have the same precedence. In C, for example, addition and subtraction are similar enough to share the same precedence.

Associativity indicates which of two *similar* operations is to be carried out first. By *similar*, we mean operations with the same precedence (for example, addition and subtraction in C). For example, C chooses to parse

```
A - B - C
```

as

```
(A - B) - C
```

whereas such languages as APL or FORTRAN use:

```
A - (B - C)
```

If the first operation is evaluated before the second (as C does), the operation is *left associative*. If the second operation is evaluated before the first (as APL does), the operation is *right associative*.

Occasionally, a compiler may have operations that are not associative. For example, FORTRAN regards:

```
A .GT. B .GT. C
```

as invalid. In this case, the operation is *nonassociative*.

You can declare the precedence and associativity of operator tokens in the declarations section by using the keywords:

```
%left
%right
%nonassoc
```

For example:

```
%left '+' '-'
```

indicates that the **+** and **–** operations have the same precedence and are left associative.

Associativity declarations should be given in order of precedence. Operations with lowest precedence are listed first, and those with highest precedence are listed last. Operations with equal precedence are listed on the same line. For example,

```
%right '='
%left  '+' '-'
%left  '*' '/' '%'
```

says that **=** has a lower precedence than **+** and **–**, which in turn have a lower precedence than *****, **/**, and **%**. **=** is also right associative, so that

```
A = B = C
```

is parsed as

```
A = (B = C)
```

Because of the way **yacc** specifies precedence and associativity, operators with equal precedence always have the same associativity. For example, if **A** and **B** have equal precedence, their precedence must have been set with one of

```
%left A B
%right A B
%nonassoc A B
```

which means **A** and **B** must have the same associativity.

The names supplied with **%right**, **%left**, and **%nonassoc** can be literals or **yacc** identifiers. If they are identifiers, they are regarded as token names. **yacc** generates a **%token** directive for such names if they have not already been declared. For example, in:

```
%left '+' '-'
%left '*' '/'
%left UMINUS
```

UMINUS is taken to be a token identifier. There is no need to define **UMINUS** as a token identifier; a **%token** directive is generated automatically if necessary. It is perfectly valid to have an explicit:

```
%token UMINUS
```

if you want; however, it must precede the **%left** declaration.

For a more technical discussion of how precedence and associativity rules affect a parser, see “Ambiguities” on page 81.

Variable and Function Declarations

The declarations section may contain standard C declarations for variables or functions used in the actions specified in the grammar rules section. All such declarations should be included in a block that begins with **%{** and ends with **%}**. For example:

```
%{
    int i, j, k;
    static float x = 1.0;
%}
```

gives a few variable declarations. These declarations are essentially transferred *as is* to the *beginning* of the source code that **yacc** produces. This means that they are *external* to **yyparse()** and therefore global definitions.

Summary

The source code produced by **yacc** contains the following:
- Code from the declarations section
- Parsing tables produced by **yacc** to represent the grammar
- The **yyparse()** routine
- Code specified in the function section

Grammar Rules Section

A **yacc** grammar rule has the general form

```
identifier : definition ;
```

A colon separates the definition from the identifier being defined. A semicolon ends the definition.

The identifiers defined in the grammar rule section are known as *nonterminal symbols*. *Nonterminal* suggests that these symbols are not final; instead, they are made up of smaller things: tokens or other nonterminal symbols.

Here is a simple example of the definition of a nonterminal symbol:

```
paren_expr : '(' expr ')' ;
```

This says that a **paren_expr** consists of a left parenthesis, followed by an **expr**, followed by a right parenthesis. The **expr** is either a token or a nonterminal symbol defined in another grammar rule. This grammar rule can be interpreted to say that a parenthesized expression consists of a normal expression inside parentheses.

A nonterminal symbol can have more than one definition. For example, you might define **if** statements with:

```
if_stat : IF '(' expr ')' stat ;
if_stat : IF '(' expr ')' stat ELSE stat ;
```

This definition assumes that **IF** and **ELSE** are tokens recognized by the lexical analyzer (which means that this parser's **yylex()** can recognize keywords). The definition also assumes that **expr** and **stat** are nonterminal symbols defined elsewhere.

When a single symbol has more than one meaning, **yacc** lets you join the various possibilities into a single definition. Different meanings are separated by "or" bars (|). Thus you can write:

```
if_stat : IF '(' expr ')' stat
        | IF '(' expr ')' stat ELSE stat
        ;
```

This technique is highly recommended, since it makes **yacc** input more readable.

Definitions in a grammar can be recursive. For example:

```
list : item
     | list ',' item
     ;
```

defines **list** to be one or more items separated by commas.

```
intexp : '(' intexp ')'
       | intexp '+' intexp
       | intexp '-' intexp
       | intexp '*' intexp
       | intexp '/' intexp
       | INTEGER
       ;
```

says that an integer expression can be another integer expression in parentheses, the sum of integer expressions, the difference of integer expressions, the product of integer expressions, the quotient of integer expressions, or an integer number standing on its own (where **INTEGER** is a token recognized by the lexical analyzer).

In recursive symbol definitions, it is often useful to have the empty string as one of the possible definitions. For example:

```
program :
        /* the empty string */
      | statement ';' program
      ;
```

defines a program as zero or more statements separated by semicolons.

The definition of **list** was an example of *left recursion* because **list** was on the left in the recursive definition. The definition of **program** was an example of *right recursion*, which is seldom recommended. For a discussion of the pros and cons of the two types of recursion, see "Right Recursion versus Left Recursion" on page 90.

Recognition Actions

In addition to defining what a nonterminal symbol *is*, a grammar rule usually describes what to do if the nonterminal symbol is encountered in parser input. This is called a *recognition action*.

Recognition actions are specified as part of the grammar rule. They are enclosed in brace brackets in the definition:

```
break_stat : BREAK ';' { breakfn(); };
```

In this definition, **break_stat** is a nonterminal symbol made up of the token known as **BREAK**, followed by a semicolon. If this symbol is recognized, the parser invokes a function named **breakfn**. Presumably this is a user-defined function that handles a **break;** statement.

Note: A semicolon is needed to mark the end of the definition, even though the recognition action ends in a brace bracket. Programmers who are used to C should bear this in mind.

For compatibility with some versions of UNIX **yacc**, z/OS UNIX **yacc** lets you put an equals sign (**=**) before the opening brace that begins a recognition action:

```
break_stat : BREAK ';' = { breakfn(); };
```

When a symbol has more than a single definition, a different recognition action may be associated with each definition. The next section shows an example of this.

Token and Symbol Values

One of the most common recognition actions is to return a value. For example, if an input is recognized as an expression to be evaluated, the parser may want to return the resulting value of the expression. To return a value, the recognition action merely assigns the value to a special variable named **$$**. For example:

```
hexdigit : '0' { $$ = 0; }
         | '1' { $$ = 1; }
                ...
         | 'A' { $$ = 10; }
         | 'B' { $$ = 11; }
         | 'C' { $$ = 12; }
         | 'D' { $$ = 13; }
         | 'E' { $$ = 14; }
         | 'F' { $$ = 15; }
         ;
```

is one way to convert hexadecimal digits into numeric values. In this case, **yylex()** just returns the digits it finds, and **yyparse()** performs the actual conversion.

Another common recognition action is to return a value based on one or more of the items that make up the nonterminal symbol. Inside the recognition action, **$1** stands for the value of the first item in the symbol, **$2** stands for the value of the second item, and so on. If the item is a token, its value is the **yylval** value returned by **yylex()** when the token was read. If the item is a nonterminal symbol, its value is the **$$** value set by the recognition action associated with the symbol. Thus you might write:

```
intexp : '(' intexp ')'        { $$ = $2; }
          /* value of parenthesized expression
             is expression inside parentheses */
       | intexp '+' intexp     { $$ = $1 + $3 ; }
          /* value of addition is sum of two
             expressions */
       | intexp '-' intexp     { $$ = $1 - $3 ; }
          /* value of subtraction is difference
             of two expressions */
       |  /* and so on */
       ;
```

This particular definition shows that each part of a multiple definition may have a different recognition action.

In the source code for **yyparse()**, this set of actions is turned into a large **switch** statement. The cases of the **switch** are the various possible recognition actions.

If no recognition action is specified for a definition, the default is:

```
{ $$ = $1 ; }
```

This action just returns the value of the first item (if the item has a value).

Precedence in the Grammar Rules

The discussion of the declarations section showed how precedence can be assigned to *operators*. Precedence can also be assigned to *grammar rules*, and this is done in the grammar rules section.

One way to give a grammar rule a precedence uses the **%prec** construct:

```
%prec TOKEN
```

in a grammar rule indicates that the rule has the same precedence as the specified token.

For example, consider the unary minus operator. Suppose your declaration section contains:

```
%left '+' '-'
%left '*' '/'
%left UMINUS
```

In the grammar rules section, you can write:

```
exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '-' exp %prec UMINUS
    | /* and so on */
    ;
```

You cannot directly set up a precedence for the unary minus, since you had already set up a precedence for the “–” token. Instead, you created a token named **UMINUS** and gave it the precedence you wanted to assign the unary minus. The grammar rule for the unary minus added:

```
%prec UMINUS
```

to show that this rule has the precedence of **UMINUS**.

As another example, you might set up precedence rules for the right shift and left shift operations of C with:

```
%left RS LS
    ...
exp :
    | exp '<' '<' exp %prec LS
    | exp '>' '>' exp %prec RS
    ...
```

In this way you give the shift operations the proper precedence and avoid confusing them with the comparison operations > and <. Of course, another way to resolve this problem is to make the lexical analyzer clever enough to recognize >> and << and to return the **RS** or **LS** tokens directly.

Although symbols like **UMINUS**, **LS**, and **RS** are treated as tokens, they do not have to correspond to actual input. They may just be placeholders for operator tokens that have two different meanings.

Note: The use of **%prec** is relatively rare in **yacc**. People do not usually think of **%prec** in their first draft of a grammar. **%prec** is added only in later drafts, when it is needed to resolve conflicts that appear when the rules are run through **yacc**.

If a grammar rule is not assigned a precedence using **%prec**, the precedence of the rule is determined by the last *token* in the rule. For example, if the rule is:

```
expr : expr '+' expr
```

the last token in the rule is "**+**" (since **expr** is a nonterminal symbol, not a token). Thus the precedence of the rule is the same as the precedence of **+**.

If the last token in a rule has no assigned precedence, the rule does not have a precedence. This can result in some surprises if you are not careful. For example, if you define:

```
expr : expr '+' expr ';'
```

the last token in the rule is "**;**"— so the rule probably does not have a precedence even if **+** does.

Start Symbol

The first nonterminal symbol defined in the rules section is called the *start symbol*. This symbol is taken to be the largest, most general structure described by the grammar rules. For example, if you are generating the parser for a compiler, the start symbol should describe what a complete program looks like in the language to be parsed.

If you do not want the first grammar rule to be taken as the start symbol, you can use the directive:

```
%start name
```

in your rules section. This indicates that the nonterminal symbol *name* is the start symbol. *name* must be defined somewhere in the rules section.

The start symbol must be all-encompassing: Every other rule in the grammar must be related to it. In a sense, the grammar rules form a *tree*: The root is the start symbol, the first set of branches are the symbols that make up the start symbol, the next set of branches are the symbols that make up the first set, and so on. Any symbol that is *outside* this tree is reported as a *useless variable* in **yacc** output. The parser ignores useless variables; it is looking for a complete start symbol, and nothing else.

End Marker

The end of parser input is marked by a special token called the *end marker*. This token is often written as **$end**; the value of the token is zero.

It is the job of the lexical analyzer **yylex()** to return a zero to indicate **$end** when the end of input is reached (for example, at end of file, or at a keyword that indicates end of input).

yyparse() terminates when it has parsed a start symbol followed by the end marker.

Declarations in yyparse()

You can specify C declarations that are local to **yyparse()** in much the same way that you specify external declarations in the Declarations Section. Enclose the declarations in **%{** and **%}** symbols, as in

```
%{
    /* External declarations */
%}
%%
/* Grammar Rules start here */
%{
    /* Declarations here are
           local to yyparse() */
%}
/* Rules */
%%
/* Function section */
```

You can also put declarations at the start of recognition action code, which is local to that action.

Function Section

The function section of **yacc** input may contain functions that should be linked in with the **yyparse()** routine. **yacc** itself does nothing with these functions; it simply adds the source code on the end of the source code produced from the grammar rules. In this way, the functions can be compiled at the same time that the **yacc**-produced code is compiled.

Of course, these additional functions can be compiled separately and linked with the **yacc**-produced code later on (after everything is in object code format). Separate compilation of modules is strongly recommended for large parsers; however, functions that are compiled separately need a special mechanism if they want to use any definitions that are defined in the **yacc**-produced code, and it is sometimes simpler to make the program part of the **yacc** input.

For example, consider the case of **yylex()**. Every time **yylex()** obtains a token from the input, it returns to **yyparse()** with a value that indicates the type of token found. Obviously, then, **yylex()** and **yyparse()** must agree on which return values indicate which kind of tokens. Since **yyparse()** already refers to tokens using compile-time constants (created in the declarations section with the **%token** directive), it makes sense for **yylex()** to use the same constants. The lexical analyzer can do this very easily if it is compiled along with **yyparse()**.

Size might be the determining factor. With very simple parsers, it is easier to put **yylex()** in the function section. With larger parsers, the advantages of separate compilation are well worth the extra effort.

If you are going to compile **yylex()** or other routines separately from **yyparse()**, use the:

```
-D file.h
```

option on the **yacc** command line. **yacc** writes out the compiler constant definitions to the file of your choice. This file can then be included (with the **#include** directive) to obtain these definitions for **yylex()** or any other routine that needs them. The constants are already included in the generated parser code, so you need them only for separately compiled modules.

Lexical Analyzer

The lexical analyzer **yylex()** reads input and breaks it into tokens; in fact, it determines what constitutes a token. For example, some lexical analyzers may return numbers one digit at a time, whereas others collect numbers in their entirety before passing them to the parser.

Similarly, some lexical analyzers may recognize such keywords as **if** or **while** and tell the parser that an **if** token or **while** token has been found. Others may not be designed to recognize keywords, so it is up to the parser itself to distinguish between keywords and other things, such as variable names.

Each token named in the declarations section of the **yacc** input is set up as a defined C constant. The value of the first token named is 257, the value of the next is 258, and so on. You can also set your own values for tokens by placing a positive integer after the first appearance of any token in the declarations section. For example:

```
%token AA 56
```

assigns a value of 56 to the definition of the token symbol **AA**. This mechanism is very seldom needed, and you should avoid it whenever possible.

There is little else to say about requirements for **yylex()**. If the function is to return the value of a token as well as an indication of its type, the value is assigned to the external variable **yylval**. By default, **yylval** is defined as an **int** value, but it can also be used to hold other types of values. For more information, see the description of **%union** in "Types" on page 80.

Internal Structures

To use **yacc** effectively, it is helpful to understand some of the internal workings of the parser that **yacc** produces. This section looks at some of these workings.

As a point of reference, consider a parser with the following grammar:

```
%token NUM
%left '+' '-'
%left '*' '/'
%%
expr : NUM
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '(' expr ')'
     ;
```

States

As the parser reads in token after token, it switches between various *states*. You can think of a state as a point where the parser says, ″I have read *this* particular sequence of input tokens and now I am looking for one of *these* tokens.″

For example, a parser for the C language might be in a state where it has finished reading a complete statement and is ready for the start of a new statement. It therefore expects some token that can legitimately start a statement (for example, a keyword such as **if** or **while**, or the name of a variable for an assignment). In this state, it reads a token. Say it finds the token corresponding to the keyword **if**. It then switches to a new state, where it says, ″I have seen an **if** and now I want to see the **(** that begins the **if** condition.″ When it finds the **(**, it switches again to a state that says, ″I have found **if(** and now I want the start of a condition expression.″

States break the parsing process into simple steps. At each step, the parser knows what it has seen and what it is looking for next.

yacc assigns numbers to every possible state the parser can enter. The 0th state is always the one that describes the parser's condition before it has read any input. Other states are numbered arbitrarily.

Sometimes a particular input is the start of only one construct. For example, the **for** keyword in C can be the start of only a **for** statement, and the **for** statement has only one form.

On the other hand, a grammar can have several nonterminal symbols that start the same way. In the sample grammar, all of:

```
expr '+' expr
expr '-' expr
expr '*' expr
expr '/' expr
```

start with **expr**. If the parser finds that the input begins with **expr**, the parser has no idea which rule matches the input until it has read the operator following the first **expr**.

The parser chooses which state it enters next by looking at the next input token. This token is called the *lookahead symbol* for that state.

Diagramming States

yacc uses simple diagrams to describe the various states of the parser. These diagrams show what the parser has seen and what it is looking for next. The diagrams are given in the parser description report produced by **yacc**. See "yacc Output" on page 75 for more information.

For example, consider the state where the parser has just read a complete **expr** at the beginning of a larger expression. It is now in a state where it expects to see one of the operators **+**, **–**, *****, or **/**, or perhaps the **$end** marker (indicating the end of input). **yacc** diagrams this state as:

```
$accept:  expr.$end
expr:     expr.'+' expr
expr:     expr.'-' expr
expr:     expr.'*' expr
expr:     expr.'/' expr
```

This lists the possible grammar constructs that the parser may be working on. (In the first line, **$accept** stands for the start symbol.) The dot (**.**) indicates how much the parser has read so far.

If the lookahead symbol is *****, the parser switches to a state diagrammed by:

```
expr:     expr '*'.expr
```

In this state, the parser knows that the next thing to come is another **expr**. This means that the only valid tokens that can be read next are "**(**" or **NUM**, since those are the only things that start a valid **expr**.

State Actions

There are several possible actions that the parser can take in a state:

- *Accept* the input
- *Shift* to a new state
- *Reduce* one or more input tokens to a single nonterminal symbol, according to a grammar rule

- *Go to* a new state
- Raise an *error* condition

To decide which action to take, the parser checks the lookahead symbol (except in states where the parser can take only one possible action, so that the lookahead symbol is irrelevant).

This means that a typical state has a series of possible actions based upon the possible values of the lookahead symbol. In **yacc** output, you might see:

```
'+'   shift 8
'-'   shift 7
'*'   shift 6
'/'   shift 5
')'   shift 9
.     error
```

This says that if the parser is in this state and the lookahead symbol is "**+**", the parser shifts to state 8. If the lookahead symbol is "**–**", the parser shifts to state 7, and so on.

The dot (**.**) in the final line stands for any other token not mentioned in the preceding list. If the parser finds any unexpected tokens in this particular state, it takes the **Error** action.

The sections that follow explain precisely what each state action means and what the parser does to handle these actions.

Accept

The **Accept** action happens only when the parser is in a state that indicates it has seen a complete input and the lookahead symbol is the end marker **$end**. When the parser takes the **Accept** action, **yyparse()** terminates and returns a zero to indicate that the input was correct.

Shift

The **Shift** action happens when the parser is partway through a grammar construct and a new token is read in. As an example, state 4 in the sample parser is diagrammed with:

```
expr:     expr.'+' expr
expr:     expr.'-' expr
expr:     expr.'*' expr
expr:     expr.'/' expr
expr:     '(' expr.')'

'+'   shift 8
'-'   shift 7
'*'   shift 6
'/'   shift 5
')'   shift 9
 .    error
```

This shows that the parser shifts to various other states depending on the value of the lookahead symbol. For example, if the lookahead symbol is "*****"—the parser shifts to state 6, which has the diagram:

```
expr:     expr '*'.expr

NUM   shift 2
```

```
'('   shift 1
 .    error

expr  goto 11
```

In this new state, the parser has further shifts it can make, depending on the next lookahead symbol.

When the parser shifts to a new state, it saves the previous state on a stack called the *state stack*. The stack provides a history of the states that the parser has passed through while it was reading input. It is also a control mechanism, as described in “yacc Output” on page 75.

Paralleling the state stack is a *value* stack, which records the values of tokens and nonterminal symbols encountered while parsing. The value of a token is the **yylval** value returned by **yylex()** at the time the token was read. The value of a nonterminal symbol is the **$$** value set by the recognition action associated with that symbol’s definition. If the definition did not have an associate recognition action, the value of the symbol is the value of the first item in the symbol’s definition.

At the same time that the **Shift** action pushes the current state onto the state stack, it also pushes the **yylval** value of the lookahead symbol (token) onto the value stack.

Reduce

The **Reduce** action takes place in states where the parser has recognized all the items that make up a nonterminal symbol. For example, the diagram of state 9 in the sample grammar is:

```
expr:     '(' expr ')'.
```

At this point, the parser has seen all three components that make up the nonterminal symbol **expr**. As the line:

```
 .    reduce (6)
```

shows, it does not matter what the lookahead symbol is at this point. The nonterminal symbol has been recognized, and the parser is ready for a **Reduce** action.

Note: The **(6)** just means that the parser has recognized the nonterminal symbol defined in rule **(6)** of the grammar. See “yacc Output” on page 75 for more information.

The **Reduce** action performs a number of operations. First, it pops states off the state stack. If the recognized nonterminal symbol had *N* components, a reduction pops *N*–1 states off the 1 stack. In other words, the parser goes back to the state it was in when it first began to gather the recognized construct.

Next, the **Reduce** action pops values off the value stack. If the definition that is being reduced consisted of *N* items, the **Reduce** action conceptually pops *N* values off the stack. The topmost value on the stack is assigned to **$N**, the next to **$N–1**, and so on down to **$1**.

Once the **Reduce** action has gathered all the **$X** values, the parser invokes the recognition action that was associated with the grammar rule being reduced. This recognition action uses the **$1–$N** values to come up with a **$$** value for the nonterminal symbol. This value is pushed onto the value stack, thereby replacing the *N* values that were previously on the stack.

If the nonterminal symbol had no recognition action, or if the recognition action did not set **$$**, the parser puts the value of **$1** back on the stack. (In reality, the value is never popped off.)

Lastly, the **Reduce** action sets things up so that the lookahead symbol seems to be the nonterminal symbol that was just recognized. For example, it may say that the lookahead symbol is now an **expr** instead of a token.

Goto

The **Goto** action is a continuation of the **Reduce** process. **Goto** is almost the same as **Shift**; the only difference is that the **Goto** action takes place when the lookahead symbol is a nonterminal symbol while a **Shift** takes place when the lookahead symbol is a token.

For example, state 6 in the sample grammar reads:

```
expr:    expr '*'.expr

NUM   shift 2
'('   shift 1
 .    error

expr  goto 12
```

The first time the parser enters this state, the lookahead symbol is a token and the parser shifts into some state where it begins to gather an **expr**. When it has a complete **expr**, it performs a **Reduce** action that returns to this state and set the lookahead symbol to **expr**. Now when the parser has to decide what to do next, it sees that it has an **expr** for the lookahead symbol and therefore takes the **Goto** action and moves to state 12.

The **Shift** action pushes the current state onto the state stack. The **Goto** does not have to do this: The state was on the stack already. Similarly, **Shift** pushes a value onto the value stack, but **Goto** does not, since the value corresponding to the nonterminal symbol was already put on the value stack by the **Reduce** action. **Goto** replaces the top of the state stack with the target stack.

When the parser reaches the new state, the lookahead symbol is restored to whatever it was at the time of the **Reduce** action.

Essentially then, a **Goto** is like a **Shift**, except that it takes place when you come *back* to a state with the **Reduce** action. Also, a **Shift** is based on the value of a single input token, whereas a **Goto** is based on a nonterminal symbol.

Error

The parser takes the **Error** action when it encounters any input token that cannot legally appear in a particular input location. When this happens, the parser raises the **error** condition. Since error handling can be quite complicated, the whole of the next section is devoted to the subject.

Error Handling

If a piece of input is incorrect, the parser can do nothing with it. Except in extreme cases, however, it is inappropriate for the parser to stop all processing as soon as an error is found. Instead, the parser should skip over the incorrect input and resume parsing as soon after the error as possible. In this way, the parser can find many syntax errors in a single pass through the input, saving time and trouble for the user.

yacc therefore tries to generate a parser that can *restart* as soon as possible after an error occurs. **yacc** does this by letting you specify points at which the parser can pick up after errors. You can also dictate what special processing is to take place if an error is encountered at one of these points.

The Error Symbol

yacc's error handling facilities use the identifier **error** to stand for erroneous input. Therefore, you should not use **error** as the name of a user-defined token or nonterminal symbol.

You should put **error** in your grammar rules where error recovery might take place. For example, you might write:

```
statement: error
    | /* other definitions of a statement */;
```

This tells **yacc** that errors may occur in statements, and that after an error, the parser is free to restart parsing at the end of a complete statement.

The Error Condition

As noted in "Internal Structures" on page 66, **yacc** takes the **Error** action if it finds an input that is not valid in a particular location. The **Error** action has the following steps:

1. See if the current state has a **Shift** action associated with the **error** symbol. If it does, shift on this action.
2. If the current state has no such action, pop the state off the stack and check the next state. Also pop off the top value on the value stack, so that the state stack and value stack stay in synch.
3. Repeat the second step until the parser finds a state that can shift on the **error** symbol.
4. Take the **Shift** action associated with the **error** symbol. This pushes the current state on the stack—that is, the state that can handle errors. No new value is pushed onto the value stack; the parser keeps whatever value was already associated with the state that can handle errors.

When the parser shifts out of the state that can handle errors, the lookahead symbol is whatever token caused the error condition in the first place. The parser then tries to proceed with normal processing.

Of course, it is quite possible that the original lookahead symbol is incorrect in the new context. If the lookahead symbol causes an error again, it is discarded and the error condition stays in effect. The parser continues to read new tokens and discard them until it finds a token that can validly follow the error. The parser then takes whatever action is associated with the valid token.

In a typical grammar, the state that has been handling errors is eventually popped off the stack in a **Reduce** operation.

Notice that the parser always shifts (through the **Shift** action) on the **error** token. It never reduces on **error**, even if the grammar has a state where **error** is associated with a **Reduce** action.

In some situations, an error condition is raised and the parser pops all the way to the bottom of the state stack without finding a state that can handle the **error** symbol. For example, the grammar may have no provisions for error recovery. In this case, **yyparse()** simply terminates and returns a 1 to its caller.

Examples

As a simple example, consider a parser for a simple desk calculator. All statements end in a semicolon. Thus you might see the rule:

```
statement : var '=' expr ';'
          | expr ';'
          | error ';'
          ;
```

When an error occurs in input, the parser pops back through the state stack until it comes to a state where the **error** symbol is recognized. For example, the state might be diagrammed as:

```
$accept:   .statement $end

error  shift 2
NUM    shift 4
.      error

var       goto 7
expr      goto 3
statement goto 5
```

If an error occurs anywhere in an input statement, the parser pops back to this state, and then shifts to state 2. State 2 looks like this:

```
statement:   error.';'

';'   shift 6
 .    error
```

In other words, the next token must be a semicolon. If it is not, another error occurs. The parser pops back to the previous state and takes the **error** shift again. Input is discarded token by token until a semicolon is found. When the semicolon is found, the parser is able to shift from state 2 to state 6, which is:

```
statement:  error ';'.

.    reduce (3)
```

The erroneous line is reduced to a **statement** nonterminal symbol.

Now this example is simple, but it has its drawbacks. It gets you into trouble if the grammar has any concept of block structure or parenthesization. Why? Once an error occurs, the rule:

```
statement : error ';'
```

effectively tells the parser to discard absolutely everything until it finds a character. If you have a parser for C, for example, it would skip over important characters such as **)** or **}** until it found a semicolon. Your parentheses and braces would be out of balance for the rest of the input, and the whole parsing process would be a waste of time. The same principle applies to any rule that shows the **error** token followed by some other nonnull symbol: It can lead to hopeless confusion in a lot of grammars.

It is safer to write the rule in a form like this:

```
statement : error
          | ';'
          | /* other stuff */
```

In this case, the **error** token matches material only until the parser finds something else it recognizes (for example, the semicolon). After this happens, the **error** state is reduced to a **statement** symbol and popped off the stack. Parsing can then proceed as usual.

Error Recognition Actions

The easiest way to generate an error message is to associate a recognition action with the grammar rule that recognizes the error. You can do something simple:

```
statement: error
    {
        printf("You made an error!\n");
    }
```

or you can be fancier:

```
line: error '\n' prompt line
    { $$ = $4; };
prompt: /* null token */
    { printf("Please reenter line.\n"); };
```

If an error occurs, the parser skips until it finds a newline character. After the newline, it always finds a null token matching **prompt**, and the recognition action for **prompt** displays the message:

```
Please reenter line.
```

The final symbol in the rule is another **line**, and the action after the **error** rule shows that the result of the rule (**$$**) should be the material associated with the second input line.

All this means that if the user makes a mistake entering an input line, the parser displays an error message and accepts a second input line in place of the first. This allows for an interactive user to correct an input line that was incorrectly typed the first time.

Of course, this setup works only if the user does not make an error the second time the line is typed too. If the next token he or she types is also incorrect, the parser discards the token and decides that it is still gobbling up the original error.

The yyclearin Macro

After an **Error** action, the parser restores the lookahead symbol to the value it had at the time the error was detected; however, this is sometimes undesirable.

For example, your grammar may have a recognition action associated with the **error** symbol, and this may read through the next lot of input until it finds the next sure-to-be-valid data. If this happens, you certainly do not want the parser to pick up the old lookahead symbol again once error recovery is finished.

If you want the parser to throw away the old lookahead symbol after an error, put:

```
yyclearin ;
```

in the recognition action associated with the **error** symbol. **yyclearin** is a macro that expands into code that discards the lookahead symbol.

The yyerror Function

The first thing the parser does when it performs the **Error** action is to call a function named **yyerror()**. This happens *before* the parser begins going down the state stack in search of a state that can handle the **error** symbol. **yyerror()** is a **lex** and **yacc** library function that simply displays a text string argument to **stderr** using **fprintf**, and returns the integer value received from **fprintf**. The user may choose to supply their own version. See “The lex Library Routines” on page 42 for information about creating a user-defined **yyerror()**.

The simplest **yyerror()** functions either abort the parsing job or just return so that the parser can perform its standard error handling.

The **yacc** passes one argument to **yyerror()**: a character string describing the type of error that just took place. This string is almost always:

```
Syntax error
```

The only other argument strings that might be passed are:

```
Not enough space for parser stacks
Parser stack overflow
```

which are used when the parser runs out of memory for the state stack.

Once **yyerror()** returns to **yyparse()**, the parser proceeds popping down the stack in search of a state that can handle errors.

If another error is encountered soon after the first, **yyerror()** is *not* called again. The parser considers itself to be in a *potential error* situation until it finds three correct tokens in a row. This avoids the torrents of error messages that often occur as the parser wades through input in search of some recognizable sequence.

After the parser has found three correct tokens in a row, it leaves the potential error situation. If a new error is found later on, **yyerror()** is called again.

The yyerrok Macro

In some situations, you may want **yyerror()** to be called even if the parser has not seen three correct tokens since the last error.

For example, suppose you have a parser for a line-by-line desk calculator. A line of input contains errors, so **yyerror()** is called. **yyerror()** displays an error message to the user, throws away the rest of the line, and prompts for new input. If the next line contains an error in the first three tokens, the parser normally starts discarding input *without* calling **yyerror()** again. This means that **yyerror()** does not display an error message for the user, even though the input line is wrong.

To avoid this problem, you can explicitly tell the parser to leave its potential error state, even if it has not yet seen three correct tokens. Simply code:

```
yyerrok ;
```

as part of the error recognition action.

For example, you might have the rule:

```
expr : error {
         yyerrok;
         printf("Please re-enter line.\n");
         yyclearin;
     }
```

yyerrok expands into code that takes the parser out of its potential error state and lets it start fresh.

Other Error Support Routines

YYABORT
: Halts **yyparse()** in midstream and immediately returns a 1. To the function that called **yyparse()**, this means that **yyparse()** failed for some reason.

YYACCEPT
: Halts the parser in midstream and returns a 0. To the function that called **yyparse()**, this means that **yyparse()** ended successfully, even if the entire input has not yet been scanned.

YYRETURN(*value)*
: Halts the parser in midstream and returns whatever *value* is. You should use this rather than simply coding **return(***value)*.

YYERROR
: Is a macro that *fakes* an error. (Note that it is uppercase.) When **YYERROR** is encountered in the code, the parser reacts as if it just saw an error and goes about recovering from the error. "Advanced yacc Topics" on page 84 gives an example of how **YYERROR** can be useful.

yacc Output

yacc can produce several output files. Options on the **yacc** command line dictate which files are actually generated.

The most important output file is the one containing source code that can be compiled into the actual parser. The name of this file is specified with the **-o** *file.c* command line option.

Another possible output file contains compile-time definitions. The name of this file is specified with **-D** *file.h* on the command line. This file is a distillation of the declarations section of the **yacc** input. For example, all the **%token** directives are restated in terms of constant definitions.

```
%token IF
```

appears as:

```
#define IF 257
```

in the definition file (assuming that **IF** is the first token in the declarations section). By including this file with:

```
#include "file.h"
```

separately compiled modules can make use of all the pertinent definitions in the **yacc** input.

The third output file that **yacc** can produce is called the parser description, where the file name is specified with the **-v** or **-V** option. *y.output* is the name of the file when **-v** is used. When you need to specify an alternate name, use **-V**. The parser description is split into three sections:

- A summary of the grammar rules
- A list of state descriptions
- A list of statistics for the parser generated by **yacc**

The sections that follow show what the parser description looks like for the following grammar:

```
%token IF ELSE A
%%
stmt : IF stmt ELSE stmt
     | IF stmt
     | A
     ;
```

Rules Summary

The rules summary section of the parser description begins with the command line used to invoke **yacc**. This is intended to serve as a heading for the output material.

Next comes a summary of the grammar rules. The example has:

```
Rules:
   (0)  $accept:  stmt $end
   (1)  stmt:     IF stmt ELSE stmt
   (2)  stmt:     IF stmt
   (3)  stmt:     A
```

The 0th rule is always the definition for a symbol named **$accept**. This describes what a complete input looks like: the **Start** symbol followed by the end marker. Other rules are those given in the grammar.

yacc puts a form-feed character (0x0C) on the line after the last grammar rule, so that the next part of the parser description starts on a new page.

State Descriptions

The parser description output contains complete descriptions of every possible state. For example, here is the description of one state from the sample grammar:

```
State 2
    stmt :  IF.stmt ELSE stmt
    stmt :  IF.stmt

  IF   shift 2
  A    shift 1
  .    error

  stmt     goto 4
```

By now, this sort of diagram should be familiar to you. The numbers after the word **shift** indicate the state to which the parser shifts if the lookahead symbol happens to be **IF** or **A**. If the lookahead symbol is anything else, the parser raises the error condition and starts error recovery.

If the parser pops back to state 2 by means of a **Reduce** action, the lookahead symbol is now **stmt** and the parser will go to state 4.

As another example of a state, here is state 1:

```
State 1
    (3)   stmt:    A.

 .     reduce (3)
```

This is the state that is entered when an **A** token has been found. The (3) on the end of the first line is a *rule number*. It indicates that this particular line sums up the whole of the third grammar rule that was specified in the **yacc** input. The line:

```
    .    reduce (3)
```

indicates that no matter what token comes next, you can reduce this particular input using grammar rule (3) and say that you have successfully collected a valid **stmt**. The parser performs a reduction by popping the top state off the stack and setting the lookahead symbol to **stmt**.

It is important to distinguish between:

```
  A  shift 1
```

in state 2 and:

```
  .  reduce (3)
```

in state 1. In the **Shift** instruction, the number that follows is the number of a *state*. In the **Reduce** instruction, the number that follows is the number of a *grammar rule* (using the numbers given to the grammar rules in the first part of the parser description). The parser description always encloses rule numbers in parentheses, and leaves state numbers as they are.

Here is the complete list of state descriptions for the grammar:

```
State 0
         $accept: .stmt $end

    IF    shift 2
    A     shift 1
    .     error

    stmt    goto 3

State 1
    (3)  stmt:     A.

    .     reduce (3)

State 2
         stmt:     IF.stmt ELSE stmt
         stmt:     IF.stmt

    IF    shift 2
    A     shift 1
    .     error

    stmt    goto 4

State 3
         $accept:  stmt.$end
    $end  accept
    .     error

State 4
         stmt:     IF stmt.ELSE stmt
    (2)  stmt:     IF stmt. { $end ELSE }

    ELSE  shift 5
    .     reduce (2)

State 5
         stmt:     IF stmt ELSE.stmt

    IF    shift 2
    A     shift 1
    .     error

    stmt    goto 6

State 6
    (1)  stmt:     IF stmt ELSE stmt.

    .     reduce (1)
```

The parser always begins in state 0, that is, in a state where no input has been read yet. An acceptable input is a **stmt** followed by the end marker. When a **stmt** has been collected, the parser goes to state 3. In state 3, the required end marker, **$end**, indicates that the input is to be accepted. Anything else found is excess input and means an error.

In state 4, the rule labeled (2) has:

```
[ $end ELSE ]
```

on the end. This just means that the parser expects to see one of these two tokens next.

Parser Statistics

The last section of the parser description is a set of statistics summarizing **yacc**'s work. Here are the statistics you see when you run the sample grammar through **yacc**:

```
4 rules, 5 tokens, 2 variables, 7 states
Memory:  max = 9K
States: 3 wasted, 4 resets
Items:  18, 0 kernel, (2,0) per state, maxival=16 (1 w/s)
Lalr:  1 call, 2 recurs, (0 trans, 12 epred)
Actions:  0 entries, gotos: 0 entries
Exceptions: 1 states, 4 entries
Simple state elim: 0%, Error default elim: 33%
Optimizer: in 0, out 0
Size of tables:  24 bytes
1 seconds, final mem = 4K
```

Some of these values are machine-independent (for example, the number of rules), others are machine-dependent (for example, the amount of memory used), and some can be different every time you run the job (for example, time elapsed while **yacc** was running).

Many of these are of no interest to the normal user; **yacc** generates them only for the use of those maintaining the **yacc** software. A number of the statistics refer to shift-reduce or reduce-reduce conflicts; for a discussion of these, see “Ambiguities” on page 81. Here is a description of the statistic lines:

`4 rules, 5 tokens, 2 variables, 7 states`
: The four rules are the grammar rules given in the first part of the parser description. The five tokens are **A**, **IF**, **ELSE**, **$endf**, and **error** (which is always defined, even if it is not used in this grammar). The two variables are the nonterminal symbols, **stmt** and the special **$accept**. The seven states are states 0 to 6.

`Memory:  max = 9K`
: This gives the maximum amount of dynamic memory that **yacc** required while producing the parser. This line may also have a *success rate*, which tells how often **yacc** succeeded in having enough memory to handle a situation and how often it had to ask for more memory.

`States: 3 wasted, 4 resets`
: The algorithm that constructs states from the grammar rules makes a guess at the number of states it needs, very early in the **yacc** process. If this guess is too high, the excess states are said to be *wasted*.

  When states from the various grammar rules are being created, a state from one rule sometimes duplicates the state from another (for example, there were two rules that started with **IF** in the previous example). In the final parsing tables, such duplicate states are merged into a single state. The number of *resets* is the number of duplicate states formed and then merged.

`Items: 18, 0 kernel, (2,0) per state, maxival=16 (1 w/s)`
: A state is made of items, and the kernel items are an important subset of these: The size of the resulting parsing tables and the running time for **yacc** are proportional to the number of items and kernel items. The rest of the statistics in this line are not of interest to normal users.

`Lalr: 1 call, 2 recurs, (0 trans, 12 epred)`
: This gives the number of calls and recursive calls to the conflict resolution routine. The parenthesized figures are related to the same process. In some ways, this is a measure of the complexity of the grammar being parsed. This line does not appear if there are no reduce-reduce or shift-reduce conflicts in your grammar.

`Actions: 0 entries, gotos: 0 entries`
: This gives the number of entries in the tables **yyact** and **yygo**. **yyact** keeps track of the possible *shifts* that a program may make and **yygo** keeps track of the *gotos* that take place at the end of states.

`Exceptions: 1 states, 4 entries`
: This gives the number of entries in the table **yygdef**, yet another table used in **yacc**. **yygdef** keeps track of the possible **Reduce**, **Accept**, and **Error** actions that a program may make.

`Simple state elim: 0%, Error default elim: 33%`
: The percentage figures indicate how much table space can be saved through various optimization processes. The better written your grammar, the greater the percentage of space that can be saved; therefore, high percentages here are an indication of a well-written grammar.

`Optimizer: in 0, out 0`
: These are optimization statistics, not of interest to typical **yacc** users.

`Size of tables: 24 bytes`
: The size of the tables generated to represent the parsing rules. This size is given in bytes on the host machine, so it is inaccurate if a cross-compiler is being used on the eventual source code output. The size does not include stack space used by **yyparse()** or debug tables obtained by defining **YYDEBUG**.

`1 second, final mem = 4K`
: The total real time that **yacc** used to produce the parser, and the final dynamic memory of the parser (in K bytes).

Types

Earlier sections mentioned that **yylval** is **int** by default, as are **$$**, **$1**, **$2**, and so on. If you want these to have different types, you can redeclare them in the declarations section of the **yacc** input. This is done with a statement of the form:

```
%union {
     /*
      * possible types for yylval and
      * $$, $1, $2, and so on
      */
}
```

For example, suppose **yylval** can be either integer or floating point. You might write:

```
%union {
     int intval;
     float realval;
}
```

in the declarations section of the **yacc** input. **yacc** converts the **%union** statement into the following C source:

```
typedef union {
    int intval;
    float realval;
} YYSTYPE;
```

yylval is always declared to have type **YYSTYPE**. If no **%union** statement is given in the **yacc** input, it uses:

```
#define YYSTYPE int
```

Once **YYSTYPE** has been defined as a union, you may specify a particular interpretation of the union by including a statement of the form:

```
%type <interpretation> symbol
```

in the declarations section of the **yacc** input. The *interpretation* enclosed in the angle brackets is the name of the union member you want to use. The *symbol* is the name of a nonterminal symbol defined in the grammar rules. For example, you might write:

```
%type <intval> intexp
%type <realval> realexp
```

to indicate that an integer expression has an integer value and a real expression has a floating-point value.

Tokens can also be declared to have types. The **%token** statement follows the same form as **%type**. For example:

```
%token <realval> FLOATNUM
```

If you use types in your **yacc** input, **yacc** enforces compatibility of types in all expressions. For example, if you write:

```
$$ = $2
```

in an action, **yacc** demands that the two corresponding tokens have the same type; otherwise, the assignment is marked as invalid. The reason for this is that **yacc** must always know what interpretation of the union is being used to generate correct code.

The Default Action

The default action associated with any rule can be written as:

```
$$ = $1
```

which means that the value of associated with **$1** on the value stack is assigned **$$** on the value stack when the rule is reduced. If, for example, **$1** is an integer, then **$$** is the same integer after the reduction occurs.

On the other hand, suppose that the recognition action associated with a rule explicitly states:

```
$$ = $1
```

This explicit assignment may not have the same effect as the implicit assignment. For example, suppose that you define:

```
%union {
    float floatval;
    int intval;
}
```

Also suppose that the type associated with **$$** is **floatval** and the type associated with **$1** is **intval**. Then the explicit statement:

```
$$ = $1
```

performs an integer to floating-point conversion when the value of **$1** is assigned to **$$**, whereas the implicit statement did an integer to integer assignment and did *not* perform this conversion. You must therefore be careful and think about the effects of implicit versus explicit assignments.

Ambiguities

Suppose you have a grammar with the rule:

```
expr : expr '-' expr ;
```

and the parser is reading an expression of the form:

```
expr - expr - expr
```

The parser reads this token by token, of course, so after three tokens it has:

```
expr - expr
```

The parser recognizes this form. In fact, the parser can reduce this right away into a single **expr** according to the given grammar rule.

The parser, however, has a problem. At this point, the parser does not know what comes next, and perhaps the entire line is something like:

```
expr - expr * expr
```

If it is, the precedence rules specify that the multiplication is to be performed before the subtraction, so handling the subtraction first is incorrect. The parser must therefore read another token to see if it is really all right to deal with the subtraction now, or if the correct action is to skip the subtraction for the moment and deal with whatever follows the second **expr**.

In terms of parser states, this problem boils down to a choice between *reducing* the expression:

```
expr - expr
```

or *shifting* and acquiring more input before making a reduction. This is known as a *shift-reduce conflict*.

Sometimes a parser must also choose between two possible reductions. This kind of situation is called a *reduce-reduce conflict*.

In case you are curious, there is no such thing as a shift-shift conflict. To see why this is impossible, suppose that you have the following definitions:

```
thing : a b
      | a c
      ;
b : T rest_of_b;
c : T rest_of_c;
```

If the parser is in the state where it has seen **a**, you have the diagram:

```
thing : a.b
thing : a.c
```

You might think that if the lookahead symbol was the token **T**, the parser would be confused, since **T** is the first token of both **b** and **c**; however, there is no confusion at all. The parser just shifts to a state diagrammed with:

```
thing : a T.rest_of_b
thing : a T.rest_of_c
```

Resolving Conflicts by Precedence

The precedence directives (**%left**, **%right**, and **%nonassoc**) let **yacc**-produced parsers resolve shift-reduce conflicts in an obvious way:

1. The precedence of a **Shift** operation is defined to be the precedence of the token on which the **Shift** takes place.
2. The precedence of a **Reduce** operation is defined to be the precedence of the grammar rule that the **Reduce** operation uses.

If you have a shift-reduce conflict, and the **Shift** and **Reduce** operations both have a precedence, the parser chooses the operation with the high precedence.

Rules to Help Remove Ambiguities

Precedence cannot resolve conflicts if one or both conflicting operations have no precedence. For example, consider the following:

```
statmt:  IF '(' cond ')' statmt
    |    IF '(' cond ')' statmt ELSE statmt ;
```

Given this rule, how should the parser interpret the following input?

```
IF ( cond1 ) IF ( cond2 ) statmt1 ELSE statmt2
```

There are two equally valid interpretations of this input:

```
IF ( cond1 ) {
     IF ( cond2 ) statmt1
     ELSE statmt2
}
```

and:

```
IF ( cond1 ) {
     IF ( cond2 ) statmt1
}
ELSE statmt2
```

In a typical grammar, the **IF** and **IF-ELSE** statements would not have a precedence, so precedence could not resolve the conflict. Thus consider what happens at the point when the parser has read:

```
IF ( cond1 ) IF ( cond2 ) statmt1
```

and has just picked up **ELSE** as the look-ahead symbol.

1. It can immediately reduce the:

   ```
   IF ( cond2 ) statmt1
   ```

   using the first definition of **statmt** and obtain:

   ```
   IF ( cond1 ) statmt ELSE ...
   ```

   thereby associating the **ELSE** with the first **IF**.
2. It can shift, which means ignoring the first part (the **IF** with **cond1**) and going on to handle the second part, thereby associating the **ELSE** with the second **IF**.

In this case, most programming languages choose to associate the **ELSE** with the second **IF**; that is, they want the parser to shift instead of reduce. Because of this (and other similar situations), **yacc**-produced parsers are designed to use the following rule to resolve shift-reduce conflicts.

Rule 1

If there is a shift-reduce conflict in situations where no precedence rules have been created to resolve the conflict, the default action is to shift.

The conflict is also reported in the **yacc** output so you can check that shifting is actually what you want. If it is not what you want, the grammar rules have to be rewritten.

The rule is used only in situations where precedence rules cannot resolve the conflict. If both the shift operation and the reduce operation have an assigned precedence, the parser can compare precedences and decide which operation to perform first. Even if the precedences are equal, the precedences must have originated from either **%left**, **%right**, or **%nonassoc**, so the parser knows how to handle the situation. The only time a rule is needed to remove ambiguity is when one or both of the shift or reduce operations does not have an assigned precedence.

In a similar vein, **yacc**-produced parsers use the following rule to resolve reduce-reduce conflicts.

Rule 2

If there is a reduce-reduce conflict, the parser always reduces by the rule that was given first in the rules section of the yacc input.

Again, the conflict is reported in the **yacc** output so that users can ensure that the choice is correct.

Precedence is *not* consulted in reduce-reduce conflicts. **yacc** always reduces by the earliest grammar rule, regardless of precedence.

The rules are simple to state, but they can have complex repercussions if the grammar is nontrivial. If the grammar is sufficiently complicated, these simple rules for resolving conflicts may not be capable of handling all the necessary intricacies in the way you want. Users should pay close attention to all conflicts noted in the parsing table report produced by **yacc** and should ensure that the default actions taken by the parser are the desired ones.

Conflicts in yacc Output

If your grammar has shift-reduce or reduce-reduce conflicts, there is also a table of conflicts in the statistics section of the parser description. For example, if you change the rules section of the sample grammar to:

```
stmt : IF stmt ELSE stmt
     | IF stmt
     | stmt stmt
     | A ;
```

you get the following conflict report:

```
Conflicts:
    State  Token      Action
        5  IF         shift 2
        5  IF         reduce (3)
        5  A          shift 1
        5  A          reduce (3)
```

This shows that state 5 has two shift-reduce conflicts. If the parser is in state 5 and encounters an **IF** token, it can shift to state 2 or reduce using rule 3. If the parser encounters an **A** token, it can shift to state 1 or reduce using rule 3. This is summarized in the final statistics with the line:

```
2 shift-reduce conflicts
```

Reading the conflict report shows you what action the parser takes in case of a conflict: The parser always takes the *first* action shown in the report. This action is chosen in accordance with the two rules for removing ambiguities.

Advanced yacc Topics

The following topics are covered in this section:
- Rules with multiple actions
- Selection preferences for rules
- Using nonpositive numbers in **$N** constructs
- Using lists and handling null strings
- Right recursion versus left recursion
- Using **YYDEBUG** to generate debugging information

- Important symbols used for debugging
- Using the **YYERROR** macro
- Rules controlling the default action
- Errors and shift-reduce conflicts
- Making **yyparse()** reentrant
- Miscellaneous points

Rules with Multiple Actions

A rule can have more than one action. For example, you might have:

```
a : A1 {action1} A2 {action2} A3 {action3};
```

The nonterminal symbol **a** consists of symbols **A1**, **A2**, and **A3**. When **A1** is recognized, **action1** is invoked; when **A2** is recognized, **action2** is invoked; and when **A3** is recognized (and therefore the entire symbol **A**), **action3** is invoked. In this case:

```
$1   – is the value of A1
$2   – is the value of $$ in action1
$3   – is the value of A2
$4   – is the value of $$ in action2
$5   – is the value of A3
```

If types are involved, multiple actions become more complicated. If **action1** mentions **$$**, there is no way for **yacc** to guess what type **$$** has, since it is not really associated with a token or nonterminal symbol. You must therefore state it explicitly by specifying the appropriate type name in angle brackets between the two dollar signs. If you had:

```
%union {
    int intval;
    float realval;
}
```

you might code:

```
$<realval>$
```

in place of **$$** in the action, to show that the result had type **float**. In the same way, if **action2** refers to **$2** (the result of action1), you might code:

```
$<realval>2
```

To deal with multiple actions, **yacc** changes the form of the given grammar rule and creates grammar rules for dummy symbols. The dummy symbols have names made up of a $ followed by the rule number. For example:

```
a : A1 {action1} A2 {action2} A3 {action3};
```

might be changed to the rules:

```
$21 : /* null */ {action1} ;
$22 : /* null */ {action2} ;
a : A1 $21 A2 $22 A3 {action3};
```

These rules are shown in the rules summary of the parser description report.

This technique can introduce conflicts. For example, if you have:

```
a : A1 {action1} A2 X;
b : A1 A2 Y;
```

These are changed to:

```
$50 : /* null */ {action1};
a : A1 $50 A2 X;
b : A1 A2 Y;
```

The definitions of **a** and **b** give a shift-reduce conflict because the parser cannot tell whether **A1** followed by **A2** has the null string for **$50** in the middle. It has to decide whether to reduce **$50** or to shift to a state diagrammed by:

```
b : A1 A2.Y
```

As a general rule, you can resolve this conflict by moving intermediate actions to just before a disambiguating token.

Selection Preference for Rules

A *selection preference* can be added to a grammar rule to help resolve conflicts. The following input shows a simple example of how a selection preference can resolve conflicts between two rules:

```
a1 : a b ['+' '-']
     { /* Action 1 */ } ;
a2 : a b
     { /* Action 2 */ } ;
```

The selection preference is indicated by zero or more *tokens* inside square brackets. If the token that follows the **b** is one of the tokens inside the square brackets, the parser uses the grammar rule for **a1**. If it is not one of the given tokens, the parser uses the rule for **a2**. In this way, the conflict between the two rules is resolved; the preference tells which one to select, depending on the value of the lookahead token.

Note: A selection preference states that a rule is to be used when the next token is one of the ones listed in the brackets and is not to be used if it is not in the brackets.

The lookahead token is merely used to determine which rule to select. It is *not* part of the rule itself. For example, suppose you have:

```
a1 : a b ['+' '-'] ;
a2 : a b ;
xx : a1 op expr ;
```

and suppose you have an **a**, a **b**, and "**+**" as the lookahead token. The **+** indicates that the **a** and **b** is to be reduced to **a1**. The parser does this and finds that the **a1** is part of the **xx** rule. The **+** lookahead token is associated with the **op** symbol in the **xx** rule. In other words, a selection preference does not *use up* an input token; it just looks at the token value to help resolve a conflict.

The square brackets of a selection preference may contain no tokens, as in:

```
x : y z [ ];
```

This says that the parser will never use this rule unless it cannot be avoided.

Selection preferences can also be stated using the construct:

```
[ˆ T1 T2 T3 ...]
```

where the first character is a caret (∧) and **T1**, **T2**, and so on are tokens. When this is put on the end of a rule, it indicates that the rule is to be used if the lookahead token is *not* one of the listed tokens. For example:

```
a1 : a b
   { /* Action 1 */ } ;
a2 : a b [^ '+' '-']
   { /* Action 2 */ } ;
```

says that rule **a2** is to be used if the token after the **b** is *not* a **+** or **–**. If the token is **+** or **–**, **a2** is not to be used (so **a1** is).

Selection preference constructs can be put in the middle of rules as well as on the end. For example, you can write:

```
expr : expr ['+' '-'] op expr
       { /* Action 1 */ }
     | expr op expr
       { /* Action 2 */ } ;
```

This states that if the first **expr** is followed by a **+** or **–** you want to use the first rule; otherwise, you want to use the second. The preference does not use up the **+** or **–** token; you still need a symbol (**op**) to represent such tokens.

Selection preferences that appear in the middle of a rule are implemented in the same way as multiple actions, using dummy rules. The previous example results in something like the following:

```
$23 : ['+' '-'] ;
expr : expr $23 op expr
       { /* Action 1 */ }
     | expr op expr
       { /* Action 2 */ } ;
```

(where the *23* in **$23** is just a number chosen at random). The dummy rule that is created is a null string with the selection preference on the end. The first token for **op** is the **+** or **–** that was the lookahead token in rule **$23**.

If a selection preference in the middle of a rule is immediately followed by an action, only one dummy rule is created to handle both the action and the preference.

In most cases, a selection preference counts as a **$N** symbol, but it has no associated value. For example, in:

```
expr : expr ['+' '-'] op expr
```

there is:

```
$1 – first expr
$2 – no value
$3 – op
$4 – second expr
```

If the preference is followed by an action, the preference and the action count as a single **$N** symbol, the value of which is equal to the **$$** value of the action. For example, in:

```
expr : expr ['+' ’-'] {action} op expr
```

there is:

```
$1 – first expr
$2 – $$ of action
$3 – op
$4 – second expr
```

The **%prec** construct is incompatible with rules that contain selection preferences, because the preference is all that is needed to resolve conflicts. For this reason, **yacc** issues an error message if a rule contains both a preference and the **%prec** construct.

Selection preferences can be used to resolve most conflicts. Indeed, there may be cases where the most practical course of action is to write a number of conflicting rules that contain selection preferences to resolve the conflicts, as in:

```
expr : expr ['+' '-'] op expr
     | expr ['*' '/' '%'] op expr
     | expr ['&'; '|'] op expr
            ...
```

Note: Selection preferences of the form:

```
[error]
[ˆ error]
```

are not useful. Selection preferences are implemented through (dummy) **Reduce** actions, but the parser's error-handling routines always look for **Shift** actions and ignore reductions.

Using Nonpositive Numbers in $N Constructs

yacc lets you use constructs like **$0**, **$-1**, **$-2**, and so on in recognition actions. These were once important, but the techniques for specifying multiple actions have made them obsolete. **yacc** supports the constructs only for compatibility with older grammars.

To understand what these constructs mean, it is important that you think in terms of the state stack. Each **$N** construct is associated with a state on the stack; the value of **$N** is the value of the token or nonterminal symbol associated with the state at the time of a **Reduce** operation. (Recall that recognition actions are performed when the appropriate **Reduce** action takes place.) **$1** is the value associated with the state that found the first component of the grammar rule, **$2** is the value associated with the second state, and so on. **$0** is the value associated with the state that was on top of the stack before the first component of the grammar rule was found. **$-1** is the value associated with the state before that, and so on. All of these states are still on the stack, and their value can be obtained in this way.

As an artificial example, suppose that a grammar has the rules:

```
stmt : IF condition stmt
     | WHILE condition stmt
condition : /* something */
          { /* action */ }
```

The action associated with the condition can use the **$-1** construct to find out if the preceding token was **IF** or **WHILE**. (Of course, this assumes that the only items that can precede a condition are the **IF** and **WHILE** tokens.) There are occasionally times when this sort of information is needed.

Using Lists and Handling Null Strings

Grammars often define lists of items. There are two common ways to do this:

```
list : item
     | list item ;
```

or:

```
list : /* null */
     | list item ;
```

The first definition means that every **list** has at least one item. The second allows zero-length lists.

Using the second definition is sometimes necessary or convenient, but it can lead to difficulties. To understand why, consider a grammar with:

```
list1 : /* null */
      | list1 item1 ;
list2 : /* null */
      | list2 item2 ;
list  : list1
      | list2 ;
```

When the parser is in a position to look for a **list**, it automatically finds a null string and then gets a conflict because it cannot decide if the null string is an instance of **list1** or **list2**. This problem is less likely to happen if you define:

```
list1 : item1
      | list1 item1 ;
list2 : item2
      | list2 item2 ;
list  : /* null */
      | list1
      | list2
      ;
```

The parser can determine if it has a **list1** or a **list2** by seeing if the list starts with **item1** or **item2**.

A **yacc**-produced parser avoids infinite recursions that result from matching the same null string over and over again. If the parser matches a null string in one state, goes through a few more states, and shifts once more into the state where the null string was matched, it does not match the null string again. Without this behavior, infinite recursions on null strings can occur; however, the behavior occasionally gets in the way if you *want* to match more than one null string in a row. For example, consider how you might write the grammar rules for types that may be used in a C cast operation, as in:

```
char_ptr = (char *) float_ptr;
```

The rules for the parenthesized cast expression might be written as:

```
cast : '(' basic_type opt_abstract ')' ;
opt_abstract : /* null */
             | abstract;
abstract : '(' abstract ')'
         | opt_abstract '[' ']'
         | opt_abstract '(' ')'
         | '*' opt_abstract
         ;
```

Consider what happens with a cast such as:

```
(int *[ ])
```

This is interpreted as a "*" followed by a null **opt_abstract** followed by a null **opt_abstract** followed by square brackets; however, the parser does *not* accept two null **opt_abstract**s in a row, and takes some other course of action. To correct this problem, you must rewrite the grammar rules. Rather than using the **opt_abstract** rules, have rules with and without an **abstract**:

```
cast : '(' basic_type abstract ')' ;
abstract : /* null */
         | abstract '[' ']'
         | '[' ']'
         | abstract '(' ')'
         | '(' ')'
         | '*' abstract
         | '*'
         ;
```

Right Recursion versus Left Recursion

“Input to yacc” on page 56 mentioned left and right recursion. For example, if a program consists of a number of statements separated by semicolons, you might define it with right recursion as:

```
program : statement
        | statement ';' program ;
```

or with left recursion as:

```
program : statement
        | program ';' statement ;
```

If you think about the way that the state stack works, you can see that the second way is much to be preferred. Consider, for example, the way something like:

```
S1 ; S2 ; S3 ; S4
```

is handled (where all the **S***n*’s are statements).

With right recursion, the parser gathers **S1;** and then go looking for a program. To gather this program, it gathers **S2**. It then looks at the lookahead symbol “;” and sees that this program has the form:

```
statement ';' program
```

The parser then gathers the program after the semicolon. But after **S3**, it finds another semicolon, so it begins gathering yet another program. If you work the process through, you find that the state stack grows to seven entries (one for each **S***n:* and one for each “**;**”) before the first **Reduce** takes place.

On the other hand, if you have the left recursion:

```
program : program ';' statement
```

and the same input, the parser performs a **Reduce** as soon as it sees:

```
S1 ; S2
```

This is reduced to a single state corresponding to the nonterminal symbol **program**. The parser reads **;S3** and reduces:

```
program ; S3
```

to **program** again. The process repeats for the last statement. If you follow it through, the state stack never grows longer than three states, as compared with the seven that are required for the right recursive rule. With right recursion, no reduction takes place until the entire list of elements has been read; with left recursion, a reduction takes place as each new list element is encountered. Left recursion can therefore save a lot of stack space.

The choice of left or right recursion can also affect the order that recognition actions are performed in. Suppose **T** is a token. If you define:

```
x : /* null */
  | y ',' x {a1} ;
y : T {a2} ;
```

then the input:

```
T , T , T
```

performs recognition actions in the order:

```
{a2} {a2} {a2} {a1} {a1} {a1}
```

The **{a2}** actions are performed each time a **T** is reduced to **y**. The **{a1}** actions do not happen until the entire list has been read, because right recursion reads the entire list before any **Reduce** actions take place.

On the other hand, if you define:

```
x : /* null */
  | x ',' y {a1} ;
y : T {a2};
```

the recognition actions for the same input take place in the order:

```
{a2} {a1} {a2} {a1} {a2} {a1}
```

With left recursion, **Reduce** actions take place every time a new element is read in for the list.

This means that if you want the action order:

```
{a2} {a2} {a2} {a1} {a1} {a1}
```

you must use right recursion even though it takes more stack space.

Using YYDEBUG to Generate Debugging Information

If you define a symbol (with the **#define** directive) named in the declarations section and set the variable **yydebug** to a nonzero value, your parser displays a good deal of debugging information as it parses input. The **-t** command line option is a convenient shortcut to defining the symbol named . Your program may set **yydebug** to a nonzero value before calling **yyparse()** or while **yyparse()** is executing. The following describes the output you may see.

Every time **yylex()** obtains a token, the parser displays:

```
read T (VALUE)
```

T is the name of the token and *VALUE* is the numeric value. Thus if **yylex()** has read an **IF** token, you might see:

```
read IF (257)
```

Every time the parser enters a state, it displays:

```
state N (X), char (C)
```

where *N* is the state number as given in the state description report, and *X* and *C* are other integers. *X* is another number for the state; **yacc** actually renumbers the states and grammar rules after it generates the state description report to improve the parser's efficiency, and *X* gives the state number after renumbering. *C* is the token type of the lookahead symbol if the symbol is a token. If the symbol is not a token, or if there is no lookahead symbol at the moment, *C* is –1. As an example:

```
state 6 (22), char (-1)
```

indicates that the parser has entered state 6 on the state description report (state 22 after renumbering) and that the current lookahead symbol is not a token.

Every time the parser performs a **Shift** action, it displays:

```
shift N (X)
```

where *N* is the number of the state that the parser is shifting to and *X* is the number of the same state after renumbering.

Every time the parser performs a **Reduce** action, it displays:

```
reduce N (X), pops M (Y)
```

This says the parser has reduced by grammar rule *N* (renumbered to *X*). After the reduction, the state on top of the state stack was state *M* (renumbered to *Y*).

Important Symbols Used for Debugging

Debugging a **yacc**-produced parser is difficult, since only part of the code is produced by user input. The remainder is standard code produced by **yacc**. This is aggravated by the fact that the state and rule numbers shown in the state description report are not the same as those used when the parser actually runs. For optimization purposes, the states are sorted into a more convenient order. Thus, the *internal* state number used by the program is usually not the same as the *external* state number known to the user.

To help you when examining parser code using a symbolic debugger, the following are a few of the important variables that the parser uses:

yyval Holds the value **$$** at the time of a reduction. This has the type **YYSTYPE**.

yychar
Holds the most recent token value returned by **yylex()**.

yystate
Is the *internal* number of the current state.

yyps Points to the current top of the state stack. Thus **yyps[0]** is the internal number of the current state, **yyps[-1]** is the internal number of the previous state, and so on.

yypv Points to the top of the current value stack. The entries in this stack have the type **YYSTYPE**. When a **Reduce** operation performs a recognition action, this pointer is moved down the stack to the point where:

```
yypv[1] = $1
yypv[2] = $2
```

and so on.

yyi Is the internal number of the rule being reduced by a **Reduce** action.

yyrmap
is an array present only when is defined. It is used to convert internal rule numbers to external ones. For example, **yyrmap[yyi]** is the external number of the rule being reduced by a **Reduce** action.

yysmap
Is an array present only when is defined. It is used to convert internal state numbers to external ones. For example, **yysmap[yystate]** is the external number of the current state.

Using the YYERROR Macro

The **YYERROR** macro creates an artificial error condition. To show how this can be useful, suppose you have a line-by-line desk calculator that allows parenthesizing expressions and suppose you have a variable *depth* that keeps track of how deeply parentheses are nested. Every time the parser finds an opening parenthesis, it adds 1 to *depth*. Every time it finds a closing parenthesis, it subtracts 1.

Consider how the following definitions work:

```
expr : lp expr ')'
          {depth--;}
     | lp error
          {depth--;}
     ;
lp : '(' {depth++;};
```

If no error occurs, the *depth* variable is incremented and decremented correctly. If an error does occur, however, what happens? Your **yyerror()** routine is called on to recover from the error in the middle of an expression. Often, it is more reasonable to postpone this recovery until you reach a point where you have a whole expression; therefore, you might use the following alternate definition:

```
expr : lp error
          {depth--; YYERROR;}
     ;
line : error ’\n' prompt line
       { $$ = $4; } ;
prompt : /* null token */
       {printf("Please reenter line.\n");};
```

Now, what happens when the grammar is asked to parse a line such as:

```
1 + (( a +
```

When the end of the line is encountered, the parser recognizes an error has occurred. Going up the stack, the first state ready to handle the error is:

```
expr : lp error ;
```

At this point, the parser reduces the input:

```
( a +
```

into an **expr**. The reduction performs the recognition action: it decrements the *depth* variable and then signals that an error has taken place. The **Error** action begins popping the stack again. It finds the previous opening parenthesis, recognizes another:

```
lp error
```

construct, and performs another reduction. The parenthesis count is again decremented, and another error condition is generated.

This time, the grammar rule that deals with the error is the definition of **line**. An error message is issued and a new line is requested. In this way, the parser has worked its way back to error-handling code that can deal with the situation. Along the way, the parser correctly decremented the *depth* variable to account for the missing parentheses.

This method of dealing with errors decrements *depth* for every unbalanced opening parenthesis on the line. This corrects the *depth* count properly. Our first definition (without the **YYERROR** call) would have decremented *depth* only once.

This example is somewhat contrived, of course; you can always just set *depth* to zero whenever you start a new line of input. The usefulness of the technique is more apparent in situations where you obtain memory with **malloc**, whenever you get an opening delimiter and free the memory with **free**, and whenever you get a closing delimiter. In this case, it is obvious that you need to do precisely as many **free** operations as **malloc** operations, so you must raise the error condition for each unbalanced opening delimiter.

You might think that the symbol **lp** is unnecessary, and you can just define:

```
expr : '(' {depth++;} expr ')' {depth--;}
     | '(' error {depth--;} ;
```

However, this does not work in general. There is no guarantee that the action:

```
{depth++;}
```

is performed in all cases, particularly if the token after the "**(**" is one that could not start an expression.

As an interesting example of another way to use **YYERROR**, consider the following (taken from a parser for the Pascal programming language):

```
program:
declaration
| program declaration
;
declaration:
LABEL label_list
| CONST const_list
| VAR var_list
| PROC proc_header
| CTION func_header
;
label_list :
label_list ',' label
| label
|  error
|  error [LABEL CONST VAR PROC FUNC BEGIN]
               { YYERROR; /*  other code */ }
;
```

This deals with errors in two different ways:

1. If an error is followed by one of the tokens **LABEL**, **CONST**, and so on (representing the beginning of new declaration sections in Pascal), the input is reduced to a complete **label_list** and an appropriate action is taken. This action uses **YYERROR** to raise the error condition, but only *after* the reduction has taken place.
2. The other rule is used when the parser finds an error that is not followed by one of the listed tokens. This corresponds to an error in the middle of a label list and requires a different sort of handling. In this case, error handling is allowed to take place immediately, without reduction, because there may be another **label_list** to come.

This kind of approach can be used to distinguish different kinds of errors that may take place in a particular situation.

Rules Controlling the Default Action

The default action is the one that is taken when the parser finds a token that has no specified effect in the current state. In a state diagram, the default action is marked with a dot (**.**). The default is always a **Reduce** or an **Error** action, chosen according to the following rules:

1. If the state has no **Shift** actions and only one **Reduce**, the default is the **Reduce** action.
2. Apart from rule 1, an empty rule never has **Reduce** as a default.
3. If a state has more than one **Reduce** action, the parser examines the *popularity* of each **Reduce**. For example, if reduction A is used with any of three different input tokens and reduction B is used with only one input token, reduction A is three times as *popular* as B. If one **Reduce** action is more than twice as popular as its closest contender (that is, if it is taken on more than twice as many input tokens), and if that **Reduce** action is associated with a rule that contains at least *five* tokens, the popular **Reduce** action is made the default.
4. In all other cases, the default action is an **Error** action. For example, **Error** is chosen when a rule has more than one **Reduce** action, and there is no **Reduce** that is more than twice as popular as all the other contenders.

Note: z/OS UNIX **yacc**'s predecessor UNIX **yacc** always chooses the most popular **Reduce** action as a default (if there is one). It does not use the same requirements as 3. As a result of this difference, z/OS UNIX **yacc**'s parser tables are about 20% larger than UNIX **yacc**'s, but an z/OS UNIX **yacc**-generated parser usually detects errors much earlier than a parser generated by UNIX **yacc**.

Errors and Shift-Reduce Conflicts

A grammar may contain shift-reduce conflicts that occur when an error is encountered. In this case, the **Shift** action is always taken and no warning message is displayed.

Making yyparse() Reentrant

If you define **YYALLOC** in the declarations section (with the **#define** directive), the state and value stacks used by **yyparse()** are allocated dynamically through **malloc** and freed before **yyparse()** returns. In other words, **yyparse()** makes itself reentrant by saving a number of externals when it begins execution and restoring them upon completion. The externals involved are:

```
yylval   yyval   yypvt
yynerrs  yychar  yyerrflag
```

If you specify **longjmp** to get out of **yyparse()** (due to an action), the externals are *not* restored, and **yyparse()** is not reentrant.

Miscellaneous Points

It is incorrect to code either:

```
%token X 0
```

or:

```
%token X 256
```

The value `0` is reserved for the end marker and 256 is reserved for **error**.

If you define **YYSTATIC**, both the state and value stacks are static; otherwise, the state stack is *auto* (allocated on the program stack) and the value stack is static. Defining **YYALLOC** saves both stack space and static space; defining **YYSTATIC** saves stack space.

If you define **YYSYNC**, the parser always has a lookahead token when it performs a shift or reduce action. If the symbol is not defined, the parser obtains a lookahead token only if the value of the token is needed.

Chapter 4. Tutorial on Using make

make can be a key factor in the successful management of software development projects, as well as any other type of project where you must keep a collection of files in synchronization with one another. **make** is used in situations where a change in one file necessitates the *updating* of one or more other files that depend on the changed file.

For example, suppose a program is built from several separate object files, each of which depends on its own source file. If you change a source file and then run **make**, **make** can automatically determine which object files are out of date (older than their corresponding source files). **make** can then recompile the changed source files to get new object modules, and then link all the component object files to get an updated version of the program.

Basic Concepts

This section discusses the major concepts that underlie the **make** command and gives some simple examples of how to use **make**.

The Makefile

To use **make**, you usually require a *makefile*, a text file that describes the interdependencies of the files that you want **make** to supervise, as well as the *recipes* for remaking files whenever necessary.

An example makes this easier to understand. (You will find this example a lot more verbose than a typical makefile, but there is no need to confuse things by taking a lot of shortcuts right now.) The following example shows the contents of a sample makefile for a small program using the **c89** compiler interface:

```
program : main.o func.o
c89 -o program main.o func.o
main.o : main.c
c89 -c main.c
func.o : func.c
c89 -c func.c
```

This makefile consists of three *rules*. The first rule is:

```
program : main.o func.o
c89 -o program main.o func.o
```

The first line in this rule states that the file **program** depends upon the two **.o** files that follow the colon (**:**). If any or all of the **.o** files have changed since the last time **program** was made, **make** attempts to remake **program**. It does this using the recipe on the next line. This recipe consists of a **c89** command that links **program** from the two object files.

Before **make** remakes **program**, it checks to see if any of the **.o** files need remaking. To do this, it checks the other rules in the makefile to determine the dependencies of the **.o** files. If any of the **.o** files need remaking (because they've become *out of date* with their associated **.c** files), **make** remakes the **.o** files first, and then makes **program**. **make** updates each object file by executing the recipe that follows the appropriate file.

Writing a Rule

The previous example showed a collection of simple rules. All the rules follow a consistent format:

```
target target ... : prerequisite
prerequisite ... <tab> recipe
```

make accepts rules with more complicated formats, but this tutorial restricts itself to this simple form for the time being.

The term *target* usually refers to a file made from other files. For example, a target could consist of an object file built by compiling a source file. **make** also recognizes a number of *special targets*, which are not files.

A rule may have several targets:

```
func1.o func2.o : includes.h
    c89 -c func1.c
    c89 -c func2.c
```

This says that if you change **includes.h**, you must update both **func1.o** and **func2.o**.

The *prerequisite* part of a rule consists of a list of files. The targets depend directly or indirectly on these files: if any of the files change, the targets require remaking. The prerequisite list appears on the same line as the targets, separated from the targets by a colon (**:**).

The *recipe* part of a rule consists of one or more commands that remake the target when necessary. The recipe usually begins on the line following the target and prerequisite list. A recipe can consist of any number of lines, but each line in the recipe must begin with a tab character.

Typing a Tab Character

If you are using the **ed** editor, you can type a tab character as a <Esc-i> sequence. After you press <Enter>, the tab character is displayed as the correct number of blanks.

If you are using the ISPF/PDF editor, you cannot type a tab character (ISPF handles only displayable characters). Instead, you can:

1. Select a character that you will not be using in the file—for example, the character `@`.
2. At the beginning of each line of the recipe, type an @ instead of a tab character.
3. When you have finished editing the file, on the command line type:

   ```
   change @
   X'05'
    all 1
   ```

   This converts the @ to the hex character 05, which is a tab.

   In ISPF Edit, the X'05' now displays as a blank space, which you cannot type over. If you use ISPF to edit or browse an existing file that has tabs in it:

   - In browse mode, the X'05' (tab) displays as a period (`.`) by default.
   - In edit mode, the X'05' displays as a blank space. When you edit the file, ISPF displays a message that the file contains “nonprintables” (meaning the tab characters) and tells you how to use the FIND command to locate them. You can change the tabs back to @ by typing this on the command line:

     ```
     change
     X'05'
      @ all 1
     ```

You can insert any number of blank lines between lines in a recipe, provided that each line begins with a tab character. A line that does not begin with a tab ends the recipe.

In the interests of efficiency, **make** executes most recipe lines itself. However, a recipe line may contain a character special to your command interpreter or shell(for example, the **>** and **<** redirection constructs). In these cases, **make** calls the command interpreter to execute the line, so that the special characters are handled properly.

Filenames Containing a Colon

Occasionally, target names may contain a colon:

```
a:file
```

Usually, **make** interprets a colon as the mark separating the target names from the prerequisite list. To avoid confusion, use quotes to enclose any filename that contains a colon:

```
"a:program" : "a:main.o" func1.o ...
    recipe
```

White Space

White space separates items in a target or prerequisite list. White space consists of one or more blanks or tab characters. You can also surround the colon between the target list and the prerequisite list with white space; however, you do not have to.

Continuation Lines

A backslash (\) as the last character of a line indicates that the line is not finished; it continues on to the next line of the file. For example:

```
target list : \
prerequisite list
```

is equivalent to:

```
target list : prerequisite list
```

You will find this useful if the length of a list makes it impossible to fit everything on one line. You can do this several times; a single line can be broken into any number of continuation lines.

Targets with More Than One Recipe

A file may appear as the target in more than one rule. If several of these rules have associated recipes, use a double colon (**::**) to separate the target and prerequisites. As an example, consider the file **A** that depends on three other files: **B**, **C**, and **D**:

```
A :: B C
        first recipe
A :: C D
        second recipe
```

If **A** is up to date with **C** and **D**, but not **B**, **make** executes only the first recipe. If **A** is out of date with **C**, **make** executes both recipes.

When a target has different recipes for different prerequisites, you must use the double colon in each of the rules associated with the target. You can use a single colon in several rules for the same target, provided that only one of those rules contains a recipe. Metarules do not follow this general rule. For more information on metarules, see “Metarules” on page 109.

As a special case, if no prerequisites are specified, the target is always remade.

Comments

A makefile may contain comments. A comment begins with a number sign character (**#**), and extends to the end of the line. Consider the following example:

```
#  This is a comment line
target : prerequisite # This is another comment
recipe # One more comment
```

make ignores the contents of all comments; they simply allow the creator of the makefile to explain its contents.

Running make

To run **make** in its most basic form, type the following command:

```
make
```

When you use **make** in this way, it expects to find your makefile in the working directory with the name **makefile**. Once it finds your makefile, **make** checks to see if the first target has become out of date with its prerequisites. Part of this process requires checking that the prerequisites *themselves* do not require remaking. **make** remakes all the files it requires to properly remake the first target.

Because of this, many users often put an *artificial* rule at the beginning of a makefile, naming all the targets they remake most frequently. The following example could serve as the first rule of a makefile:

```
all : target1  target2 ...
```

The file named **all** does not exist, but when **make** tries to remake **all**, it automatically checks **all**'s specified prerequisites to ensure they do not require remaking. **make** looks through the makefile for any rules that have **all**'s prerequisites as targets. **make** remakes any that have become out of date with their own specific prerequisites. When **make** remakes the files, it displays the recipe lines as it runs them.

You can also specify *targetnames* on the command line:

```
make target1  target2
```

make attempts to remake only the given targets, plus any prerequisites of those targets that need remaking. For example, you could type the following command:

```
make func1.o func2.o
```

make then remakes the given **.o** files, if they require it.

If you give your makefile a name other than **makefile**, or place it in a separate directory, you have to specify the name of the file you want **make** to use. You do this with the **-f** option:

```
make -f filename
```

In this case, you indicate a makefile called *filename*. You can combine these two options; you can specify particular targets *and* a different name for the makefile:

```
make -f filename  target1  target2 ...
```

One other interesting option is **-n**. When you specify this option (before any target names), **make** displays the commands it must execute to bring the targets up to date, but does not actually execute the commands. Consider the following example:

```
make -n program
```

make displays the commands needed to bring **program** up to date. You will find this option useful if you have just created a makefile and you want to check it to see if it behaves the way you expect. In effect, it gives you a dry run of the updating process.

There are a large number of other options for the **make** command. This tutorial discusses a few of these options. The full list of options is provided with the **make** command description in *z/OS UNIX System Services Command Reference* .

Macros

Suppose you are using **make** to maintain a C program that you are compiling with the **c89** command. The **c89** command features a **-L** option that allows you to specify a directory to add to the search path when **c89** searches for libraries.

All the modules that make up this C program should be compiled with libraries from the same directory. This means that you can set up your makefile as follows:

```
module1.o : module1.c
    c89 -L libdir -c module1.c
module2.o : module2.c
    c89 -L libdir -c module2.c
# And so on
```

These commands all use libraries from the directory **libdir**. (They also use the **-c** option, which compiles the source code but does not link it.)

Now suppose that you want to use the libraries stored in the directory **libdir2** instead of those stored in **libdir**. You need to go back to your makefile and change all the:

```
-L libdir
```

references into:

```
-L libdir2
```

This task is time consuming and error-prone. You may easily miss one of the recipes that have to be changed, or make a typing mistake while you are editing the file.

Macros simplify this kind of situation. The term *macro* refers to a symbol that stands for a string of text. The following example demonstrates the form used to create a macro:

```
macro_name = string
```

When **make** encounters the construction:

```
$(macro_name)
```

it expands it to the *string* associated with *macro_name*.

For example, consider the following:

```
CC = c89
CFLAGS = -L libdir
module1.o : module1.c
    $(CC) -c $(CFLAGS) module1.c
module2.o : module2.c
    $(CC) -c $(CFLAGS) module2.c
# And so on
```

The first line creates a macro named **CC**. The makefile assigns the string **c89** (the command that invokes your compiler) to the macro. The second line creates a macro named **CFLAGS**, which contains the options you want to specify to the compiler. Throughout the makefile, the example uses `$(CC)` and `$(CFLAGS)` in place of the compilation command and its options.

This makefile works exactly the same as the previous one; however, it is much easier to change. If you decide that you want to compile with libraries from the directory **libdir2** instead of **libdir**, you just have to change the **CFLAGS** definition to:

```
CFLAGS = -L libdir2
```

By changing the one line, you can change all the appropriate recipes in the file. In the same way, you can add more standard options to your definition of **CFLAGS**.

By changing the definition of **CC**, you can switch to an entirely different C compiler. The following example shows the same makefile in terms of a hypothetical C compiler invoked by **ccomp**.

```
CC = ccomp
CFLAGS = −L libdir
module1.o : module1.c
    $(CC) -c $(CFLAGS) module1.c
module2.o : module2.c
    $(CC) -c $(CFLAGS) module2.c
# And so on
```

You did not need to modify the rules and recipes, just the two macro definitions.

Naming Macros

Any sequence of uppercase or lowercase letters, digits, or underscores (_) may form the name of a macro. The first character cannot be a digit. Traditionally, macros are given uppercase names to stand out more clearly in your makefile.

Because **make** assumes the **$** represents the beginning of a macro expansion when it appears in a makefile, you must type two **$** characters to represent an actual (literal) **$** character. The following example creates a macro named **DOLLAR** containing the single character **$**.

```
DOLLAR = $$
```

Macro Examples

For example, if you are using **c89**, you might have a makefile with these definitions:

```
USER = /usr/jsmith
# directory where object modules are kept
DIROBJ = $(USER)/project/obj
# directory where src modules are kept
DIRSRC = $(USER)/project/src
$(DIROBJ)/module.o : $(DIRSRC)/module.c
   # compile the file
   $(CC) -c $(DIRSRC)/module.c
   # and move the object file to the specified directory
   mv $(DIRSRC)/module.o $(DIROBJ)/module.o
```

This makefile defines macros for the directories that contain source files and object modules. These macros can be changed easily. For example, if you want to store all the object files in a different directory, just change the definition of **DIROBJ**.

The next example comes from a difference between various C compilers. Some compilers put compiled object code into files ending with **.obj** and executable code into files ending with **.exe**, whereas others put the object code into files ending with **.o** and executable code into files with no suffix. If you plan to switch from one system to another, you might use the following macro definitions:

```
O = .obj
E = .exe
program$(E) : module1$(O) module2$(O) ...
    recipe
module1$(O) : ...
```

If you change to a compiler that uses the **.o** suffix for object files, you can just change the definition of **O** to change all the suffixes in the file. Similarly, if you change to a system that does not use suffixes with executable programs, you can define:

```
E =
```

so that `$(E)` expands to an empty (null) string.

When a macro name consists of a single character, **make** lets you omit the parentheses, so that, for example, you can write the macro **$(E)** as **$E**. You will find this useful if you use common suffix macros:

```
program$E : module1$O module2$O ...
    recipe
module1$O : ...
```

Command-Line Macros

The command line that you use to call **make** may contain macro definitions. You place these after any options and before any targets:

```
make -f makefile DIROBJ=/usr/rhood program
```

The macro definition:

```
DIROBJ=/usr/rhood
```

assigning **DIROBJ** the value of **/usr/rhood** follows the **make -f** option and precedes the target **program**.

A macro definition on the command line always overrides any macro definitions inside the makefile.

If a command-line macro definition contains white space, you must enclose it in quotes or apostrophes, as in the following example:

```
make ’FILES = a.c b.c' target target...
```

Variations

You can contain a macro name within braces (**{}**) as well as parentheses. The following two forms are equivalent:

```
$(macro)
```

and:

```
${macro}
```

A **$(name)** construct can contain other **$(name)** constructs. For example, suppose you have a program suitable with either the **c89** compiler interface and the hypothetical **ccomp** compiler. You might write the following in your makefile:

```
CFLAGS_C89 = −L libdir
CFLAGS_CCOMP = −l libdir
CC_C89 = c89
CC_CCOMP = ccomp
module1.o : module1.c
    $(CC_$(COMP)) -c $(CFLAGS_$(COMP)) module1.c
```

You can then call **make** with the following command line:

```
make "COMP=C89"
```

Inside the construct **$(CC_$(COMP))** the **$(COMP)** is replaced with **C89**. The original construct becomes:

```
$(CC_C89)
```

which then expands to **c89**. Similarly, the following transformations occur, in order:

```
$(CFLAGS_$(COMP)) expands to $(CFLAGS_C89)
$(CFLAGS_C89) expands to −L libdir
```

On the other hand, if you call **make** with:

```
make "COMP=CCOMP"
```

the macro expansions produce **CC_CCOMP** and **CFLAGS_CCOMP**. These, in turn, produce **ccomp** and **-I libdir**.

Special Runtime Macros

In addition to the macros already discussed, **make** lets you use a number of *special runtime macros* that **make** expands as it carries out a recipe. These macros yield meaningful results only when they appear in the recipe part of a rule, except for the dynamic prerequisite macros (which are useful outside recipe lines).

The most straightforward of the special macros is **$@**. When this appears in a recipe, it expands to the name of the target currently being updated. For example, suppose we have the rule:

```
file1.o file2.o : includes.h
    cp $@ /backup
    rm $@
    # commands to remake file
```

This rule has two targets. When either target needs remaking, the recipe uses the **cp** command to copy the current target file to the **/backup** directory and then uses the **rm** command to delete the current file. **make** then goes on to remake the file. In this instance, the **$@** conveniently stands for whichever file is being remade. You do not want to delete one of the targets if it was not being remade.

The special macro **$*** stands for the name of the target, with its suffix omitted. For example, if the target is:

```
/dir1/dir2/file.o
```

then **$*** is:

```
/dir1/dir2/file
```

Consider this example of using **$*** in a makefile:

```
file1.o file2.o : include.h
    $(CC) -c $(CFLAGS) $*.c
```

If **include.h** changes, **make** updates **file1.o** by compiling **file1.c**, and updates **file2.o** by compiling **file2.c**. Remember that this form can appear only in the recipe part of a rule, not in the prerequisite list.

The special construct **$&** stands for all the prerequisites of a target in all the rules that apply to that target. **$^** stands for all the prerequisites of a target in the single rule the recipe of which is being used to remake the target. For example, consider:

```
A : B C
    recipe ...
A : D
```

Inside the recipe, **$^** stands for `B C`, whereas **$&** stands for `B C D`.

Note: The **$^** symbol is an extension not found in traditional implementations of **make**.

The **$<** macro is similar to **$∧**,but it only gives the names of the prerequisites that prompt the execution of the associated rule (for normal rules, those newer than the target). In the previous example, if **B** is newer than **A**, but **C** is older, **$<** stands for **B** inside the recipe.

Several other macros of this kind exist. For more detail on runtime macros, see "Runtime Macros" on page 141.

Dynamic Prerequisites

The special macros discussed in the previous section become useful only when used in the recipe part of a rule. There are similar constructs that you can use in the prerequisite part of a rule, written as **$$@**, and **$$***. You can use these constructs to create *dynamic prerequisites*.

When **$$@** appears in the prerequisite list, it stands for the target name. If you are building a library, it stands for the name of the *archive library*. For example, the two following rules are equivalent:

```
file1 : $$@.c
file1 : file1.c
```

Similarly, the following rule uses the dynamic prerequisite symbol as well as one of the special runtime macros discussed in the previous section:

```
file1 file2 file3 : $$@.c
    $(CC) -c $(CFLAGS) $@.c
```

When **$$*** appears in the prerequisite list, it stands for the name of the target, but without the *suffix*.

See "Modified Expansions" for examples that make use of the **$$@** dynamic prerequisite. There are other dynamic prerequisite macros. For more detail see "Dynamic Prerequisites" on page 141 and the **make** command description in *z/OS UNIX System Services Command Reference*.

Modified Expansions

You can modify the way in which **make** expands macros. This section describes extensions not found in traditional implementations of **make**.

The following example shows you how macro modification works. If the macro **FILE** represents the full pathname of a file, then**$(FILE:d)** expands to the name of the directory that contains the file.

For example, if you define:

```
FILE = /usr/george/program.c
```

then **$(FILE:d)** expands to **/usr/george**. The macro modifier **d.** stands for *directories only*. To modify a macro, put a colon followed by one or more modifiers after the macro name.

If a filename has no explicit directory, the **:d** modifier produces dot (**.**), standing for the working directory.

Consider these two other macro modifiers:

```
b (base)  — file portion of name, not including suffix
f (file)  — file portion of name, with suffix
```

Using the previous definition of **$(FILE)**, the two other macro modifiers produce these results:

```
$(FILE:b) expands to   program
$(FILE:f) expands to   program.c
```

You can combine modifiers. For example:

```
$(FILE:db) expands to   /usr/george/program
```

If a macro consists of several pathnames, modifiers apply to each appropriate pathname in the expansion. For example, suppose you define:

```
LIST = /d1/d2/d3/file.ext x.ext d4/y.ext
```

Then you have the following sample macro expansions:

```
$(LIST:d)  →  /d1/d2/d3 . d4
${LIST:b}  →  file x y
${LIST:f}  →  file.ext x.ext y.ext
$(LIST:db) →  /d1/d2/d3/file x d4/y
```

You can apply modifiers to special runtime macros and to the dynamic prerequisite symbol. For example, consider:

```
all: file1.o file2.o

file1.o file2.o : $$(@:b).c
    $(CC) -c $(CFLAGS) $(@:b).c
```

make evaluates these statements as:

```
all: file1.o file2.o

file1.o : file1.c
    $(CC) -c $(CFLAGS) file1.c

file2.o : file2.c
    $(CC) -c $(CFLAGS) file2.c
```

Substitution Modifiers

The substitution modifier is another extension not found in traditional implementations of **make**. It is similar to the modifiers discussed in the previous section but somewhat more complicated.

The substitution modifier has the following form:

```
s/original/replacement/
```

The *original* string normally appears in the macro expansion, and the substitution modifier will replace *original* with the *replacement* string.

As an example, using the previous definition for **$(LIST)**:

```
$(LIST:s/ext/abc/) expands to /d1/d2/d3/file.abc x.abc d4/y.abc
```

Every occurrence of **ext** is replaced with **abc**. As another example:

```
FILE = /usr/jsmith/file.c
$(FILE) : $(FILE:s/jsmith/mjones/)
    cp $(FILE:s/jsmith/mjones/) $(FILE)
```

is equivalent to:

```
/usr/jsmith/file.c : /usr/mjones/file.c
    cp /usr/mjones/file.c /usr/jsmith/file.c
```

You can combine the substitution modifier with other modifiers, and **make** applies the modifiers in order from left to right. For example:

```
$(LIST:s/ext/abc/:f)  expands to  file.abc x.abc y.abc
```

Tokenization

The tokenization modifier is another extension not found in traditional implementations of **make**. For **make**'s purposes, a *token* represents a sequence of characters lacking any blanks or tab characters. **make** interprets a string enclosed in quote characters as a single token, even if the quoted string includes blanks or tabs.

The construct:

```
$(macro:t"string")
```

expands the given macro and puts the given *string* between each token in the expanded macro. This process is called *tokenization*. For example, if you define:

```
LIST = a b c
```

the tokenization construct

```
$(LIST:t"+")
```

produces:

```
a+b+c
```

make places the **+** *between* each pair of tokens; however, it does not add it after the last token. This more useful example puts a **+** and a newline character (**\en**) between pairs of tokens:

```
$(LIST:t"+\n") expands to a+
b+
c
```

"Recipes" on page 117 tells how to use this kind of expansion with linkers.

Prefix and Suffix Operations

The prefix and suffix modifiers:

```
:^"prefix"
:+"suffix"
```

add a prefix or suffix to each space-separated token in the expanded macro. Consider the following macro definition:

```
test = main func1 func2
```

This definition of **test** produces the following expansions:

```
$(test:^"/src/")
```

expands to:

```
/src/main /src/func1 /src/func
```

and:

```
$(test:+".c")
```

expands to:

```
main.c func1.c func2.c
```

You can combine these modifiers:

```
$(test:^"/src/":+".c")
```

expands to:

```
/src/main.c /src/func1.c /src/func2.c
```

If the prefix and suffix strings themselves consist of a blank-separated list of tokens, the expansion produces the cross-product of both lists. For example, given the following macro assignment:

```
test = a b c
```

the following expansions occur:

```
$(test:^"1 2 3") expands to 1a 1b 1c 2a 2b 2c 3a 3b 3c
$(test:+"1 2 3") expands to a1 b1 c1 a2 b2 c2 a3 b3 c3
```

In combination, **make** produces this expansion:

```
$(test:^"1 2 3":+"1 2 3")

expands to 1a1 1b1 1c1 2a1 2b1 2c1 3a1 3b1 3c1
           1a2 1b2 1c2 2a2 2b2 2c2 3a2 3b2 3c2
           1a3 1b3 1c3 2a3 2b3 2c3 3a3 3b3 3c3
```

Inference Rules

So far, you have had to create explicit recipes for remaking every target. You would find it useful, however, if **make** offered a way to state general guidelines, like this: “If you want to remake an object file, compile the source file with the same basename.”

Metarules create such guidelines. Metarules employ a form similar to normal rules; however, they describe general guidelines, not specific recipes for specific rules. This section examines the ways you create and use metarules.

Note: The new metarule format, discussed in this chapter, may not be recognized by older versions of **make**. Older versions of **make** need the less general *suffix rules*. For compatibility, **make** also supports suffix rules; see “Suffix Rules” on page 110 for more information.

Metarules

Consider this simple example of a metarule:

```
%.o : %.c
     $(CC) -c $(CFLAGS) $<
```

The first line says “If the name of a target ends with the suffix **.o** and you do not have an explicit rule, the prerequisite of the target has the same base name but with the suffix **.c**.” After that comes the recipe line, which uses the special **$<**macro to refer to the single prerequisite in this rule (that is, the **.c** file).

As an example of a makefile that uses metarules, consider the following:

```
CC = c89
CFLAGS = −O
FILES=main func
program : $(FILES:+".o")
    $(CC) $(CFLAGS) $& -o program
%.o : %.c
    $(CC) -c $(CFLAGS) $*.c
```

When **make** tries to remake **program**, it checks the two specified object files to see if either needs *remaking*. **make** notes that these files end in the **.o** suffix. Since there is no explicit rule for these files, **make** uses the metarule for targets ending in **.o**:

```
%.o : %.c
    $(CC) -c $(CFLAGS) $*.c
```

make therefore checks on the **.c** files that correspond to the **.o** files. If any of the **.o** files are out of date with respect to their corresponding **.c** files, **make** uses the metarule recipe to remake the **.o** files from the **.c** source.

Note: There is no need for specific rules for any of the **.o** files; the general metarule covers them all.

If a rule is given without a recipe, and a metarule applies, the metarule and the prerequisites in the explicit rule are combined. For example:

```
file.o : includes.h
%.o : %.c
    $(CC) -c $(CFLAGS) $*.c
```

states that **file.o** depends on **includes.h** as well as **file.c**. The metarule is used to remake **file.o** if it is out of date with respect to either **includes.h** or **file.c**.

Suffix Rules

Suffix rules are an older form of inference rule. They have the form:

```
.suf1.suf2:
recipe...
```

make matches the suffixes against the suffixes of targets with no explicit rules. Unfortunately, they don't work quite the way you would expect. The rule

```
.c.o :
recipe...
```

says that **.o** files depend on **.c** files. Compare this with the usual rules

```
file.o : file.c # compile file.c to get file.o
```

and you will see that suffix rule syntax seems backward! This, by itself, serves as good reason to avoid suffix rules.

You can also specify single-suffix rules such as:

```
.c:
recipe...
```

which match files ending in **.c**.

For a suffix rule to work, the component suffixes must appear in the prerequisite list of the .SUFFIXES special target. You turn off suffix rules by placing:

```
.SUFFIXES:
```

in your makefile. This clears the prerequisites of the .SUFFIXES target, which prevents the enactment of any suffix rules. The order in which the suffixes appear in the .SUFFIXES rule determines the order in which **make** checks the suffix rules.

The following steps describe the search algorithm for suffix rules:

1. Extract the suffix from the target.

2. Is it in the .SUFFIXES list? If not, quit the search.
3. If it is in the .SUFFIXES list, look for a double suffix rule that matches the target suffix.
4. If you find one, extract the basename of the file, add on the second suffix, and see if the resulting file exists. If it doesn't, keep searching the double suffix rules. If it does exist, use the recipe for this rule.
5. If no successful match is made, the inference has failed.
6. If the target did not have a suffix, check the single suffix rules in the order that the suffixes are specified in the .SUFFIXES target.
7. For each single suffix rule, add the suffix to the target name and see if the resulting filename exists.
8. If the file exists, execute the recipe associated with that suffix rule. If the file doesn't exist, continue trying the rest of the single suffix rules. If no successful match is made, the inference has failed.

Try some experiments with the **-v** option specified to see how this works.

make also provides a special feature in the suffix rule mechanism for archive library handling. If you specify a suffix rule of the form:

```
.suf.a:
     recipe
```

the rule matches any target having the LIBRARYM attribute set, regardless of what the actual suffix was. For example, if your makefile contains the rules:

```
SUFFIXES: .a .o
.o.a:
echo adding $< to library $@
```

then if **mem.o** exists,

```
make "mylib(mem.o)"
```

causes:

```
adding mem.o to library mylib
```

to be printed.

See "Libraries" on page 119 for more information about libraries and the .LIBRARY and .LIBRARYM attributes.

The Default Rules File

When you run **make**, it usually begins by examining the startup file that contains the default rules.("Command-Line Options" on page 123 explains how to use the **-r** option to prevent **make** from using the default rules in the startup file.)

The startup file is created at the time that you install**make** on your system. The name of the file is **/etc/startup.mk**.

The startup file contains a number of macro definitions andoption settings, as well as various metarules. **make** processes the information in the startup file beforeyour makefile, so you can think of the default information as *predefined*.

Consider the metarules in the startup file.For example, this file contains:

```
O = .o
%$O : %.c
     $(CC) -c $(CFLAGS) $<
```

The definition of the **O** macro gives the standard suffix for object files. The metarule that follows the definition tells how object files can be obtained from **.c** files.

The metarule makes several assumptions:

- The macro **CC** gives the name of the command to invoke the compiler.When you install **make**, you tell the installation procedure which C compiler you are using. The installation procedure then sets things up so that the **CC** macro refers to your choice of C compiler.
- The **CFLAGS** macro specifies any compiler arguments that appear before the name of the source file. You can redefine your own **CFLAGS** macro to specify any standard flags. Again, the installation procedure sets up a default value for **CFLAGS** based on the compiler you use.
- A **-c** option is specified. This option indicates that the source file is only to be compiled, not linked.
- The rule ends with **$<**.Recall that, in normal rules, this special runtime macro stands for the list of prerequisites in the rule that prompt the rule's execution; in this metarule, it stands for the **.c** file associated with the object file being remade.

If some of these assumptions are not useful to you, you may consider changing the startup file.For example, you might change the default definition of **CFLAGS** to a set of compilation options that you intend to use frequently. You can edit the startup file with any text editor.

Controlling the Behavior of make

There are several methods for controlling the way that **make** does its work. This discussion of **make** touches on *attributes*, *special targets*, and *control macros*.

Some Important Attributes

Attributes are qualities which you may attach to targets. When **make** finds it necessary to update a target that has one or more attributes, the attributes cause **make** to take special actions. This section covers only a few of the attributes available; see "Using Attributes to Control Updates" on page 134 for a complete list.

The first attribute is .IGNORE.If **make** encounters an error when trying to remake a target with this attribute, it ignores the error, and goes on trying to remake other targets. (Normally, if **make** encounters an error, it just issues an error message and stops all processing.)

You can assign attributes to targets in two different ways. First, your makefile can contain a separate line of the following form:

```
attribute attribute ... : target target...
```

For example:

```
.IGNORE : file.o
```

indicates that **file.o** has the .IGNORE attribute. Errors that arise while making **file.o** are ignored.

You can also specify attributes inside a rule. The rule would then have the following form:

```
targets attribute attribute ... :
prerequisites
    recipe
```

This assigns attributes to the given targets as well as stating the prerequisites and recipes for the targets. Consider the following example:

```
file.o .IGNORE : file.c
    $(CC) -c $(CFLAGS) file.c
```

indicates that **make** may ignore errors when remaking **file.o**.

When **make** remakes a target, it normally displays the recipe lines that are being used in the operation; however, if a target has the .SILENT attribute, **make** does not display these lines. In addition, **make** does not issue any warnings that might normally result.

The .PRECIOUSattribute may be used in a rule. .PRECIOUS tells **make** that it must not remove the associated target. For example, you can use the following rule to protect object files employed in making a program:

```
.PRECIOUS : main.o func1.o func2.o
```

You will find .PRECIOUS useful because **make** normally removes intermediate targets that did not exist before **make** started execution. For example, if you have a target with dependencies on **main.o**, **func1.o**, and **func2.o**, **make** compiles **main.c**, **func1.c**, and **func2.c** to produce them. These **.o** files are intermediate targets. If they did not exist before **make** is invoked, they are deleted after the target is created. Marking these object files as .PRECIOUS avoids this deletion.

Some Important Special Targets

The *special targets* of **make** are not really targets at all; they are keywords that control the behavior of **make**. These keywords are called *targets* because they appear as targets in lines that have the same format as normal rules.

A rule with a special target may not have any other targets (normal or special); however, some special targets *may* be given attributes.

The sections that follow discuss some useful special targets. "Special Target Directives" on page 135 provides complete details on all the recognized special targets.

The .ERROR Target

A rule of the form

```
.ERROR : prerequisites
     recipe
```

tells **make** to execute the given recipe if it encounters an error in other processing.

For example, you might code:

```
.ERROR :
     echo "We had an error! Removing tempfile."
rm tempfile
```

to issue an error message. Normally, this is not necessary, since **make** displays error messages of its own; however, you can use the .ERROR rule to perform extra *cleanup* actions after errors.

If a special .ERROR rule has prerequisites, all the prerequisites are brought up to date if an error occurs.

Including Other Makefiles

You use the .INCLUDE special target in a rule of the form:

```
.INCLUDE : file1 file2 ...
```

When **make** encounters a rule like this in a makefile, it reads in the contents of the given files (in order from left to right) and uses their contents as if they had appeared in the current makefile. For example, suppose the file **macrodef** contains a set of macro definitions. Then:

```
.INCLUDE : macrodef
```

obtains those macro definitions and processes them as if they actually appeared at this point in the makefile.

It is possible to store *includable* files under other directories. To do this, you use another special target:

```
.INCLUDEDIRS : dir1 dir2 ...
```

specifies a list of directories to be searched if **make** cannot find a relative name in an .INCLUDE rule in the working directory. For example, with:

```
.INCLUDEDIRS : /usr/dir1
.INCLUDE : file1
```

make searches for **file1** in the working directory first, and then in **/usr/dir1**.

If you enclose the filenames in an .INCLUDE rule in angle brackets:

```
.INCLUDE : <file1> <dir/file2>
```

make does not look for these files in the working directory. It goes straight to the directories named in any preceding .INCLUDEDIRS rule. This lets you obtain input for **make** from other directories without worrying about conflicts with files in the working directory.

If a filename given in an .INCLUDE rule is an absolute name (for example, **/usr/jsmith/file**), **make** uses the name as is. In the case of a relative name, **make** looks for the file in the include directories as described earlier.

An included file may contain .INCLUDE rules of its own. This process is called *nesting* include files.

If **make** cannot find a file you want to .INCLUDE , **make** normally issues an error message and quits. However, you can give the .IGNOREattribute to the .INCLUDE target:

```
.INCLUDE .IGNORE : file
```

If **make** cannot find the given file, it simply continues processing the current makefile. .IGNORE is the only attribute that can be given to .INCLUDE.

Environment Variables

The .IMPORT special target imports environment variables and defines them as macros. For example:

```
.IMPORT : SHELL
```

obtains the value of the **SHELL** environment variable. It creates a macro named **SHELL** containing the current value of the **SHELL** environment variable.

If you try to import a currently undefined environment variable, **make** issues an error message and quits. However, you can use the .IGNORE attribute to tell **make** to ignore this error:

```
.IMPORT .IGNORE: HOME
```

The special rule:

```
.IMPORT : .EVERYTHING
```

imports all the currently defined environment variables, and sets up appropriate macros.

You use the .EXPORT special target to export variables to the environment of subsequently run commands. The following line exports environment variables that have the same names as the given macros:

```
.EXPORT : macro1 macro2 ...
```

make assigns the current values of the macros to the environment variables. **make** ignores any attributes attached to this special target. Environment changes do not affect the environment of the process that called **make** (usually your command interpreter).

Some Important Control Macros

Control macros are special macros that give information to **make** and obtain information in return. For example, the **PWD** control macro contains the name of the working directory. Thus you can use **$(PWD)** to refer to the working directory in a makefile.

Some control macros let you control how **make** behaves. For example, you can use the **SHELL** macro to indicate the command interpreter that **make** uses to execute certain recipecommand lines.

The sections that follow describe some useful control macros. “Control Macros” on page 138 provides complete descriptions of all the recognized control macros.

Information Macros

You can obtain certain types of information with *information macros* while using **make**.

DIRSEPSTR
Gives the characters that you can use to separate parts of a file name. This is usually just the slash (**/**) character.

MAKEDIR
Gives the full pathname of the working directory from which **make** was called.

NULL Contains the null string (that is, a string with no characters). This section describes one use of this, later on.

OS Contains the name of the operating system you are using.

PWD Gives the full pathname of the working directory.

make automatically sets all these information macros.

Attribute Macros

You can set attributes for **make** using *attribute macros*. These macros all follow the same pattern. If the macro has a NULL value, **make** turns off the associated attribute. If the macro has a non-NULL value, **make** turns on the associated attribute for all subsequent targets.

As an example, the **.IGNORE** attribute macro lets you assign the .IGNORE attribute to all the targets named in the makefile.

```
.IGNORE = yes
```

turns on the option. **make** gives the .IGNORE attribute to every target and ignores all errors. The following macro assignment assigns the null string to the **.IGNORE** control macro.

```
.IGNORE = $(NULL)
```

After this, **make** only ignores errors in targets that explicitly have the .IGNORE attribute. Note the use of the **NULL** macro in turning off the option.

Similarly, the macros **.PRECIOUS** and **.SILENT** give all targets the associated attributes.

Other Control Macros

Consider this list of some other useful control macros.

MAKESTARTUP
Contains the full pathname of the startup file. A built-in rule sets this to **/etc/startup.mk**, but you can change it on the command line or in the environment.

SHELL
Names a file that contains a shell. Normally, **make** tries to execute recipe lines without calling a shell; however, some recipe lines require execution by a shell to work properly. For example, lines that employ the redirection constructs **>** or **<** require execution by a shell. The **SHELL** macro tells **make** where to find the appropriate shell. The startup file specifies this macro's value.

SHELLFLAGS
Gives a collection of flags to pass to the shell if and when **make** invokes it to execute a recipe command line. The startup file specifies the default value for **SHELLFLAGS**, based on the value of **SHELL**.

SHELLMETAS
Contains a string of characters for which **make** keeps watch when examining recipe command lines. If a command line contains any of the characters in the string line, **make** passes the command line to the shell specified by the **SHELL** macro. If a command line does not contain any of these characters, **make** executes it directly.

As an example, you want the **SHELLMETAS** macro to contain the redirection symbols **<** and **>** as part of its value. Command recipes commonly employ redirection, but **make** must perform redirection through a shell; **make** cannot directly perform redirection. The startup file specifies a default value for **SHELLMETAS**, based on the value of **SHELL**.

Recipes

Recipe Lines

Until now, examples have placed all recipe lines after the first line of a rule, starting every recipe line with a tab. In fact, you can put a recipe on the same line as the prerequisite list if you put a semicolon (**;**) after the list. For example, you can write:

```
 %.o : %.c ; $(CC) -c $(CFLAGS) $<
```

The recipe comes immediately after the semicolon.

As another feature, **make** lets you designate special processing for particular recipe lines. If the tab at the beginning of a recipe line is immediately followed by an at character (**@**), **make** does *not* echo the line when it is executed. Using the **@** this way affects **make** like .SILENT, but for one line only:

```
file1.o : file1.c
    @cp file1.o /backup
    $(CC) -c $(CFLAGS) file1.c
```

make does not show the **cp** command when executing it, but does display the compilation command.

A minus sign (**\-**) immediately following the initial tab of a recipe line, affects **make** like .IGNORE,but for one line only:

```
file1.o : file1.c
    -cp file1.o /backup
    $(CC) -c $(CFLAGS) file1.c
```

make does not stop if the **cp** command gets an error (for example, because the device with the directory **/backup** is full). More technically, when minus sign precedes a command line, **make** ignores any nonzero return value the command produces.

A plus sign (**+**) immediately following the initial tab of a recipe line,forces **make** to execute the recipe line even when you specify the **-n**, **\-q**, or **\-t** options. You will find this particularly useful when doing a recursive make. For example, suppose you have the following rule in your recipe:

```
dir :
+make -c subdir
```

and you invoke **make** in the following way:

```
make -n
```

make simply prints most commands. However, **make** executes this recipe line allowing you to see what **make** will build in **subdir**. Because **make** will place **-n** in the **MAKEFLAGS** inherited by the child process, it also will print rather than execute. This allows you to see all of the commands that would be executed, not just the ones in the working directory.

You can combine these markers in any order:

```
file1.o : file1.c
    -@+cp file1.o /backup
    $(CC) -c $(CFLAGS) file1.c
```

Executing Regular Recipes

To update a target, **make** expands and executes a recipe. The expansion process replaces all macros and text diversions within the recipe. Then **make** either executes the commands directly, or passes them to a shell.

When **make** calls a regular recipe, it executes each line of the recipe separately (using a new shell for each, if a shell is required). This means that the effect of some commands does not persist across recipe lines. For example, a change directory (**cd**) request in a recipe line changes only the current working directory for that recipe line. The next recipe line reverts to the previous working directory.

The value of the macro **SHELLMETAS** determines whether **make** uses a shell to execute a command. If **make** finds any character in the value of **SHELLMETAS** in the expanded recipe line, it passes the command to a shell for execution; otherwise, it executes the command directly. Also, if the makefile contains the .POSIX target, **make** always uses the shell to execute recipe lines.

To force **make** to use a shell, you can add characters from **SHELLMETAS** to the recipe line.

The value of the macro **SHELL** determines the shell that **make** uses for execution. The value of the macro **SHELLFLAGS** provides the options that **make** passes to the shell. Therefore, the command that **make** uses to run the expanded recipe line is:

```
$(SHELL) -$(SHELLFLAGS) expanded_recipe_line
```

When **make** is about to invoke a recipe line, it normally writes the line to the standard output. If the .SILENT attribute is set for the target or the recipe line (using **@**), **make** does not echo the line.

Group Recipes

make supports *group recipes*, but traditional implementations of **make** do not. A group recipe signifies a collection of command lines fed as a unit to the command interpreter. By contrast, **make** executes commands in normal recipe one by one.

You enclose a group recipe's command lines in square brackets. The opening square bracket ([) must appear as the first non-white space character in a line. The closing square bracket (]) must also appear as the first non-white space character in a line. The square brackets can enclose as many command lines as you want. Recipe lines must begin on the line following the opening square bracket.

A typical group recipe might involve special command constructs, such as the looping constructs of the z/OS Shell. Consider the following example:

```
book : chap1.tr chap3.tr
[
    >book
    for i in $&
    do
        fmt -j -l 66 $$i >>book
    done
]
```

This creates a shell **for** loop that uses the **fmt** command to format each file under the **dir** directory and append the formatted material to the **book** file. A normal rule cannot be written in this way, because the recipe command lines in a normal rule are executed one by one.

Note: **make** expands the group recipe; therefore, you must write the **$i** shell variable as **$$i**; otherwise, **make** attempts to expand the **$i make** variable.

The command lines inside a group recipe do not require an initial tab character. Also, an @ character, a + character, or a – character immediately after the opening ([) has the same effect as in a normal recipe, for the entire group recipe:

1. @ silences the group recipe execution
2. + causes the recipe always to be executed regardless of the option flags set
3. – ignores error returns

Special Group Recipe Constructs

You can set the **GROUPSHELL** control macro to indicate which command interpreter will receive your group recipes. For example, you might set:

```
SHELL = rsh
GROUPSHELL = sh
```

so that you pass normal recipes to the restricted shell and group recipes to the full z/OS Shell. The default rules specify the same value for the **GROUPSHELL** as for **SHELL**.

When **make** encounters group recipes, it creates a temporary file to hold the command lines and then submits this temporary file to the shell.

The **GROUPFLAGS** control macro lets you specify any option flags **make** uses when invoking a group recipe. This is similar to the **SHELLFLAGS** control macro used for normal recipe lines.

Executing Group Recipes

Group recipe processing is similar to that of regular recipes, except that **make** always invokes a shell. **make** writes the entire group recipe to a temporary file, with a suffix provided by the **GROUPSUFFIX** macro. **make** then submits this temporary file to a command interpreter for execution. The value of **GROUPSHELL** provides the appropriate command interpreter, and **make** provides the flags from the value of **GROUPFLAGS**.

If you have set the .PROLOG attribute for the target being made, **make** adds the recipe associated with the special target .GROUPPROLOG at the beginning of the group recipe. If you have also set the .EPILOG attribute, **make** adds the recipe associated with the special target .GROUPEPILOG onto the end of the group recipe. You can use this facility to append a common header or trailer to group recipes.

make echoes group recipes to standard output just like standard recipes.

Libraries

It is often good programming practice to save compiled object code in an *object library*, a collection of object modules stored in a single file. When a library is linked with your code, only the object modules referred to in the library are actually linked into the final program.

If object code is stored in a library, your makefile must have access to the code from that library. This means you have to tell **make** when a target is a library, since **make** requires special handling to check whether library members are up to date.

To make a library, specify the library as a target with the .LIBRARY attribute, and give as prerequisites the object files that you want to make members. If you specify the prerequisites in the form:

```
name (member)
```

then **make** automatically sets the .LIBRARY attribute for the target, and interprets the *member* inside the parentheses as a prerequisite of the library target.

make employs the .LIBRARY attribute to determine if a particular target is a library:

```
LIBOBJS = mod1 mod2 mod3
userlib$(LIBSUFFIX) .LIBRARY : $(LIBOBJS:+"$O")
```

This example tells **make** that **userlib$(LIBSUFFIX)** has the .LIBRARY attribute and is therefore a library. The prerequisites for this target are the object files

```
mod1$O mod2$O mod3$O
```

This example makes use of the **LIBSUFFIX** macro defined in the startup file.**LIBSUFFIX** specifies the usual suffix for libraries, just as **O** specifies the usual suffix for object files. (For brevity, the default rules also define the **A** macroequal in value to **LIBSUFFIX**.)

make gives the prerequisites of a .LIBRARY target the .LIBRARYM attribute. The library name is also internally associated with the prerequisites. This lets the file binding mechanism look for the member in an appropriate library if an object file cannot be found.

Using these features, you can write:

```
mylib$A : mylib$A(mem1$O) mylib$A(mem2$O)
    recipe for making library
```

Note that **make** gives the **A** macro the same value as the **LIBSUFFIX** macro in the startup file.

In any rule, you may use a construct of the form:

```
libname$(LIBSUFFIX)(member)
```

to refer to an object file contained in a library. This kind of construct may appear as a target or prerequisite. For example, you might have:

```
 prog$E : prog$O mylib$(LIBSUFFIX)(module$O)
     # recipe for linking object and library
```

make infers the following information from this:

- The file **mylib$(LIBSUFFIX)** is a library.
- The module **module$O** is a member of that library; and therefore, it is a prerequisite for the library.
- The **module$O** module inside the library is a prerequisite of **prog$E** (that is, the program links in that module).

The recipe in this rule should tell **make** how to link the object file with the library module. The library metarules in the standard startup file specify themeans for updating libraries.

If a target or prerequisite has the form:

```
name ((entry))
```

make gives the *entry* the .SYMBOL attribute, and gives the target *name* the .LIBRARY attribute. **make** then searches the library for the entry point, and returns not only the modification time of the member which defines the entry, but also the name of the member file. This name then replaces *entry*, and **make** uses it for making the member file. Once bound to a library member, **make** removes the .SYMBOL attribute from the target.

Metarules for Library Support

The startup file defines several macros and metarules that are useful in manipulating libraries.**LIBSUFFIX** and **A** both give the standard suffix for a library, and **O** gives the standard suffix for an object file. The **AR** macro specifies the librarian program. By default, the macro contains the **ar** program provided with **make**. By default, **ARFLAGS** contains the string **-ruv**. These flags cause **ar** to update the library with all the specified members that have changed since the library was last updated. **ar** updates libraries stored in the standard library format. You can assign the **ARFLAGS** macro any option flags used in the library updating process; the default rules set the flags to update an existing library, or create a new library as appropriate.

For further information on the **ar** command, see the command description in *z/OS UNIX System Services Command Reference*.

The startup file contains the following metarule:

```
%$(LIBSUFFIX) .LIBRARY .precious :
    $(AR) $(ARFLAGS) $@ $?
```

With this metarule, you need not directly use the **ar** command in your makefile. **make** automatically rebuilds a library with the appropriate suffix when any of the prerequisite object modules are out of date.

You can accomplish your library handling simply by specifying the names of the object members of the library:

```
LIBOBJS= mod1 mod2 mod3
userlib$(LIBSUFFIX) .LIBRARY: $(LIBOBJS:+"$O")
```

As an example of the effect of this metarule, suppose that a makefile contains:

```
lib$A .LIBRARY : mod1$O mod2$O mod3$O
```

make gives the .LIBRARY attribute to the **lib$A** target, so the metarule applies:

```
make lib.a
```

The startup file contains a metarule for making executable files from object files. This metarule adds the value of the macro **LDLIBS** as a list of libraries to be linked with the object files. If you have several programs, all of which depend on the same library, you can add the name of your library to the definition of **LDLIBS**, and automatically get it linked when using the metarule. For example, assume this metarule for your compiler:

```
 %$E :  %$O
      $(LD) $(LDFLAGS) -o $@ $< $(LDLIBS)
```

You can add the following lines to your makefile:

```
LDLIBS += mylib$A
program1$E : mylib$A
program2$E : mylib$A
```

The first line adds **mylib$A** to the current definition of **LDLIBS**. Subsequent lines describe the programs you want to build using this library; because a recipe is not given, **make** uses the metarule from the startup file to relink the programs. Thus, the command:

```
make program1
```

remakes the library **mylib.** if required, and then relinks **program1** from **program1.o** using the libraries specified in **LDLIBS**.

Chapter 5. More Information on make

The following example describes the general form of the **make** command line:

```
make [ options ] [ macro definitions ] [
  target ... ]
```

You can omit items shown between [and] brackets. The brackets are part of the standard documentation style; they enclose optional items and are not used on **make**'s actual command line.

The *target*s specified on the command line are usually filenames. **make** attempts to update these *target*s, if necessary, using the rules defined in a startup file and rules taken from a user makefile.

If you do not specify any *target* names on the command line, **make** attempts to find a makefile. It also updates the first nonspecial target specified in the makefile. ("Special Target Directives" on page 135 describes special targets.)

The *macro definitions* specified on the command line have the same form as macro definitions in a makefile. Command-line *macro definitions* take effect after any definitions in the startup file and the user makefile. See "Macros" on page 129 for more information.

Command-Line Options

You can specify a number of options on the **make** command line. Most take the form of a minus sign (**–**) followed by a single letter. The case of the letter is significant; for example, **-e** and **-E** are different *options* and have different effects.

If a command line has several such *options*, they can be *bundled* together. For example, the following two command lines are equivalent:

```
make -i -e
make -ie
```

The following list explains all the command line options of **make**. Many of these match options in other versions of **make**.

-c *dir* Attempts to change into the specified directory when **make** starts up. If **make** can't change the directory, an error message is printed. This is useful for recursive makefiles when building in a different directory.

-E Suppresses reading of the environment. Normally when **make** starts up, it reads all strings defined in the environment into the corresponding macros. For example, if you have an environment variable named **PATH** defined, **make** creates a macro with the same name and value. If you specify neither **\-E** nor **\-e**, **make** reads the environment *before* reading the makefile.

-e Reads the environment *after* reading the makefile. If you specify neither **-e** nor **-E**, **make** reads the environment *before* reading the makefile.

-f *file* Tells **make** to use *file* as the makefile. If you specify a minus sign (**–**) in place of *file*, **make** reads the standard input. (In other words, **make** expects you to enter the makefile from the terminal or redirect it from a file.)

-i Tells **make** to ignore all errors and continue making targets. This is equivalent to the .IGNORE attribute or macro.

-k Makes all independent targets, even if an error occurs. Ordinarily, **make** stops after a command returns a nonzero status. Specifying **-k** tells **make** to ignore the error and continue to make other targets, as long as they are unrelated to the target that received the error. **make** does not attempt to update anything that depends on the target that was being made when the error occurred.

-n Displays the commands that need to be run to update the chosen targets, but does not actually run the commands. This feature works with group recipes, but in this case, **make** will run the commands. If **make** finds the string **$(MAKE)** in a recipe line, that line is run with **$(MAKE)** replaced by:

```
make -n $(MAKEFLAGS)
```

(**MAKEFLAGS** is described in "Special Macros" on page 138). This lets you see what recursive calls to **make** do. ("Makefile Input" on page 125 explains group recipes.)

-p Prints the digested makefile. This display is in a human-readable form useful for debugging, but you cannot use it as input to **make**.

-q Checks whether the target is up to date. If it is up to date, **make** exits with a status of `0`; otherwise, it exits with a status of 1 (typically interpreted as an error by other software). No commands are run when **-q** is specified.

-r Tells **make** not to read the startup file. "Finding the Makefile"

-S Terminates **make** if an error occurs during operations to bring a target up to date (opposite of **-k**). This is the default.

-s Tells **make** to do all its work silently. **make** does not display the commands it is running or any warning messages. This is equivalent to setting the .SILENT attribute, or assigning a nonnull value to the **.SILENT** macro.

-t Touches the targets to mark them as up to date, without actually running any commands to change the targets. Use the **-t** option with caution: careless use may cause **make** to consider files as recently changed (because they have been touched), even though you have not changed them. This can result in a target that isn't brought up to date when required.

-u Forces an unconditional update: **make** behaves as if all the prerequisites of the given target are out of date.

-V Prints the version number of **make**. It also prints the *built-in rules* of this version of **make**. For more about built-in rules, see "Finding the Makefile".

-v Causes **make** to display a detailed account of its progress. This includes:
- What files it reads
- The definition and redefinition of each macro
- Metarule and suffix rule searches
- Other information

-x Exports all macro definitions to the environment. This happens just before making any targets, but after the entire makefile has been read.

Finding the Makefile

make works with information from several different sources:

Built-in rules
The **make** program itself contains built-in rules. They may change from one release to the next, but you cannot change them yourself. The command **make –V** displays the built-in rules for your version of **make**.

Default rules

The standard startup file contains a group of default rules used by **make**. You can specify the name of this startup file by setting the value of the **MAKESTARTUP** environment variable. If **MAKESTARTUP** contains a *null* value (the default), then **make** uses **/etc/startup.mk**. You can use a different file by assigning a filename to **MAKESTARTUP** on the **make** command line as if it were a macro. You can edit the contents of the startup file with a normal text editor. When **make** is installed, the startup file is set up according to your specifications. You should not customize this file until you are familiar with **make** and have decided how you want to control its behavior. This file defines various control macros and default rules; if you lose this file or put incorrect material into it, **make** will not work as documented here. The standard startup file specifies default values for all required control macros and default metarules.

A local default rules file

As distributed, the last line of the startup file prompts **make** to read the local **startup.mk** file, if such a file exists.

The makefile

A makefile is just a normal text file that you create with any text editor. It provides specific rules for remaking your targets. (If you use a word processor or editor that inserts embedded control characters, you have to save the file as a normal text file, without those control characters.)

When you invoke **make**, it first tries to find a startup file and then tries to find a user makefile. **make** follows these steps to find the startup file:

- If the command line contains a macro definition for **MAKESTARTUP**, **make** uses that value as the name of a different startup file. If the file can be read, **make** uses it as the startup file.
- If the command line does not have a **MAKESTARTUP** macro, or if **make** cannot read the file it names, **make** checks the environment for a variable named **MAKESTARTUP**. If this variable exists, **make** attempts to read its value as the startup file.
- If neither of these is successful, **make** looks for the file named **startup.mk** as defined in the built-in rules.

You can therefore use a **MAKESTARTUP** macro definition on the command line or in the environment to obtain a different startup file.

The special target .MAKEFILES determines the location of your makefile. This is discussed in “Special Target Directives” on page 135. The built-in rules version of .MAKEFILES tells **make** to look for **makefile** or **Makefile** in the working directory. **makefile** is tried first; **Makefile** is used only if **makefile** cannot be found. You can also use the **-f** *file* option to give the name of the user makefile explicitly.

If you specify the **-r** option on the command line, **make** does not attempt to read a startup file. Instead, it uses the built-in rules and attempts to find a user makefile directly.

Makefile Input

A makefile can contain any or all of the following:
- Macro definition lines
- Target definition lines
- Recipe lines

- Comments

The ordering of these within a makefile is very flexible. There are only two restrictions:

- The recipe lines for a target must immediately follow the target definition line.
- The recipe describing how to make a target cannot span more than one makefile.

For a discussion of how to use more than one makefile, see the explanation of .INCLUDE in "Special Target Directives" on page 135.

If a makefile line cannot fit on a single text line, you can break it over several text lines by putting a backslash (\) at the end of each partial line. For example:

```
macro = abc\
def
```

is the same as:

```
macro = abcdef
```

If you are using the **-n** option to display what **make** would execute, **make** puts backslash and line-feed characters at the end of each partial line so that the output resembles the makefile input.

Comments

Comments begin with the **#** character and extend to the end of line, as in:

```
# This is a comment
```

make itself ignores all comment text. If you need to put a **#** in your makefile without creating a comment, put a backslash (\) in front of it, or enclose it in double quotes.

Rules

A makefile contains a series of *rules* that specify targets, dependencies, and recipes. For example, a rule might state that an object file depends on a source file; if you change the source file, you want **make** to remake the object file using the changed source.

Files that depend on other files are called *targets*. The files that a target depends on are called *prerequisites*.

This is the general format of a rule:

```
targets [attributes]
ruleop [prerequisites] [; recipe ]
{<tab> recipe}
```

You need to include items enclosed by [ ]; items within { } can appear zero or more times. In a rule:

targets
: Represents a list of one or more dependent files.

attributes
: Represents a list, possibly empty, of attributes to apply to the list of targets. See "Using Attributes to Control Updates" on page 134 for more details.

ruleop
: Represents an operator that separates the target names from the prerequisite names, and optionally affects the processing of the specified

targets. All rule operators begin with a colon (**:**). For more information about rule operators, see “Rule Operators”.

prerequisites
Represents a list of filenames on which the specified targets depend.

recipe May follow on the same line as the prerequisites, separated from them by a semicolon. If such a recipe exists, **make** uses it as the first in a list of recipe lines defining a method for remaking the named targets. Additional recipe lines may follow the first line of the rule. Each such recipe line must begin with a tab character. For more about recipes, see “Recipes” on page 129.

As an example of a simple rule, consider the following:

```
main.o : include.h
```

This rule contains a single target, **main.o**, and a single prerequisite, **include.h**. The rule states that if **include.h** changes, **main.o** will require remaking. A typical makefile does not specify a recipe for making **main.o** from **main.c**; instead, the default rules provide the recipe using a metarule or suffix rule. These rules are discussed in “Using Inference Rules” on page 143.

When **make** parses rules, it treats the targets and prerequisites as tokens separated by white space (one or more blank or tab characters). In addition, **make** treats the rule operator (*ruleop*) as a token, but does not require white space around it.

Makefiles can contain special rules that control the behavior of **make** instead of stating a dependency between targets and prerequisites. For more information about such rules, see “Special Target Directives” on page 135.

Rule Operators

The *rule operator* in a rule separates the targets from the prerequisites. Rule operators also let you modify the way in which **make** handles the making of the associated targets. **make** recognizes the following rule operators:

: Separates targets and prerequisites. The same target may have many **:** rules stating different prerequisites for the target, but only one such rule can specify a recipe for making the target, except with metarules. Within metarules, you can specify more than one recipe for making the target. If the target has more than one associated metarule, **make** uses the first metarule that matches.

:: If no prerequisites are specified, the target is always remade. Otherwise, this indicates that this rule may not be the only rule with a recipe for the target. There may be other **::** rules that specify a different set of prerequisites, with different recipes for updating the target. **make** builds any such target if any of the rules find the target out of date with any related prerequisites. **make** then uses the corresponding recipe to perform the update. You can find an example later in this section.

:! Tells **make** to execute the recipe for the associated targets once in turn for each recently changed prerequisite. Ordinarily, **make** executes the recipe only once for all recently changed prerequisites at the same time.

:^ Tells **make** to insert the specified prerequisites before any other prerequisites already associated with the specified targets.

:- Forces **make** to clear the previous list of prerequisites before adding the new prerequisites. Thus, you can replace:

```
.SOURCE
.SOURCE: dir1 dir2
```

with the following:

```
.SOURCE :- dir1 dir2
```

However, the old form still works as expected. See “Special Target Directives” on page 135.

:| Used only in metarules, tells **make** to treat each metadependency as an independent metarule; for example:

```
%.o :| archive/%.c rcs/%.c /srcarc/RCS/%.c
recipe...
```

is equivalent to:

```
%.o : archive/%.c
recipe...
%.o : rcs/%.c
recipe...
%.o : /srcarc/rcs/%.c
recipe...
```

You will find this operator particularly useful for searching for **rcs** file archives. If the **RCSPATH** variable used by **rcs** contains the following value:

```
archive/%f;rcs/%f;/srcarc/rcs/%f
```

then the metarule:

```
% :| $(RCSPATH:s/%f/%/:s/;/ /)
co -l $<
```

searches the path looking for an **rcs** file and checks it. See “Pattern Substitution” on page 131 for an explanation of macro expansion.

It is meaningless to specify **:!**, **:–**, or **:^** with an empty list of prerequisites (although this is not considered an error).

The following example shows how **::** works. Suppose a makefile contains:

```
a.o :: a.c b.h
# first recipe for making a.o

a.o :: a.y b.h
# second recipe for making a.o
```

If **make** finds **a.o** out of date with respect to **a.c**, it uses the first recipe to make **a.o**. If **a.o** is found out of date with respect to **a.y**, **make** uses the second recipe. If **make** finds **a.o** out of date with respect to **b.h**, it calls both recipes to make **a.o**. In the last case, the order of invocation matches the order of the rule definitions in the makefile.

Remember that you should use the **::** operator if a target has more than one associated recipe, unless you form metarules. For more information on metarules, see “Metarules” on page 144.

The following example is an error:

```
joe : fred ... ; recipe
joe : more ... ; recipe  #error
```

Recipes

The recipe consists of a list (possibly empty) of lines defining the actions **make** carries out to update a target. **make** defines recipe lines as arbitrary strings that may contain macro expansions. These follow a target-prerequisite line, and you can space them apart by comment or blank lines. You terminate a recipe by a new target description, a macro definition, or end of file.

Each recipe line *must* begin with a tab character. Optionally, you can place **–**, **@**, **+** (or any combination) directly after the tab.

- **–** instructs **make** to ignore nonzero exit values when it executes this recipe line; otherwise, **make** stops processing after an error.
- **@** instructs **make** *not* to echo the recipe line to the standard output prior to its execution; otherwise, **make** prints each line as it executes the line.
- **+** instructs **make** to always execute the recipe line, even when you have specified the **-n**, **-q**, or **-t** options.

See "Special Target Directives" on page 135 for other ways to obtain this behavior.

make also accepts *group recipes*. A group recipe begins with an opening bracket ([) in the first non-white-space position of a line, and ends with a closing bracket (]) in the first non-white-space position of a line. In this format, recipe lines do not require a leading tab character.

make passes group recipes, as a single unit, to a command interpreter for execution whenever the corresponding target requires updating. If the [that starts the group immediately precedes one or more of **–**, **+**, or **@**, they apply to the entire group in the same way that **–**, **+**, and **@** apply to single recipe lines.

As noted earlier, rules can have *;recipe* on the same line as the target definition line. If additional lines with a leading tab character follow the rule definition, *;recipe* is used as the first recipe line, and the additional lines follow it. Otherwise, the text after the **;** is used as the entire recipe. If the semicolon is present but the rest of the recipe line is empty, **make** interprets this as an empty recipe.

Missing Recipes

If **make** cannot find a recipe for a particular target, it normally displays a message on the standard error stream, in the form:

```
Don't know how to make target
```

make does *not* generate this message if a rule has an explicitly empty recipe.

Macros

A *macro* fulfills a function similar to a programming language's variable: You can assign a value to a macro, and then use this value in subsequent operations by referring to the macro. You can define **make** macros within the makefile or on the command line, or by importing them from the environment. For instructions on importing environment variables as macros, see "Special Target Directives" on page 135.

On the command line and inside a makefile, you have three ways to create a macro. **make** recognizes the first form (most other versions of **make** do as well):

```
macro = string
```

This example gives the value of *string* to *macro*.

The other two forms are not found in traditional implementations of **make**:

```
macro := string
```

expands *string* (including any macros it contains) and then assigns the expanded string to *macro*.

```
macro += string
```

changes the current value of *macro* by adding a single space and then the value of *string*. In this case, **make** does *not* expand *string*.

When **make** defines a macro other than definitions read from the environment, it strips any leading and trailing white space from the macro value. White space consists of any combination of blanks or tabs.

After you have defined a macro, you can use it in any makefile line. Whenever **make** finds one of the following constructs in a makefile:

```
$(macro)
${macro}
```

it replaces *macro* with its associated, predefined string. Thus, **$(TEST)** causes an expansion of the macro variable named **TEST**. If you have defined **TEST**, **make** expands any reference to **$(TEST)** to your associated string. If you haven't defined **TEST** at that time, **$(TEST)** expands to the **NULL** string (a string containing no characters). This is equivalent to the following macro definition:

```
TEST=
```

If the name of a macro consists of a single character, you can omit the parentheses or braces. Thus, **$X** is equivalent to **$(X)**.

make processes macro definitions on the command line last; they will override definitions for macros of the same name found within the makefile. Therefore, definitions found inside the makefile cannot redefine macros defined on the command line.

Modified Macro Expansions

make supports several new macro expansion expressions, of the form:

```
$(macro_name:modifier_list:modifier_list:...)
```

Each *modifier_list* consists of one or more characters that tell **make** to extract only part of the string associated with the given macro. A list of characters and their meanings follows:

```
b or B – File portion of all pathnames, without suffix
d or D – Directory portion of all pathnames
f or F – File portion of all pathnames, including suffix
s or S – Simple pattern substitution (see "Pattern Substitution" on page 131)
t or T – Tokenization (see "Tokenization" on page 131)
u or U – All characters in the expansion are mapped into uppercase
l or L – All characters in the expansion are mapped into lowercase
^    – token prefixing (see "Prefix and Suffix Operations" on page 131)
+ – token suffixing (see "Prefix and Suffix Operations" on page 131)
```

You can use either uppercase or lowercase for modifier letters. Suppose, for example, you define a macro with:

```
test = D1/D2/d3/a.out f.out d1/k.out
```

Then the following macro expansions take on the values shown.

```
$(test:d)           → D1/D2/d3 . d1
$(test:b)           → a f k
$(test:F)           → a.out f.out k.out
${test:DB}          → D1/D2/d3/a f d1/k
${test:s/out/in/}   → D1/D2/D3/a.in f.in d1/k.in
$(test:t"+")        → D1/D2/D3/a.out+f.out+d1/k.out
$(test:u)           → D1/D2/D3/A.OUT F.OUT D1/K.OUT
$(test:l)           → d1/d2/d3/a.out f.out d1/k.out
$(test:^"/rd/")     → /rd/D1/D2/d3/a.out /rd/f.out /rd/d1/k.out
$(test:+".Z")       → D1/D2/d3/a.out.Z f.out.Z d1/k.out.Z
```

The **:d** modifier gives a **.** for names that do not have explicit directories.

Pattern Substitution

You use the substitution modifier to substitute strings in a macro definition:

```
:s/pattern/replace/
```

You can use any printing character in place of the **/** character to delimit the pattern and replacement text, as long as you use it consistently within the command.

For compatibility with UNIX System V, **make** also supports the suffix replacement modifier:

```
$(name:oldsuffix=newsuffix)
```

This expands **$(name)** normally, and then replaces any occurrences of the suffix *oldsuffix* with *newsuffix*. **make** replaces the **o** string only when it appears in the position of a suffix:

```
LIST = apple.o orange.o object.o
$(LIST:o=c) → apple.c orange.c object.c
```

Tokenization

The tokenization modifier:

```
:t"string"
```

expands the macro value into tokens (strings of characters separated by white space) separated by the quoted *string* that follows the **t** modifier. **make** does not append the separator string to the last token. The following list shows the special escape sequences that may appear in the separator string and their meanings:

```
\" → "
\\ → \
\a → alert (bel)
\b → backspace
\f → formfeed
\n → newline
\r → carriage return
\t → horizontal tab
\v → vertical tab
\ooo → EBCDIC character octalooo>
```

Thus, using the previous definition of **$test**, the following expansion occurs:

```
$(test:f:t"+\n") expands to a.out+
        ;                   f.out+
                            k.out
```

Prefix and Suffix Operations

You use prefix and suffix modifiers:

```
:^"prefix"
:+"suffix"
```

to add a prefix or suffix to each space separated token in the expanded macro.

For example, suppose you specify the following macro definition:

```
test = main func1 func2
```

Then the following expansions occur:

```
$(test:^"/src/")expands to /src/main /src/func1 /src/func2
$(test:+".c") expands to main.c func1.c func2.c
```

You can combine these two macro references:

```
$(test:^"/src/":+".c")
```

expands to:

```
/src/main.c /src/func1.c/src/func2.c
```

If the prefix and suffix strings themselves consist of a list of tokens separated by blanks, the resulting expansion is the cross-product of both lists.

For example, if you specify the following definition of **test**:

```
test = a b c
```

Then the following expansions occur:

```
$(test:^"1 2 3") expands to 1a 1b 1c 2a 2b 2c 3a 3b 3c
$(test:+"1 2 3") expands to a1 b1 c1 a2 b2 c2 a3 b3 c3
```

You can combine these two references:

```
$(test:^"1 2 3":+"1 2 3")

expands to              1a1 1b1 1c1 2a1 2b1 2c1 3a1 3b1 3c1
                        1a2 1b2 1c2 2a2 2b2 2c2 3a2 3b2 3c2
                        1a3 1b3 1c3 2a3 2b3 2c3 3a3 3b3 3c3
```

Nested Macros

make also allows the values of macros to control the expansion of other macros. You can include such nested macros in the following ways:

```
$(string)
```

or

```
${string}
```

where *string* contains additional **$(...)** or **${...}** macro expansions. Consider the following example:

```
$(CFLAGS$(_HOST)$(_COMPILER))
```

make first expands **$(_HOST)** and **$(_COMPILER)** to get results and then uses those results as the name of the macro to expand. This is useful when you write a makefile for more than one target environment. Suppose you import **$(_HOST)** and **$(_COMPILER)** from the environment and they represent the host machine type and the host compiler, respectively. If the makefile contains the following macro definition, **CFLAGS** takes on a value that corresponds to the environment in which **make** is being called:

```
CFLAGS_VAX_CC = -c -O
  # for _HOST == "_VAX", _COMPILER == "_CC"
CFLAGS_PC_MSC = -c -ML
  # for _HOST == "_PC",  _COMPILER == "_MSC"
CFLAGS := $(CFLAGS$(_HOST)$(_COMPILER))
```

Text Diversion

With text diversion you can directly create files from within a recipe. This feature is an extension to traditional **make** systems and probably absent from other implementations.

In a recipe, you can use a construct of the form:

```
<+ text +>
```

where the given *text* can stand for anything; several lines long if desired, each beginning with a tab, as must all recipes. When **make** encounters this construct, it creates a temporary file with a unique name, and copies the given *text* to that file. Then **make** executes the recipe with the name of the temporary file inserted in place of the diversion. When **make** finishes processing, it removes all the temporary files. (You can use the **-v** option to have **make** show the names of these temporary files, and leave them around to be examined.)

make places temporary files in the **/tmp** directory unless the **TMPDIR** environment variable is set.

make expands macro references inside the text in the normal way, so that the file contains the text with all macro references replaced by the associated strings. Newline characters are copied as they appear in the diversion.

Normally, **make** does *not* copy white space at the beginning of each line of the *text* into the temporary file, unless you put a backslash at the front of a white space character, in which case the white space from that point on is copied into the temporary file:

```
<+
    This line does not begin with white space
\   This one does.
+>
```

As a simple example of text diversion, suppose that the **CC** macro currently contains **c89** (the **c89** compiler interface). If **make** encounters the recipe line:

```
copy <+ Using $(CC) as compiler
     +> hifile
```

it creates a temporary file containing:

```
Using c89 as compiler
```

Since **make** strips white space from the beginning of the second line, the contents of the temporary file end at the newline character at the end of the first line.

The temporary file that the text diversion process creates has a unique name. Suppose that the name is **temp**. **make** changes the original recipe line to:

```
copy temp hifile
```

with the result that the line:

```
Using c89 as compiler
```

is copied into **hifile**.

Consider a more realistic example of how you can use this feature:

```
OBJECTS=program$O module1$O module2$O
program: $(OBJECTS)
        link @<+ $(OBJECTS:t"+\n")
                 $@/noignorecase
 $(NULL)
 $(LDLIBS)
             +>
```

The tokenizing expression:

```
$(OBJECTS:t"+\n")
```

adds a **+** and a newline after each token in the **OBJECTS** macro. The runtime macro **$@** stands for the name of the target being made (as explained in “Special Macros” on page 138). As a result, the temporary file created by the text diversion contains:

```
program.o+
module1.o+
module2.o
program/noignorecase
```

which is the sort of input file that the **link** command can handle. The recipe therefore consists of the following command:

```
link @tempfile
```

tempfile stands for the name of the temporary file holding the text diversion.

Creating a text diversion in this way is complicated, but it may be the only way to handle some situations.

Using Attributes to Control Updates

make defines several target attributes. You can assign attributes to a single target, a group of targets, or to all targets in the makefile. Attributes affect what **make** does when it needs to update a target. **make** recognizes the following attributes:

.EPILOG
Inserts shell epilog code when executing a group recipe associated with any target having this attribute set. (See also .PROLOG).

.IGNORE
Ignores any errors encountered when trying to make a target with this attribute set.

.LIBRARY
Indicates that *target* is a library. If **make** finds a target of the form *lib(member)* or *lib((entry))*, **make** automatically gives the .LIBRARY attribute to the target named *lib*. For further information, see “Libraries” on page 119.

.PRECIOUS
Tells **make** not to remove this target under any circumstances. Any automatically inferred prerequisite inherits this attribute. For an explanation of why this is provided, see the discussion of .REMOVE in “Special Target Directives” on page 135.

.PROLOG
Inserts shell prolog code when executing a group recipe associated with any target having this attribute set.

.SETDIR
Changes the working directory to a specified directory when making associated targets. The syntax of this attribute is:

```
.SETDIR=path
```

where *path* represents the pathname of the desired working directory.

.SILENT
Does not echo the recipe lines when making any target with this attribute set, and does not issue any warnings.

You can set any of the previous attributes. **make** recognizes two more attributes which you cannot set: the .LIBRARYM and .SYMBOL attributes.

.LIBRARYM
Indicates that *target* is a library member. You cannot explicitly set this attribute; **make** automatically gives it to targets or prerequisites of the form *lib(entry)*; that is, *lib* sets the .LIBRARY attribute, and *entry* gets the .LIBRARYM attribute.

.SYMBOL
Indicates that *target* is the library member with a given entry point. You cannot explicitly set this attribute; **make** automatically gives it to targets or prerequisites of the form *lib((entry))*.

You can use attributes in several ways:

```
targets attribute_list :
prerequisites attribute_list : targets
```

Both of these examples assign the attributes specified by *attribute_list* to each of the *targets*.

A line of the form:

```
attribute_list :
```

(with no *targets*) applies the list of attributes to all targets in the makefile. Traditional versions of **make** may let you do this with the .IGNORE attribute, but not with any others attributes.

You can use any attribute with any target (including special targets). Some combinations are useless (for example, .INCLUDE .PRECIOUS: ...). Other combinations are quite useful:

```
.INCLUDE .IGNORE : "startup.mk"
```

This example tells **make** not to complain if it cannot find **startup.mk** using the include file search rules. If you do not use a specified attribute with the special target, **make** issues a warning and ignores the attribute.

Special Target Directives

Special targets are called *targets* because they appear in the target position of rules; however, they really function as keywords, not targets; and the rules in which they appear serve as *directives*, which control the behavior of **make**.

The special target must be the only target in a special target rule; you cannot list other normal or special targets.

Some attributes do not affect special targets. You can give any attribute to any special target, but often the combination is meaningless and the attribute has no effect.

.BRACEEXPAND
Cannot have prerequisites or recipes associated with it. If set, the .BRACEEXPAND special target allows use of the brace expansion feature from previous versions of **make**. If you have old makefiles that use the now-outdated brace expansion feature, you can use this special target to continue using them without modification. For more information about brace expansion, see *z/OS UNIX System Services Command Reference*.

.DEFAULT
Takes no prerequisites, but does have a recipe associated with it. If **make** cannot find a mechanism to build a target, it uses the recipe from the .DEFAULT rule. If your makefile contains:

```
.DEFAULT:
echo no other rule found
echo so doing default rule for $<
```

and no other rule for **file.c**, then:

```
make file.c
```

displays:

```
no other rule found
so doing default rule for file.c
```

.ERROR
If defined, prompts the execution of the recipe associated with this target whenever **make** detects an error condition. You can use any attribute with this target. **make** brings any prerequisites of this target up to date during its processing.

Note: make ignores any errors while making this target.

.EXPORT
Prompts **make** to determine which prerequisites associated with this target correspond to macro names. **make** exports these to the environment, with the values they hold, at the point in the makefile at which **make** reads this rule. **make** ignores any attributes specified with this target. Although **make** exports the value specified to the environment at the point at which it reads the rule, no actual execution of commands takes place until the entire makefile is read. Only the final exported value of a given variable affects executed commands.

.GROUPEPILOG
Prompts **make** to add the recipe associated with this target after any group recipe for a target that has the .EPILOG attribute. See "Executing Regular Recipes" on page 118 for further information.

.GROUPPROLOG
Puts the recipe associated with this target in before any group recipe for a target that has the .PROLOG attribute. See "Executing Regular Recipes" on page 118 for further information.

.IMPORT
Prompts **make** to search for the associated prerequisite names in the environment. **make** defines the names it finds as macros with the value of the macro taken from the environment. If it cannot find a name, it issues an

error message; however, if you specify the .IGNORE attribute, **make** does not generate an error message and does not change the macro value.

If you give the prerequisite .EVERYTHING to .IMPORT, **make** reads in the entire environment. (Requiring this special prerequisite instead of an empty string helps to avoid accidentally importing the entire environment by expanding a null macro as the prerequisite of .IMPORT.)

Note: Normally **make** imports the entire environment unless suppressed by the **-E** option.

.INCLUDE

Tells **make** to process one or more additional makefiles, as if their contents had been inserted at the line where **make** found the .INCLUDE in the current makefile. You specify the makefiles to be read as the prerequisites for .INCLUDE. If the list contains more than one makefile, **make** reads them in order from left to right.

make uses the following search rules when trying to find the makefile:

- If a relative filename is enclosed in quotes (") or is not enclosed, **make** begins its search in the working directory. If the file is not found, **make** then searches for it in each directory specified by the .INCLUDEDIRS special target.
- If a relative filename is enclosed with **<** and **>**, (as in <file>), **make** searches only in the directories specified by the .INCLUDEDIRS special target.
- If an absolute (fully qualified) filename is given, **make** looks for that file, and ignores the .INCLUDEDIRS list.

If **make** cannot find a file, it normally issues an error message and ends; however, if the .IGNORE attribute is specified, **make** just ignores missing files. The .IGNORE attribute is the only attribute that can be specified with .INCLUDE.

For compatibility with **make** on UNIX System V:

```
include file
```

at the beginning of a line has the same meaning as:

```
.INCLUDE: file
```

.INCLUDEDIRS

Contains a list of specified prerequisites that define the set of directories to search when trying to include a makefile.

.MAKEFILES

Contains a list of prerequisites that name a set of files to try to read as the user makefile. **make** processes these files in the order specified (from left to right) until it finds one up to date. The built-in rules specify:

```
.MAKEFILES : makefile Makefile
```

.POSIX

Causes **make** to process the makefile as specified in the POSIX.2 standard. This special target must appear before the first noncomment line in the makefile. This target may have no prerequisites and no recipes associated with it. The .POSIX target does the following:

- It causes **make** to use the shell when running all recipe lines (one per shell).

- It disables any brace expansion (set with the `.BRACEEXPAND` special target).
- It disables metarule inferencing.
- It disables conditionals.
- It disables **make**'s use of dynamic prerequisites.
- It disables **make**'s use of group recipes.
- **make** will *not* check for the string `$(MAKE)` when run with the **-n** option specified.

.REMOVE
Causes **make** to remove intermediate targets. In the course of making some targets, **make** may create new files as intermediate targets. For example, if **make** creates an executable file, it may have to create some object files if they don't currently exist. **make** tries to remove any such intermediate targets that did not exist initially. It does this by using the recipe associated with the .REMOVE special target. The startup file set up an appropriate **rm** command to serve as a default for .REMOVE. If you want to avoid this automatic removal for certain targets, give those targets the .PRECIOUS attribute. (.PRECIOUS is especially useful for marking libraries, since you usually want them to remain.)

.SOURCE
Contains a prerequisite list that defines a set of directories to check when trying to locate a target filename. For more information, see "Binding Targets" on page 142.

.SOURCE.x
Is similar to .SOURCE, except that **make** searches the .SOURCE.x list first when trying to locate a file with a name ending in the suffix **.x**.

.SUFFIXES
Contains a prerequisite list of this target, which defines a set of suffixes to use when trying to infer a prerequisite for making a target. There is no need to declare suffixes. If the .SUFFIXES rule has no prerequisites, the list of suffixes is cleared, and **make** does not use suffix rules when inferring targets.

Special Macros

make defines two classes of special macros: control macros and runtime macros.

The *control macros* control **make**'s behavior. If you have several ways of doing the same thing, using the control macros is preferable. A control macro having the same function as a special target or attribute also has the same name.

make defines the *runtime macros* when making targets, and they are usually useful only within recipes. The exceptions to this are the dynamic prerequisite macros, discussed later in this chapter.

Control Macros

There are two groups of control macros:
- String-valued macros
- Attribute macros

make automatically creates internally defined macros. You can use these macros with the usual **$**(*name*) construct. For example, you can use **$(PWD)** to obtain the working directory name.

String-Valued Macros

DIRSEPSTR
Is defined internally. It gives the characters that you can use to separate components in a pathname. This is usually just `/`. If **make** finds it necessary to make a pathname, it uses the first character of **DIRSEPSTR** to separate pathname components.

GROUPFLAGS
Is set by the startup file and can be changed by you. This macro contains the set of flags to pass to the command interpreter when **make** calls it to execute a group recipe. See the discussion of **MFLAGS** for more about switch characters.

GROUPSHELL
Is set by the startup file and can be changed by you. It defines the full path to the executable image used as the shell (command interpreter) when processing group recipes. This macro must be defined if you use group recipes. It is assigned the default value in the standard startup file.

GROUPSUFFIX
Is set by the startup file and can be changed by you. If defined, this macro gives the string used as a suffix when **make** creates group recipe files to be handed to the command interpreter. For example, if it is defined as **.sh**, all group recipe files created by **make** end in the suffix **.sh**.

INCDEPTH
Is defined internally. It gives the current depth of makefile inclusion. This macro contains a string of digits. In your original makefile, this value is `0`. If you include another makefile, the value of **INCDEPTH** is `1` while **make** processes the included makefile, and goes back to `0` when **make** returns to the original makefile.

MAKE Is set by the startup file and can be changed by you. The standard startup file defines it as:

```
$(MAKECMD) $(MFLAGS)
```

make itself does not use the **MAKE** macro, but it recognizes the string **$(MAKE)** when using the **-n** option for single-line recipes.

MAKECMD
Is defined internally. It gives the name you used to call **make**.

MAKEDIR
Is defined internally. It contains the full path to the directory from which you called **make**.

MAKEFLAGS
Contains all the flags specified in the **MAKEFLAGS** environment variable plus all the flags specified on the command line, with the following exceptions. It is an error to specify **\-c**, **-f**, or **\-p** in the environment variable, and any specified on the command line do not appear in the **MAKEFLAGS** macro. Flags in the **MAKEFLAGS** environment variable can optionally have leading dashes and spaces separating the flags. **make** strips these out when the **MAKEFLAGS** macro is constructed.

MAKESTARTUP
May be set by you, but only on the command line or in the environment. This macro defines the full path to the startup file. The built-in rules assign a default value to this macro.

MFLAGS
Is defined internally. It gives the list of flags given to **make** including a leading dash. That is, **$(MFLAGS)** is the same as **–$(MAKEFLAGS)**.

NULL Is defined internally. It is permanently defined to be the **NULL** string. This is useful when comparing a conditional expression to a **NULL** value and in constructing metarules without ambiguity. See "Metarules" on page 109 for more information.

OS Is defined internally. It contains the name of the operating system you are running.

PWD Is defined internally. It represents the full path to the working directory in which **make** runs.

SHELL
Is set by the default rules and can be changed by you. It defines the full path to the executable image used as the shell (command interpreter) when processing single-line recipes. This macro must be defined if you use recipes that require execution by a shell. The default rules assign a default value to this macro by inspecting the value of the **SHELL** environment variable.

Note: The startup file must explicitly import the **SHELL** environment variable. The default importation of the environment does not apply to **SHELL**.

SHELLFLAGS
Is set by the startup file and can be changed by you. This macro specifies the list of options (flags) to pass to the shell when calling it to execute a single-line recipe. The flags listed in the macro do not possess a leading dash.

SHELLMETAS
Is set by the startup file and can be changed by you. This macro defines a list of characters that you want **make** to search for in a single recipe line. If **make** finds any of these characters in the recipe line, **make** uses the shell to call the recipe; otherwise, **make** calls the recipe without using the shell.

Attribute Macros

The attribute macros let you turn global attributes on or off. You use the macros by assigning them a value. If the value does not contain a **NULL** string, **make** sets the attribute *on* and gives all targets the associated attribute. If the macro *does* contain a **NULL** string, **make** sets the attribute *off*.

The following macros correspond to attributes of the same name:

```
.EPILOG
.IGNORE
.PRECIOUS
.PROLOG
.SILENT
```

See "Using Attributes to Control Updates" on page 134 for more information.

Runtime Macros

Runtime macros receive values as **make** is making targets. They take on different values for each target. These are the recognized runtime macros:

$@	Evaluates to the full name of the target, when building a normal target. When building a library, it expands to the name of the archive library. For example, if the target is **mylib(member)**, **$@** expands to **mylib**.
$%	Also evaluates to the full name of the target, when building a normal target. When building a library, it expands to the name of the archive member. In the previous example, **$%** expands to **member**.
$&	Evaluates to the list of all prerequisites, in all rules that apply to the target.
$?	Evaluates to the list of all prerequisites that are newer than the target. In inference rules, however, this macro evaluates to the same value as the **$^** macro.
$>	Evaluates to the name of the library if the current target is a library member. For example, if the target is **mylib(member)**, **$>** expands to **mylib**.
$^	Evaluates to the list of prerequisites given in the rule that contains the recipe **make** is executing.
$<	In normal rules, it evaluates the same as **$?**. In inference rules it evaluates to the single prerequisite that causes the execution of the rule.
$*	Is equivalent to **$(%:db)**. This expands to the target name with no suffix.
$$	Expands to **$**.

The following example illustrates the difference between these:

```
a.o : a.c
a.o : b.h c.h
      recipe for making a.o
```

Assume **a.c** and **c.h** are newer than **a.o**, whereas **b.h** is not. When **make** executes the recipe for **a.o**, the macros expand to the following values:

```
$@  →  a.o
$*  →  a
$&  →  a.c b.h c.h
$?  →  a.c c.h
$^  →  b.h c.h
$<  →  b.h c.h
```

Consider this example of a library target:

```
mylib(mem1.o):
recipe...
```

For this target, the internal macros then expand to:

```
$@  →  mylib
$*  →  mem1
$>  →  mylib
```

Dynamic Prerequisites

You can use the symbols **$$@**, **$$%**, **$$***, and **$$>** to create dynamic prerequisites (that is, prerequisites calculated at the time that **make** tries to update a target). Only these runtime macros yield meaningful results outside of recipe lines.

When **make** finds **$$@** in the prerequisite list, the macro expands to the target name. If you are building a library, it expands to the name of the *archive library*. With the line:

```
fred : $$@.c
```

make expands **$$@** when making **fred**, so the target name **fred** replaces the macro.

You can modify the value of **$$@** with any of the macro modifiers. For example, in:

```
a.c : $$(@:b).c
```

the **$$(@:b)** expands to **a**.

You can apply modifiers to special runtime macros and to the dynamic prerequisite symbol. For example, consider:

```
all: file1 file2

file1 file2: $$@.c
    $(CC) $(CFLAGS) -o $@ $@.c
```

make evaluates these statements as:

```
all: file1 file2

file1: $$@.c
    $(CC) $(CFLAGS) -o file1 file1.c

file2: $$@.c
    $(CC) $(CFLAGS) -o file2 file2.c
```

When **make** finds **$$%** in the prerequisite list, it also stands for the name of the target, but when building a library, it stands for the name of the *archive member*.

When **make** finds **$$*** in the prerequisite list, it stands for the name of the target, but without the *suffix*.

You can use the **$$>** macro in the prerequisite list only if you are building a library. In this case, it stands for the name of the *archive library*. Otherwise, its use is invalid.

For more information on dynamic prerequisites and their use, see *z/OS UNIX System Services Command Reference*.

Binding Targets

Makefiles often specify target names in the shortest manner possible, relative to the directory that contains the target files. **make** possesses relatively sophisticated techniques of searching for the file that corresponds to a target name in a makefile.

Assume that you try to bind a target with a name of the form **pathname.ext**, where **.ext** is the suffix and **pathname** is the stem portion (that is, that part which contains the directory and the basename). **make** performs all search operations relative to the working directory except when the given name is a full pathname starting at the root of a file system.

1. Look for **pathname.ext** relative to the working directory, and use it if it is found.
2. Otherwise; if the .SOURCE.ext special target is defined, search each directory given in its list of prerequisites for **pathname.ext**. If **.ext** is a **NULL** suffix (that

is, **pathname.ext** is really just **pathname**) use .SOURCE.NULL instead. If it is found, use that file. If it is still not found, try this step again using the directories specified by .SOURCE .

3. If it is still not found, and the target has the library member attribute (.LIBRARYM) set, try to find the target in the library of which the target is a member (see "Libraries" on page 119).

   Note: This same set of rules is used to bind a file to the library target at an earlier stage of the makefile processing.

4. If still not found, the search fails. **make** returns the original name **pathname.ext**.

If at any point the search succeeds, **make** replaces the name **X.a** of the target with the new bound name and then refers to it by that name internally.

There is potential here for a lot of search operations. The trick is to define .SOURCE.x special targets with short search lists and leave .SOURCE undefined, or as short as possible. Initially, **make** simply defines .SOURCE as:

```
.SOURCE : .NULL
```

In this context, .NULL tells **make** to search the working directory by default.

The search algorithm has the following useful side effect. When **make** searches for a target that has the .LIBRARYM (library member) attribute, **make** first searches for the target as an ordinary file. When a number of library members require updating, it is desirable to compile all of them first and to update the library at the end in a single operation. If one of the members does not compile and **make** stops, you can fix the error and run **make** again. **make** does not remake any of the targets with object files that have already been generated as long as none of their prerequisite files have been modified.

If a target has the .SYMBOL attribute set (see "Libraries" on page 119), **make** begins its search for the target in the library. If **make** finds the target, it searches for the member using the search rules. Thus, **make** first binds library entry point specifications to a member file, and then checks that member file to see if it is out of date.

When defining .SOURCE or .SOURCE.x targets, the construct:

```
.SOURCE :
.SOURCE : fred gerry
```

is equivalent to:

```
.SOURCE :- fred gerry
```

More generally, the processing of the .SOURCE special targets is identical to the processing of the .SUFFIXES special targets.

Using Inference Rules

Specifying recipes for each and every target becomes tedious and error-prone. For this reason, **make** provides a number of mechanisms allowing you to specify generic rules for a particular type of target. These mechanisms are called *inference rules*. There are two major types: suffix rules and metarules.

Suffix rules are a historical mechanism that matches the suffix of a target against a list of special suffixes and rules to find a recipe to use. For more information, see "Suffix Rules" on page 110.

The second mechanism is called *metarules*. These *pattern rules* are a more recent invention provided by a number of modern versions of **make**. They are much more flexible and general than the older suffix rules. You should use the metarules rather than the suffix rules. **make** provides the suffix rules primarily for compatibility reasons. A final way to specify a recipe to a target that doesn't have any other rule is through the .DEFAULT special target. See "Special Target Directives" on page 135.

Here is the search order for the various mechanisms:
1. Search explicit rules in the makefile.
2. Check to see if an appropriate metarule exists.
3. Check to see if an appropriate suffix rule exists.
4. Check to see if the .DEFAULT target was defined; otherwise, display an error and stop.

Metarules

A metarule states, in general, that targets with names of a particular form depend on prerequisites with names of a related form. The most common example is that targets with a name ending in **.o** depend on prerequisites with the same basename, but with the suffix **.c**. The process of deriving a specific rule from a metarule is called *making an inference*.

Consider this example, which explains the general metarule format:

```
%.o : %.c
$(CC) -c $(CFLAGS) $<
```

This rule states that any target file that has the suffix **.o**, and doesn't have an explicit rule, depends on a prerequisite with the suffix **.c** and the same basename. For example, **file.o** depends on **file.c**. The recipe that follows the command tells how to compile the **.c** file to get a corresponding **.o** file.

As another example, consider the following metarule:

```
%.c .PRECIOUS : RCS/%.c,v
    -co -q $<
```

Anyone who uses the public-domain application **rcs** to manage C source files will find this useful. The metarule says that any target with the suffix **.c** depends on a prerequisite that has the same filename, but is found in the subdirectory **RCS** under the same directory that contains the target. For example, **dir/file.c** is checked out of **dir/RCS/file.c,v**. The recipe line uses the special **$<** macro to stand for the prerequisite (in the **RCS** directory).

The general metarule format is:

```
pre%suf :
prerequisite prerequisite...     recipes
```

where *pre* and *suf* are arbitrary (possibly empty) strings. If the **%** character appears in the prerequisite list, it stands for whatever the **%** matched in the target.

Here is an inference rule that omits both the *suf* and *pre* strings:

```
% .PRECIOUS: RCS/%,v
    -co -q $<
```

This rule matches any target and tries to check it out from the **rcs** archive.

A number of technical considerations dictate the order in which **make** tries to make inferences. If several metarules can apply to the same target, there is no way to control the one that **make** actually uses. You can use the **-v** and **-n** options to find out what **make** chooses. A well-designed set of metarules yields only one rule for a particular target.

A metarule may specify attributes for a target. If **make** attempts to make a target that has a particular attribute, it first checks for a metarule that applies to the target and specifies the given attribute. If no such metarule exists, **make** looks for a metarule that does not specify the attribute. This lets you specify different metarules for targets with different attributes. **make** performs this test for all attributes except .SILENT, .IGNORE, and .SYMBOL.

Suffix Rules

Suffix rules are an older form of inference rule. They have the form:

```
.suf1.suf2:
recipe...
```

make matches the suffixes against the suffixes of targets with no explicit rules. Unfortunately, they don't work quite the way you would expect.

The rule:

```
.c.o :
recipe...
```

says that **.o** files depend on **.c** files. Compare this with the usual rules:

```
file.o : file.c
compile file.c to get file.o
```

and you will see that suffix rule syntax is backward! This, by itself, gives good reason to avoid suffix rules.

You can also specify single-suffix rules similar to the following, which match files ending in **.c**:

```
.c:
recipe...
```

For a suffix rule to work, the component suffixes must appear in the prerequisite list of the **.SUFFIXES** special target. The way to turn off suffix rules is simply to place:

```
.SUFFIXES:
```

in your makefile with no prerequisites. This clears the prerequisites of the **.SUFFIXES** targets and prevents any suffix rules from firing. The order in which suffixes appear in the **.SUFFIXES** rule determines the order in which **make** checks the suffix rules.

Here is the search algorithm for suffix rules:

1. Extract the suffix from the target.
2. If it does not appear in the **.SUFFIXES** list, quit the search.

3. If it is in the **.SUFFIXES** list, look for a double suffix rule that matches the target suffix.
4. If you find one; extract the basename of the file, add on the second suffix, and see if the resulting file exists. If it doesn't, keep searching the double suffix rules. If it does exist, use the recipe for this rule.
5. If no successful match is made, the inference has failed.
6. If the target did not have a suffix, check the single suffix rules in the order that the suffixes are specified in the **.SUFFIXES** target.
7. For each single suffix rule, add the suffix to the target name and see if the resulting filename exists.
8. If the file exists, execute the recipe associated with that suffix rule. If the file doesn't exist, continue trying the rest of the single suffix rules. If no successful match is made, the inference has failed.

Try some experiments with the **-v** option specified to see how this works.

There is a ″special″ feature in the suffix rule mechanism that wasn't described earlier. It is for archive library handling. If you specify a suffix rule of the form:

```
.suf.a:
recipe
```

the rule matches any target having the **LIBRARYM** attribute set, regardless of the target's actual suffix.

For example, suppose your makefile contains the rules, and **mem.o** exists:

```
.SUFFIXES: .a .o
.o.a:
echo adding $< to library $@
```

Then, the following command:

```
make "mylib(mem.o)"
```

causes **make** to print the following line:

```
adding mem.o to library mylib
```

Refer to "Libraries" on page 119 for more information about libraries and the .LIBRARY and .LIBRARYM attributes.

Compatibility Considerations

make attempts to remain compatible with versions of **make** found on UNIX and POSIX-conforming systems, while meeting the needs of differing environments. This section examines ways in which **make** may differ from traditional versions.

Conditionals

Conditionals let you selectively include or exclude parts of a makefile. This lets you write rules that have different formats for different systems.

Note: Traditional implementations of **make** do not recognize conditionals. They are extensions to the POSIX standard.

A conditional has the following format:

```
.IF expression
input1
.ELSIF expression
input2
.ELSE
input3
.END
```

The *expression* has one of the following forms:

```
text
text == text
text != text
```

The value of the first form is *true* if the given text is not null; otherwise, it is *false*. The value of the second form is *true* if the two pieces of text are equal, and the value of the last form is *true* if the two pieces of text are not equal.

When **make** encounters a conditional construct, it begins by evaluating the *expression* after the .IF. If the value of the expression is *true,* **make** processes the first piece of input (*input1*) and ignores the second; if the value is *false,* **make** processes the second (*input2*) and ignores the first. Otherwise, it processes the third input.

The .IF , .ELSE , .ELSIF, and .END keywords must begin in the first column of an input line (no preceding white space).

You may be used to indenting material inside **if-else** constructs; however, you should *not* use tabs to indent text inside conditionals (except, of course, for recipe lines, which are always indented with tabs). The text inside the conditional should have the same form that you would use outside the conditional.

You can omit the .ELSE part of a conditional.

BSD UNIX make

The following is a list of the notable differences between z/OS UNIX **make** and the 4.2 or 4.3 BSD UNIX version of **make**.

- BSD UNIX **make** supports wildcard filename expansion for prerequisite names. Thus, if a directory contains **a.h**, **b.h**, and **c.h**, BSD UNIX **make** performs the following expansion:

  ```
  target: *.h expands to target: a.h b.h c.h
  ```

  z/OS UNIX **make** does not support this type of filename expansion.
- Unlike BSD UNIX **make**, *touching* library members causes **make** to search the library for the member name and to update the time stamp if the member is found.
- z/OS UNIX **make** does not support the BSD **VPATH** variable. A similar and more powerful facility is provided through the .SOURCE special target.

System V AUGMAKE

The following special features have been implemented to make **make** more compatible with System V AUGMAKE:

- You can use the word **include** at the start of a line instead of the .INCLUDE: construct that is normally understood by **make**.

- **make** supports the macro modifier expression **$(macro:str=sub)** for suffix changes.
- When defining special targets for the suffix rules, the special target **.X** is equivalent to **.X.NULL**.

Improving make Performance

For possible improvement in the performance of make, set the shell variable **_MAKE_BI=YES**. When this shell variable is set, sh will invoke the built-in make, built-in c89, built-in c++, and built-in cc instead of the /bin commands. Using built-in make may be especially beneficial when making large applications since the sh does not need to start another process but instead calls the built-in make. As with other shell variables, this may be set in **/etc/profile, $HOME/.profile** or on command line by a user. For more information about built-in commands, see *z/OS UNIX System Services Command Reference* under sh utility (Built-in Commands section).

Chapter 6. Debugging z/OS C/C++ Programs

This chapter describes the z/OS UNIX services available to you for debugging z/OS C/C++ application programs as you develop them.

This chapter describes:

- How to control processes for an application and interrupt applications
- The formal debugging services available for z/OS C/C++ applications, including the z/OS UNIX **dbx** utility
- Debugging from the z/OS UNIX shells with **dbx**

The chapter concludes with a list of **dbx** subcommands.

The z/OS UNIX System Services web page also has information about **dbx**. Go to:

`http://www9.s390.ibm.com/products/oe/dbx/`

If you're interested in porting applications to z/OS UNIX, see the online z/OS UNIX System Services Porting Guide at:

`http://www.s390.ibm.com/products/oe/bpxa1por.html`

If you are viewing this book using IBM BookManager BookServer, you can click on either url above and be automatically linked to its associated page.

Controlling Processes

This section discusses how you find out information about processes associated with z/OS UNIX C/MVS application programs so that you can end them if the application program develops problems.

Obtaining the Status of z/OS UNIX Application Program Processes

When the z/OS UNIX C/MVS application program you are developing runs, you can check the processes assigned to it to determine where it is running and how much processor time it is using as it runs.

In order for you to be able to check the status of your application program processes, the program must run in a shell or create processes by requesting z/OS UNIX C/MVS services. You can check the status of processes for executable files running in the following environments:

- Shell foreground
- Shell background
- TSO/E foreground, when processes are started
- MVS batch, when processes are started

Note: If the application program is submitted for MVS batch processing using the BPXBATCH program, a JCL job stream is used to invoke a batch program, which executes a shell environment from which the application program executable file is run.

You must also enter a shell **ps** or **jobs** command to determine the process IDs for the application if you do not already have them recorded.

To check the status of your z/OS UNIX C/MVS application program processes, follow these procedures:

1. Record the process IDs when the application is started.

2. Issue the process status or job status command when you want to check on the shell-initiated application:

   ```
   ps
   ```

   The preceding command displays the status of only those processes associated with the default user ID. The following command, when entered by a user with superuser authority, displays the status of all active processes. Otherwise, it displays the status of all processes associated with the default user ID.

   ```
   ps -A
   ```

   The following command displays the status of all active jobs in the shell—including process ID and user ID information:

   ```
   jobs -l
   ```

   Note: If the application was started from TSO/E for MVS batch submission and you go into the shell (using the TSO/E OMVS command to create a shell environment or the PA1 key from the TSO escape prompt to return to an existing shell environment), you can use the **ps** command to determine the status of the processes associated with your user ID. If you stay in the TSO/E environment, you can use TSO/E commands to check the MVS batch job queues.
3. From the resulting status display, check the status of the process and any of its child processes.
4. Record the necessary process IDs for the application program's job.

If you are running the application program executable file from a shell and it appears to be hung, and you cannot enter shell commands, you can query the process status and IDs either by using the **open** subcommand or by creating another shell session. When the second shell session is started, enter:

```
ps -ef
```

and record the process ID for the application. You can then kill the C/MVS application program process. For information on killing processes, see Killing a Runaway Process. To return to the first shell session, enter: `exit`.

The output from the **ps** command is displayed to **stdout**, which is normally the terminal. You can redirect output from the command using the > character and a specified HFS filename.

Killing a Runaway Process

When the z/OS UNIX C/MVS application program you are developing and testing runs unchecked, the processes assigned to it can be out of the normal control of the application user. These processes need to be ended to free address spaces and associated storage. You may need to *kill* an application program's active processes to do this.

Before you can kill an application program's processes, you must know the process IDs to send a kill signal to. See "Obtaining the Status of z/OS UNIX Application Program Processes" on page 149 for information on how to do this.

You can, if you want, code the C/MVS source such that the application program checks the status of its processes and raises a flag if it detects and intercepts a program signal that a process is not behaving as anticipated. The application program can then kill the flagged process and take the appropriate processing action. You need to use the z/OS UNIX C/MVS signal functions to do this. See *z/OS C/C++ Run-Time Library Reference*, for the descriptions of the signal-handling functions. For information on z/OS UNIX C/MVS considerations for signal delivery, see *z/OS C/C++ Programming Guide*. For more information on POSIX signal handling, see *The POSIX.1 Standard: A Programmer's Guide,* by Fred Zlotnick (Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991). This book contains a thorough discussion of POSIX.1-defined signal handling.

Follow these procedures to kill a process:

- Interactively, you can do one of three things to end a runaway process:
  - Enter the "break" keystroke sequence (<Ctrl-C>) from the terminal.
  - Issue the **kill** command specifying the kill signal and the process IDs to end the processes and free memory. The following example shows two ways to kill the same z/OS shell processes:
    - **kill -S KILL 2819 15163**
    - **kill -9 2819 15163**
  - If the shell prompt is not available, invoke the shell again. Then enter the **kill** command for the correct process ID from a user ID that has the appropriate privilege to kill the process.
- From within an application program, you can:
  1. Intercept a signal indicating that something has gone wrong with a process.
  2. Call the z/OS UNIX C/MVS **getpid()** or **getppid()** function to get the process ID in question.
  3. Call the **kill(***pid,sig)* function to pass the signal on to the process identified. The process responds to the signal according to how the process has been coded to handle signals.
  4. Call the **sigaction()** function if the action indicated by the **kill** command is to be changed.

  Note: There is no special method for killing a process under the TSO/E environment if the z/OS UNIX application program is started from that environment.

After you kill a process, control is returned to the parent process or the application continues on with conditional processing.

Introduction to the z/OS Debugger

You need to create an z/OS UNIX C/MVS application program that will compile, link-edit, and run successfully. After your program has been developed, you can take advantage of the z/OS UNIX debugger (with its **dbx** utility) to debug the program from within the shell environment on an MVS system.

Using **dbx**, you can debug your program at the source level and at the machine level. Source-level debugging allows you to:

- Set breakpoints at selected statements with conditions for activation
- Hold and release thread execution
- Run a program one line at a time
- Access variables symbolically and display them in the correct format
- Examine the source text using simple search functions or the **ed** editor
- Debug processes that contain **fork()** and **exec()** functions
- Interrupt and examine a program that is already in progress

- Trace execution of a program by line, routine, or variable
- Display expressions using a wide range of operators
- Print a list of the active routines and their parameters (stack traceback)
- Print declarations of variables, along with their fully qualified names
- Modify the directory list from which to search for source files
- Determine the application programs loaded into a process
- Debug applications involving threads
- Display information about thread, condition variable, and mutex objects
- Debug applications involving DLLs
- Refer to, display, and modify program variables that contain doublebyte character set (DBCS) characters
- View MVS dumps of C/C++ programs

Machine-level debugging allows you to:
- Set breakpoints at selected machine instructions with conditions for activation
- Hold and release thread execution
- Run a program one instruction at a time
- Display or modify the contents of machine registers and memory
- Debug processes that contain **fork()** and **exec()** functions
- Interrupt and examine a program that is already in progress
- Trace execution of a program by instruction
- Display expressions using a wide range of operators
- Determine the application programs loaded into a process
- Debug applications involving threads
- Display information about thread, condition variable, and mutex objects
- View MVS dumps

You can tailor the **dbx** utility to:
- Customize your interface to the **dbx** utility with command aliases
- Customize your debugging environment with an initialization file
- Invoke your choice of an editor (the default editor is **ed**) and use shell commands during the debugging session
- Enter commands from either standard input or a named file
- Reroute standard output and standard error to HFS files
- Set or change predefined variables
- Change the command prompt

so that **dbx** fits your work preferences.

Your application program may or may not strictly conform to the z/OS UNIX-supported POSIX standards. You can debug POSIX-conforming C/MVS applications in three environments available to most z/OS UNIX application programmers:
- A POSIX-conforming workstation environment.
- A shell environment.
- A shell environment started through MVS batch. From an MVS batch environment, use the BPXBATCH program to enter the **dbx** command, passing to it the appropriate subcommands in a command file. **dbx** is run from the shell through MVS batch, and subcommands are entered from a command file to do the appropriate debugging. You need to specify the *decimal* value of process ID (PID) for the running application with the **dbx -a** option. The results of the debugging session can be directed to an HFS file. See “Running the dbx Utility”

on page 157 for more information on use of the **-a** option. See *C/C++ User's Guide* for more information on using the BPXBATCH program to run applications from the shell through MVS batch.

If an z/OS UNIX C/MVS application program being developed is POSIX-conforming, it can be compiled and link-edited at a workstation that conforms to the same POSIX standards supported by the system where it will eventually run. A UNIX-related source-level debugging facility can then be used at the workstation to do some debugging of the application.

Prepare your C/MVS application program source for debugging using the z/OS UNIX **c89** utility. For a discussion of how to set the necessary compiler and linkage editor options for debugging, see *z/OS Language Environment Debugging Guide*.

An z/OS UNIX POSIX-conforming C application can be debugged to some extent at a POSIX-conforming workstation. However, final debugging and verification of the application's correctness should be performed from the MVS system. The z/OS UNIX **dbx** utility is the C/MVS source-level debugging solution supported by z/OS UNIX for application programs.

The application source most likely will be created and coded on one or more workstations connected to an MVS system. Assuming that the application is POSIX-conforming and the workstations used to develop it run POSIX-conforming operating systems, you can do more than just edit your application source at a workstation.

From a workstation operating system that conforms to z/OS UNIX-supported standards, you can:
- Create and code application source files.
- Create and store the correct POSIX and user-defined include file libraries.
- Store user or application data.

  Note: To create and test an z/OS UNIX C/MVS application program on such a workstation using user or application data, you must first understand all the ASCII-EBCDIC data conversion considerations and plan for them.
- Use a compiler.
- Test the application modules.
- Use a UNIX-style source-level debugger (for example, the **dbx** debugger).

You can then move the application source to an MVS system to be compiled, link-edited, and tested before putting it into a production environment.

Using the z/OS UNIX Debugger to Debug Your Application

You debug your z/OS UNIX C/MVS application on an MVS system using the **dbx** utility running in the shell environment.

1. Run the z/OS UNIX debugger's **dbx** utility, specifying the name of the executable file to be debugged and any of the options desired.

   For example, to start source-level debugging for a file named **payroll** which is in your working directory, and search the **/u/user/src** directory for the source files, specify:

   ```
   dbx -I /u/user/src payroll
   ```
2. When prompted by **dbx**, supply the subcommands you plan to use to debug the program source.

Note: You can use **dbx** to debug an z/OS UNIX C/MVS application program that you are running under the MVS batch environment through use of the BPXBATCH program.

For more information on how to use the **dbx** utility from the shell, see “Debugging from the Shell with the dbx Utility” on page 157. For a list of the **dbx** subcommands, see page 157.

For more information on the **dbx** command, see *z/OS UNIX System Services File System Interface Reference*.

Using the z/OS UNIX Debugger with Multithreaded Applications

The **dbx** debugger allows you to debug multithreaded applications at the source level or at the machine level. **dbx** supports three objects related to multithreaded applications: threads, mutexes, and condition variables.

In a multithreaded process, **dbx** will give control to the user in the context of a single thread called the *current thread*. Many commands operate in the context of the current thread—for example, commands that display information about variables within threads. To use **dbx** commands on a given thread, you must first make it the current thread.

When **dbx** gives control to the user, all threads must be stopped. All other threads in the process are stopped while the debugger is working on a particular thread. The debugger assigns a thread variable—for example, `$t1`—to each thread in the process. This variable is like a temporary name, and is easier to use when referring to the thread than the hexadecimal thread ID. When tracing program execution, the debugger displays this variable for the thread that causes each breakpoint.

Note: The debugger must obtain information about threads, mutexes, and condition variables from LE/370. If you do not want the debugger to maintain this information, use the **nodebug** option when you compile your application program. The **nodebug** option tells LE/370 not to inform **dbx** about condition variables and mutex objects. It nonetheless maintains almost all information about multiple threads.

z/OS UNIX Debugger Restrictions and Debugging Limitations

Although the z/OS UNIX debugger is provided for your application development use in a POSIX-conforming environment, it cannot differentiate between POSIX-conforming code and non-POSIX-conforming code. As long as your application program meets the requirements for using the z/OS UNIX debugger and is not dependent on any listed restrictions, you may be able to debug it using the debugger. The feature is limited, however, in its ability to step through application programs that use application program interfaces (APIs) not provided through a POSIX-conforming C programming language.

Application programs containing CICS statements *cannot* be debugged with the z/OS UNIX debugger.

IBM provides the debugger to debug z/OS UNIX C/MVS application programs. Other MVS debuggers, such as INSPECT, cannot be used.

The Debugger has the following restrictions

- **dbx** can be run only from one of the z/OS UNIX shells. It cannot be used from the TSO/E environment.

- Core file debugging (dump analysis) is not supported.

The debugger provides fully supported source-level debugging of z/OS UNIX C/MVS application programs if you ensure that:

- The application program was compiled using the **c89 -g** option.
- The executable file resides in the hierarchical file system (HFS), or resides in an MVS partitioned data set linked to by a file in the HFS that has the sticky bit on
- The source files reside in the HFS or as MVS partitioned data sets (PDS) members. If the source is in a PDS, the PDS must have organization VB. **dbx** does support source in FB PDS's.
- The application program was compiled as reentrant but link-edited as a nonreentrant z/OS UNIX application executable file. For more information on preparing your z/OS UNIX C/MVS application program for debugging, see *z/OS Language Environment Debugging Guide*.
- The application program is loaded into the user read/write storage.
- The application program does not load additional executable files into its address space. The **dbx** utility cannot support **fetch()** function calls.

  Note: To avoid dynamic loading by an application program, link-edit all parts of the application program. Doing so makes it easier to get full debugging support. **dbx** does support source level debugging of programs that contain DLLs. Source symbolic information for a DLL is processed by **dbx** after a DLL is loaded, before code is executed in the DLL.
- The application program to be debugged runs in an z/OS UNIX POSIX *process space*. A process space is an MVS address space that was created either through a **fork()** function or through a request for z/OS UNIX services from within a non-POSIX application program.
- In a multiprocessing environment, specified with the **dbx multproc on** subcommand setting, you must enter **dbx -A** once for each new child process to be debugged. This requires a separate shell session for each process to be debugged. For more information on how to debug a multiprocess application program, see "Debugging Programs Involving Multiple Processes" on page 168.
- You must code the application to report information about condition variable, mutex objects, and thread object events from LE/370. If your application is not coded in one of the following ways, neither condition variable nor mutex information nor stack size information for threads will be available. Also, trace output from threads ($tv events), mutexes ($mv events), and condition variables ($cv events) will not be available. Other thread information will be available, however. To enable your program for thread debugging, you must **either:**
  - Add the following line at the top of the C program:
    ```
    #pragma runopts(TEST(ALL))
    ```
    Or:
  - Code an assembler program, CEEUOPT, to invoke the CEEXOPT macro, which specifies **TEST(ALL)**. For examples of how to code this program, see *z/OS Language Environment Debugging Guide*.
- This setup is also needed for **dbx** to do proper source level stepping when the **$LE_hookstep** is set and for proper DLL processing when the **$LE_dlls** is set. Note that both these flags are set by default.

Note: If not all the conditions described are met, you may still be able to use the **dbx** utility at a reduced level of effectiveness. The limitations of debugging for each case are outlined below.

Debugging limitations of z/OS UNIX dbx

- If the source that contains the **main()** function is compiled without the C/MVS compiler TEST(ALL) and GONUM options (by specifying **c89 -s** or not specifying **c89 -g**), **dbx** cannot:
  - Set breakpoints at source statements or function entry points
  - Use symbolic names to display or alter data, storage, and program instructions
  - Support source-level debugging
- If the application program is running without the POSIX(ON) runtime option, **dbx** cannot:
  - Use POSIX synchronous signaling
  - Support process check intercepts
- If the source files reside somewhere outside the hierarchical file system, **dbx** can debug them if you compiled and link-edited them into the file system using **c89** and identified the files as MVS PDS members with the **use //** subcommand.

  Note: The MVS data sets must be variable block (VB) data sets. Fixed block (FB) data sets are not supported by **dbx**.
- If the executable file is loaded into a read-only subpool, **dbx** cannot:
  - Set breakpoints
  - Support instruction stepping
- If the application program dynamically loads other program modules, **dbx** cannot:
  - Access the modules loaded by the application program
  - Determine entry points
  - Support source-level debugging with symbolic access to storage for such modules
  - Set breakpoints
- Do not use the **step** subcommand to leave a signal catcher that was entered as the result of the delivery of a signal. **dbx** may not give you control again where you expect it and may not give you control until the program exits.

Restrictions on dbx for C++

Using the OS/390 V2R4 compiler and LE prelinker removes most of the restrictions that existed previously. The following is an updated list of restrictions on **dbx** for C++ that reflect compiler symbolic limitations as of OS/390 Version 2 Release 4:

- The OS/390 V2R4 version of dbx cannot source level debug C++ programs created with -g by the 3.2 or lower compiler versions. If you need to source level debug these types of programs, you should keep an older version of **dbx**. The better option is to recompile the older C++ programs using the new compiler/prelinker, which allows the new **dbx** to function properly.
- Type-checking for reference types fails because the information is not available in the symbolics.
- The definition for anonymous class class/struct/union/enum field items appears both in the normal location inside the { }, but also outside in the enclosing { }.
- Unreferenced member functions appear in a class as **function_name:1** instead of the correct **return_type function_name(parms)** definition.
- Symbolics for **inline, virtual, pure** are not available, so **dbx** cannot report them.
- You cannot step/next into an inline function. However, you can set a stop in an inline function and use run or continue to get it. This includes explicit and implicit inlines. Implicit inlines are member functions that completely reside inside a class and are not just prototyped.

- When the **$LE-hookstep** flag is set and the **LE RUNOPTS(TEST(ALL))** flag is set, **dbx** can now step to any source line where the source file has been compiled with **—g.**

Debugging from the Shell with the dbx Utility

This section provides information on how to do the following tasks:
- Running the **dbx** utility
  - Using the **dbx** utility (examples)
  - Running shell commands from **dbx**
- Controlling program execution
  - Setting and deleting breakpoints to step through a program
  - Running a program
  - Consulting a stopped program
  - Tracing execution
- Displaying and manipulating the source file
  - Changing the source directory path
  - Displaying the current file
  - Changing the current file or procedure
  - Editing source files while debugging a program
- Debugging application programs involving multiple processes
- Examining program data
  - Handling signals
  - Displaying a stack traceback
  - Displaying and modifying variables
  - Scoping of names
  - Understanding operators and modifiers allowed in expressions
  - Understanding type checking in expressions
  - Converting variables to lowercase and uppercase
  - Changing print output with special debugging variables
- Debugging application programs involving threads
  - Examining multithread program status
  - Controlling multithread program execution
- Debugging at the machine level
  - Using machine registers
  - Examining memory addresses
  - Running a program at the machine level
- Source level debug of DLLs
- Customizing the **dbx** debugging environment
  - Defining a new **dbx** prompt
  - Creating subcommand aliases
  - Using the **.dbxsetup** file
  - Using the **.dbxinit** file
  - Reading **dbx** subcommands from a file

Running the dbx Utility

There are several common ways to start a debug session using **dbx** options. By default, **dbx** prepares the named program for execution by forking a new process and loading the program into that child process. **dbx** then prompts you to enter debugger subcommands. You can then begin setting breakpoints, single-stepping instructions, displaying variables, and other debugging aids. If no executable program is specified when **dbx** is entered, the default executable file **a.out.** is assumed.

There are options for **dbx** you can specify that allow you to modify the default behavior of the **dbx** utility:

dbx -r

Use the **-r** option if you want to enter the **dbx** utility only when your program ends abnormally or is interrupted by a signal. If the executable program ends successfully, **dbx** exits. Otherwise, the reason for termination or interruption is reported and you enter the **dbx** utility.

Note: The **dbx** utility will know about a condition variable or mutex object only if **dbx** is active when LE/370 creates the object for the application program. Therefore, any mutex or condition variable activity will be lost up to the point where **dbx** starts debugging your program. Also, **dbx** cannot display information about the state of mutexes or condition variables that were created before the you entered the debugger.

Syntax: dbx -r *[options] [executable [CommandArguments]]*

dbx -a

Use the **-a** option if you want to debug a process that is already in progress. To use this option, you must have permission to kill the process that has the specified process ID.

dbx connects to the process if access is allowed, determines the full pathname of the executable file, reads in the symbolic information, and then prompts for commands. Interaction at that point is no different from if you had invoked **dbx** to begin execution of the process.

Note: You can specify the **dbx -a** option and supply the process ID for an application executable file being run from the shell through the MVS batch environment using the BPXBATCH program. You must make sure that the process ID you pass to **dbx** is in decimal format.

Syntax: dbx *[options]* -a *processid*

Note: The **dbx** utility will know about a condition variable or mutex object only if **dbx** is active when LE/370 creates the object for the application program. Therefore, any mutex or condition variable activity will be lost up to the point where **dbx** starts debugging your program. Also, **dbx** cannot display information about the state of mutexes or condition variables that were created before you entered **dbx** with the **-a** option.

dbx -A

Use the **-A** option if you want to debug a process created by a program being debugged when the **multproc** subcommand is set to `on`. Under these circumstances, **dbx** enters a message telling you to enter `dbx -A` *`processid`* to begin debugging the new process. To use this option, you must have permission to kill the process that has the specified process ID.

dbx interrupts the process if access is allowed, determines the full pathname of the executable file, reads the symbolic information, and then prompts for commands. Interaction at that point is no different from if you had invoked **dbx** to begin execution of the process. Use the UNIX command **ps** to determine the process ID of the program to be debugged.

Syntax: dbx *[options]* -A *processid*

dbx -I

Use the **-I** option to add the specified directories to the list of directories that are searched when **dbx** looks for a source file. **dbx** normally looks for source files in the working directory and in the directory where the executable file is located. If your source is in **/u/user/src** and **/u/group/src**, and the executable file is in **/u/user/bin**, you should specify the **-I** option so that **dbx** can find the source automatically.

Syntax: dbx -I *directory1*-I *directory2...* [*options*] [*executable*]

For example, to add two directories to the list of directories to be searched for the source file of an executable file **objfile**, enter:

```
dbx -I /u/user/src -I /u/group/src objfile
```

or:

```
dbx -I /u/user/src:/u/group/src objfile
```

You can specify the **use** subcommand to do directory searches for the source file after you have invoked **dbx**. The **use** subcommand differs from the **dbx -I** option in that it resets the list of directories to be searched, whereas **-I** adds a directory to the list.

If your C/MVS source files are MVS data sets, you can specify the **dbx use** subcommand to search for an MVS data set for source-level debugging. You indicate that the source is in an MVS data set with a double-slash (//) prefix. For example, to search for the source file of an executable file **objfile**, enter:

```
dbx objfile
Entering debugger ...
dbx for MVS
Type 'help' for help.
reading symbolic information  ...

(dbx) use //

(dbx)
```

dbx -c

Use the **-c** option to run the list of **dbx** commands in the specified file before accepting commands from standard input.

Note: You can use the **source** subcommand for this purpose after **dbx** is invoked.

Syntax: dbx -c *file [options] [executable]*

dbx -C

Use the **-C** option to put **dbx** in dump processing mode. In this mode, dbx can operate on an MVS dump as if it were a running program. The exception is any operations that involve modifying or executing the program is not supported in dump processing mode.

Syntax: dbx -C *dump -name[options]*

dbx supports specification of program arguments on the **dbx** command line and use of the **cont** subcommand when the **dbx** command prompt is first displayed. This eliminates the need for the **kill/fork/exec** overhead that is required for the first run.

As an example, if you specified the following:

```
$ dbx myprogram
(dbx) run a b c
```

You can specify:

```
$ dbx myprogram a b c
(dbx) cont
```

For a complete discussion of the **dbx** command's syntax, options, and subcommands, see *z/OS UNIX System Services Command Reference*.

Examples of Using the dbx Utility

To examine the state of the process in memory for a sample program **/mylog/appl/execut/samp**:

```
main() {
   int *p1;
   p1 = 0;
   *p1 = 888; }
```

that is the object from a compile using the **c89 -g** option, enter:

```
c89 -g -o samp samp.c
dbx -r samp
```

dbx runs the program until it reaches an abnormal termination condition and then prompts you for a debugging subcommand:

```
Entering debugger ...
dbx for MVS
Type 'help' for help.
reading symbolic information  ...

segmentation violation in main at line 4
    4   *p1 = 888; }
(dbx) quit
```

As another example, consider that the following program, **looper.c**, can never end because the value of **i** is never incremented:

```
main()
{
        int i,x[64];

        for (i = 0; i < 10;)
          printf(x[i]);
}
```

Compile the program with the **c89 -g** option to produce an executable program with symbolic debugging capability:

```
c89 -g -o looper looper.c
```

Then run the program from the command line:

```
looper &
```

Seeing that your program does not end as expected, you want to debug it while it continues to run. To attach **dbx** to a process that was just started in the background, type the following:

```
dbx -a $!
```

or perform the following steps:

1. Determine the ID number associated with the process that is running to attach to **looper**. You must open another shell session if you did not run **looper** as a background process. From the second shell session you just established, enter the following shell command:

```
ps -u userid -o pid,tty,time,comm
```

where *userid* is your TSO/E user ID that is running **looper**. All active processes that belong to that user ID are displayed:

```
      PID TT              TIME COMMAND
   655362 ?           00:00:10
   458755 ttyp0000    00:00:00 /bin/sh
   524292 ttyp0000    00:00:02 looper
        6 ttyp0000    00:00:00 /bin/ps
```

The process ID associated with **looper** is 524292.

2. To attach **dbx** to **looper,** enter:

```
dbx -a 524292
```

dbx attaches to the process running the program, displays the last instruction processed, and prompts you for a debugging subcommand:

```
Waiting to attach to process 524292 ...
Determining program name ...
Successfully attached to /tmp/looper...
dbx for MVS
Type 'help' for help.
reading symbolic information  ...
6       printf(x[i]);
(dbx)
```

You can now query and debug the process as if it had been originally started with **dbx**. When you are finished debugging the process and have ended **dbx**, enter the shell **exit** command to end the second shell session and return to your initial session.

Running Shell Commands from dbx

You can run shell commands without exiting from the **dbx** utility by using the **sh** subcommand.

If **sh** is entered without any commands specified, the shell is entered for use until it is exited, at which time control returns to **dbx**. The **SHELL** environment variable determines which shell is used. For example:

```
(dbx) sh echo $SHELL
/bin/sh
(dbx) sh
$ echo 'This is the shell.' #You will remain in the shell until you exit.
This is the shell.
$ exit
(dbx)
```

Controlling Program Execution

The **dbx** utility allows you to set breakpoints (stopping places) in the program. After entering **dbx**, you can specify which lines or addresses are to be breakpoints, and then run the program with **dbx**. When the program reaches a breakpoint, it halts and reports that it has reached a breakpoint. You can then use **dbx** subcommands to examine the state of your program.

Setting and Deleting Breakpoints to Step through a Program

An alternative to setting breakpoints is to run your program one line or instruction at a time, a procedure known as single-stepping. This section discusses how to set and delete breakpoints, begin program execution, and control program execution.

Use the **stop** subcommand to set breakpoints in **dbx**. There are four variations of the **stop** subcommand for programs compiled with the debug flag (the **c89 -g** option):

stop at *linenumber [***if** *Condition]*
: Stops the program at a specified source line number. The *linenumber* parameter consists of an optional filename and a **:**; (colon), followed by a line number. For example, “hello.c”:23 and 23 are valid *linenumber* parameters. The optional **if** *condition* flag specifies that execution should be halted at the specified line number if the condition is true when the line number is reached. Line numbers are relative to the beginning of the source file. A condition is an expression that evaluates to true or false. For example:

```
(dbx) stop at "zinfo.c":57
```

stop in *procedure [***if** *condition]*
: Stops the program at the first executable statement in a procedure or function. For example:

```
(dbx) stop in main
```

stop *variable [***in** *procedure* ³ **at** *linenumber] [***if** *condition]*
: Stops the program when the value of *variable* changes.

For example:

```
(dbx) stop x
```

stop if *condition*
: Stops the program whenever *condition* evaluates to true.

For example:

```
(dbx) stop if (x > y) and (x < 20000)
```

Note: See *z/OS UNIX System Services Command Reference* for more information on the **stop** subcommand.

After any of the preceding subcommands, **dbx** responds with a message reporting the event it has built as a result of your command. The message includes the event ID associated with your breakpoint along with an interpretation of your command. The syntax of the interpretation might not be exactly the same as your command. The following are examples:

```
(dbx) stop in main
[1] stop in main
(dbx) stop at 19 if x == 3
[2] if x = 3 { stop } at "hello.c":19
```

The numbers in the brackets are the event identifiers associated with the breakpoints. When the program is halted as a result of one of the events, the event identifier is displayed along with the current line to show what event caused the program to stop. The events you create exist with internal events created by **dbx**, so event identifiers might not always be sequential.

Use the **status** command to display all current events. You can redirect output from **status** to an HFS file. Each event is displayed in the same form as it was when stored.

The **clear** and **delete** commands remove breakpoints. The **clear** command deletes breakpoints by line number. The **delete** command eliminates events by event identifier. Use **delete all** to remove all breakpoints and trace events.

The following examples show how to display the active events and remove them:

```
(dbx) status

[1] stop in main
[2] if x = 3 { stop } at "hello.c":19
(dbx) delete 1
(dbx) status

[2] if x = 3 { stop } at "hello.c":19
(dbx) clear 19
(dbx) status
(dbx)
```

Running a Program

The **run** subcommand starts your program.

Note: See *z/OS UNIX System Services Command Reference* for more information on the **run** subcommand and its format.

The **run** subcommand tells **dbx** to begin running an executable file and passes arguments just as if they were typed on the shell command line. Input can be redirected from a file and output redirected to a file.

Note: The **rerun** subcommand has the same form as **run**; the difference between them is that for **rerun** if no arguments are passed, the argument list from the previous execution is used.

After your program begins, it continues until one of the following occurs:

- The program reaches a breakpoint.
- A signal occurs that is not ignored, such as **INTERRUPT** or **QUIT**.
- A multiprocess event occurs while multiprocess debugging is enabled.
- The program completes.
- A DLL has been loaded.

In each case, the **dbx** utility receives control and displays a message explaining why the program stopped.

Continuing a Stopped Program

There are several ways, using **dbx** subcommands, to continue the program after it is stopped:

- **cont** *[signalname ³ signalnumber]* continues the program from the current stopping point until either the program finishes, another breakpoint is reached, or a signal is received which is trapped by **dbx**. If a signal is specified, the process continues as though it received the signal. If a signal is not specified and the **dbx** debugging program variable *$sigblock* is not set and a signal caused the program being debugged to stop, using **cont** causes the program to continue as if it had received the original signal. If a signal is not specified and the *$sigblock* variable is set and a signal caused the program being debugged to stop, the program resumes running.

  Signals can be specified by name, number, or name without the **SIG** prefix. Signal names can be either lowercase or uppercase. The following **cont** subcommands are equivalent:

  ```
  cont SIGQUIT
  cont 24
  cont quit
  ```

- **step** *[nNumber]* runs one or a specified *nNumber* of source lines.

  next *[number]* runs up to the next source line, or runs a specified *number* of source lines.

A common method of debugging is to step through your program one line at a time. The **step** and **next** subcommands serve that purpose. The distinction between these two commands is apparent only when the next source line to be run involves a call to a subprogram. In this case, the **step** subcommand stops in the subprogram; in contrast, the **next** subcommand runs until the subprogram has finished and then stops at the next instruction after the call.

With the **step** subcommand, the **dbx** utility stops after each machine instruction to see if the program has reached any line numbers. With the **next** subcommand, the **dbx** utility sets an internal breakpoint at the address associated with the next line number and runs until that breakpoint is reached. For that reason, **next** runs much more quickly than **step**, and the difference is most noticeable when **step** is run from a line that calls a subroutine that has not been compiled with the debug flag. Also, **next** is faster than **step** if a line calls a subroutine loaded into processor read-only storage, such as a link pack area (LPA). The **dbx** utility cannot store breakpoints in read-only storage, so **dbx** sets a breakpoint at every line in the current function. Those subroutines do not have line numbers, and the **dbx** utility may have to run and stop at thousands of machine instructions until it reaches a point that corresponds to a line number. You should use **next** on any line that contains any subroutine and especially on those that have not been compiled with the **c89 -g** option or have been loaded into read-only storage.

There is no event number associated with these stops, because there is no permanent event associated with stopping a program.

- **return** *[procedure]* continues execution until a return to *procedure* is encountered, or until the current procedure returns if *procedure* is not specified.

  If you accidentally step into a subroutine that you do not want to step through, you can use the **return** subcommand to run through the current procedure or to a specified procedure.
- **skip** *[number]* continues execution until the end of the program or until *number*+1 breakpoints are executed.

Tracing Execution

The **trace** subcommand tells **dbx** to print information about the state of the program while the program is running. **trace** can slow your program considerably, depending on how much work **dbx** has to do.

For applications with one or more threads, each thread in the process has a number assigned by **dbx**. The **trace** subcommand displays information about the thread that causes a breakpoint to occur, or the current thread. The current thread number is displayed in parentheses and to the right of the **dbx** breakpoint response as shown in the example in this section. All threads in the process are stopped when **dbx** encounters a breakpoint in the current thread.

trace does not work unless you first enter a **stop in main** subcommand to stop at the program's **main()** function. You can then continue to step through the program and enter **trace** subcommands. The following example shows what happens when you try to run **trace** without stopping at the **main()** function:

```
dbx samp

Entering debugger ...
dbx for MVS
Type 'help' for help.
reading symbolic information ...
```

```
(dbx) trace

Cannot determine where to set a breakpoint.  You must specify
'Procedure', 'SourceLine' or 'Address'.
```

Here's how to run **trace** correctly:

```
(dbx) stop in main
[1] stop in main
(dbx) run
[1] stopped in main at line 6 ($t1)
    6    x = 0;
(dbx) trace
[2] trace
(dbx) cont
trace:    7    y = 1;
trace:    8    }
program exited
(dbx)
```

There are six forms of program tracing:

trace Single-steps the program, printing out each source line that is run. This can be very slow for the same reasons that **step** can be slow.

trace in *procedure [***if** *condition]***;**
: Restricts the printing of source lines to when the specified *procedure* is active. You can specify an optional *condition* to control when trace information should be produced. For example:

```
(dbx) trace in sub2,

[1] trace in sub2
(dbx) run

trace in hellosub.c:  8  printf("%s",s);
trace in hellosub.c:  9  i = '5';
trace in hellosub.c:  10  }
```

trace *procedure [***in** *procedure] [***if** *condition]***;**
: Displays a message each time *procedure* is called or returned from. When *procedure* is called, the information includes passed parameters and the name of the calling routine. On a return, the information includes the return value from *procedure*. The following is an example:

```
(dbx) trace sub

[1] trace sub
(dbx) run

calling sub(s = "hello", a = -1, k = delete) from
function main
returning "hello" from sub
```

trace *linenumber [***if** *condition]***;**
: Prints the specified source line when the program reaches that line.

trace *expression* **at** *linenumber [***if** *condition]*
: Prints the value of *expression* when the program reaches the specified source line. The line number and file are printed, but the source line is not. For example:

```
(dbx) trace x*17 at "hellosub.c":8 if (x > 0)

[1] if x > 0 { trace x*17 } at "hellosub.c":8
```

```
(dbx) **run**

at line 8 in file "hellosub.c": x*17 = 51
```

trace *variable [***in** *procedure] [***if** *condition]*

Prints the location in the program and the value of *variable* each time the memory location associated with *variable* is modified. This is the slowest form of trace. The following is an example:

```
(dbx) trace x

[1] trace x
initially (at line 4 in "hello.c"): x = 0
after line 17 in "hello.c": x = 3
```

Deleting trace events is the same as deleting stop events. When the **trace** subcommand is run, the event ID associated is displayed along with the internal representation of the event.

Displaying and Manipulating the Source File

You can use the **dbx** utility to search through and display portions of the source files for a program. For z/OS UNIX application program debugging with **dbx**, program source files can exist as HFS files or as MVS data sets. For **dbx** to be able to search and display program source in MVS data sets, you must enter the **dbx use** subcommand to specify a search path of double-slash (//). This indicates that the source for the executable file is to be searched for as an MVS data set.

The **dbx** utility keeps track of the current file, current procedure, and current line. The current line and current file are set initially to the line and file containing the source statement where the process ended.

Note: This is true only if the process stopped in a location compiled for debugging.

Changing the Source Directory Path

The **dbx** utility normally searches the working directory and the directory where the program is located for source files for the program. If the source file cannot be found, the debugger runs without displaying source statements.

You can change this with the **-I** option on the **dbx** invocation line, or with the **use** command within **dbx**.

Displaying the Current File

The **list** subcommand allows you to list source lines.

Some special symbols representing *SourcelineExpression* are useful with the **list** command, and with the **stop** and **trace** subcommands. These symbols are $ and @:

$ Represents the next line to be run.

@ Represents the next line to be listed.

You can use simple integer expressions involving addition and subtraction in *SourcelineExpression* expressions.

For example:

```
(dbx) list $

4 {
(dbx) list 5
```

```
5 char i = '4';
(dbx) list sub

23 char *sub(s,a,k)
24 int a;
25 enum status k; ...
```

The **move** subcommand changes the next line number to be listed.

```
(dbx) move 25
(dbx) list @ -2

23 char *sub(s,a,k)
```

Changing the Current File or Procedure

You can use the **func** and **file** subcommands to change the current file, current procedure, and current line within **dbx** without having to run any part of your program.

Note: If your C/MVS application program source is in MVS data sets and you compiled the application program with the **c89 -g** option for debugging, you can specify the **dbx use** subcommand to identify the C/MVS source MVS data set for source-level debugging.

You can search through the current file for text matching regular expressions; **dbx** supports the basic regular expressions described in *z/OS UNIX System Services User's Guide*. If a match is found, the current line is set to the line containing the matching text. The syntax of the search subcommand is:

/ *RegularExpression* **[/]**
It searches forward in the current source file for the given expression.

? *RegularExpression* **[?]**
It searches backward in the current source file for the given expression.

If you repeat the search without arguments, the search wraps around the end of a file.

For example:

```
(dbx) func
sub2
(dbx) file
hellosub.c
(dbx) func sub

(dbx) func
sub
(dbx) file
hello.c
(dbx) / i

5 static int x;
(dbx) /

6 int i = 0xfffffff;
(dbx) ? static

5 static int x;
```

Editing Source Files While Debugging a Program

You can also invoke an external text editor for your source using the **edit** subcommand. The default editor invoked is **ed**. You can override this default by setting the environment variable **EDITOR** to your desired editor before invoking **dbx**.

Control returns to **dbx** after the return from an editing session.

Debugging Programs Involving Multiple Processes

Application programs involving multiple processes are those that call the **fork()** and **exec()** functions. When a program forks, the operating system creates another process that has the same image as the original; the original is called the *parent process*, and the created process is called the *child process*.

When a process calls an **exec()** function, a new program takes over or overlays the original program. Under default circumstances, the **dbx** utility can debug only the original or parent program; however, **dbx** can follow the execution and debug the program.

You must enter the **multproc on** subcommand to enable **dbx** to debug application programs that call an **exec()** function or to debug child processes created through use of the **fork()** function.

When multiprocess debugging is enabled and a fork occurs, both the parent and the child process are halted in the fork. A separate shell session needs to be opened for a new version of **dbx** to be started to control the running of the child process. When the fork occurs, execution is stopped in the parent, and **dbx** displays the state of the program:

```
(dbx) multproc on
(dbx) multproc

multiprocess debugging is enabled
(dbx) run

application forked, child pid=65544, process stopped, awaiting input
use 'dbx -A 65544' on another terminal to establish a debug session
for the child pid

stopped due to fork with multiprocessing enabled in fork at 0x2a89074
0x2a89074 bfffd084     icm    $r15,X'F',X'84'($sp)
(dbx)
```

Another shell session must now be opened to debug the child process. On this second shell session, enter the **dbx -A** option with the process ID for the child:

```
dbx -A 65544

Waiting to attach process 65544 ...

attached in fork at 0x2a89074
0x2a89074 bfffd084     icm    $r15,X'F',X'84'($sp)
(dbx)
```

At this point there are two distinct debugging sessions. The debugging session for the child process retains all the breakpoints from the parent process, but only the parent process can be rerun.

When the program calls an **exec()** function while in multiprocess debugging mode, the program overwrites itself and the original symbol information becomes obsolete.

All breakpoints are deleted when the **exec()** occurs, and the new program is stopped and identified in order for the debugging to be meaningful. **dbx** attaches itself to the new program image, reports the name, and then prompts for input:

```
(dbx) multproc

multiprocess debugging is enabled
(dbx) run

Attaching to program from exec ...
Determining program name ...
Successfully attached to /u/user/execprog ...
Reading symbolic information ...
(dbx)
```

Enter the **map** subcommand to determine the name of the new program being debugged.

When you are finished debugging the new program and end the **dbx** debugging session, enter the **exit** shell command to end the shell session and return to your first shell session.

Using _BPX_PTRACE_ATTACH to Load Programs into User Modifiable Storage

When the environment variable **_BPX_PTRACE_ATTACH** is set to YES, programs invoked via the spawn, exec and attach_exec callable services or via the C language **spawn()** and **exec()** family of functions are loaded into user modifiable storage. Then these target programs can be debugged using **dbx**. The programs that are loaded into storage during the execution of the target program, except for modules loaded from LPA, are also loaded.

For more information on **_BPX_PTRACE_ATTACH**, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

Examining Program Data

This section discusses the following topics:
- Handling signals
- Displaying a stack traceback
- Displaying and modifying variables
- Scoping of names
- Understanding operators and modifiers allowed in expressions
- Understanding type checking in expressions
- Converting variables to lowercase and uppercase
- Changing print output with special debugging variables

Handling Signals

The **dbx** utility can either trap or ignore signals before they are sent to your program. Each time your application program is to receive a signal, **dbx** is notified. If the signal is to be ignored, it is passed to your program; otherwise, **dbx** stops the program and notifies you that a signal has been trapped. All signals are caught by default except the **SIGHUP**, **SIGCHILD**, **SIGALRM**, and **SIGKILL** signals. Change the default handling with the **catch** and **ignore** subcommands.

Note: The **dbx** utility cannot ignore the **SIGTRAP**, **SIGILL**, **SIGCONT**, **SIGDUMP**, or **SIGKILL** signal. The **SIGKILL** signal terminates the process without giving **dbx** a chance to trap it.

In the following example, a program uses **SIGINT** to catch <Ctrl-c> from the keyboard. In order not to stop **dbx** each time one of these signals is received, enter:

```
(dbx) ignore SIGINT
(dbx) ignore

HUP INT KILL ALRM CHLD
```

To make **dbx** stop again when **SIGINT** is received, enter:

```
catch SIGINT
```

If a signal is not specified and the **dbx** debugging program variable *$sigblock* is not set and a signal caused the program being debugged to stop, using **cont** causes the program to continue as if it had received the original signal. If a signal is not specified and the *$sigblock* variable is set and a signal caused the program being debugged to stop, the program resumes running.

Displaying a Stack Traceback

To get a listing of the procedure calls preceding a program halt, use the **where** subcommand.

In the following example, the executable file **hello** consists of two source files and three procedure files, including the standard procedure **main**. The program stopped at a breakpoint in procedure **sub2**.

```
(dbx) run

[1] stopped in sub2 at line 4 in file "hellosub.c"
(dbx) where

sub2(s = "hello", n = 52), line 4 in "hellosub.c"
sub(s = "hello", a = -1, k = delete), line 31 in "hello.c"
main(), line 19 in "hello.c"
```

The stack traceback shows the call in *reverse* order. Starting at the bottom, the following events occurred:

1. The shell called **main()**.
2. The **main()** called procedure **sub** at line 19 with values:
   ```
   s = "hello"
   a =-1
   k = delete.
   ```
3. **sub** called procedure **sub2** at line 31 with values:
   ```
   s = "hello"
   n = 52
   ```
4. The program stopped in procedure **sub2** at line 4.

Note: Set the **dbx** utility variable **$noargs** to turn off the display of arguments passed to procedures.

You can also display portions of the stack with the **up** and **down** subcommands. For example:

```
(dbx) up 0

sub2(s = "hello", n = 54), line 4 in "hellosub.c"
(dbx) up 2

main(), line 19 in "hello.c"
```

```
(dbx) **down**

sub(s = "hello", a = -1, k = delete), line 31 in "hello.c"
```

Displaying and Modifying Variables

To display an expression, use the **print** subcommand. To print the names and values of variables, use the **dump** subcommand. If the given procedure name is **.** (a period), all active variables are printed. To modify the value of a variable, use the **assign** subcommand. If you use **print** *expression*, the expression cannot invoke a function or procedure call.

Note: You cannot assign the value of a literal string to a character pointer. For example, the following is not supported:

```
assign char_ptr="hello world"
```

For example, in a C/MVS program, you have an automatic integer variable *x* with value 7, and you are in the **sub2** procedure with parameters **s** and **n**:

```
(dbx) print x, n

7 52
(dbx) assign x = 3*x
(dbx) print x

21
(dbx) dump

sub2(s = "hello", n = 52)
x = 21
```

Scoping of Names

Names resolve first, using the static scope of the current function. The dynamic scope is used if the name is not defined in the first scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message **using** *QualifiedName* is printed. You can override the name-resolution procedure by qualifying an identifier with a block name (such as *Module.Variable*). Source files are treated as modules named by the filename without the suffix. For example, the *x* variable, which is declared in the **sub** procedure inside the **hello.c** file, has the fully qualified name **hello.sub.x**. The program itself has the name **.** (a period).

Two **dbx** subcommands are helpful in determining which symbol is found when multiple symbols with the same name exist: the **which** subcommand and the **whereis** subcommand.

The following is an example after stopping in the **sub2** procedure:

```
(dbx) which s

hellosub.sub2.s
(dbx) whereis s

hellosub.sub2.s
hello.sub.s
hello.main.s
```

The example shows there are three procedures in the program that have a symbol named **s**.

Understanding Operators and Modifiers Allowed in Expressions

The **dbx** program can display a wide range of expressions.

*** (asterisk)** *or* **ˆ (circumflex)**
: Denotes indirection *or* pointer dereferencing.

[ ] (brackets) *or* **() (parentheses)**
: Denotes subscript array expressions.

. (period)
: Use this field reference operator with pointers and structures. This makes the C programming language operator → (arrow) unnecessary, although it is allowed.

& (ampersand)
: Gets the address of a variable.

.. (two periods)
: Separates the upper and lower bounds when specifying a subsection of an array. For example: **n[1..4]**.

The following types of operations are valid in expressions:

Algebraic
: **=**, **–**, ***,/**(floating division); **div** (integral division); **mod**; **exp** (exponentiation)

Bitwise
: **–**, ^{3}, **bitand**, **xor**, **˜**, **<<**, **>>**

Logical
: **or**, **and**, **not**, $^{3\,3}$, **&&**;

Comparison
: **<**, **>**, **<=**, **>=**, **<>** or **!=**, **=** or **==**

Other **sizeof**

Note: Functions cannot be used in **dbx** expressions.

Logical and comparison expressions are allowed as conditions in **stop** and **trace** subcommands.

Understanding Type Checking in Expressions

Expression types are checked. You can override the type of an expression by using a renaming or casting operator. There are two forms of type renaming:
typename (expression)
expression\ typename

The following is an example where the **x** variable is an integer with value 97:

```
(dbx) print x

97
(dbx) print char (x), x \ char, x

'a' 'a' 97
```

The **whatis** subcommand prints the declaration of an identifier, which can then be qualified with block names as in the previous example.

whatis *name*

The following is an example:

```
(dbx) whatis sub2

int sub2(s,n)
```

```
char *s;
int n;
(dbx) whatis hello.sub.k

enum status k;
```

You can also print the declaration of an enumeration, structure, or union tag. The construct **$$***tagname* is used for that purpose:

```
(dbx) whatis $$status

enum $$status { run, create, delete, suspend };
```

The type of the **assign** subcommand expression must match the variable type it is being assigned. If the types do not match, an error message is displayed. Change the expression type using a type renaming. You can disable type checking by setting a special **dbx** utility **$unsafeassign** variable.

For example, using **n** and **status** as in the previous example:

```
(dbx) assign n = delete

incompatible types
(dbx) assign n = int (delete)
(dbx) print n, $$status (n)

2 delete
(dbx) set $unsafeassign
(dbx) assign n = suspend; print n

3
```

Converting Symbols to Lowercase and Uppercase

By default, **dbx** converts symbols based on the current language. If the current language is C/MVS or undefined, the symbols are not converted and they are interpreted as they actually appear. The current language is undefined if the program is in a section of code that has not been compiled with the debug flag. You can override default handling with the **case** command.

Using **case** without arguments tells you how case is currently being handled.

Changing Print Output with Special Debugging Variables

You can use the **set** subcommand to set the following special **dbx** utility variables to get different results from the **print** subcommand:

$asciichars
    Prints characters interpreting the binary data of the characters as ascii.

$asciistrings
    Prints strings interpreting the binary data of the strings as ascii.

$hexints
    Prints integer expressions in hexadecimal.

$hexchars
    Prints character expressions in hexadecimal.

$hexstrings
    Prints the address of the character string, not the string itself.

$octints
    Prints integer expressions in octal.

$expandunions
    Prints fields within a union.

Use the **unset** subcommand to reset special **dbx** utility variables. Set and reset these variables to get the desired results. For example:

```
(dbx) whatis x; whatis i; whatis s

int x;
char i;
char *s;
(dbx) print x, i, s

375 'c' "hello"
(dbx) set $hexints; set $hexchars
(dbx) print x, i, s

0x177 0x63 "hello"
(dbx) unset $hexchars; set $octints
(dbx) print x, i

0567 'c'
```

The variable **$catchbp** is available for use with the **next** subcommand. Normally, execution of **next** does not honor breakpoints. In order to honor breakpoints with **next**, set the special **dbx** utility variable **$catchbp.**

Debugging Application Programs Involving Threads

z/OS UNIX **dbx** allows you to debug multithreaded applications at the source level or the machine level. It maintains state information about the following three types of multithread application objects:

Threads
: Portable facilities that support concurrent programming allowing an application to perform many actions simultaneously.

Mutexes
: Mutual exclusion locks (mutexes) that allow shared variables to be seen by other threads in a consistent state.

Condition variables
: A synchronization object, used with a mutex, allows a thread to block until some event occurs, and allows for communication among multiple threads.

The **dbx** command updates information about these objects as your program runs. The following **dbx** subcommands are provided so that you can view and modify these multithread objects:

- The **thread** subcommand displays a list of active threads for the application program. You can list all active threads as the default, or you can list specific threads using the **number** parameter. You can also select threads by their states—for example, active or asynchronous. You can control thread execution with the **hold** and **release** options.
- The **mutex** subcommand displays a list of active mutex objects for the application program. You can list all active mutexes as the default, or you can list a specific mutex object using the **number** parameter. You can also select locked or unlocked, and wait or no-wait, mutexes with a series of parameters supplied with the subcommand.
- The **condition** subcommand displays a list of active condition variables for the application program. You can list all active condition variables as the default, or you can list a specific condition variable using the **number** parameter. You can also select wait or no-wait condition variables with parameters supplied with the subcommand.
- The **readwritelock** subcommand displays a list of active read/write lock objects for the application program. You can list all active read/write locks as the default, or you can list a specific read/write lock object using the **number** parameter. You

can also select locked or unlocked, and wait or no-wait, read/write locks with a series of parameters supplied with the subcommand.

For more information about these **dbx** subcommands, see *z/OS UNIX System Services Command Reference*.

Examining Multithread Program Status

The **dbx** command provides three subcommands—**condition**, **mutex**, **readwritelock**, and **thread**—for examining thread-related objects. To check on the status of all threads in your application, enter the **thread** subcommand with no operands. The following example shows a sample output of the **thread** subcommand:

```
(dbx) thread
thread  thread_id           state  substate  held  exit_status
>$t1    0x03567d9000000001  activ             no    0x00000000
 $t2    0x0356831000000002  activ   cv_wait   no    0x00000000
 $t3    0x035688a000000003  activ   mu_wait   no    0x00000000
 $t4    0x03568e2000000004  activ   cv_wait   no    0x00000000
 $t5    0x035693b000000005  activ   jn_wait   no    0x00000000
 $t6    0x0356993800000006  activ   cv_wait   no    0x00000000
(dbx)
```

The > (greater than) sign in the left margin marks the *current thread*, which is the thread last notified of an event. A **dbx** internal name such as *$t1* is assigned to each thread for easy reference. Also, a data structure type is associated with the thread object for referring to individual elements of the thread object. For an example that uses the **dbx whatis** subcommand to display the data structure for an individual thread, see "Examining the Status of Individual Threads" on page 176.

Note: Thread internal names change as the program executes and the debugger automatically updates thread names. For example, if you have three threads—$t1, $t2, and $t3—and $t2 finishes running, the remaining two threads are renumbered to $t1 and $t2. You should keep this in mind when using the **dbx** subcommands that refer to thread names. Break and trace points that refer to thread names are automatically updated to their new names when necessary.

The `thread_id` column lists a constant hexadecimal value that is assigned to the thread when it is created.

The `state` column indicates the execution status of a thread. Possible states include:

activ
The thread is currently executing, or it is executing in a wait state. The `substate` field indicates if the thread is waiting: `cv_wait` for a thread waiting for a condition variable, `mu_wait` for a thread waiting for a mutex, and `jn_wait` for a thread waiting for a **pthread_join** call to return.

async
The thread is not currently executing on a task. For example, if the BPXPRMxx PARMLIB member MAXTHREADS is set to 1000, but MAXTHREADTASKS is set to 50, there can be only 50 threads in the **activ** state. If a program creates more than 50 threads, the ones greater than 50 will have a state of **async**.

dead
The thread has finished processing. The `exit_status` field contains data about the exit status of the thread when it finished processing.

pcanc
The thread is pending cancellation by either explicitly disabling cancellation or waiting for controlled cancellation at a specified cancellation point.

The `held` field indicates whether a thread is being held by **dbx**. The thread will not execute until it is released, allowing you to focus attention on other threads. For example, if a variable changes, you can be certain that it was not changed by the held thread.

Displaying Full Information about a Thread: The **thread** subcommand can be used with the **info** parameter to return more detailed information about a particular thread. For example, to display the full information about thread number 2, *$t2*, enter:

```
(dbx) thread info 2
 thread thread_id           state  substate  held  exit_status
>$t2    0x0356831000000002  activ             no   0x00000000
     pending signals:
        None
     general:
        detached    = no
        asynch      = no
        pthread     = no
        weight      = heavy
        stack size  = 5200

(dbx)
```

The weight of the thread is either medium or heavy. The default process thread is always heavy.

Displaying and Switching the Current Thread: All **dbx** subcommands return values based on the context of the current thread (indicated by >). To display the current thread, enter:

```
(dbx) thread current
 thread  thread_id           state  substate  held  exit_status
>$t1     0x0356e28800000001  activ             no   0x00000000
```

Thread number 1, *$t1*, is the current thread, as indicated by the > symbol. To display or modify variables local to another thread, you must first make it the current thread by entering:

```
(dbx) thread current 2
```

Then enter the **thread current** command to display information about the current thread:

```
(dbx) thread current
 thread thread_id           state  substate  held  exit_status
>$t2    0x03567d9000000001  activ             no   0x00000000
(dbx)
```

In these examples, thread number 2, *$t2*, replaces thread number 1, *$t1*, as the current thread.

Examining the Status of Individual Threads: After you know the names of the individual threads in your application program, you can access information about each thread using other **dbx** subcommands. Thread information can include the thread ID, the execution state of the thread, and its exit status, as shown in the preceding example. For example, you can use the **set** subcommand to display hexadecimal information about a particular thread:

```
(dbx) set $hexchars
(dbx) set $hexints
(dbx) print $t2
(thread_id = (0x3, 0x56, 0x9d, 0x90, 0x0, 0x0, 0x0, 0x1), state = 0x80,
kernel_attrs = 0x0, exit_status = (nil), signal_mask = (0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0))
(dbx)
```

You can also use the **whatis** subcommand to display the data structure associated with a thread. For example:

```
(dbx) whatis $t1
struct {
    $char thread_id[8];
    $char state;
    $short kernel_attrs;
    $void *exit_status;
    $char signal_mask[8];
} $t1;

(dbx)
```

Other **dbx** subcommands return information as usual in the context of the current thread.

Displaying Information about Mutex Objects: To check on the status of all mutex objects in your application, enter the **mutex** subcommand with no operands. The example below shows a sample output of the **mutex** subcommand:

```
(dbx) mutex
 mv      obj_addr    type  lock  owner   # wait  # recur  waiters
 $m1     0x341c418         yes   $t1          0        0
 $m2     0x341c404         yes   $t4          2        0  $t10, $t15
 $m3     0x341c410   recu  yes   $t2          0        0
 $m4     0x341c3f8   recu  yes   $t6          0        3
 $m5     0x341c3fc         no                 0        0
 $m6     0x341c408   recu  no                 0        0
(dbx)
```

A **dbx** internal name such as *$m1* is assigned to each mutex object for easy reference. Also, a data structure type is associated with the mutex for referring to individual elements of the mutex object. For an example that uses the **dbx whatis** subcommand to display the data structure for an individual thread, see “Examining the Status of Individual Mutex Objects” on page 178.

Note: Mutex internal names change as the program executes and the debugger automatically updates mutex names. For example, if you have three mutexes—$m1, $m2, and $m3—and $m2 is destroyed, the remaining two mutexes are renumbered to $m1 and $m2. You should keep this in mind when using the **dbx** subcommands that refer to mutex names. Break and trace points that refer to mutex names are automatically updated to their new names when necessary.

The `obj_addr` column contains the address of the object as allocated by the application program.

The `type` column indicates whether the mutex is recursive, and the `lock` column indicates whether the mutex is locked. If the mutex is locked, the `owner` column lists the name of the owning thread. If the mutex is locked, and there are threads waiting for the mutex, the `# wait` column shows how many threads are waiting for the mutex, and the `waiters` column lists the names of the waiting threads.

If the mutex is recursive and it is locked more than once by the same thread, the # recur column shows how many times that thread has locked the mutex after the first lock.

Examining the Status of Individual Mutex Objects: After you know the names of the individual mutex objects in your application, you can access information about each mutex object using other **dbx** subcommands. Mutex information can include the mutex ID, and its lock, wait, and recursion status, as shown in the preceding example. For example, you can use the **set** subcommand to display hexadecimal information about a particular mutex object:

```
(dbx) set $hexchars
(dbx) set $hexints
(dbx) print $m1
(type = "recu", lock = 0x1, num_wait=0x10, num_recur = 0x0)
(dbx)
```

You can also use the **whatis** subcommand to display the data structure associated with a mutex object. For example:

```
(dbx) whatis $m1
struct {
    enum { , recu } type;
    $char lock;
    $integer num_wait;
    $integer num_recur;
} $m1;

(dbx)
```

Displaying Information about Condition Variables: To check on the status of all condition variables in your application program, enter the **condition** subcommand with no operands. The following example shows a sample output of the **condition** subcommand:

```
(dbx) condition
cv       obj_addr    mutex   # wait  waiters
$c1      0x3340562   $m1          4  $t1 $t2 $t3 $t4
$c2      0x34030f0                0
(dbx)
```

A **dbx** internal name such as *$c1* is assigned to each condition variable for easy reference. Also, a data structure type is associated with the condition variable for referring to individual elements of the condition variable. For an example that uses the **dbx whatis** subcommand to display the data structure for an individual thread, see “Examining the Status of Individual Condition Variables” on page 179.

Note: Condition variable internal names change as the program executes and the debugger automatically updates condition variable names. For example, if you have three condition variables—$c1, $c2, and $c3—and $c2 is destroyed, the remaining two condition variables are renumbered to $c1 and $c2. You should keep this in mind when using the **dbx** subcommands that refer to condition variable names. Break and trace points that refer to condition variable names are automatically updated to their new names when necessary.

The obj_addr column contains the address of the object as allocated by the application program.

The `mutex` column shows the mutex object associated with the condition variable. If there are any threads waiting for the condition variable, the `# wait` column shows how many threads are waiting, and the `waiters` column lists the names of the waiting threads.

Examining the Status of Individual Condition Variables: After you know the names of the individual condition variables in your application program, you can access information about each condition variable using other **dbx** subcommands. Information can include the condition variable ID and wait status, as just shown. For example, you can use the **set** subcommand to display hexadecimal information about a particular condition variable:

```
(dbx) set $hexchars
(dbx) set $hexints
(dbx) print $c1
(mutex = 0x340e410, num_wait = 0)
(dbx)
```

You can also use the **whatis** subcommand to display the data structure associated with a condition variable. For example:

```
(dbx) whatis $c1
struct {
    $integer *mutex;
    $integer num_wait;
} $c1;

(dbx)
```

Displaying Information about Read/Write Lock Objects: To check on the status of all read/write lock objects in your application, enter the **mutex** subcommand with no operands. The example below shows a sample output of the **readwritelock** subcommand:

```
(dbx) mutex
 mv      obj_addr    type  shr  lock  holder   # wait  # recur  waiters
 $l1     0x341c418         yes  yes   $t1         0       0
 $l2     0x341c404         no   yes   $t4         2       0   $t10, $t15
 $l3     0x341c410   recu  no   yes   $t2         0       0
 $l4     0x341c3f8   recu  no   yes   $t6         0       3
 $l5     0x341c3fc         yes  no                0       0
 $l6     0x341c408   recu  no   no                0       0
(dbx)
```

A **dbx** internal name such as *$l1* is assigned to each mutex object for easy reference. Also, a data structure type is associated with the mutex for referring to individual elements of the mutex object. For an example that uses the **dbx whatis** subcommand to display the data structure for an individual thread, see “Examining the Status of Individual Mutex Objects” on page 178.

Note: Read/Write lock internal names change as the program executes and the debugger automatically updates read/write lock names. For example, if you have three read/write locks $l1, $l2, and $l3 and $l2 is destroyed, the remaining two mutexes are renumbered to $l1 and $l2. You should keep this in mind when using the **dbx** subcommands that refer to mutex names. Break and trace points that refer to read/write lock names are automatically updated to their new names when necessary.

The `obj_addr` column contains the address of the object as allocated by the application program.

The `type` column indicates whether the read/write lock is recursive, and the `lock` column indicates whether the read/write lock is locked. If the read/write lock is locked, the `holder` column lists the name of the owning thread. If the read/write lock is locked, and there are threads waiting for the read/write lock, the `# wait` column shows how many threads are waiting for the read/write lock, and the `waiters` column lists the names of the waiting threads.

If the read/write lock is recursive and it is locked more than once by the same thread, the `# recur` column shows how many times that thread has locked the read/write lock after the first lock.

Examining the Status of Individual Read/Write Lock Objects: After you know the names of the individual read/write lock objects in your application, you can access information about each read/write lock object using other **dbx** subcommands. Read/write lock information can include the read/write lock ID, and its lock, wait, and recursion status, as shown in the preceeding example. For example, you can use the set subcommand to display hexadecimal information about a particular read/write lock object:

```
(dbx) set $hexchars
(dbx) set $hexints
(dbx) print $l1
(type = "recu", lock = 0x1, num_wait=0x10, num_recur = 0x0, share = 0x0)
(dbx)
```

You can also use the **whatis** subcommand to display the data structure associated with a read/write lock object. For example:

```
(dbx) whatis $m1
struct {
    enum { , recu } type;
    $char lock;
    $integer num_wait;
    $integer num_recur;
    $char share;
} $m1;
(dbx)
```

Controlling Multithread Program Execution

z/OS UNIX threads are process-oriented threads that are maintained by the kernel. They depend on one another in order to run successfully. If you stop execution of any one thread, the entire process and all other threads in the process also stop.

Such **dbx** events as breakpoints are not specific to any single thread but apply to all threads within a process. If one thread reaches a breakpoint, the process and its threads also stop.

Using Breakpoint Subcommands: You can use the **next(i)** and **step(i)** subcommands to set a breakpoint within the program and continue execution until the breakpoint is reached. Since breakpoints are not specific to any one thread, the **step** or **next** subcommands can be interrupted by another thread or threads reaching the breakpoint intended for the **next** or **step** command.

To prevent other threads from reaching the specified breakpoint, you can prevent them from running. For example, you can run the **thread hold** subcommand to hold all threads, and then the **thread release number** subcommand to release the current thread. You can make this process automatic by setting the variable *$hold_next*; **dbx** then holds all threads except the current one before reaching the specified breakpoint, and releases all threads afterward.

Note: The execution of one thread may be dependent on the action of another thread—for example, waiting for the release of a mutex held by another thread. In this case, holding a thread may cause the program to deadlock.

Holding and Releasing Threads: You can control the running of your program by holding and releasing individual threads. When a thread is held, it is not allowed to run until it is released. For example, to hold thread number 4, enter:

```
(dbx) thread hold 4
(dbx)
```

Note: A program deadlocks if all its threads are held. In addition, the execution of one thread may depend on the action of another thread—for example, waiting for the release of a mutex held by another thread. In this case, holding even one thread may cause the program to deadlock.

Setting Per-Thread Breakpoints with a Conditional Breakpoint: Normal breakpoints are not specific to any one thread. If a thread reaches a breakpoint, all threads and the process stop. However, you can use conditional breakpoints to specify breakpoints for any one particular thread by checking the execution state of the thread. For example, if you need a breakpoint for thread number 3 at line 42, enter:

```
(dbx) stop at 42 if $t3==$current
(dbx)
```

Note: Be careful when determining where and how a conditional breakpoint is placed. This operation may be very slow if the breakpoint is placed at a high traffic area. Also, the subcommand **stop at 42 if $t3==$current** is much faster than the subcommand **stop in func if $t3==$current**, because a conditional breakpoint with the **stop in** subcommand forces **dbx** to check the condition at every line within the function. Only one check is needed when you enter an unconditional **stop at** subcommand.

Modifying Thread-Related Objects: You can use the **dbx** subcommand **assign** with thread objects. For example, you can set the exit status for a particular thread:

```
(dbx) set $hexchars
(dbx) set $hexints
(dbx) print $t2
(thread_id = (0x3, 0x56, 0x9d, 0x90, 0x0, 0x0, 0x0, 0x1),
state = 0x80,
kernel_attrs = 0x0, exit_status = (nil),
signal_mask = (0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0))
(dbx) assign $t2.exit_status=&$void(0x2d48692)
(dbx) print $t2.exit_status
0x2d58692
(dbx)
```

Note: Currently, the `exit_status` field in the thread object is the only modifiable field in a thread object, condition variable, or mutex object.

Preventing Unexpected Debugging Behavior: You can set the *$hold_next* parameter to prevent the unexpected debugging behavior that results when multiple threads are executing the same code where a breakpoint is set. When more than one thread reaches a breakpoint at the same time, **dbx** processes all event information for the first thread that reached the breakpoint before processing the subsequent threads. As part of the event processing for the first thread, the breakpoint can be deleted. All subsequent threads appear as though they received a **SIGTRAP** signal instead of reaching the breakpoint.

The *$hold_next* parameter affects **dbx** operation whenever it does machine-level instruction stepping, as well as during a **next**, **nexti**, **step**, or **stepi**. There are cases where **dbx** must do machine-level instruction stepping to properly stop or to provide trace information for a particular event. For example, the **stop var** subcommand causes **dbx** to evaluate the value of *var* after every machine instruction.

Debugging at the Machine Level

You can use the **dbx** utility to examine the program at the machine language level. You can display and modify memory addresses, display machine instructions, single-step instructions, set breakpoints and trace events at memory addresses, and display the registers.

In the commands and examples that follow, an address is an expression that evaluates to a memory address. The most common forms of addresses are integers and expressions that involve taking the address of an identifier with the **&** (ampersand) operator. You can also specify an address as an expression enclosed in parenthesis in machine-level commands. Addresses can be composed of other addresses and the operators **+** (plus), **–** (minus), and indirection (unary *****).

Using Machine Registers

Use the **registers** subcommand to see the value of the machine registers.

Registers are divided into two groups:

- *General-purpose registers* are denoted by **$r***number*, where *number* represents the number of the register. Supported register numbers are in the range 0–15.
- *Floating-point registers* are denoted by **$fr***number*, where *number* represents the number of the register. Supported register numbers are 0, 2, 4, and 6. Floating-point registers in long double format are denoted **$qfr0** and **$qfr4**.

  Floating-point registers are not displayed by default. Use the **unset** subcommand to reset the **$noflregs dbx** utility variable to enable the floating-point registers display (`unset $noflregs`).

Supported system-control registers are:

$iar $pc
: The instruction-address register

$sp $fp $stkp
: The stack or frame pointer

$psw $psw0 $psw1
: The program status words

$rtrn The return address register

$dest The destination address register

Note: If using the `XPLINK` output of the registers command $r4 changes to $sp instead of $r13.

Examining Memory Addresses

Use the following to print the contents of memory starting at the first address and continuing up to the second address, or until *Count* items are displayed. The mode specifies how memory is to be printed.

address, address /[mode][> file]
address / [count][mode] [>file]

If *mode* is omitted, the previous mode specified is reused. The initial mode is **X**. The following modes are supported:

b	Prints a byte in octal
c	Prints a byte as a character
C	Prints a wchar_t character
d	Prints a short word in decimal
D	Prints a long word in decimal
f	Prints a single-precision floating-point number
g	Prints a double-precision floating-point number
h	Prints a byte in hexadecimal
i	Prints the machine instruction
l	Prints a wint_t character
o	Prints a short word in octal
O	Prints a long word in octal
q	Prints a long double-precision floating-point number
s	Prints a string of characters terminated by a null byte
S	Prints a wchar_t string
W	Prints a wint_t string
x	Prints a short word in hexadecimal
X	Prints a long word in hexadecimal

Note the following example:

```
(dbx) print &x

0x3fffe460
(dbx) &x/X;

3fffe460: 31323300
(dbx) &x,&x+12/x

3fffe460: 3132 3300 7879 7a5a 5958 5756 003d 0032
(dbx) ($pc)/2i

100002cc (sub)  7c0802a6        mflr   r0
100002d0 (sub + 0x4)    bfc1fff8       stm      r30,-8(r1)
```

Expressions in parentheses can be used as an address.

Running a Program at the Machine Level

The commands for debugging your program at the machine level are similar to those at the symbolic level. The **stopi** subcommand stops the machine when the address is reached, or the condition is true, or the variable is changed. The **tracei** subcommands are similar to the symbolic trace subcommands. The **stepi** subcommand executes one or *number* machine instructions.

The **nexti** subcommand executes up to the next *number* machine instructions. The **nexti** subcommand does not follow branch-and-link instructions; it continues until execution returns to the next instruction. The **gotoi** subcommand changes the program counter address.

Other subcommands such as **assign** and **return** work at the machine level. In fact, if there are no symbolic addresses applicable; **stop**, **trace**, **step**, and **next** can be used equivalently with the machine-level counterparts.

When your program stops in a procedure that has not been compiled for debugging, the next instruction to be executed is displayed along with the current machine address. This is analogous to the current line at the symbolic level. For example:

```
(dbx) stopi at &sub

[1] stopi at 0x0010015e (sub)
(dbx) run

[1] stopped in sub at 0x10015e
0x0010015e (sub)            90bfd034      stm $r11,$r15,X'34'($sp)
(dbx) step 9

stopped in sub at 0x10017a
0x0010017a (sub+0x1c) 0def                basr $r14,$r15  (sub2)
```

If you performed another **stepi** at this point, you would stop at address `0x220226f4`, the entry point of procedure **printf**. Issuing the **nexti** subcommand at this point continues execution to `0x00100180` automatically.

Customizing the dbx Debugging Environment

You can customize your shell debugging environment by:
- Defining a prompt
- Creating subcommand aliases
- Using the **.dbxinit** file
- Passing subcommands from an HFS file

Defining a New dbx Prompt

The **dbx** prompt is normally the name with which you invoke the **dbx** utility. If you specified `/bin/dbx a.out` on the command line, your prompt would be:

```
(/bin/dbx)
```

You can change the prompt to suit your preference with the **prompt** subcommand. For example, to change the **dbx** prompt, enter the following to start the **dbx** utility for the program to be debugged:

```
/bin/dbx hello

dbx for MVS
Type 'help' for help.
reading symbolic information ...
(/bin/dbx)
```

The cursor is now positioned after the prompt for **(/bin/dbx)**. To change the prompt to `debug subcommand:` enter:

```
(/bin/dbx)
prompt "debug subcommand:"

debug subcommand:
```

You can also use the **prompt** line in your **.dbxinit** file to specify a different prompt. This causes your prompt to be used instead of the default each time you initialize **dbx**.

For example, to define a new prompt in the **.dbxinit** file, enter the following line in your **.dbxinit** file to initialize **dbx** with the debug prompt `debug subcommand:`

```
prompt "debug subcommand:
```

See “Using the .dbxinit. File” on page 186 for more information on using the **.dbxinit** file.

Creating dbx Subcommand Aliases

You can build your own commands from the **dbx** primitive subcommand set. The following commands allow you to build a user alias from the arguments specified. All commands in the replacement string for the alias must be **dbx** primitive subcommands. You can then use your aliases in place of the **dbx** primitives:

alias *[aliasname[commandname]]*
alias *aliasname "CommandString"*
alias *aliasname* (*parameter1, parameter2,...*) *"CommandString"*

The **alias** subcommand with no arguments displays the current aliases in effect; with one argument the command displays the replacement string associated with that alias.

The first two forms of **alias** are used simply to substitute the replacement string for the **alias** each time it is used. For example:

```
(dbx) alias rr rerun
(dbx) alias printandstep "print n; step"
(dbx)
```

Each time **rr** is typed at the subcommand prompt, **dbx** performs a **rerun** subcommand. Similarly, **printandstep** results in two subcommands being executed: **print n** and then **step.**

The third form of aliasing is a limited macro facility. Each parameter specified in the **alias** subcommand is substituted for in the replacement string. This can be useful in eliminating excessive typing:

```
(dbx) alias px(n) "set $hexints; print n; unset $hexints"
(dbx) alias a(x,y)"print symname[x] -> symvalue._n_n.name.Id[y]"
(dbx) px(126)

0xfe
```

The alias **px** in the previous example prints a value in hexadecimal without permanently affecting the debugging environment. The following aliases and associated subcommand names are supplied by **dbx** by default:

c	**cont**
cv	**condition variable**
d	**delete**
e	**edit**
h	**help**
j	**status**
l	**list**
m	**map**
mu	**mutex**
n	**next**
p	**print**
q	**quit**
r	**run**
s	**step**
st	**stop**
t	**where**
th	**thread**
x	**registers**

You can remove an alias with the **unalias** subcommand.

Using the .dbxinit. File

Each time you begin a debugging session, **dbx** searches for a special initialization file named **.dbxinit**. This file is searched for first in your working directory and then in the home directory. The **.dbxinit** file should contain a list of **dbx** subcommands to run each time you begin a debugging session. These subcommands are run before **dbx** begins to read subcommands from standard input.

Subcommands from the home directory **.dbxinit** file are processed before the subcommands from the working directory **.dbxinit** file. Normally, **.dbxinit** contains **alias** subcommands, but it can contain any valid **dbx** subcommands. For example:

```
# cat .dbxinit

alias si "stop in"
prompt "debug subcommand:"
# dbx a.out
dbx for MVS
Type 'help' for help.
reading symbolic information ...
debug subcommand: alias

si stop in
t  where
⋮
debug subcommand:
```

Reading dbx Subcommands from a File

The **-c** invocation option and **.dbxinit** provide mechanisms for executing **dbx** subcommands before reading from standard input. There is also a way to read **dbx** subcommands from a file after the debugging sessions has begun, using the **source** subcommand:

```
source filename
```

This reads **dbx** subcommands from the given file. For example, to read and run the list of subcommands in the **dbx** command file **cmdfile**, enter:

```
(dbx) source cmdfile
```

After running the list of commands in the **cmdfile** file, **dbx** displays a prompt and again waits for input.

You can also use the **-c** option to specify a list of subcommands to be run when initially invoking **dbx**.

```
dbx -cCommandFile [options][executable]
```

This runs the **dbx** subcommands in *CommandFile* before accepting subcommands from standard input. The **source** subcommand can be used for this purpose after **dbx** is invoked.

dbx Environment Variables

EDITOR
The editor to use during processing of the **dbx** edit subcommand. Set to ″vi″ by default.

HOME Your home directory.

PAGER
The pager to use during processing of the **dbx** help subcommand. Set to ″pg″ by default.

PATH Your program search path.

SHELL
: The shell to use during processing of the **dbx** sh subcommand. Set to ″sh″ by default.

_DBX_GCORECLISTDSN
: Alternate name for the SYS1.SBLSCLI0 MVS PDS data set that contains the IPCS clists, in particular BLSCDDIR is used to allocate a dump directory by the BPXGCORE kernel service. Unset by default

_DBX_GCOREDIRSTR
: Used to tailor the use of the IPCS dump directory. This string is used on the invocation of the BLSCDDIR command and may contain any of the parameters that are accepted by BSLCDDIR. The string may be used to tailor the creation of a temporary VSAM dump directory, or may be specified to request the use of an existing dump directory. Unset by default

_DBX_GCOREEXEDSN
: Alternate name for the 'SYS1.SBPXEXEC' MVS PDS data set that contains the REXX exec BPXTIPCS, used to create a dump directory and establish the IPCS environment for dump reading by the BPXGCORE kernel service. Unset by default

_DBX_GCORELOGDSN
: MVS data set name to contain the log created by the BPXGCORE kernel service. Unset by default

dbx External Program Usage

dbx uses the external programs that are listed in the following to perform various functions. Including a program in your PATH that has the same name as one of the programs **dbx** uses, and which does not behave as the system installed version, may cause **dbx** to behave unexpectedly. This includes any modified behavior that may result from creating shell aliases, environment variables, and/or a ″login script″ ($ENV).

echo
: **dbx** uses echo to parse command line options passed to your program from the **dbx** command line and on the run/rerun/sh **dbx** subcommands.

vi (EDITOR)
: **dbx** uses the program specified in the EDITOR environment variable during processing of the edit subcommand.

pg (PAGER)
: **dbx** uses the program specified in the PAGER environment variable during processing of the help subcommand.

sh (SHELL)
: **dbx** uses the program specified in the SHELL environment variable during processing of the sh subcommand.

Debugging MVS dumps

You can use the **dbx** utility to view MVS dumps to determine the cause of program failure. Full source level debug is available for C/C++ programs that have been compiled with -g. Machine level debug is also always available.

Commands that attempt to execute or modify the program are not available in dump processing mode. The following subcommands are available:

- alias
- args

- case
- down
- edit
- exmane storage
- file
- func
- help
- history
- list
- listfiles
- listfuncs
- listi
- map
- move
- object
- print
- prompt
- quit
- record
- registers
- set
- sh
- source
- thread
- unalias
- up
- use
- whatis
- where
- whereis
- which

For example:

```
dbx -I /u/fred/src -C tsimple.dmp
FDBX0089: dbx for MVS.
FDBX0399: Compiled: Jul  7 2000 10:04:54 GMT as BFP
FDBX0400: OS level: 09.00 02, LE level: 2.9 without CWIs.
FDBX0100: Type 'help' for help.
FDBX0750: Initializing dump tsimple.dmp.  This may take a while...
FDBX0751: BPXGMCDE token=0x241696f0, release=1, level=0
FDBX0752: BPXGMCDE Starting The TSO environment
FDBX0755: BPXGMCDE BPXTIPCS allocating dump directory via BLSCDDIR
FDBX0757: BPXGMCDE BPXTIPCS invoking IPCS
FDBX0760: BPXGMCDE Dump analysis processing ASIDs: 28 of 29
FDBX0762: Using ASID=0x0018
FDBX0763: Using PID=50331657
FDBX0732: interrupt code=0x4, abend code=0x940C4000, abend reason code=0x4, inst
ruction length=0x4
FDBX0099: reading symbolic information ...
FDBX0900: reading symbols for ./tsimple ...
```

```
segmentation violation in f3 at line 23 ($t1)
   23      printf("in f3() %d %p %s %d %d\n",bool,cptr,arry8,a,b);
(dbx) l 22
   22      fptr=0; *fptr=1;
(dbx) p fptr
0x0
(dbx)
```

The FDBX0762 message shows that dbx has found the ASID in the dump that caused the problem and will use that ASID for further processing. The FDBX0763 message shows that dbx has found the PID in the dump that casued the problem and will use that PID for further processing. The FDBX0732 message shows various information about the abend.

Finally dbx shows where the problem that occured, in this case a segmentation violation and the source line after the problem occured. Listing the previous line or lines will show the source line that was being processed when the error occured. Commands such as where and print will allow viewing the final state of the program and help to determine what went wrong. In our example, we see that the value of fptr is zero, and storing into location zero caused the segmentation violation.

An example of how to invoke dbx against a dump in an MVS data set:

```
dbx -I /u/fred/src -C "//'sys1.dump00'"
```

Ananlyzing dump initialization problems

dbx calls the BPXGMCDE kernel service which in turn calls IPCS to initialize the dump processing environment. Various problems can arise during this processing. dbx provides two methods for diagnosis/fixing dump initialization problems.

1. dbx lists any error messages that was produced by the BPXGMCDE call following the dbx message FDBX0379. This output can be used by an MVS systems programmer to fix the dump initialization problem.
2. By allocating an MVS data set and exporting the _DBX_GCORELOGDSN environment variable before running dbx against a dump, the BPXGMCDE service will store log information into the data set that may can be used by an MVS systems programmer to fix the dump initialization problem.

Maximizing source level debug of in production C/C++ program dumps

Due to limitations on the C/C++ symbolics generated by the compiler, dbx cannot perform source level debug of optimized compiled programs. Therefore, to allow some source level debug, but still retain the speed of an optimized program, some program restructure must be done. The key is to organize the program so that types/classes/structs and other definitions are kept in one C/C++ part along with global data that is compiled -g. The rest of the application references/modifies the data in this part. When the application abends and takes a dump, the state/important information is contained in the -g part, so dbx can now display/format the data in the dump in source level mode. By casting the types/structs/classes from this part to an address, dbx can display the data item symbolically.

Debugging Considerations/Setup On MVS

Setting up CEEEVDBG for ptrace/dbx's use

CEEEVDBG is a debugger exit that is provided by LE that dbx is now using for more operations than just obtaining thread/mutex and condition variable information. It is now also used for source level stepping and DLL processing.

In order for dbx to use this interface, a sample assembler program is supplied in SYS1.SAMPLIB(CEEEVDBG) that needs to installed in some loadlib in LNKLIST or a STEPLIB for example.

The following is a simple way to see if you have it installed correctly:

```
# dbx hello_world_simple_program
(dbx) stop in main
(dbx) cont
(dbx) step
>>>> if the step just runs the program,
>>>> CEEEVDBG is not installed correctly

>>>> if you stopped at the next source line,
>>>> then CEEEVDBG is installed correctly
```

The following is some sample JCL that can be used to install CEEEVDBG into a 'USERID.LOADLIB' PDS:

```
//CEEHASM  JOB (0000),'CWU',CLASS=A,MSGCLASS=X,
//         REGION=4M,NOTIFY=&SYSUID;
//*********************************************************************
//ASM      EXEC PGM=ASMA90,PARM='OBJECT,NODECK,LINECOUNT(55)'
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//         DD DSN=SYS1.MODGEN,DISP=SHR
//         DD DSN=CEE.SCEEMAC,DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1;,DISP=(,PASS),UNIT=SYSDA,
//         SPACE=(CYL,(1,1))
//SYSLIN   DD DSN=&&SYSLIN;,DISP=(,PASS),UNIT=SYSDA,
//         SPACE=(CYL,(1,1)),DCB=BLKSIZE=3200
//SYSIN    DD DSN=SYS1.SAMPLIB(CEEEVDBG),DISP=SHR
//*******************************************************************
//LINK     EXEC PGM=HEWL,PARM='LIST,XREF,MAP,RENT,REUS'
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLIB   DD DISP=SHR,DSN=CEE.SCEELKED
//         DD DISP=SHR,DSN=SYS1.CSSLIB
//SYSLMOD  DD DSN=USERID.LOADLIB(CEEEVDBG),DISP=SHR
//SYSLIN   DD DSN=&&SYSLIN;,DISP=(OLD,DELETE,DELETE)
//         DD *
  NAME  CEEEVDBG(R)
/*
```

Once USERID.LOADLIB(CEEEVDBG) is built, an MVS systems programmer can add this loadlib to the systems LNKLIST or it can be STEPLIB'ed to from the shell before running dbx:

```
export STEPLIB='USERID.LOADLIB'
```

Attach (-a) considerations

Some setup that is normally done by dbx when a program is started by dbx must be done when the program is run outside of dbx, then dbx is attached to the program at a later time. Two environment variables must be exported before the program is run:

export _BPX_PTRACE_ATTACH=yes
Tells the kernel to always load programs into a R/W subpool

export _CEE_RUNOPTS=″test(all)″
Tells LE to load the CEEEVDBG debugger exit and send it debug events.

Programs with DLL's

When building DLLs it is important not to **link** the DLL with -g. When linked -g, a DLL becomes non-reentrant. Therefore, when the DLL is loaded by LE, a new copy at a new address gets loaded. dbx will produce a warning when this condition occurs. The flow of the program from this point on is questionable since there is more than one copy of the DLL in storage and also more than one copy of writable static data in storage.

dbx does not load DLLs for the program, dbx only waits to get notified by LE when a DLL has been loaded so that dbx can read the symbolics for the DLL.

dbx does not know any symbolic information about a DLL until after the DLL gets loaded into storage by LE and dbx reads symbolics for the DLL. The default behavior for dbx is to stop on the first reference of a DLL so that if debugging of the DLL is desired, breakpoints can then be set or static DLL data can be displayed/modified.

The default behavior of dbx to stop at the first reference of a DLL and to process DLLs at all can be modified by the two dbx set variables $dll_loadstop and $dll_loads which are set by default. In a **.dbxsetup** or at any time during debugging the **$dll_loads** variable can be unset as in unset $dll_loads. This tells dbx to ignore any DLL loads and do not process any symbols for the DLLs. This is useful if the applications contains DLLs, however, no debugging of the DLLs is desired. The benefit is faster execution of the application since dbx does not have to pause the application to read symbolics for the newly loaded DLLs.

If debugging of the DLLs is still desired, IE, display static DLL data, however, the only important stop/trace locations is in the main part of the program, then the following command can be used to tell dbx not to stop at DLL references: *unset $dll_loadstop*. When this set flag is unset, dbx will still read symbolics for DLLs as they are loaded, however, will allow the application to continue running after symbolics are processed.

If debugging of only certain DLLs is desired, the $dll_ask dbx set variable can be set which is unset by default as in: *set $dll_ask*. Once set, before dbx reads symbolics for a DLL, it prompt the user for a Y]N answer, telling dbx to read symbolics for the DLL or not. This is useful when the program contains large DLLs and/or a large number of DLLs and only a small number or one DLL needs to be debugged. The benefit is to cut down on symbolics reading time for undesired DLLs.

If stopping only in a specific function in a specific DLL is desired, the **onload** dbx subcommand can be used in conjunction with $dll_loadstop. The following dbx subcommand is placed in a **.dbxsetup** file: *unset $dll_loadstop*. This tells dbx to not stop at the first reference of a DLL after DLL symbolics reading has completed. The following dbx subcommand tells dbx to activate the stop event when the function becomes defined in a DLL that was loaded: *onload stop in myfunc*. This allows a deferred event to be setup at any time, in any DLL, only activated when the particular file/symbol/line is known to dbx.

Programs that run fine only under dbx

When dbx is in control, the kernel loads programs into R/W storage so that dbx can set breakpoints. If a program runs fine in this situation, it may be that the program is attempting to write over the program code and/or some other write protected area of the program.

This can be verified by running the program in R/W storage outside of dbx control. To force the program to get loaded into R/W storage, before running the program export the following environment variable:

```
export _BPX_PTRACE_ATTACH=yes
```

Now any program that is loaded will load into R/W storage, including any DLLs.

Multiprocess debugging

The default for multiprocess debugging is off. When dbx receives a fork event from your application, dbx detaches itself from the child process of the fork, allowing the child to execute without any dbx intervention. The parent remains under dbx control and can be debugged as usual.

To debug both the parent and the child of the fork, you must first enter the dbx subcommand *multproc on*. When your application forks, both the parent and the child processes stop and dbx sends out a prompt that explains how to reattach another dbx to the child process so that now there will be two dbxs. One debugging the parent process and one debugging the child process.

To debug only the child of the forks, you must first enter the dbx subcommand *multproc child*. When your application forks, dbx detaches itself from the parent process of the fork, allowing the parent to execute with any dbx intervention. The child remains under dbx control and can be debugged as usual.

Programs started via JCL/daemons

dbx can only debug or attach to processes. The following C program lines will cause even a non-posix MVS program to become a process and wait for dbx to attach to it:

```
printf("My PID is: %d\n",getpid());
sleep(30);
```

If the program was not yet a process, the getpid() will automatically turn the program into a process. The resulting PID that is printed is the one that dbx can attach to, IE, *dbx -a 42*.

There are other authority considerations for debugging daemaons that call such set functions as setuid() and setgrp(). This is beyond the scope of this dbx chapter. Special authorization in the security facilty, IE, RACF, may need to be setup by an MVS system's programmer to allow the authorized programs to run successfully when using dbx to debug.

Programs that exist in MVS data sets rather than the HFS

dbx can debug programs that have source or loadmodules in MVS data sets as compared to the heirarchtical file system (HFS).

dbx can only locate source in MVS data sets when:

1. The program was compiled with the source in an MVS data set or partioned data set. This is required because the compiler puts the source file name inside the symbolics of the loadmodule.
2. The source is in a variable block/record data set or partioned data set. dbx does not support source code in fixed block/record data sets or partioned data sets.
3. The user places the special path // into the dbx source path directory list, IE, use //. This tells dbx to specially search for MVS data sets.

dbx does not get proper events for DLLs or loadmodules that exist in MVS data sets or MVS partioned data sets, therefore cannot support these programs directly.

However, dbx does support loadmodules in MVS data sets when they are loaded via external links in the hierarchtical file system. This program arrangement is usually required for authorized programs such as daemons to run on MVS. In addition, for performance reasons, dbx by default does not process symbolics for MVS loadmodules that it finds in the load map. The following line must be added to a **.dbxsetup** so that dbx will process MVS loadmodules: *set $sticky_debug*.

Performance considerations (symbolics reading)

By default dbx will attempt to spawn the program to be debugged so that dbx and the program can exist in the same address space. This allows dbx to do direct read of information in the debuggee which increases symbolics reading performance greatly and reduces storage use since dbx can now point to the information directly in the debuggee.

If dbx is in a different address than the debuggee because of a -a, -A or the spawn startup failed, then dbx will use captured storage to map areas of the debuggee into its own address space. This is faster than using ptrace() to read every piece of information, however, does require table lookups and an address translation for every storage access, so it is not as fast as being in the same address space.

If for some reason the above two fail, then dbx must resort to calling ptrace() for every block of information it needs to processing in the debuggee. This is the slowest access method to the debuggee.

Performance considerations (program execution)

As in symbolics reading, when dbx can use direct address space reads and writes, performance is the best, followed by captured storage followed by the slowest ptrace() calls. This is because dbx must inquire/change information in the debuggee to performance operations like variable value queries and setting/removing breakpoints.

There are some dbx events that will cause dbx to source level step the debuggee which will cause a noticable speed degradation in the debuggee. Command examples:

```
trace
trace in myfunc
```

There are some dbx events that will cause dbx to instruction step the debuggee which will cause a significant speed degradation in the debuggee. Any events that require dbx to query the value of storage or a variable will cause dbx to execute the program in this mode. Command examples:

```
stop if a>2
trace x
trace if b==3
stop in myfunc if x<55
```

In the above examples, the last example limits the scope of the instruction stepping when the program execution is within function myfunc.

Limiting the scope of where the instruction/single stepping occurs by using **in***func-name* will help to maintain program execution. Setting stop/trace events at specific lines is the fastest since dbx only needs to stop and query the program's state at a specific line. Command examples:

```
st at "myfile.c":52 if x>2
trace x at 99
```

List of dbx Subcommands

The **dbx** utility provides subcommands for the following task categories:
- Debugging environment control
- Displaying the source file
- Ending program execution
- Machine-level debugging
- Multiprocess debugging
- Multithread debugging
- Printing and modifying variables, expressions, and types
- Procedure calling
- Running your program
- Setting and deleting breakpoints
- Signal handling
- Tracing program execution

The complete alphabetized list of **dbx** subcommands is:

/	Searches forward in the current source file for a pattern.
?	Searches backward in the current source file for a pattern.
address **/**	Just specifying an address displays the contents of memory.
alias	Displays and assigns aliases for **dbx** subcommands.
assign	Assigns a value to a variable or change the exit status of a thread.
case	Changes the way that **dbx** interprets symbols. Symbols are normally interpreted as they appear.
catch	Starts trapping a signal before that signal is sent to the application program.
clear	Removes all stops at a given source line.
cleari	Removes all breakpoints at an address.
condition	Displays a list of active condition variables for a multithread program.
cont	Continues program execution from the current breakpoint until the program finishes, another breakpoint is encountered, or a signal that cannot be ignored is received.
delete	Removes the traces and stops corresponding to the specified numbers.
display memory	See *address*.
down	Moves the current function down the stack.
dump	Displays the names and values of variables in the specified procedure.
edit	Invokes an editor on the specified file.
file	Changes the current source file to the specified file.
func	Changes the current function to the specified function or procedure.
goto	Causes the specified source line to be the next line executed.
gotoi	Changes program counter addresses.
help	Displays a list of commonly used **dbx** commands.

ignore
Stops trapping a signal before that signal is sent to the application program.

list Displays lines of the current source file.

listi Lists instructions from the application program.

map Displays address maps and loader information for the application program.

move Changes the next line to be displayed.

multproc
Enables or disables multiprocess debugging.

mutex Displays a list of active mutex objects for the application program.

next Runs the application program up to the next source line. When you use **next** with **$hold_next**, the current thread executes and all others are held.

nexti Runs the application program up to the next machine instruction. When you use **nexti** with **$hold_next**, the current thread executes and all others are held.

object *filename*
Loads the specified executable file, without the overhead of reloading **dbx**.

print Prints the value of expressions.

prompt
Changes the **dbx** prompt to the specified string.

quit Quits the **dbx** utility.

registers
Displays the values of all general-purpose registers, system-control registers, floating-point registers, and the current instruction register.

rerun Restarts execution of an application program using the parameters from a previous **run** or **rerun** command.

return Continues execution of the application program until a return to the specified procedure is reached.

run Begins execution of an application program. Can pass parameters to the application program.

set Defines a value for a nonprogram, condition, mutex, or thread variable.

sh Passes a command to the shell for execution.

skip Continues execution from the current stopping point and skips some number of breakpoints.

source
Reads **dbx** commands from a file.

status Displays the currently active **trace** and **stop** subcommands.

step Runs a single source line. When you use **step** with **$hold_next**, the current thread executes and all others are held.

stepi Runs a single machine instruction. When you use **stepi** with **$hold_next**, the current thread executes and all others are held.

thread
Displays a list of active threads for the application program and can be used to hold, release, and switch the current thread.

stop Stops execution of the application program.

stopi Sets a stop at a specified location.

trace Prints tracing information.

tracei Turns on tracing using a machine instruction address.

unalias
Removes an alias.

unset Deletes a nonprogram variable.

up Moves the current function up the stack.

use Sets the list of directories to be searched when looking for a source file (HFS file or MVS data set).

whatis
Displays the declaration of application program components.

where Displays a list of all active procedures and functions.

whereis
Displays the full qualifications of all the symbols whose names match the specified identifier.

which Displays the full qualification of the specified identifier.

See *z/OS UNIX System Services Command Reference* for a complete discussion of the format and use of these **dbx** subcommands.

Debugging Environment Control

The **dbx** subcommands for debugging environment control are:

alias Displays and assigns aliases for **dbx** subcommands.

help Displays a list of commonly used **dbx** commands.

prompt
Changes the **dbx** prompt to the specified string.

sh Passes a command to the shell for execution.

source
Reads **dbx** commands from a file.

unalias
Removes an alias.

Debugging Threads

condition
Displays a list of active condition variables for a multithread program.

mutex Displays a list of active mutex objects for the application program.

thread
Displays a list of active threads for the application program and can be used to hold, release, and switch the current thread.

Displaying the Source File

The **dbx** subcommands for displaying the source file are:

/ Searches forward in the current source file for a pattern.

? Searches backward in the current source file for a pattern.

edit Invokes an editor on the specified file.

file Changes the current source file to the specified file.

func Changes the current function to the specified function or procedure.

list Displays lines of the current source file.

listi Lists instructions from the application program.

move Changes the next line to be displayed.

use Sets the list of directories to be searched when looking for a source file (HFS file or MVS data set).

Ending Program Execution

The **dbx** subcommands for ending program execution are:

quit Quits the **dbx** utility.

Machine-Level Debugging

The **dbx** subcommands for machine-level debugging are:

address **/**
Just specifying an address displays the contents of memory.

display memory
Displays the contents of memory.

gotoi Changes program counter addresses.

map Displays address maps and loader information for the application program.

nexti Runs the application program up to the next machine instruction.

registers
Displays the values of all general-purpose registers, system-control registers, floating-point registers, and the current instruction register.

stepi Runs one source instruction.

stopi Sets a stop at a specified location.

tracei Turns on tracing using a machine instruction address.

Multiprocess Debugging

The **dbx** subcommand for multiprocess debugging is:

multproc
Enables or disables multiprocess debugging.

Printing and Modifying Variables, Expressions, and Types

The **dbx** subcommands for printing and modifying variables, expressions, and types are:

assign
Assigns a value to a variable.

case Changes the way in that **dbx** interprets symbols. Symbols are normally interpreted as they appear.

condition
Displays a list of active condition variables for a multithread program.

dump Displays the names and values of variables in the specified procedure.

mutex Displays a list of active mutex objects for the application program.

print Prints the value of expressions.

set Assigns a value for a nonprogram variable.

thread
Displays a list of active threads for the application program and can be used to hold, release, and switch the current thread.

unset Deletes a nonprogram variable.

whatis
Displays the declaration of application program components.

whereis
Displays the full qualifications of all the symbols whose names match the specified identifier.

which Displays the full qualification of the specified identifier.

Procedure Calling

dbx does not support a print procedure or print function.

Running Your Program

The **dbx** subcommands for running your program are:

cont Continues program execution from the current breakpoint until the program finishes, another breakpoint is encountered, or a signal that cannot be ignored is received.

down Changes the current scoping content to the next stack frame.

goto Causes the specified source line to be the next line executed.

gotoi Changes program counter addresses.

mutex Displays a list of active mutex objects for the application program.

next Runs the application program up to the next source line. When used with **$hold_next**, the current thread executes and all others are held.

nexti Runs the application program up to the next machine instruction. When used with **$hold_next**, the current thread executes and all others are held.

object *filename*
Loads the specified executable file, without the overhead of reloading **dbx**.

rerun Restarts execution of an application program using the parameters from a previous **run** or **rerun** command.

return Continues execution of the application program until a return to the specified procedure is reached.

run Begins execution of an application program.

skip Continues execution from the current stopping point and skips some number of breakpoints.

step Runs a single source line.

stepi	Runs a single source instruction.
thread	Displays a list of active threads for the application program and can be used to hold, release, and switch the current thread.
up	Changes the current scoping content to the previous stack frame.

Setting and Deleting Breakpoints

The **dbx** subcommands for setting and deleting breakpoints are:

clear	Removes all stops at a given source line.
cleari	Removes all breakpoints at an address.
delete	Removes the traces and stops corresponding to the specified numbers.
status	Displays the currently active **trace** and **stop** subcommands.
stop	Stops execution of the application program.

Signal Handling

The **dbx** subcommands for signal handling are:

catch	Starts trapping a signal before that signal is sent to the application program.
ignore	Stops trapping a signal before that signal is sent to the application program.

Tracing Program Execution

The **dbx** subcommands for tracing program execution are:

trace	Prints tracing information.
tracei	Turns on tracing using a machine instruction address.
where	Displays a list of all active procedures and functions.

Appendix A. TSO/3270 Passthrough Mode

Overview

TSO/3270 passthrough mode allows full-screen 3270 interactive applications to be invoked from and run in a shell environment. A full-screen 3270 interactive application can exercise significant control over the 3270 terminal device by sending and receiving 3270 data, and can thus take full advantage of the display and input/output capabilities of 3270 devices, as opposed to a line-mode only application.

TSO/3270 passthrough mode allows a POSIX application program that is invoked from a shell command prompt to change its mode of terminal interaction from line mode to *TSO/3270 passthrough mode*. In line-mode interaction between the terminal (in this case a 3270 device) and an application program, the application can read and write lines of text data delimited by newline characters. While in TSO/3270 passthrough mode the application can read and write 3270 data, allowing it the same degree of control over the 3270 device as an application that uses the TSO TPUT and TGET APIs.

Note: z/OS UNIX does no validation of the data stream prepared by the application; therefore, sending data that is inappropriate for the device (for example, that exceeds the device's capabilities) may cause unpredictable results.

TSO/3270 passthrough mode is an extension to that subset of the POSIX General Terminal Interface (GTI) supported by z/OS UNIX for 3270 devices. The interface provides the following functions:

- The application program can detect whether a file descriptor represents a 3270 terminal that can be put into TSO/3270 passthrough mode.
- The application program can change from line mode to TSO/3270 passthrough mode and back again.
- An application can determine the current mode (line mode or TSO/3270 transparent mode).
- An application can indirectly issue certain TSO APIs (such as TPUT, TGET, and TPG) to send or receive 3270 data to the terminal.
- An application can indirectly issue the IKJEFTSR API to invoke most TSO commands.
- A program can return from 3270 transparent to line mode at appropriate times, such as upon normal or abnormal termination of a child process that had been operating in TSO/3270 passthrough mode.

TSO/3270 passthrough mode provides basic functions to read and write 3270 data from a program that is invoked exclusively in the shell environment provided by the combination of the z/OS UNIX shells and utilities, the OMVS command processor, and related kernel pseudoterminal and line discipline functions. The programming interfaces (using commands that are imbedded in the data read and written) are direct mappings of TSO TGET and TPUT.

Supported TSO Functions

TSO/3270 passthrough mode supports only the following APIs:

- GTDEVSIZ
- GTSIZE
- GTTERM

  ATTRIB=, PRMSIZE=, and ALTSIZE= are always specified (that is, all 3 fields are always passed back to the TSO/3270 passthrough mode application).
- IKJEFTSR
- STCOM
- STFSMODE (RSHWKEY=n is not supported)
- STLINENO
- STSIZE
- STTMPMD
- TCLEARQ
- TGET
- TPG
- TPUT

HIGHP, ASID=, USERIDL=, TOKNIN= options are not supported, because OMVS is not an authorized program. This prevents TSO/3270 passthrough mode applications from sending messages to other users.

TSO Extensions Version 2 Programming Guide and *TSO Extensions Version 2 Programming Services* fully describe the services that can be invoked with the TSO/3270 passthrough mode facility.

Using the TSO/3270 Passthrough Data Stream

To use the TSO/3270 Passthrough facility, a typical application program would:

1. Include the **fomth32p.h** header file, which contains the layout of the TSO/3270 passthrough data stream.
2. Use **tcgetattr**() against STDIN_FILENO or the controlling terminal to verify that the terminal supports TSO/3270 passthrough mode. If the PKT3270 bit in termios is set, the terminal supports TSO/3270 passthrough mode.
3. Use **tcsetattr**() to set the PTU3270 bit in termios. This sets the terminal into TSO/3270 passthrough mode.
4. Indirectly issue TSO APIs using the TSO/3270 passthrough data stream:
   - The application uses **write()** and **read()** to exchange data with the passthrough-mode terminal. When the terminal is in TSO/3270 passthrough mode, only the special TSO/3270 passthrough data stream can be sent. Ordinary character data must not be sent to the terminal while it is in passthrough mode.
   - The TSO/3270 passthrough data stream consists of requests written to TSO and responses from OMVS or TSO. Each request or response consists of a 12-byte header optionally followed by variable length binary data.
   - Requests can be sent in a single **write()** to the TTY, or a single passthrough request can be split across many separate **write()** calls to the TTY. More than one request can be combined in a single **write()**.

- When reading responses, it is possible to get back part of a response when doing **read()** to the TTY. Also, data from a single TTY **read()** can contain more than one TSO/3270 passthrough response.

  After TSO/3270 passthrough mode is entered, the first byte **read()** will be the first byte of the first response header. The program can determine how long this response is from the data length field in the response header. It can then determine where the data for this response ends in the incoming data stream. The next response header will immediately follow the end of the previous response in the data stream. The TSO/3270 passthrough mode application must accumulate or split up the incoming data stream into individual responses.
- The application can use STFSMODE and STTMPMD to put the TSO session into TSO fullscreen mode. It then can use TGET, TPUT, and TPG to send 3270 fullscreen data to the terminal.

  A typical TPUT request would contain the 12-byte header followed by data to be passed to TPUT. The various options on the TPUT macro are indicated in the request header by coded values. TPUT responses are 12 bytes long, and may contain error return codes from OMVS or from TSO.

  TGET requests are 12 bytes long (they contain no optional data). The request header contains coded values for the requested TGET options. TGET responses contain a 12-byte header followed by any data returned from TGET. The response header contains return code information from OMVS and TSO, along with the length of the data from TGET.
- Typically, a program issues a request to TSO and then waits for the response. It checks the return code in the response and handles any received data. The response header can contain error return codes from TSO or the OMVS command itself:
  - If the TSO/E OMVS command detects an error in the passthrough request (unknown TSO API requested, for example), it rejects the request without passing it through to TSO. The __error field in the response header is set to one of the codes described in the next section.
  - If the TSO/E OMVS command does not find errors, the request is passed through to TSO with the macro options in the request header and any optional data. The results from TSO, along with any data from TSO, are packaged into a response and sent back to the application through the TTY. The application issues one or more **read()**s to get the results and data. The **fomth32p.h** header file example shown later in this appendix ("TSO/3270 Passthrough Mode Data Stream" on page 210) describes how results and errors from each TSO command are passed back in the response.
- Applications can indirectly invoke TSO commands using the IKJEFTSR request and response.

  The IKJEFTSR response contains only return code information from TSO and the invoked command. Actual output from the invoked TSO command is not returned as data following the response header.

5. Return the terminal to normal mode, using **tcsetattr**() to reset the PTU3270 bit in termios. This will end TSO/3270 passthrough mode. The application should read in any expected responses before issuing **tcsetattr**().

Preliminary Processing of TSO/3270 Passthrough Mode Requests

Preliminary error checks are performed on all TSO/3270 passthrough mode requests received while the TTY is in TSO/3270 passthrough mode. If the preliminary check fails, the request is not passed to TSO, and the error is reported in the response.

The following severe errors can be reported:

__error	**Error description**
0xC1	First byte of TSO/3270 passthrough mode request is not 0xFF. When the TSO/E OMVS command reads a TSO/3270 passthrough mode request from the master TTY, it expects the first byte of data to be 0xFF, the TSO/3270 passthrough mode request introductory byte. If the first byte is not 0xFF, this error occurs. This error can occur if non-TSO/3270 data is written to the slave TTY, or if the length field in the previous TSO/3270 passthrough mode request was shorter than the amount of 3270 data following it. 3270 application errors can also cause this problem. Non-3270 data can be written to the TTY from background processes, from the job-control shell, or by inter-user message programs. This non-3270 data can get intermixed with the 3270 data stream, causing the end of the 3270 data to be treated as the start of the next (bad) TSO3270 request.
0xC2	Length field is too long. This error occurs when the __l field in the TSO/3270 passthrough mode request is longer than 32767 bytes, which is not allowed for any request.

Note: In addition to passing back the __error field listed in the TSO/3270 passthrough mode response, the __rc field is set to -1. OMVS also ends TSO/3270 passthrough mode when one of these severe errors occurs. OMVS passes back a TSO/3270 passthrough mode response with __error set to the error code given above. OMVS then waits a few seconds, in case the 3270 application has not already issued **read()** to get the TSO/3270 passthrough mode response. OMVS then ends TSO/3270 passthrough mode. The TTY is set back to normal (non-TSO/3270 passthrough) mode. The TSO/E OMVS command then sends SIGWINCH to all processes in the foreground process group. The 3270 application may end when this error is detected (either SIGWINCH is received or the __error field is seen). It should catch SIGWINCH or else do frequent **tcgetattr**() to detect when OMVS ends TSO/3270 passthrough mode on the TTY.

The following error conditions are also detected in the preliminary error check. When one of these errors occurs, the requested TSO function is not invoked. The __error field in the TSO/3270 passthrough mode response is set to the value indicated, and the __rc field is set to -1.

__error	**Error description**
0x81	The __fcn field contains an unknown TSO function code. This error occurs when the __fcn field is not one of the known coded values.

0x82	This TSO function required no data, but data was provided.
	This error occurs when the TSO function accepts no 3270 data, but the __l field in the TSO/3270 passthrough mode request was non-zero. The __l field must be zero in this case, and no data can be passed in the TSO/3270 passthrough mode request. When processing the failing request, OMVS steps past __l bytes of data in the __d field and then looks for the start of the next TSO/3270 passthrough mode request.
0x91	The __p1 field contains an unknown coded value for this TSO/E function.
	This error occurs when the requested TSO function uses the __p1 field as an input parameter. The __p1 field does not contain one of the allowed values for this TSO function.
0x92	The __p2 field contains an unknown coded value for this TSO/E function.
	This error occurs when the requested TSO function uses the __p2 field as an input parameter. The __p2 field does not contain one of the allowed values for this TSO function.
0x93	The __p3 field contains an unknown coded value for this TSO/E function.
	This error occurs when the requested TSO function uses the __p3 field as an input parameter. The __p3 field does not contain one of the allowed values for this TSO function.
0xA1	Not enough storage was available to invoke the requested TSO/E function.
	This error occurs when the TSO/E OMVS command cannot get enough storage (usually only a few hundred bytes) to call the requested TSO/E function. The lack of storage may be of short or long duration. If this TSO/3270 passthrough mode request is reissued, it may succeed, or it could fail again.

In addition, the following warning conditions can be detected. If one of these errors is detected, the __error field in the response is set as indicated, and the requested TSO/E service is invoked. The __rc field, and the __l and __d fields are set based on the TSO/E return code and any returned data. (See the discussion of TSO return code processing that follows.) These warning conditions can be reported:

__error	**Error description**
0x41	Reserved __p1 field should be 0, but was non-zero.
	This error occurs when the __p1 field in the TSO/3270 passthrough mode request is not used for the requested TSO/E function. This unused field is reserved, and should be set to zero in the TSO/3270 passthrough mode request.
0x42	Reserved __p2 field should be 0, but was non-zero.
	This error occurs when the __p2 field in the TSO/3270 passthrough mode request is not used for the requested TSO/E function. This unused field is reserved, and should be set to zero in the TSO/3270 passthrough mode request.
0x43	Reserved __p3 field should be 0, but was non-zero.

	This error occurs when the __p3 field in the TSO/3270 passthrough mode request is not used for the requested TSO/E function. This unused field is reserved, and should be set to zero in the TSO/3270 passthrough mode request.
0x49	One or more reserved bits in the __p1 field were on. This error occurs when at least one of the reserved bits in the __p1 field is not used for the requested TSO/E function. All unused bits are reserved, and should be set to zero in the TSO/3270 passthrough mode request.
0x4A	One or more reserved bits in the __p2 field were on. This error occurs when at least one of the reserved bits in the __p2 field is not used for the requested TSO/E function. All unused bits are reserved, and should be set to zero in the TSO/3270 passthrough mode request.

Only one of these warning conditions is reported. The user 3270 application can choose to ignore these warnings, but the application may fail in the future if a reserved bit or field becomes meaningful. The 3270 application should treat these warnings as unexpected errors. The 3270 application may be designed to fail whenever the __error field is non-zero.

If none of these errors or warnings is detected, the __error field in the response will be set to 0, and the requested TSO/E service is invoked. The __rc field, and the __l and __d fields will be set based on the TSO/E return code and any returned data.

Processing of Return Codes from Invoked TSO Services

If all preliminary checks are correct, OMVS passes the request to TSO. The TSO return code and any data are packaged into a response, depending on the TSO return code.

This section describes how OMVS handles return codes from the invoked TSO functions. See *TSO Extensions Version 2 Programming Guide* and *TSO Extensions Version 2 Programming Services* for more information about each return code from the TSO/E function.

Note: When the OMVS error action says that OMVS ends (because the terminal has logged off), the master TTY is closed. This eventually causes **read()** from the slave TTY to fail.

- GTDEVSIZ

r/c	OMVS processing
0(0)	Pass back: __rc=0, __l=8, __d = data from registers 0 and 1
4(4)	Pass back: __rc=4, __l=0 -- (Parameter specified. This error should not occur.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- GTSIZE

r/c	OMVS processing
0(0)	Pass back: __rc=0, __l=8, __d = data from registers 0 and 1
4(4)	Pass back: __rc=4, __l=0 -- (Parameter specified. This error should not occur.)

other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- GTTERM

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=8, __d = 8 bytes of data, as described in **fomth32p.h** (see below)
8(8)	Pass back: __rc=8, __l=0 -- (Not a 3270 terminal. This error should not occur.)
12(C)	Pass back: __rc=12, __l=0 -- (Missing PRMSIZE parameter. This error should not occur.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- IKJEFTSR

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=12, __d = 12 bytes of data, as described in **fomth32p.h** (Normal completion)
4(4)	Pass back: __rc=0, __l=12, __d = 12 bytes of data, as described in **fomth32p.h** (Non-zero return code from command. This return code is reported in the response data.)
8(8)	Pass back: __rc=0, __l=12, __d = 12 bytes of data, as described in **fomth32p.h** (Attention ended the command.)
12(C)	Pass back: __rc=0, __l=12, __d = 12 bytes of data, as described in **fomth32p.h** (Command abended. The abend code and reason code are in the response data.)
other	Pass back: __rc=0, __l=12, __d = 12 bytes of data, as described in **fomth32p.h** (These errors should not occur.)

- STCOM

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (Invalid parameter specified. This error should not occur.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- STFSMODE

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (Invalid parameter specified. This error should not occur.)
8(8)	Pass back: __rc=8, __l=0 -- (Not a 3270 terminal. This error should not occur.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- STLINENO

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (Invalid parameter specified. This error should not occur.)

8(8)	Pass back: __rc=8, __l=0 -- (Not a 3270 terminal. This error should not occur.)
12(C)	Pass back: __rc=12, __l=0 -- (Line number was 0 or too high. This could be a bad parameter from the TSO/3270 passthrough mode application.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- STSIZE

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (Invalid parameter specified. This error should not occur.)
8(8)	Pass back: __rc=8, __l=0 -- (SIZE= or LINE= was invalid. This could be a bad parameter from the TSO/3270 passthrough mode application.)
12(C)	Pass back: __rc=12, __l=0 -- (Screen size unknown. This could be a bad parameter from the TSO/3270 passthrough mode application.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- STTMPMD

r/c	**OMVS processing**
0	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (Invalid parameter specified. This error should not occur.)
8(8)	Pass back: __rc=8, __l=0 -- (Not a display terminal. This error should not occur.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- TCLEARQ

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (Invalid parameter specified. This error should not occur.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- TGET

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=register 1, d = __l bytes of data **Note:** If register 1 exceeds 32767, __error = 0xB1, __rc = -1, __l = 0, and no data will be passed back. This error should not occur.)
4(4)	Pass back: __rc=4, __l=0 -- (NOWAIT, and no data available; not an error.)
8(8)	Pass back: __rc=8, __l=0 -- (Attention occurred.) If ENDPASSTHROUGH(ATTN) was specified on the TSO/E OMVS command, TSO/3270 passthrough mode will end (see "The ENDPASSTHROUGH key" later in this appendix).
12(C)	Pass back: __rc=12, __l=register 1, d = __l bytes of data

This TSO/E return code occurs when not all the available data fits in the buffers passed to TGET. This is not an error. The next TGET will obtain more of the available data.

Note: When TGET returns, if register 1 exceeds 32767, __error = 0xB1, __rc = -1, __l = 0, and no data will be passed back. This error should not occur.

16(10) Pass back: __rc=16, __l=0 -- (Invalid parameters; this error should not occur.)

20(14) Pass back: __rc=20, __l=0 -- (Terminal logged off. OMVS ends after TSO/3270 passthrough mode ends.)

24(18) Pass back: __rc=24, __l=register 1, d = __l bytes of data

Data was received in NOEDIT mode. This is the same as return code=0, but in NOEDIT mode.

When TGET returns, if register 1 exceeds 32767, __error = 0xB1, __rc = -1, __l = 0, and no data is passed back. This error should not occur.

28(1C)

Pass back: __rc=28, __l=register 1, d = __l bytes of data

This TSO/E return code occurs when not all the available data fits in the buffers passed to TGET. This is not an error. The next TGET will obtain more of the available data.

When TGET returns, if register 1 exceeds 32767, __error = 0xB1, __rc = -1, __l = 0, and no data is passed back. This error should not occur.

other Pass back: __rc=other, __l=0 -- (This error should not occur.)

- TPG

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (No TSO buffers)
8(8)	Pass back: __rc=8, __l=0 -- (Attention occurred) **Note:** if ENDPASSTHROUGH(ATTN) was specified on the TSO/E OMVS command, TSO/3270 passthrough mode will end (see "the ENDPASSTHROUGH key" later in this appendix).
12(C)	Pass back: __rc=12, __l=0 -- (Invalid parameters; this error should not occur.)
20(14)	Pass back: __rc=20, __l=0 -- (Terminal logged off. OMVS ends after TSO/3270 passthrough mode ends.)
other	Pass back: __rc=other, __l=0 -- (This error should not occur.)

- TPUT

r/c	**OMVS processing**
0(0)	Pass back: __rc=0, __l=0
4(4)	Pass back: __rc=4, __l=0 -- (No TSO buffers)
8(8)	Pass back: __rc=8, __l=0 -- (Attention occurred)

Note: if ENDPASSTHROUGH(ATTN) was specified on the TSO/E OMVS command, TSO/3270 passthrough mode will end (see "The ENDPASSTHROUGH key" later in this appendix).

12(C) Pass back: __rc=12, __l=0 -- (ASID not receiving. This error should not occur.)

16(10) Pass back: __rc=16, __l=0 -- (Invalid parameters. This error should not occur.)

20(14) Pass back: __rc=20, __l=0 -- (Terminal logged off. OMVS will end after TSO/3270 passthrough mode ends.)

24(18) Pass back: __rc=24, __l=0 -- (Send not permitted. This error should not occur.)

28(1C)
Pass back: __rc=28, __l=0 -- (Receiver insecure. This error should not occur.)

32(20) Pass back: __rc=32, __l=0 -- (Not enough TSO storage)

other Pass back: __rc=other, __l=0 -- (This error should not occur.)

TSO/3270 Passthrough Mode Data Stream

User applications can optionally include the **fomth32p.h** header file to map the TSO/3270 passthrough mode data stream.

```
/*****START OF SPECIFICATIONS*******************************************
*
*    $MAC (fomth32p.h) COMP(SCPX4) PROD(FOM):
*
*01* macro NAME: fomth32p.h
*
*01* DSECT NAME: fomth32p.h
*
*01* DESCRIPTIVE NAME: TSO/3270 passthrough mode data stream
*
*02*   ACRONYM: fomth32p.h
*                                                                     */
 /*01* PROPRIETARY STATEMENT=                                         */
 /***PROPRIETARY_STATEMENT********************************************/
 /*                                                                   */
 /*                                                                   */
 /* LICENSED MATERIALS - PROPERTY OF IBM                              */
 /* THIS macro IS "RESTRICTED MATERIALS OF IBM"                       */
 /* 5655-068 (C) COPYRIGHT IBM CORP. 1995                             */
 /*                                                                   */
 /* STATUS= HOT1130                                                   */
 /*                                                                   */
 /***END_OF_PROPRIETARY_STATEMENT*************************************/
/*
*01* EXTERNAL CLASSIFICATION:  GUPI
*01* END OF EXTERNAL CLASSIFICATION:
*
*
*01* FUNCTION:
*
*    This header file defines structures and manifest constants used
*    to map the TSO/3270 passthrough mode data stream.
*
*
*01* METHOD OF ACCESS:
*
*02*   C/370:
*
```

```
*      - #include <fomth32p> -or-
*      - #include "fomth32p"
*
*      - When sending data, establish a pointer of type
*        __tso3270_request_t* to the start of the data area.
*
*        Fill in the various request parameters using the fields in
*        __tso3270_request_s.
*
*      - After receiving data from the TTY, determine where
*        in the data the next response starts.  If there is no
*        leftover (unread) data from a prior response, the new
*        response will begin at the start of the read-in data.
*        Establish a pointer of type __tso3270_response_t* to the start
*        of the response, and use the fields in the
*        __tso3270_response_s structure to look at the received data.
*

**********************************************************************/
*   General layout for TSO/3270 passthrough mode requests           */
**********************************************************************/

typedef struct __tso3270_request_s
{
   unsigned char   __ff;              /*(+00) introductory byte --
                                         must be 0xFF              */

   unsigned char   __fcn;             /*(+01) Function code -- for
                                         values, see below         */

   unsigned char   __p1;              /*(+02) First parameter byte --
                                         contents depend on which
                                         function is being
                                         requested                 */

   unsigned char   __p2;              /*(+03) Second parameter byte --
                                         contents depend on which
                                         function is being requested
                                                                   */
   int             __p3;              /*(+04) Third parameter --
                                         contents depend on which
                                         function is being requested
                                                                   */

   unsigned int    __l;               /*(+08) number of bytes of data
                                         that follow -- can be 0,
                                         meaning no data           */

   char            __d[1]              /*(+0C) variable number
                                         of data bytes -- can be 0  */

} __tso3270_request_t;


#define _TSO3270_REQH_L    12U        /* Length of request when there
                                         is no data                */


/*********************************************************************
*
*   General Layout for TSO/3270 passthrough mode responses
*   -------------------------------------------------------
*
**********************************************************************/

typedef struct __tso3270_response_s
{
   unsigned char   __fe;              /*(+00) introductory byte --
                                         will be 0xFE              */

   unsigned char   __fcn;             /*(+01) Function code --
                                         normally echoed back from
                                         the original request -- if
                                         __error = 0xC1, __fcn will
                                         be 0x00                   */

   unsigned char   __error;           /*(+02) error code from the
                                         TSO/E OMVS command itself --
                                         0 means no error from OMVS,
                                         and that the requested TSO/E
                                         service was invoked.  That
                                         service completed (or failed)
                                         with the return code in the
                                         __rc field, below         */

   char            __r0;              /*(+03) (reserved)           */

   int             __rc;              /*(+04) Return code (usually
                                         from register 15) from the
                                         TSO/E service invoked -- if
                                         OMVS itself detected an
                                         error, __rc will be set to
                                         -1                        */


   unsigned int    __l;               /*(+08) number of bytes of data
                                         that follow -- can be 0,
                                         meaning no data           */

   char            __[1]               /*(+0C) variable number
                                         of data bytes -- can be 0  */

} __tso3270_response_t;


#define _TSO3270_RSPH_L    12U        /* Length of response when
                                         there is no data          */


/***********************************************************************
 *
 *
 *   General constants in TSO/3270 passthrough mode requests/responses
 *   ==================================================================
 *
 *
 **********************************************************************/

#define _TSO3270_FF    0xFFU           /* 0xFF value used as the
                                         introductory byte of each
                                         request                   */

#define _TSO3270_FE    0xFEU          /* 0xFE value used as the
                                         introductory byte of each
                                         response                  */

#define _TSO3270_L0    0U             /* __l = 0, in a request or
                                         response means the there is
                                         no data following the length
                                         field                     */

#define _TSO3270_LMAX  32767U         /* __l = 32767, which is the
                                         maximum allowed data length
                                                                   */
```

```
#define _TSO3270_RCBAD (-1)            /* __rc = -1 -- indicates that
                                          OMVS found an error and that
                                          the requested TSO/E function
                                          was not invoked            */


/*------------------------------------------------------------------
 *
 *   TSO/3270 passthrough mode function codes for requests and responses
 *
 *------------------------------------------------------------------*/

#define _TSO3270_TPUT      0x11U       /* function code for TPUT
                                          request and response       */

#define _TSO3270_TGET      0x12U       /* function code for TGET
                                          request and response       */

#define _TSO3270_TPG       0x13U       /* function code for TPG
                                          request and response       */

#define _TSO3270_GTDEVSIZ  0x21U       /* function code for GTDEVSIZ
                                          request and response       */

#define _TSO3270_GTSIZE    0x22U       /* function code for GTSIZE
                                          request and response       */

#define _TSO3270_GTTERM    0x23U       /* function code for GTTERM
                                          request and response       */

#define _TSO3270_STFSMODE  0x31U       /* function code for STFSMODE
                                          request and response       */

#define _TSO3270_STLINENO  0x32U       /* function code for STLINENO
                                          request and response       */

#define _TSO3270_STTMPMD   0x33U       /* function code for STTMPMD
                                          request and response       */

#define _TSO3270_STCOM     0x34U       /* function code for STCOM
                                          request and response       */

#define _TSO3270_STSIZE    0x35U       /* function code for STSIZE
                                          request and response       */

#define _TSO3270_TCLEARQ   0x41U       /* function code for TCLEARQ
                                          request and response       */

#define _TSO3270_IKJEFTSR  0x51U       /* function code for IKJEFTSR
                                          request and response   @D1A*/



/*------------------------------------------------------------------
 *
 *   __error field in TSO/3270 passthrough mode responses
 *
 *    note: When an error or severe error code is set in the __error
 *          field, __rc is set to -1.  If a warning code is set in
 *          the _error field, the __rc field is set to the /return code from
 *          the invoked TSO/E service.
 *
 *    note: if more than one error exists for a given TSO/3270 request,
 *          the TSO/E OMVS command will generally report the first
 *          error found.  This will usually be the first applicable
 *          error in the following list:
```

```
 *
 *------------------------------------------------------------------*/


#define _TSO3270_ERROR_OK      0x00U   /* error = 0: no error found by
                                          TSO/E OMVS command before
                                          the TSO/E service was invoked
                                          -- means the service was
                                          invoked, but errors may have
                                          been reported by the invoked
                                          TSO/E service -- also, no
                                          OMVS-detected warning
                                          condition were found       */


/*
 *  Severe errors -- TSO/3270 connection is ended
 *  =============    ----------------------------
 *
 *  The TSO/3270 request is not processed.  The response is written
 *  back on the TTY.  TSO/3270 passthrough mode is ended, and the TTY
 *  is placed back in normal operational mode.  The rest of the
 *  request data is then treated as normal session data and is written
 *  into the output area of the shell session.
 */

#define _TSO3270_ERROR_NOTFF   0xC1U   /* __ff field was not 0xFF,
                                          probably indicating that the
                                          __l field in the prior
                                          request was incorrect, or
                                          that too little or too much
                                          data was present in the
                                          prior request.             */

#define _TSO3270_ERROR_TOOLONG 0xC2U   /* __l field is too long. The
                                          data length must be always
                                          be 32767 or less for any
                                          request                    */


/*
 *  Errors -- TSO/3270 request is not passed to TSO/E
 *  ======    ---------------------------------------
 *
 *  The complete request and __l bytes of data in the __d field are
 *  read and flushed.  The TSO/E OMVS command then looks for the 0xFF
 *  byte at the start of the next request.
 *
 */


#define _TSO3270_ERROR_UNKFCN  0x81U   /* __fcn field contains an
                                          unknown Function code      */


#define _TSO3270_ERROR_LNOT0   0x82U   /* __l field was not zero, as
                                          required for this TSO/3270
                                          request                    */

#define _TSO3270_ERROR_P1BAD   0x91U   /* __p1 field contains an
                                          incorrect value for this
                                          TSO/3270 passthrough mode
                                          request                    */

#define _TSO3270_ERROR_P2BAD   0x92U   /* __p2 field contains an
                                          incorrect value for this
                                          TSO/3270 passthrough mode
                                          request                   */


#define _TSO3270_ERROR_P3BAD   0x93U   /* __p3 field contains an
                                          incorrect value for this
                                          TSO/3270 passthrough mode
                                          request                   */

#define _TSO3270_ERROR_NOSTG   0xA1U   /* There is not enough storage
                                          to invoke the requested TSO/E
                                          service -- this error
                                          can also cause the TSO/E OMVS
                                          command to end suddenly    */

/*
 *  Errors -- TSO/3270 request was passed to TSO/E, data passback is
 *  ======    suppressed
 *            -----------------------------------------------------
 *
 *  The TSO/E service was invoked, but a proper TSO/3270 response
 *  cannot be passed back.
 *
 */


#define _TSO3270_ERROR_R1LONG  0xB1U   /* After TGET, data length in
                                          register 1 was more than
                                          32767 bytes, or was longer
                                          than the OMVS TGET buffer
                                          size. (Note: the OMVS TGET
                                          buffer can be longer than
                                          the __p3 (length) field of
                                          the current TGET request)  */


/*
 *  Warnings - TSO/3270 passthrough mode request is passed to TSO/E
 *  ========   ---------------------------------------------------
 *
 *  note: The warning code is placed in the response along with the
 *        /return code and any other passed back data from the invoked TSO/E
 *        service.  The warning condition should not affect the
 *        results from the invoked TSO/E service.
 */


#define _TSO3270_ERROR_P1NOT0  0x41U   /* __p1 field should be 0 for
                                          this TSO/3270 request, but
                                          was not                   */

#define _TSO3270_ERROR_P2NOT0  0x42U   /* __p2 field should be 0 for
                                          this TSO/3270 request, but
                                          was not                   */

#define _TSO3270_ERROR_P3NOT0  0x43U   /* __p3 field should be 0 for
                                          this TSO/3270 request, but
                                          was not                   */

#define _TSO3270_ERROR_P1RES1  0x49U   /* one or more reserved bits
                                          in the __p1 field are not
                                          zero, as they should be    */

#define _TSO3270_ERROR_P2RES1  0x4AU   /* one or more reserved bits
                                          in the __p2 field are not
                                          zero, as they should be    */
```

```
/*******************************************************************
 *
 *
 *   TSO/3270 Passthrough requests and responses
 *   ===========================================
 *
 *
 *******************************************************************/


/*-------------------------------------------------------------------
 *
 * GTDEVSIZ request layout
 * -----------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_GTDEVSIZ
 *
 * __p1  = 0
 * __p2  = 0
 * __p3  = 0
 *
 * __l   = 0 (no data is passed to GTDEVSIZ)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *------------------------------------------------------------------*/


/*-------------------------------------------------------------------
 *
 * GTDEVSIZ response layout
 * ------------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_GTDEVSIZ
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from GTDEVSIZ in register 15
 *           (-1 is set if error prevented the call to GTDEVSIZ)
 *
 * __l     = 0 -- if error prevented GTDEVSIZ  from being invoked
 *           8 -- if GTDEVSIZ was invoked
 *
 * __d     = not present, if GTDEVSIZ was not invoked (__l = 0)
 *           Otherwise, 8 bytes of data from GTDEVSIZ:
 *
 *           Bytes 0-3 are register 0 (lines on screen, or 0)
 *           Bytes 4-7 are register 1 (characters per line)
 *
 *
 *------------------------------------------------------------------*/

/*
 *  Layout of returned terminal size from GTDEVSIZ
 *  ----------------------------------------------
 */
 typedef struct __tso3270_gtdevsiz_data_s
 {
   unsigned int   __reg0;                /* value returned in register
                                            0 from GTDEVSIZ         */
   unsigned int   __reg1;                /* value returned in register
                                            1 from GTDEVSIZ         */
 } __tso3270_gtdevsiz_data_t;


/*
 *  GTDEVSIZ return codes in __rc field
 *  ---------------------------------
 */

#define _TSO3270_GTDEVSIZ_RC_OK    0   /* GTDEVSIZ successful        */
#define _TSO3270_GTDEVSIZ_RC_PARM  4   /* Unwanted parm present      */


/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 *
 * macros to extract information from GTDEVSIZ reponse
 * ---------------------------------------------------
 *
 *
 * _TSO3270_GTDEVSIZ_LENGTH(p)    -- extract logical screen length from
 *                                   GTDEVSIZ response (register 0)
 *
 *                                   returned type = unsigned int
 *
 *
 * _TSO3270_GTDEVSIZ_LINESIZE(p)  -- extract logical line size from
 *                                   GTDEVSIZ response (register 1)
 *
 *                                   returned type = unsigned int
 *
 *
 *
 * notes: "p" is the address of the start of a TSO/3270 passthrough
 *        response from a successful invocation of GTSIZE. "p" must
 *        point to the 12-byte response header, not the data (__d)
 *        field.  The entire 20-byte response from GTSIZE is assumed
 *        to be present in a contiguous area starting at "p".
 *
 *        "p" must be castable to type (void *)
 *
 *
 *
 *
 * Example:
 * =======
 *
 * char read_buf[...]
 * size_t screen_area ...
 * ...
 * ... issue GTDEVSIZ request using write() ...
 * ... do read() to get response from GTDEVSIZ into read_buf ...
 * ...
 *
 * screen_area = (size_t)_TSO3270_GTDEVSIZ_LENGTH(read_buf) *
 *               (size_t)_TSO3270_GTDEVSIZ_LINESIZE(read_buf);
 *
 *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

#define _TSO3270_GTDEVSIZ_LENGTH(p)                         /* @D1A*/\
(                                                           /* @D1A*/\
 (                                                          /* @D1A*/\
  (__tso3270_gtdevsiz_data_t *)(void *)                     /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)              /* @D1A*/\
 )                                                          /* @D1A*/\
 ->__reg0                                                   /* @D1A*/\
)                                                           /* @D1A*/


#define _TSO3270_GTDEVSIZ_LINESIZE(p)                       /* @D1A*/\
(                                                           /* @D1A*/\
 (                                                          /* @D1A*/\
  (__tso3270_gtdevsiz_data_t *)(void *)                     /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)              /* @D1A*/\
 )                                                          /* @D1A*/\
 ->__reg1                                                   /* @D1A*/\
)                                                           /* @D1A*/



/*---------------------------------------------------------------------
 *
 * GTSIZE request layout
 * ---------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_GTSIZE
 *
 * __p1  = 0
 * __p2  = 0
 * __p3  = 0
 *
 * __l   = 0 (no data is passed to GTSIZE)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-------------------------------------------------------------------*/


/*---------------------------------------------------------------------
 *
 * GTSIZE response layout
 * ----------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_GTSIZE
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from GTSIZE in register 15
 *           (-1 is set if error prevented the call to GTSIZE)
 *
 * __l     = 0 -- if error prevented GTSIZE from being invoked
 *           8 -- if GTSIZE was invoked
 *
 * __d     = not present, if GTSIZE was not invoked (__l = 0)
 *           Otherwise, 8 bytes of data from GTSIZE:
 *
 *           Bytes 0-3 are register 0 (lines on screen, or 0)
 *           Bytes 4-7 are register 1 (characters per line)
 *
 *
 *-------------------------------------------------------------------*/
```

```
/*
 *  Layout of returned terminal size data from GTSIZE
 *  ------------------------------------------------
 */

 typedef struct __tso3270_gtsize_data_s
 {
   unsigned int   __reg0;                /* value returned in register
                                            0 from GTSIZE           */
   unsigned int   __reg1;                /* value returned in register
                                            1 from GTSIZE           */
 } __tso3270_gtsize_data_t;


/*
 *  GTSIZE return codes in __rc field
 *  ---------------------------------
 */


#define _TSO3270_GTSIZE_RC_OK     0    /* GTSIZE successful          */
#define _TSO3270_GTSIZE_RC_PARM   4    /* Unwanted parm present      */


/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 *
 * macros to extract information from GTSIZE response
 * -------------------------------------------------
 *
 *
 * _TSO3270_GTSIZE_LENGTH(p)      -- extract screen length from
 *                                   GTSIZE response (register 0)
 *
 *                                   returned type = unsigned int
 *
 *
 * _TSO3270_GTSIZE_LINESIZE(p)    -- extract line size from GTSIZE
 *                                   response (register 1)
 *
 *                                   returned type = unsigned int
 *
 *
 *
 * notes: "p" is the address of the start of a TSO/3270 passthrough
 *        response from a successful invocation of GTSIZE. "p" must
 *        point to the 12-byte response header, not the data (__d)
 *        field.  The entire 20-byte response from GTSIZE is assumed
 *        to be present in a contiguous area starting at "p".
 *
 *        "p" must be castable to type (void *)
 *
 *
 *
 *
 * Example:
 * =======
 *
 * char read_buf[...] ...
 * size_t screen_area ...
 * ...
 * ... issue GTSIZE request using write() ...
 * ... do read() to get response from GTSIZE into read_buf ...
 * ...
 *
 * screen_area = (size_t)_TSO3270_GTSIZE_LENGTH(read_buf) *
 *               (size_t)_TSO3270_GTSIZE_LINESIZE(read_buf);
 *
 *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

#define _TSO3270_GTSIZE_LENGTH(p)                              /* @D1A*/\
(                                                              /* @D1A*/\
 (                                                             /* @D1A*/\
  (__tso3270_gtsize_data_t *)(void *)                          /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)                 /* @D1A*/\
 )                                                             /* @D1A*/\
 ->__reg0                                                      /* @D1A*/\
)                                                              /* @D1A*/


#define _TSO3270_GTSIZE_LINESIZE(p)                            /* @D1A*/\
(                                                              /* @D1A*/\
 (                                                             /* @D1A*/\
  (__tso3270_gtsize_data_t *)(void *)                          /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)                 /* @D1A*/\
 )                                                             /* @D1A*/\
 ->__reg1                                                      /* @D1A*/\
)                                                              /* @D1A*/



/*--------------------------------------------------------------------
 *
 * GTTERM request layout
 * ---------------------
 *
 *  Note: The ALTSIZE= and ATTRIB= parameters are always set, so that
 *        GTTERM always returns the primary and alternate screen
 *        sizes and the terminal attributes.
 *
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_GTTERM
 *
 * __p1  = 0
 * __p2  = 0
 * __p3  = 0
 *
 * __l   = 0 (no data is passed to GTTERM)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-------------------------------------------------------------------*/


/*--------------------------------------------------------------------
 *
 * GTTERM response layout
 * ----------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_GTTERM
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from GTTERM in register 15
 *           (-1 is set if error prevented the call to GTTERM)
 *
 * __l     = 0 -- if error prevented GTTERM from being invoked
 *           8 -- if GTTERM was invoked
 *
 * __d     = not present, if GTTERM was not invoked (__l = 0)
 *           Otherwise, 8 bytes of data from GTTERM:
 *
 *           Byte 0    -- first byte of PRMSIZE (primary screen rows)
 *           Byte 1    -- second byte of PRMSIZE (primary screen cols)
 *           Byte 2    -- first byte of ALTSIZE (alternate screen rows)
 *           Byte 3    -- second byte of ALTSIZE (alternate screen col)
 *           Bytes 4-7 -- 4 bytes of ATTRIB field (terminal attributes)
 *
 *-----------------------------------------------------------------*/

/*
 * Layout of 2-byte screen size field from PRMSIZE or ALTSIZE
 * ---------------------------------------------------------
 */

typedef struct __tso3270_gtterm_size_s /* returned PRMSIZE/ALTSIZE   */
{
  unsigned char   __rows;              /* number of rows             */
  unsigned char   __columns;           /* number of columns          */

} __tso3270_gtterm_size_t;             /* defined type               */


/*
 * Layout of 4-byte terminal attributes (returned ATTRIB field)
 * -----------------------------------------------------------
 */

typedef struct __tso3270_gtterm_attr_s  /* returned ATTRIB field     */
{
  unsigned                     :8;      /* ATTRIB byte 0 (unused)    */

  unsigned        __dbcs       :1;      /* on if DBCS supported      */
  unsigned        __language   :7;      /* 7-bit language field      */

  unsigned                     :4;      /* ATTRIB byte 2 (unused)    */
  unsigned        __ascii_type :2;      /* ASCII-7 or ASCII-8 ID     */
  unsigned                     :2;      /* (unused)                  */

  unsigned                     :6;      /* ATTRIB byte 3 (unused)    */
  unsigned        __ascii      :1;      /* on if ASCII device        */
  unsigned        __query      :1;      /* on if Query supported     */

} __tso3270_gtterm_attr_t;              /* defined type              */


#define _TSO3270_GTTERM_DEFAULT  0U     /* American English (default)*/
#define _TSO3270_GTTERM_ENU      1U     /* American English          */
#define _TSO3270_GTTERM_KATAKANA 17U    /* Katakana                  */

#define _TSO3270_GTTERM_ASCII_7  0U     /* ASCII-7 device            */
#define _TSO3270_GTTERM_ASCII_8  1U     /* ASCII-8 device            */


/*
 * Layout of 8-byte combined data from GTTERM in the TSO3270 response
 * -----------------------------------------------------------------
 */

typedef struct __tso3270_gtterm_data_s    /* combined GTTERM output  */
{
  __tso3270_gtterm_size_t   __pri;        /* primary screen size
```

```
                                          (PRMSZE) from GTTERM    */

  __tso3270_gtterm_size_t   __alt;        /* alternate screen size
                                             (ALTSZE) from GTTERM    */

  __tso3270_gtterm_attr_t   __attr;       /* terminal attributes
                                             (ATTRIB) from GTTERM    */
} __tso3270_gtterm_data_t;                /* defined type            */


/*
 *  GTTERM return codes in __rc field
 *  --------------------------------
 */

#define _TSO3270_GTTERM_RC_OK      0   /* GTTERM successful          */
#define _TSO3270_GTTERM_RC_NODISP  8   /* Not a display              */
#define _TSO3270_GTTERM_RC_PARM   12   /* Parm list error            */


/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 *
 * macros to extract information from GTTERM response
 * --------------------------------------------------
 *
 *
 * _TSO3270_GTTERM_PRI_ROWS(p)    -- extract number of rows when
 *                                   primary screen size is active
 *
 *                                   returned type = unsigned char
 *
 *
 * _TSO3270_GTTERM_PRI_COLUMNS(p) -- extract number of columns when
 *                                   alternate screen size is active
 *
 *                                   returned type = unsigned char
 *
 *
 * _TSO3270_GTTERM_ALT_ROWS(p)    -- extract number of rows when
 *                                   alternate screen size is active
 *
 *                                   returned type = unsigned char
 *
 *
 * _TSO3270_GTTERM_ALT_COLUMNS(p) -- extract number of columns when
 *                                   alternate screen size is active
 *
 *                                   returned type = unsigned char
 *
 *
 * _TSO3270_GTTERM_ATTR(p)        -- extract terminal attributes
 *
 *                                   returned type =
 *                                             __tso3270_gtterm_attr_t
 *
 *
 * notes: "p" is the address of the start of a TSO/3270 passthrough
 *        response from a successful invocation of GTTERM. "p" must
 *        point to the 12-byte response header, not the data (__d)
 *        field.  The entire 20-byte response from GTTERM is assumed
 *        to be present in a contiguous area starting at "p".
 *
 *        "p" must be castable to type (void *)
 *
 *
 *
 *
 * Example:
 * =======
 *
 * char read_buf[...] ...
 * size_t size_1ry ...
 * int query_supported ...
 * int dbcs_supported ...
 * ...
 * ... issue GTTERM request using write() ...
 * ... do read() to get response from GTTERM into read_buf ...
 * ...
 *
 * size_1ry = (size_t)_TSO3270_GTTERM_PRI_ROWS(read_buf) *
 *            (size_t)_TSO3270_GTTERM_PRI_COLUMNS(read_buf);
 *
 * query_supported = (int)(_TSO3270_GTTERM_ATTR(read_buf).__query);
 * dbcs_supported  = (int)(_TSO3270_GTTERM_ATTR(read_buf).__dbcs);
 *
 *
 *- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/


#define _TSO3270_GTTERM_PRI_ROWS(p)                             /* @D1A*/\
(                                                               /* @D1A*/\
 (                                                              /* @D1A*/\
  (__tso3270_gtterm_data_t *)(void *)                           /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)                  /* @D1A*/\
 )                                                              /* @D1A*/\
 ->__pri.__rows                                                 /* @D1A*/\
)                                                               /* @D1A*/


#define _TSO3270_GTTERM_ALT_ROWS(p)                             /* @D1A*/\
(                                                               /* @D1A*/\
 (                                                              /* @D1A*/\
  (__tso3270_gtterm_data_t *)(void *)                           /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)                  /* @D1A*/\
 )                                                              /* @D1A*/\
 ->__alt.__rows                                                 /* @D1A*/\
)                                                               /* @D1A*/


#define _TSO3270_GTTERM_PRI_COLUMNS(p)                          /* @D1A*/\
(                                                               /* @D1A*/\
 (                                                              /* @D1A*/\
  (__tso3270_gtterm_data_t *)(void *)                           /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)                  /* @D1A*/\
 )                                                              /* @D1A*/\
 ->__pri.__columns                                              /* @D1A*/\
)                                                               /* @D1A*/

#define _TSO3270_GTTERM_ALT_COLUMNS(p)                          /* @D1A*/\
(                                                               /* @D1A*/\
 (                                                              /* @D1A*/\
  (__tso3270_gtterm_data_t *)(void *)                           /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)                  /* @D1A*/\
 )                                                              /* @D1A*/\
 ->__alt.__columns                                              /* @D1A*/\
)                                                               /* @D1A*/

#define _TSO3270_GTTERM_ATTR(p)                                 /* @D1A*/\
(                                                               /* @D1A*/\
 (                                                              /* @D1A*/\
  (__tso3270_gtterm_data_t *)(void *)                           /* @D1A*/\
  (((__tso3270_response_t *)(void *)(p))->__d)                  /* @D1A*/\
 )                                                              /* @D1A*/\
 ->__attr                                                       /* @D1A*/\
)                                                             /* @D1A*/


/*--------------------------------------------------------------------
 *
 * IKJEFTSR request layout
 * -----------------------
 *
 *  note: IKJEFTSR optional parameters 7, 8, and 9 are not used.
 *
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_IKJEFTSR
 *
 * __p1  = 0
 * __p2  = 0
 *
 * __p3  = IKJEFTSR parameter 1.  This field is passed through to
 *         IKJEFTSR with no checking.
 *
 * __l   = number of bytes of data to be passed to IKJEFTSR.  This must
 *         be less than 32768.  This length is passed to IKJEFTSR as
 *         parameter 3.
 *
 * __d   = data to be passed to IKJEFTSR in parameter 2.  This is the
 *         TSO command text string.
 *
 *-------------------------------------------------------------------*/


/* Values for IKJEFTSR parameter 1 (in __p3 field)                   */
/* ------------------------------                                    */

#define _TSO3270_IKJEFTSR_P3_AUTH    0x00000000 /* authorized    @D1A*/
#define _TSO3270_IKJEFTSR_P3_UNAUTH  0x00010000 /* unauthorized  @D1A*/
#define _TSO3270_IKJEFTSR_P3_NODUMP  0x00000000 /* no dump       @D1A*/
#define _TSO3270_IKJEFTSR_P3_DUMP    0x00000100 /* dump          @D1A*/
#define _TSO3270_IKJEFTSR_P3_COMMAND 0x00000001 /* Invoke cmd    @D1A*/
#define _TSO3270_IKJEFTSR_P3_PROGRAM 0x00000002 /* Invoke pgm    @D1A*/
#define _TSO3270_IKJEFTSR_P3_EITHER  0x00000005 /* Invoke either @D1A*/


/*--------------------------------------------------------------------
 *
 * IKJEFTSR response layout
 * ------------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_IKJEFTSR
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from IKJEFTSR in register 15
 *           (-1 is set if error prevented the call to IKJEFTSR)
 *
 * __l     = 0  -- If error prevented IKJEFTSR from being invoked
 *           12 -- IF IKJEFTSR was invoked
 *
 * __d     = not present, if IKJEFTSR was not invoked (__l = 0)
 *           Otherwise 12 bytes of data from IKJEFTSR
 *
 *           Bytes 0-3  -- Function return code from IKJEFTSR parameter 4
 *           Bytes 4-7  -- Reason code from IKJEFTSR parameter 5
 *           Bytes 8-12 -- abend code code from IKJEFTSR parameter 6
 *
 *---------------------------------------------------------------------*/


/* Layout of returned parameters 4, 5, and 6 from IKJEFTSR           */
/* ----------------------------------------------------------           */

typedef struct __tso3270_abend_s       /* combined abend code    @D1A*/
{                                      /*                        @D1A*/
   unsigned           __flags  :8;     /* abend flags from TCB   @D1A*/
   unsigned           __system :12;    /* system abend code      @D1A*/
   unsigned           __user   :12;    /* user abend code        @D1A*/

} __tso3270_abend_t;                   /*                        @D1A*/


typedef struct __tso3270_ikjeftsr_data_s /* IKJEFTSR results     @D1A*/
{                                      /*                        @D1A*/
  int                 __frc;           /* Return code (parm 4)   @D1A*/
  int                 __reason;        /* Reason code (parm 5)   @D1A*/
  __tso3270_abend_t   __abend;         /* abend code  (parm 6)   @D1A*/

} __tso3270_ikjeftsr_data_t;           /*                        @D1A*/


/* IKJEFTSR return codes (reg 15) in __rc field                      */
/* ---------------------------------------------                      */

#define _TSO3270_IKJEFTSR_RC_OK      0 /* Command Ran OK         @D1A*/
#define _TSO3270_IKJEFTSR_RC_CMDRC   4 /* Non-zero command return code   @D1A*/
#define _TSO3270_IKJEFTSR_RC_ATTN    8 /* ATTN ended the command @D1A*/
#define _TSO3270_IKJEFTSR_RC_abend  12 /* Command abended        @D1A*/
#define _TSO3270_IKJEFTSR_RC_BADADR 16 /* Bad address in parm    @D1A*/
#define _TSO3270_IKJEFTSR_RC_PARM   20 /* Parameter list error   @D1A*/
#define _TSO3270_IKJEFTSR_RC_ERROR  24 /* Unexpected error       @D1A*/
#define _TSO3270_IKJEFTSR_RC_31BIT  28 /* Unexpected 31-bit addr @D1A*/


/* Function return codes (parm 4) in __frc field                     */
/* ----------------------------------------------                     */

#define _TSO3270_IKJEFTSR_FR_NONE (-1) /* Fcn return code not filled in  @D1A*/
#define _TSO3270_IKJEFTSR_FR_OK      0 /* Normal return code             @D1A*/


/* IKJEFTSR reason codes (parm 5) in __reason field                  */
/* --------------------------------------------------                  */

#define _TSO3270_IKJEFTSR_R_NONE  (-1) /* Reason not filled in   @D1A*/
#define _TSO3270_IKJEFTSR_R_PLENGTH  4 /* Invalid Plist length   @D1A*/
#define _TSO3270_IKJEFTSR_R_BADFLG1  8 /* 1st flag byte non-zero @D1A*/
#define _TSO3270_IKJEFTSR_R_BADFLG4 12 /* 4th flag byte invalid  @D1A*/
#define _TSO3270_IKJEFTSR_R_PARM7   16 /* Unwanted 7th parameter @D1A*/
#define _TSO3270_IKJEFTSR_R_BADFLG3 20 /* 3rd flag byte invalid  @D1A*/
#define _TSO3270_IKJEFTSR_R_NOTSO   24 /* Not TSO/E environment  @D1A*/
#define _TSO3270_IKJEFTSR_R_TOOLONG 28 /* Text >32763 bytes long @D1A*/
#define _TSO3270_IKJEFTSR_R_BADADR7 32 /* Bad address in parm 7  @D1A*/
#define _TSO3270_IKJEFTSR_R_BADPRM7 36 /* Parm 7 is invalid      @D1A*/
#define _TSO3270_IKJEFTSR_R_NOFOUND 40 /* Command not found      @D1A*/
#define _TSO3270_IKJEFTSR_R_SYNTAX  44 /* Command syntax error   @D1A*/
#define _TSO3270_IKJEFTSR_R_PERCENT 48 /* CMD started with %     @D1A*/
#define _TSO3270_IKJEFTSR_R_BACKG   52 /* Unsupported backgd cmd @D1A*/
#define _TSO3270_IKJEFTSR_R_AUTHLIB 56 /* Not in auth library    @D1A*/
#define _TSO3270_IKJEFTSR_R_AUTH    60 /* Authorized command     @D1A*/
#define _TSO3270_IKJEFTSR_R_TOKEN   64 /* Invalid token          @D1A*/
```

```
#define _TSO3270_IKJEFTSR_R_ESTAE  204 /* ESTAE error           @D1A*/
#define _TSO3270_IKJEFTSR_R_STAX   208 /* STAX error            @D1A*/
#define _TSO3270_IKJEFTSR_R_PUTGET 212 /* PUTGET error          @D1A*/
#define _TSO3270_IKJEFTSR_R_SCAN   216 /* IKJSCAN error         @D1A*/
#define _TSO3270_IKJEFTSR_R_BLDL   220 /* BLDL error            @D1A*/
#define _TSO3270_IKJEFTSR_R_TBLS   224 /* IKJTBLS error         @D1A*/
#define _TSO3270_IKJEFTSR_R_ATTACH 228 /* ATTACH error          @D1A*/
#define _TSO3270_IKJEFTSR_R_LOAD   236 /* LOAD error            @D1A*/
#define _TSO3270_IKJEFTSR_R_LINK   240 /* LINK error            @D1A*/
#define _TSO3270_IKJEFTSR_R_IKJ441 244 /* IRXTVARS IKJCT441 err  @D1A*/
#define _TSO3270_IKJEFTSR_R_DMSRVA 248 /* IRXTVARS DMSRVA error  @D1A*/
#define _TSO3270_IKJEFTSR_R_CLENUP 252 /* IRXTVARS cleanup error @D1A*/
#define _TSO3270_IKJEFTSR_R_STACK  256 /* STACK error           @D1A*/
#define _TSO3270_IKJEFTSR_R_TERM   260 /* TMP termination       @D1A*/



/* abend codes (parm 6) in __abend field                           */
/* ----------------------------------------                        */

#define _TSO3270_IKJEFTSR_A_NONE  (-1) /* abend not filled in    @D1A*/




/*-------------------------------------------------------------------
 *
 * STCOM request layout
 * --------------------
 *
 *  Note: The TSO/E OMVS command does not save the STCOM setting when
 *        TSO/3270 passthrough mode is entered, nor does is restore or
 *        reset it when TSO/3270 passthrough mode ends.  Changes to the
 *        STCOM setting may persist after the TSO/3270 passthrough mode
 *        application has ended.
 *
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_STCOM
 *
 * __p1  = 0, or one of the following:
 *
 *         _TSO3270_STCOM_YES  - do STCOM YES (default)
 *         _TSO3270_STCOM_NO   - do STCOM NO
 *
 *
 * __p2  = 0
 *
 * __p3  = 0
 *
 * __l   = 0 (no data is passed to STCOM)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-----------------------------------------------------------------*/

#define _TSO3270_STCOM_YES     0U       /*  STCOM YES              */
#define _TSO3270_STCOM_NO      1U       /*  STCOM NO               */


/*-------------------------------------------------------------------
 *
 * STCOM response layout
 * ---------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_STCOM
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from STCOM in register 15
 *           (-1 is set if error prevented the call to STCOM)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *-------------------------------------------------------------------*/

/*
 *  STCOM return codes in __rc field
 *  --------------------------------
 */

#define _TSO3270_STCOM_RC_OK       0   /* STCOM successful           */
#define _TSO3270_STCOM_RC_PARM     4   /* Bad parameter              */




/*---------------------------------------------------------------------
 *
 * STFSMODE request layout
 * -----------------------
 *
 *  note: The STFSMODE RSHWKEY=n option is not supported.  The default
 *        reshow key, PA2, will be used.
 *
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_STFSMODE
 *
 * __p1  = 0 or any valid combination (added or logically ORed
 *         together) of the following 4 sets of options:
 *
 *         at most one of the following:
 *
 *         _TSO3270_STFSMODE_ON        - STFSMODE ON  (default)
 *         _TSO3270_STFSMODE_OFF       - STFSMODE OFF
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_STFSMODE_NOINITIAL - STFSMODE INITIAL=NO (default)
 *         _TSO3270_STFSMODE_INITIAL   - STFSMODE INITIAL=YES
 *
 *            note: _TSO3270_STFSMODE_INITIAL is ignored (by the
 *                  STFSMODE macro) if _TSO3270_STFSMODE_OFF is
 *                  specified.
 *
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_STFSMODE_EDIT      - STFSMODE NOEDIT=NO (default)
 *         _TSO3270_STFSMODE_NOEDIT    - STFSMODE NOEDIT=YES
 *
 *            note: _TSO3270_STFSMODE_NOEDIT is not allowed in
 *                  combination with _TSO3270_STSFMODE_OFF.  (The
 *                  STFSMODE macro does not allow this combination.)
 *
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_STFSMODE_NOPARTION - STFSMODE PARTION=NO (default)
 *         _TSO3270_STFSMODE_PARTION   - STFSMODE PARTION=YES
 *
 *
 *
 * __p2  = 0
 * __p3  = 0
 *
 * __l   = 0 (no data is passed to STFSMODE)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-------------------------------------------------------------------*/

#define _TSO3270_STFSMODE_ON        0x00  /* STFSMODE ON             */
#define _TSO3270_STFSMODE_OFF       0x01  /* STFSMODE OFF            */

#define _TSO3270_STFSMODE_NOINITIAL 0x00  /* STFSMODE INITIAL=NO     */
#define _TSO3270_STFSMODE_INITIAL   0x04  /* STFSMODE INITIAL=YES    */

#define _TSO3270_STFSMODE_EDIT      0x00  /* STFSMODE NOEDIT=NO      */
#define _TSO3270_STFSMODE_NOEDIT    0x02  /* STFSMODE NOEDIT=YES     */

#define _TSO3270_STFSMODE_NOPARTION 0x00  /* STFSMODE PARTION=NO     */
#define _TSO3270_STFSMODE_PARTION   0x08  /* STFSMODE PARTION=YES    */


/*
 *   Map of bit subfields in __p1 for STFSMODE request
 *   ------------------------------------------------
 */

typedef struct __tso3270_stfsmode_p1_s
{
  unsigned                 :4;            /* reserved               */
  unsigned    __partion    :1;            /* STFSMODE PARTION=YES    */
  unsigned    __initial    :1;            /* STFSMODE INITIAL=YES    */
  unsigned    __noedit     :1;            /* STFSMODE NOEDIT=YES     */
  unsigned    __off        :1;            /* STFSMODE OFF            */

} __tso3270_stfsmode_p1_t;


/*-------------------------------------------------------------------
 *
 * STFSMODE response layout
 * ------------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_STFSMODE
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from STFSMODE in register 15
 *           (-1 is set if error prevented the call to STFSMODE)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *-------------------------------------------------------------------*/

/*
 *  STFSMODE return codes in __rc field
 *  ----------------------------------
 */

#define _TSO3270_STFSMODE_RC_OK     0  /* STFSMODE successful        */
#define _TSO3270_STFSMODE_RC_PARM   4  /* Bad parameter              */
#define _TSO3270_STFSMODE_RC_NO3270 8  /* Not VTAM 3270 terminal     */




/*---------------------------------------------------------------------
 *
 * STLINENO request layout
 * -----------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_STLINENO
 *
 * __p1  = 0, or one of the following:
 *
 *         _TSO3270_STLINENO_OFF  - do STLINENO MODE=OFF (default)
 *         _TSO3270_STLINENO_ON   - do STLINENO MODE=ON
 *
 *
 * __p2  = 0
 *
 * __p3  = line number to pass to STLINENO in the LINE=nn option
 *
 * __l   = 0 (no data is passed to STLINENO)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-------------------------------------------------------------------*/

#define _TSO3270_STLINENO_OFF  0U       /*  STLINENO MODE=OFF        */
#define _TSO3270_STLINENO_ON   1U       /*  STLINENO MODE=ON         */


/*---------------------------------------------------------------------
 *
 * STLINENO response layout
 * ------------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_STLINENO
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from STLINENO in register 15
 *           (-1 is set if error prevented the call to STLINENO)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *-------------------------------------------------------------------*/

/*
 *  STLINENO return codes in __rc field
 *  -----------------------------------
 */

#define _TSO3270_STLINENO_RC_OK       0 /* STLINENO successful       */
#define _TSO3270_STLINENO_RC_PARM     4 /* Bad parameter             */
#define _TSO3270_STLINENO_RC_NODISP   8 /* Not display terminal      */
#define _TSO3270_STLINENO_RC_BADLINE 12 /* Bad line number           */




/*-------------------------------------------------------------------
 *
 * STSIZE request layout
 * ---------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_STSIZE
 *
 * __p1  = 0
 *
 * __p2  = logical line size to pass to STSIZE in the SIZE=nn operand
 *
 *         Note: This number is limited to 255 both by STSIZE
 *               and the 1-byte length of this field in the TSO/3270
 *               data stream.
 *
 * __p3  = number of lines to pass to STSIZE in the LINE=nn operand
 *
 * __l   = 0 (no data is passed to STSIZE)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-------------------------------------------------------------------*/


/*-------------------------------------------------------------------
 *
 * STSIZE response layout
 * ----------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_STSIZE
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from STSIZE in register 15
 *           (-1 is set if error prevented the call to STSIZE)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *-------------------------------------------------------------------*/

/*
 *  STSIZE return codes in __rc field
 *  --------------------------------
 */

#define _TSO3270_STSIZE_RC_OK      0   /* STSIZE successful          */
#define _TSO3270_STSIZE_RC_PARM    4   /* Invalid parm               */
#define _TSO3270_STSIZE_RC_VALUES  8   /* Invalid LINE/SIZE          */
#define _TSO3270_STSIZE_RC_SIZE   12   /* term size mismatch         */




/*-------------------------------------------------------------------
 *
 * STTMPMD request layout
 * ----------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_STTMPMD
 *
 * __p1  = 0, or any valid combination (added or logically ORed
 *         together) of the following 2 sets of options:
 *
 *         at most one of the following:
 *
 *         _TSO3270_STTMPMD_ON       - do STTMPMD ON   (default)
 *         _TSO3270_STTMPMD_OFF      - do STTMPMD OFF
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_STTMPMD_KEYSNO   - do STTMPMD KEYS=NO  (default)
 *         _TSO3270_STTMPMD_KEYSALL  - do STTMPMD KEYS=ALL
 *
 *
 *
 * __p2  = 0
 *
 * __p3  = 0
 *
 * __l   = 0 (no data is passed to STTMPMD)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-------------------------------------------------------------------*/

#define _TSO3270_STTMPMD_ON      0x00U  /* STTMPMD ON                */
#define _TSO3270_STTMPMD_OFF     0x01U  /* STTMPMD OFF               */

#define _TSO3270_STTMPMD_KEYSNO  0x00U  /* STTMPMD KEYS=NO           */
#define _TSO3270_STTMPMD_KEYSALL 0x02U  /* STTMPMD KEYS=ALL          */


/*
 *   Map of bit subfields in __p1 for STTMPMD request
 *   ----------------------------------------------
 */

typedef struct __tso3270_sttmpmd_p1_s
```

```
{
  unsigned                 :6;            /* reserved               */
  unsigned    __keysall    :1;            /* STTMPMD KEYS=ALL       */
  unsigned    __off        :1;            /* STTMPMD OFF            */

} __tso3270_sttmpmd_p1_t;


/*-----------------------------------------------------------------------
 *
 * STTMPMD response layout
 * -----------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_STTMPMD
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from STTMPMD in register 15
 *           (-1 is set if error prevented the call to STTMPMD)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *----------------------------------------------------------------------*/

/*
 *  STTMPMD return codes in __rc field
 *  ----------------------------------
 */

#define _TSO3270_STTMPMD_RC_OK     0   /* STTMPMD successful         */
#define _TSO3270_STTMPMD_RC_PARM   4   /* Bad parameter              */
#define _TSO3270_STTMPMD_RC_NODISP 8   /* Not display terminal       */




/*-----------------------------------------------------------------------
 *
 * TCLEARQ request layout
 * ----------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_TCLEARQ
 *
 * __p1  = 0, or one of the following:
 *
 *         _TSO3270_TCLEARQ_INPUT  - do TCLEARQ INPUT (default)
 *         _TSO3270_TCLEARQ_OUTPUT - do TCLEARQ OUTPUT
 *
 *
 * __p2  = 0
 *
 * __p3  = 0
 *
 * __l   = 0 (no data is passed to TCLEARQ)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *
 *-----------------------------------------------------------------*/

#define _TSO3270_TCLEARQ_INPUT  0U      /*  TCLEARQ INPUT            */
#define _TSO3270_TCLEARQ_OUTPUT 1U      /*  TCLEARQ OUTPUT           */


/*-----------------------------------------------------------------
 *
 * TCLEARQ response layout
 * -----------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_TCLEARQ
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from TCLEARQ in register 15
 *           (-1 is set if error prevented the call to TCLEARQ)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *-----------------------------------------------------------------*/

/*
 *  TCLEARQ return codes in __rc field
 *  ----------------------------------
 */

#define _TSO3270_TCLEARQ_RC_OK     0   /* TCLEARQ successful         */
#define _TSO3270_TCLEARQ_RC_PARM   4   /* Bad parameter              */




/*-----------------------------------------------------------------
 *
 * TGET request layout
 * -------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_TGET
 *
 *
 * __p1  = 0 or any valid combination (added or logically ORed
 *         together) of the following 2 sets of options:
 *
 *         at most one of the following:
 *
 *         _TSO3270_TGET_EDIT     - do TGET EDIT    (default)
 *         _TSO3270_TGET_ASIS     - do TGET ASIS
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_TGET_WAIT     - do TGET WAIT    (default)
 *         _TSO3270_TGET_NOWAIT   - do TGET NOWAIT
 *
 *
 *
 * __p2  = 0
 *
 *
 * __p3  = Buffer size used on the TGET request
 *
 *         Must be from 0 to 32767 bytes.
 *
 *
 * __l   = 0 (no data is passed to TGET)
 *
 * __d   = must not be present in the data stream.  The FF byte of the
 *         next TSO/3270 request must immediately follow the __l field
 *         from this request
 *
 *--------------------------------------------------------------------*/

#define _TSO3270_TGET_EDIT     0x00U   /* do TGET EDIT                */
#define _TSO3270_TGET_ASIS     0x01U   /* do TGET ASIS                */

#define _TSO3270_TGET_WAIT     0x00U   /* do TGET WAIT                */
#define _TSO3270_TGET_NOWAIT   0x10U   /* do TGET NOWAIT              */


/*
 *   Map of bit subfields in __p1 for TGET request
 *   --------------------------------------------
 */

typedef struct __tso3270_tget_p1_s
{
  unsigned                 :3;         /* reserved                    */
  unsigned    __nowait     :1;         /* TGET ,,NOWAIT               */
  unsigned                 :2;         /* reserved                    */
  unsigned    __edit       :2;         /* TGET ASIS/EDIT field        */

} __tso3270_tget_p1_t;


/*---------------------------------------------------------------------
 *
 * TGET response layout
 * -------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_TGET
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from TGET in register 15
 *           (-1 is set if error prevented the call to TGET)
 *
 * __l     = number of bytes of data returned by TGET (register 1)
 *
 * __d     = __l bytes of data from TGET
 *
 *           note: If TGET reports more than 32767 bytes, or more
 *                 data than the OMVS TGET buffer can hold, __error
 *                 is set to 0xB1, and none of the data is returned
 *                 (__l will be zero).
 *
 *                 Note that the OMVS TGET buffer size can exceed the
 *                 length in __p3, so OMVS may not always return an
 *                 error when __l exceeds __p3.
 *
 *                 (This error is not expected.  The TSO/E OMVS
 *                 can also end suddenly when this error occurs.)
 *
 *-------------------------------------------------------------------*/

/*
 *  TGET return codes in __rc field
 *  ------------------------------
 */

#define _TSO3270_TGET_RC_EDIT      0   /* EDIT/ASIS mode data        */
#define _TSO3270_TGET_RC_NOWAIT    4   /* NOWAIT - no data available */
#define _TSO3270_TGET_RC_ATTN      8   /* Attention occurred         */
#define _TSO3270_TGET_RC_MORE     12   /* More data than will fit    */
#define _TSO3270_TGET_RC_PARM     16   /* Bad parameters             */
#define _TSO3270_TGET_RC_LOGOFF   20   /* Terminal was logged off    */
#define _TSO3270_TGET_RC_NOEDIT   24   /* NOEDIT mode data           */
#define _TSO3270_TGET_RC_MORENOED 28   /* NOEDIT -- more data avail  */




/*-------------------------------------------------------------------
 *
 * TPG request layout
 * ------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_TPG
 *
 * __p1  = 0 or any valid combination (added or logically ORed
 *         together) of the following 3 sets of options:
 *
 *         at most one of the following:
 *
 *         _TSO3270_TPG_NOEDIT    - do TPG NOEDIT  (default)
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_TPG_WAIT      - do TPG WAIT    (default)
 *         _TSO3270_TPG_NOWAIT    - do TPG NOWAIT
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_TPG_NOHOLD    - do TPG NOHOLD  (default)
 *         _TSO3270_TPG_HOLD      - do TPG HOLD
 *
 *
 * __p2  = 0
 * __p3  = 0
 *
 * __l   = number of bytes of data to be passed to TPG.  This must be
 *         less than 32768.
 *
 * __d   = data to be passed to TPG.
 *
 *
 *-------------------------------------------------------------------*/

#define _TSO3270_TPG_NOEDIT     0x00U   /* do TPG NOEDIT             */

#define _TSO3270_TPG_WAIT       0x00U   /* do TPG WAIT               */
#define _TSO3270_TPG_NOWAIT     0x10U   /* do TPG NOWAIT             */

#define _TSO3270_TPG_NOHOLD     0x00U   /* do TPG NOHOLD             */
#define _TSO3270_TPG_HOLD       0x08U   /* do TPG HOLD               */
```

```
/*
 *   Map of bit subfields in __p1 for TPG request
 *   --------------------------------------------
 */

typedef struct __tso3270_tpg_p1_s
{
  unsigned                :3;          /* reserved                  */
  unsigned     __nowait   :1;          /* TPG ,,NOWAIT              */
  unsigned     __hold     :1;          /* TPG ,,,HOLD               */
  unsigned                :3;          /* reserved                  */

} __tso3270_tpg_p1_t;


/*-------------------------------------------------------------------
 *
 * TPG response layout
 * -------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_TPG
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from TPG in register 15
 *           (-1 is set if error prevented the call to TPG)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *------------------------------------------------------------------*/

/*
 *  TPG return codes in __rc field
 *  ------------------------------
 */

#define _TSO3270_TPG_RC_OK        0    /* TPG successful            */
#define _TSO3270_TPG_RC_NOWAIT    4    /* NOWAIT - no buffer avail.  */
#define _TSO3270_TPG_RC_ATTN      8    /* Attention occurred        */
#define _TSO3270_TPG_RC_PARM     16    /* Bad parameters            */
#define _TSO3270_TPG_RC_LOGOFF   20    /* Terminal was logged off    */




/*-------------------------------------------------------------------
 *
 * TPUT request layout
 * -------------------
 *
 * __ff  = 0xFF
 *
 * __fcn = _TSO3270_TPUT
 *
 * __p1  = 0 or any valid combination (added or logically ORed
 *         together) of the following 4 sets of options:
 *
 *         at most one of the following:
 *
 *         _TSO3270_TPUT_EDIT     - do TPUT EDIT    (default)
 *         _TSO3270_TPUT_NOEDIT   - do TPUT NOEDIT
 *         _TSO3270_TPUT_ASIS     - do TPUT ASIS
 *         _TSO3270_TPUT_CONTROL  - do TPUT CONTROL (may not be useful)
 *         _TSO3270_TPUT_FULLSCR  - do TPUT FULLSCR
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_TPUT_WAIT     - do TPUT WAIT    (default)
 *         _TSO3270_TPUT_NOWAIT   - do TPUT NOWAIT
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_TPUT_NOHOLD   - do TPUT NOHOLD  (default)
 *         _TSO3270_TPUT_HOLD     - do TPUT HOLD
 *
 *         combined with at most one of the following:
 *
 *         _TSO3270_TPUT_NOBREAK  - do TPUT NOBREAK (default)
 *         _TSO3270_TPUT_BREAKIN  - do TPUT BREAKIN
 *
 *
 * __p2  = 0
 * __p3  = 0
 *
 * __l   = number of bytes of data to be passed to TPUT.  This must be
 *         less than 32768.
 *
 * __d   = data to be passed to TPUT.  This includes all bytes needed
 *         by TPUT (including the initial 0x27 byte if doing TPUT
 *         FULLSCR)
 *
 *-------------------------------------------------------------------*/

#define _TSO3270_TPUT_EDIT     0x00U   /* do TPUT EDIT              */
#define _TSO3270_TPUT_NOEDIT   0x80U   /* do TPUT NOEDIT            */
#define _TSO3270_TPUT_ASIS     0x01U   /* do TPUT ASIS              */
#define _TSO3270_TPUT_CONTROL  0x02U   /* do TPUT CONTROL           */
#define _TSO3270_TPUT_FULLSCR  0x03U   /* do TPUT FULLSCR           */

#define _TSO3270_TPUT_WAIT     0x00U   /* do TPUT WAIT              */
#define _TSO3270_TPUT_NOWAIT   0x10U   /* do TPUT NOWAIT            */

#define _TSO3270_TPUT_NOHOLD   0x00U   /* do TPUT NOHOLD            */
#define _TSO3270_TPUT_HOLD     0x08U   /* do TPUT HOLD              */

#define _TSO3270_TPUT_NOBREAK  0x00U   /* do TPUT NOBREAK           */
#define _TSO3270_TPUT_BREAKIN  0x04U   /* do TPUT BREAKIN           */


/*
 *   Map of bit subfields in __p1 for TPUT request
 *   ---------------------------------------------
 */

typedef struct __tso3270_tput_p1_s
{
  unsigned    __noedit     :1;         /* TPUT ,NOEDIT              */
  unsigned                 :2;         /* reserved                  */
  unsigned    __nowait     :1;         /* TPUT ,,NOWAIT             */
  unsigned    __hold       :1;         /* TPUT ,,,HOLD              */
  unsigned    __breakin    :1;         /* TPUT ,,,,BREAKIN          */
  unsigned    __edit       :2;         /* EDIT/ASIS/CONTROL/FULLSCR */

} __tso3270_tput_p1_t;


/*-------------------------------------------------------------------
 *
 * TPUT response layout
 * --------------------
 *
 * __fe    = 0xFE
 *
 * __fcn   = _TSO3270_TPUT
 *
 * __error = 0, or one of the errors listed earlier
 *
 * __r0    = 0
 *
 * __rc    = Return code from TPUT in register 15
 *           (-1 is set if error prevented the call to TPUT)
 *
 * __l     = 0
 *
 * __d     = not present
 *
 *--------------------------------------------------------------------*/

/*
 *  TPUT return codes in __rc field
 *  -------------------------------
 */

#define _TSO3270_TPUT_RC_OK       0    /* TPUT successful            */
#define _TSO3270_TPUT_RC_NOWAIT   4    /* NOWAIT - no buffer avail.  */
#define _TSO3270_TPUT_RC_ATTN     8    /* Attention occurred         */
#define _TSO3270_TPUT_RC_ASID    12    /* ASID rejected message      */
#define _TSO3270_TPUT_RC_PARM    16    /* Bad parameters             */
#define _TSO3270_TPUT_RC_LOGOFF  20    /* Terminal was logged off    */
#define _TSO3270_TPUT_RC_NOAUTH  24    /* Not authorized to send     */
#define _TSO3270_TPUT_RC_SECURE  28    /* receiver not secure enough */
#define _TSO3270_TPUT_RC_NOSTG   32    /* No storage available       */


/*--------------------------------------------------------------------*/
```

Miscellaneous Programming Notes

When designing an application with the 3270 passthrough facility, consider the following:

- Checking for TSO/3270 passthrough mode support

  Use **isatty**(STDOUT_FILENO) to verify that a TTY is present, or open the controlling terminal. Then, use **tcgetattr**() to get a copy of termios. If the PKT3270 bit is set, TSO/3270 passthrough mode is supported.
- Redirecting STDERR (and STDOUT) using freopen()

  To prevent LE from writing error messages to the slave TTY while it is in TSO/3270 passthrough mode, STDERR_FILENO should be redirected to something other than the TTY before the TTY is placed in TSO/3270 passthrough mode. This will prevent any unexpected error messages from getting intermixed with the 3270 data stream being sent to the terminal. You may want to do the same thing with STDOUT_FILENO, and use some different file descriptor when writing the TSO/3270 passthrough mode requests.
- Setting TTY permissions

  You may want to change the permissions of the TTY before going into TSO/3270 passthrough mode. If the permissions are set to rwx------, no other users (except the superuser) should be able to **write()** messages to this TTY. These messages could get mixed up with the 3270 data stream being sent to the terminal. The program must be sure to reset the permissions back to normal, whether it ends normally or abnormally.

- Setting TSO/3270 passthrough mode

  Use **tcgetattr**() and **tcsetattr**() to set the TSO/3270 passthrough mode bit in termios. As mentioned earlier, it may be desirable to set the termios TOSTOP bit at the same time.

  When using **tcsetattr**, you may want to use the TCSADRAIN option, to wait for the TSO/E OMVS command to **read()** any queued non-3270 data first. Note that the OMVS command user may have switched to another session or may have escaped to TSO just before TSO/3270 passthrough mode was started. In these cases, the TCSADRAIN operation may take a long time. If there is any queued non-3270 data at the master TTY, and TSO/3270 passthrough mode is set without draining the queued output data, OMVS will detect a severe error on the first TSO/3270 passthrough mode request. OMVS will end TSO/3270 passthrough mode immediately. (The TSO/3270 passthrough mode application will get SIGWINCH, as usual.)

- Initial screen and TSO state

  After TSO/3270 passthrough mode is started, OMVS clears the screen before processing the first TSO/3270 passthrough mode request. Also, the TSO terminal is placed back into default state by issuing:
  - STAX DEFER=NO (to undefer attentions)
  - STFSMODE OFF (return to TSO line mode)
  - STTMPMD OFF (re-allow session manager)
  - TCLEARQ INPUT (to get rid of any queued-up input)

  If the TSO/3270 passthrough mode application wants to send fullscreen 3270 data to the terminal, it must use STFSMODE, STTMPMD, or STLINENO, as required to set up the TSO terminal before sending any data.

- Using TPUT FULLSCR

  When invoking TPUT FULLSCR via the TSO/3270 passthrough mode interface, make sure to include the initial 0x27 byte in the data stream. The TSO/E OMVS command does not automatically supply this byte.

- STCOM

  If STCOM is used to set the intercom on or off, the TSO/E OMVS command does not restore the initial setting when TSO/3270 passthrough mode ends. Any changes to the intercom status remain in effect until the next STCOM is done.

- Handling the TSO refresh AID

  The PA2 AID byte from the 3270 is the TSO refresh indication. The TSO/E OMVS command passes this refresh AID back to the TSO/3270 passthrough mode application.

  Note that the TSO/3270 passthrough mode interface does not allow the 3270 application to change the TSO refresh indication to a different AID byte, so it is always PA2.

- The CLEAR key

  If the TSO/3270 passthrough mode application does not request STTMPMD KEYS=ALL, it (and the OMVS command, too) will not know when the CLEAR key has erased the screen. This is a standard TSO feature, and the TSO/E OMVS command does nothing to shield the TSO/3270 passthrough mode application from it.

- Handling SIGWINCH

If an error causes OMVS to switch the TTY out of TSO/3270 passthrough mode, OMVS will send SIGWINCH to all members of the foreground process group. The 3270 application should establish a catcher for SIGWINCH. When SIGWINCH is received, the application should do **tcgetattr**() to see if the TTY is no longer in TSO/3270 passthrough mode. If not, the application should restore the TTY permissions, and perhaps write out error messages.

Of course, the application could also re-establish TSO/3270 passthrough mode, and completely re-establish TSO FULLSCREEN mode and repaint the screen.

- Handling SIGTTIN and SIGTTOU

  The default action for these signals is to stop the process. When the process is moved back into the foreground, SIGCONT should restart the process. However, the TTY may no longer be in TSO/3270 passthrough mode. To make a TSO/3270 passthrough mode application work properly, it may be necessary to catch SIGTTIN, SIGTTOU and maybe SIGCONT. After getting SIGTTIN or SIGTTOU, the application should either restore the TTY permissions and end, or wait for SIGCONT and then restore the TTY to TSO/3270 passthrough mode and repaint the screen.

- Running in the TSO/E address space

  In some cases, the 3270 application may run directly in the TSO/E address space. This would occur, for example, if the OMVS SHAREAS option was used (or defaulted), and the _BPX_SHAREAS environment variable was set. In other cases, the TSO/3270 passthrough mode application will run in a non-TSO address space. In this case, the only access to the TSO terminal is through the TSO/3270 passthrough mode data stream.

- Handling sudden TSO/E LOGOFF and related errors

  The TSO/E OMVS command may close the master end of the TTY when certain errors occur. When this occurs, the 3270 application should close the slave TTY. TTY cleanup is not required (nor is it usually possible).

- Cleaning up

  Usually, the TSO/3270 passthrough mode application should restore the TTY to the same state that it was in before TSO/3270 passthrough mode was entered. This would include restoring the TTY file permissions. If the TSO/3270 passthrough mode application ends while in background mode (after getting SIGTTOU/SIGTTIN) **tcsetattr**() cannot be used to restore termios. The file permissions can still be restored, however.

  Usually, the TSO/3270 passthrough mode application should catch all possible signals (like SIGILL and SIGSEGV) and should restore the TTY to its original state before ending.

The ENDPASSTHROUGH Key

TSO/3270 passthrough mode supports the ENDPASSTHROUGH option on the TSO/E OMVS command.

```
OMVS   other options

              .
              .
              .

 ENDPASSTHROUGH(ATTN|CLEAR|ENTER|NO|PA1|PA3|PF1|PF2|PF3 ... PF24|
                CLEARPARTITION|SEL)
              .
              .
              .
```

This option sets up a 3270 key that allows the user to break out of TSO/3270 passthrough mode and return to the shell session. Pressing this key will often end the 3270 application, but some 3270 applications might not give up control of the terminal. Using this option prevents the 3270 application from seeing the specified key when it is pressed. When the selected key is pressed, the OMVS command gives the user no second chance to retract it.

This key may be used during 3270 application development, to set up a *panic button* that can be used end certain application hangs. This ENDPASSTHROUGH key may not work if the keyboard is locked, or if the 3270 application unlocks the keyboard but reads no input data. The ENDPASSTHROUGH key might also not work if the application causes the 3270 to send in structured fields (from explicit created partitions).

The default is ENDPASSTHROUGH(NO), which means that all 3270 keys can be used by the 3270 application and there is no breakout 3270 key. When some other key is specified, the 3270 application cannot use that key. The selected key should be one that is not used by the application. The possible keys are:

ATTN The 3270 Attention key

In some 3270 applications this key may be changed to PA1 before it is seen by the TSO/E OMVS command. If so, OMVS will never see the Attention key; specify PA1 instead of ATTN.

With some terminal connections, the ATTN key may not be available.

CLEARPARTITION
The 3270 Clear Partition key

This key is only effective if the application is using explicit 3270 partitions.

CLEAR
The 3270 CLEAR key

In some 3270 applications the CLEAR key may not be seen by the TSO/E OMVS command when it is pressed. If so, OMVS will never break out of TSO/3270 passthrough mode. Specify some other 3270 key.

ENTER
The 3270 ENTER key

This key is useful only if the 3270 application is completely driven by PF/PA keys.

NO No breakout key (default)

PA1 The 3270 PA1 or Program Attention 1 key

Note that for some 3270 applications, the Attention (ATTN) keypress may be converted to the PA1 keypress before being passed to the TSO/E OMVS command.

PA3 The 3270 PA3 or Program Attention 3 key

The PA3 key may not be available on some keyboards.

PFn 3270 Function keys 1, 2, 3 ... 8, 9

PFnn 3270 Function keys 10, 11, 12, 13 ... 23, 24

SEL 3270 Cursor Select or light pen

This key is useful only when the 3270 application creates light-pen-selectable fields on the 3270 screen.

If some 3270 key other than Attention is specified in the OMVS ENDPASSTHROUGH option, the key will be recognized only when TGET is issued. In TSO/3270 passthrough mode, OMVS does TGET only when requested by the user 3270 application. After each TGET, the TSO/E OMVS command will check the incoming AID byte (if the incoming data stream is just normal 3270 data that the TSO/E OMVS command can understand) for the ENDPASSTHROUGH key. If the ENDPASSTHROUGH key was pressed, OMVS will end TSO/3270 passthrough mode (sending SIGWINCH as usual), and will return the display to the shell session.

If the TGET buffer length is very short, OMVS may miss ENDPASS AID bytes that are received inside inbound structured fields. This occurs if the AID byte is not received with the first TGET for this inbound 3270 data stream.

Use of TCLEARQ INPUT may cause OMVS to miss ENDPASSTHROUGH AID bytes from the 3270 (other than ATTN), if the inbound 3270 data is cleared from the input queue before TGET is done to receive it.

Pressing the ENDPASS key as an application is abending may prevent the usual LE/370 abend message from appearing on the screen.

The Attention key can be used as a more effective ENDPASSTHROUGH key, because (under certain circumstances) TGET is not required to find out that the Attention key has been pressed.

If the TSO/3270 passthrough mode application requests STTMPMD ON,KEYS=ALL, TSO changes the incoming attention AID to PA1 before the OMVS command sees it. In this case, specifying OMVS ENDPASSTHROUGH(ATTN) does not provide the user with a working ENDPASSTHROUGH key. OMVS ENDPASSTHROUGH(PA1) can be used, but the ENDPASSTHROUGH key will be effective only when TGET requests are done. Also, STTMPMD ON,KEYS=NO causes TSO to suppress incoming CLEAR key AID bytes, so ENDPASS(CLEAR) will not work in this case.

If STTMPMD KEYS=ALL is not done, the Attention key may function normally (if the terminal connection provides a real Attention key). However, attentions are deferred across all kernel SYSCALLs that OMVS makes. If OMVS ENDPASSTHROUGH(ATTN) is specified and STTMPMD KEYS=ALL is not issued, OMVS will not issue kernel SYSCALLs that wait forever to get the next output data from the TSO/3270 passthrough mode application. Instead, OMVS will periodically wake up to check for a TSO attention. If the user has caused attention, OMVS will end TSO/3270 passthrough mode (sending SIGWINCH, as usual), and will return the terminal to the shell session.

ENDPASSTHROUGH Specification Results

The following chart shows what happens when you press various keys that were specified on the **ENDPASSTHROUGH** option of the OMVS command.

```
*************************************************************************
*
*
*            +----------------------------------------------------+
*            |                                                    |
*            |             TSO Programming Environment            |
*            |                                                    |
*            +-------------------------+--------------------------+
*            |                         |                          |
*            | Unedited input          | Edited input             |
*            | ..............          | ............             |
*            |                         |                         |
*            | - after TGET ASIS       | - after TGET EDIT with  |
*            | - after TGET EDIT, with |   STFSMODE OFF          |
*            |   STFSMODE ON,NOEDIT    |   or                    |
*            |                         |   STFSMODE ON,EDIT      |
*       :    +------------+------------+------------+------------+
* OMVS  :Press|            |            |            |            |
* ENDPASS:this|  STTMPMD   |  STTMPMD   |  STTMPMD   |  STTMPMD   |
* operand: key|  KEYS=ALL  |  KEYS=NO   |  KEYS=NO   |  KEYS=ALL  |
*       :    |            |            |            |            |
*============+============+============+============+============+
*            |Action      |       No effect         | No effect  |
*            |only after  |                         |            |
*   CLEAR    |TGET returns| (CLEAR is absorbed by   | (CLEAR     |
*            |simple I/B  | TSO)                    | AID byte   |
*            |data or     |                         | not        |
*            |unsplit I/B |                         | returned   |
*            |SF          |                         | from TGET) |
*            |(only if KB |                         |            |
*            |unlocked)   |                         |            |
*-----+------+------------+------------+------------+------------+
*     :      |Action      |                         |            |
*     : press|only after  |                         |            |
*     : <PA1>|TGET returns|                         |            |
*     :      |simple I/B  |                         |            |
*     :      |data or     |                         | No effect  |
*     :      |unsplit I/B |                         |            |
*     :      |SF          |       No effect         |            |
*     :      |(only if KB |                         | (PA1 AID   |
*     :      |unlocked)   | (appears as ATTN -- use | byte not   |
* PA1 +------+------------+ ENDPASS(ATTN))          + returned   +
*     :      |Action      |                         | from TGET) |
*     :  press|only after |                         |            |
*     : <ATTN>|TGET returns|                        |            |
*     :      |simple I/B  |                         |            |
*     :      |data or     |                         |            |
*     :      |unsplit I/B |                         |            |
*     :      |SF          |                         |            |
*-----+------+------------+------------+------------+------------+
*     : press|            | After TGET/TPG/TPUT     |            |
*     : <PA1>| No effect  | (rc=8), maybe in-       | No effect  |
*     :      |            | between.  Effective     |            |
*     :      | (appears   | only if keyboard is     | (PA1 AID   |
*     :      | as PA1 --  | unlocked.               | byte not   |
* ATTN:------+ use        +------------+------------+ returned   +
*     : press| ENDPASS(   | After TGET/TPG/TPUT     | from TGET) |
*     :<ATTN>| PA1))      | (rc = 8), maybe in-     |            |
*     :      |            | between.  Can be        |            |
*     :      |            | effective even when     |            |
*     :      |            | keyboard is locked      |            |
*-----+------+------------+------------+------------+------------+
*    PFnn    |Action only after TGET   |                         |
*    ENTER   |that returns simple I/B  |       No effect         |
*    SEL     |data stream or unsplit   |                         |
*    PA3     |I/B SF                   |                         |
*            |(Works only if keyboard  |                         |
*            |is unlocked.)            |    (no AID byte         |
*-----+------+------------+------------+    returned from        +
*            |Action only after TGET   |    TGET)                |
* CLEAR-     |that returns unsplit I/B |                         |
* PARTITION  |SF.  3270 must be in     |                         |
*            |explicit partition mode. |                         |
*            |(Works only if Keyboard  |                         |
*            |is unlocked.)            |                         |
*-----+------+------------+------------+------------+------------+
*     : press|                                                   |
* NO  : any  |             No effect                             |
*     : key   |                                                 |
*-----+-------+------------+------------+------------+------------+
*
* Notes:
*   An "unsplit I/B SF" is inbound 3270 structured field data that
*   is long enough to include the imbedded 3270 AID byte in the first
*   TGET.  (In other words, the 1st TGET for this screen of 3270 data
*   must specify a buffer size long enough (perhaps 8-10 bytes or more)
*   to receive all inbound 3270 structured field data up to and
*   including the AID byte from the incoming 3270 partition.)
*   With some terminal connections, the <ATTN> key is not supported.
*   Pressing <ATTN> does nothing in these cases.
*
*
*********************************************************************
```

Other Documentation

TSO Extensions Version 2 Programming Guide and *TSO Extensions Version 2 Programming Services* fully describe the services that can be invoked with the TSO/3270 passthrough mode facility. (Note that only a small subset of TSO services can be invoked, however.)

The 3270 data stream is fully described in *3270 Information Display System: Data Stream Programmer's Reference*.

Usage Scenario

A typical usage of a TSO/3270 passthrough mode application might resemble the following:

1. Use the TSO/E OMVS command to start the z/OS shell session.

   When invoking the TSO/E OMVS command, the user should already know if TSO/3270 passthrough mode will be used later, and which applications will be used. The user may need to pick the proper ENDPASSTHROUGH key (*panic button*), based on instructions from the 3270 application programmer. If (for example) the panic button should be the 3270 attention key, enter;

   ```
   OMVS ENDPASSTHROUGH(ATTN)
   ```

2. Prepare to run the 3270 application.

   It may be a good idea to shut off any shell-provided messaging facility. Also, it is best to end any background jobs before running the TSO/3270 passthrough mode application.
3. Run the 3270 application.

   Run the 3270 application using instructions provided by the application programmer.
4. If the 3270 application hangs.

   It may be possible to press the ENDPASSTHROUGH key (defined earlier) to unhang the terminal. If not, the user may have to logoff using the SYSREQ key.
5. If the terminal displays stray data.

   Stray data may appear on the 3270 screen while the 3270 application is running. It may be possible to clear this up by pressing PA2 or CLEAR, depending on the 3270 application. The application provider may give more detailed instructions about this.
6. If background jobs break into the 3270 session.

   The screen may be returned to the shell session, and error messages may be displayed at the top of the screen.

If this happens, the recovery directions provided with the 3270 application should be followed. It may be possible to restart the 3270 application using job control commands, or the 3270 application may have ended by itself, or the 3270 application might have to be killed.

7. After the 3270 application ends.

   The normal shell session should be re-displayed on the terminal.

   If the ENDPASS key is used to end TSO/3270 passthrough mode, or some other error causes TSO/3270 passthrough mode to end, stray data may appear on the screen. Also, you may have to press <ENTER> more than once to get back to the shell prompt. The shell may issue error messages complaining about invalid input.

Sample Programs

Two sample programs are supplied in SYS1.SAMPLIB.

- Simple example

  This program is a simple TSO/3270 passthrough mode program that gets into and out of TSO/3270 passthrough mode. It ends with cleanup when any detectable error occurs. It does not guard against user messages, and does not try to recover any errors.

```
/*********************************************************************/
/*                                                                   */
/* Sample TSO/3270 passthrough mode program                          */
/* ---------------------------------------                           */
/*                                                                   */
/* This is a simple example of a TSO/3270 passthrough program.       */
/*                                                                   */
/* It enters passthrough mode and puts up a message to ask for user  */
/* input.  The input is then echoed for 10 seconds before the        */
/* program leaves passthrough mode and exits.                        */
/*                                                                   */
/*                                                                   */
/* Note: In order to simplify things, this program does not follow   */
/*       many of the programming guidelines for TSO/3270 passthrough */
/*       mode.  Although it does check for errors, it does not:      */
/*                                                                   */
/*       - handle command line redirection of STDIN_FILENO or        */
/*         STDOUT_FILENO.  (does not open the controlling terminal). */
/*                                                                   */
/*       - check for non-3270 terminals before trying to enter       */
/*         TSO/3270 passthrough mode.                                */
/*                                                                   */
/*       - set TOSTOP, or set up any signal catchers, or have any    */
/*         error recovery.  In other words, it cannot tolerate       */
/*         TSO/3270 passthrough mode interruptions caused by         */
/*         background jobs writing to the TTY.                       */
/*                                                                   */
/*       - handle unexpected TSO return codes (like attentions)      */
/*                                                                   */
/*********************************************************************/

#pragma csect(code,  "EXAMPCOD")
#pragma csect(static,"EXAMPSTA")
#pragma runopts(EXECOPS,POSIX(ON),ALL31(ON))

#define _ALL_SOURCE 1

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
```

```
#include <sys/types.h>
#include <unistd.h>
#include "fomth32p.h"             /* TSO/3270 passthrough mode .h    */

int  __errno2(void);              /* not in library headers          */
int *__err2ad(void);              /* not in library headers          */


/*********************************************************************/
/* MACROs and manifest constants                                     */
/*********************************************************************/

/* Messages for TPUT                                                 */
/* -----------------                                                 */

#define MESSAGE_1 "Session is now in TSO/3270 Passthrough Mode --"
#define MESSAGE_2 "To continue, enter some data."


/* reset errno and errno2 before issuing a function                  */
/* ------------------------------------------------                  */

#define RESET_ERRNO                                                  \
{                                                                    \
  errno = 0;                                                         \
  *__err2ad() = 0;                                                   \
}


/* complain and exit, if TSO R/C is abnormal in received response   */
/* -------------------------------------------------------------    */

#define CHECK_RSP(fcn, rc)                                           \
{                                                                    \
  if (rsp_p->__rc != _TSO3270_##fcn##_RC_##rc)                       \
  {                                                                  \
    reset_passthrough();                                             \
    fprintf(stderr, "error: " #fcn " R/C = %02X\n", rsp_p->__rc);   \
    return 0;                                                        \
  }                                                                  \
}


/* complain and exit after unexpected R/C is received                */
/* --------------------------------------------------                */

#define ERROR_RC(fcn, rc)                                            \
{                                                                    \
  reset_passthrough();                                               \
  fprintf(                                                           \
    stderr,                                                          \
    #fcn "() error: R/C=%d, errno=%d, errno2=%08X, msg=\"%s\"\n",   \
    rc,                                                              \
    errno,                                                           \
    __errno2(),                                                      \
    strerror(errno)                                                  \
    );                                                               \
  exit(0);                                                           \
}


/********************************************************************/
/* Static variables                                                 */
/********************************************************************/

static int in_passthrough_mode = 0; /* 1 = in passthrough mode      */
```

```
/* Maximum-sized buffers for TSO/3270 passthrough request/response  */
/* --------------------------------------------------------------  */

static char req_buf[_TSO3270_LMAX+_TSO3270_REQH_L] = "";
static char rsp_buf[_TSO3270_LMAX+_TSO3270_RSPH_L] = "";

static __tso3270_request_t *req_p =
      (__tso3270_request_t *)(void *)req_buf;

static __tso3270_response_t *rsp_p =
      (__tso3270_response_t *)(void *)rsp_buf;


/*******************************************************************/
/*                                                                 */
/* reset_passthrough() -- end TSO/3270 passthrough mode (if needed) */
/* ===================    ----------------------------------------- */
/*                                                                 */
/* notes: Assumes STDIN_FILENO is the TTY to be reset              */
/*                                                                 */
/*                                                                 */
/*******************************************************************/

static void reset_passthrough(void)
{
  int            tc_rc     = 0;
  struct termios termios_v = {0};


 /*
  * Return TTY to normal operation only if required
  * -----------------------------------------------
  */

  if (in_passthrough_mode == 1)
  {
    in_passthrough_mode = 0;       /* prevent error recursion       */

    RESET_ERRNO
    tc_rc = tcgetattr(STDIN_FILENO, &termios-v);
    if (tc_rc != 0) ERROR_RC(tcgetattr, tc_rc)

    termios_v.c_cflag &= ~(unsigned)PTU3270;

    RESET_ERRNO
    tc_rc = tcsetattr(STDIN_FILENO, TCSAFLUSH, &termios-v);
    if (tc_rc != 0) ERROR_RC(tcgetattr, tc_rc)
  }

  return;
}


/*******************************************************************/
/*                                                                 */
/* send_request() -- send TSO/3270 passthrough request to TTY      */
/* ==============    ----------------------------------------      */
/*                                                                 */
/*                                                                 */
/* notes: Always writes to STDOUT_FILENO                           */
/*                                                                 */
/*        Assumes TTY is in blocking mode, etc.                    */
/*                                                                 */
/*        Exits if write() error occurs                            */
/*                                                                 */
/*                                                                 */
```

```
/*********************************************************************/


static void send_request(
  unsigned char fcn,           /* passthrough function (e.g. TPUT) */
  unsigned char p1,            /* value for __p1 field in req hdr  */
  unsigned char p2,            /* value for __p2 field in req hdr  */
  int           p3,            /* value for __p3 field in req hdr  */
  size_t        data_l,        /* data length (can be 0) for __l   */
  char         *data_p         /* ptr to data (NULL OK if data_l=0)*/
  )
{
  ssize_t write_rc   = 0;
  size_t  write_l    = data_l + _TSO3270_REQH_L; /* total write len */


 /*
  * Fill in passthrough request (with data) from caller's parameters
  * ----------------------------------------------------------------
  */

  req_p->__ff   = _TSO3270_FF;
  req_p->__fcn  = fcn;
  req_p->__p1   = p1;
  req_p->__p2   = p2;
  req_p->__p3   = p3;
  req_p->__l    = data_l;

  if (data_l > 0)
    memcpy((void *)(req_p->__d), (void *)data_p, data_l);

  write_l = data_l + _TSO3270_REQH_L;


 /*
  * Send passthrough request to TTY, exit if any errors
  * ---------------------------------------------------
  */

  RESET_ERRNO
  write_rc = write(STDOUT_FILENO, (void *)req_buf, write_l);
  if (write_rc != write_l) ERROR_RC(write, write_rc)

  return;
}



/*********************************************************************/
/*                                                                   */
/* receive_response() -- receive TSO/3270 passthrough response      */
/* =================    -------------------------------------      */
/*                                                                   */
/*        This routine issues one or more read() requests to the    */
/*        TTY, until a complete TSO/3270 passthrough mode response  */
/*        has been received (i.e. __l bytes of data have been       */
/*        read in).                                                  */
/*                                                                   */
/*                                                                   */
/* notes: Always assumes that next byte from TTY starts a response  */
/*                                                                   */
/*        Assumes that only one response is outstanding (i.e.       */
/*        no extra data from next response comes in on this        */
/*        read() from the TTY).                                      */
/*                                                                   */
/*        Assumes TTY is in blocking mode, etc.                     */
/*                                                                   */
/*        Exits if read() error occurs, or 1st read byte does not   */
/*        start a response                                          */
/*                                                                  */
/*                                                                  */
/********************************************************************/

static void receive_response(void)
{
  ssize_t read_rc = 0;
  size_t  data_l  = 0;
  size_t  read_l  = sizeof rsp_buf;
  char *  read_p  = rsp_buf;


  /*
   *  Loop to accumulate a complete response in the buffer
   *  ====================================================
   *
   *  Keep issuing read() requests until __l bytes of data have been
   *  received (in addition to the response header).
   */

   while (
          (data_l < _TSO3270_RSPH_L)   /* bypass __l until filled-in */
          ||
          (data_l < (rsp_p->__l + _TSO3270_RSPH_L))
         )
   {
    /*
     * Wait for 1st/next part of response to come in
     * --------------------------------------------
     */

     RESET_ERRNO
     read_rc = read(STDIN_FILENO, (void *)read_p, read_l);

     if (read_rc <= 0) ERROR_RC(read, read_rc)


    /*
     *  adjust read pointer and read length for next part of response
     *  -------------------------------------------------------------
     */

     data_l += (size_t)read_rc;
     read_l -= (size_t)read_rc;
     read_p += (size_t)read_rc;


    /*
     * If enough data received so far, check for 0xFE, to make sure
     * we really have the start of a response.
     * -------------------------------------------------------------
     */

     if (data_l > 0U)
     {
       if (rsp_p->__fe != _TSO3270_FE)
       {
         reset_passthrough();
         fprintf(
           stderr,
           "error: __fe = %02X in response\n",
           rsp_p->__fe
           );
         exit(0);
       }
```

```
      }


     /*
      * If enough data received so far, make sure we have only data
      * belonging to this response (i.e. only 1 response is expected)
      * ------------------------------------------------------------
      */

      if (
          (data_l     >= _TSO3270_RSPH_L) /* complete hdr already?
                                           (OK to look at __l)      */
          &&;
          (data_l     >  _TSO3270_RSPH_L + rsp_p->__l)/* some data
                                           past end of this rsp?    */
         )
      {
          reset_passthrough();
          fprintf(
            stderr,
            "error: too much data, __l=%d, data_l=%d\n",
            rsp_p->__l, data_l
            );
          exit(0);
      }
    }                            /* end of main loop                */


   /*
    *  Make sure no pre-TSO errors occurred during request processing
    *  --------------------------------------------------------------
    */

    if (rsp_p->__error != _TSO3270_ERROR_OK)
    {
      reset_passthrough();
      printf("error: __error = %02X\n");
      exit(0);
    }

    return;
  }



  /******************************************************************/
  /*                                                                */
  /* main() -- solicit input and echo it in TSO/3270 passthrough mode */
  /* ====      ---------------------------------------------------- */
  /*                                                                */
  /*          - enter passthrough mode                              */
  /*          - issue two TPUTs to ask for user input               */
  /*          - issue TGET to wait for and obtain user input        */
  /*          - issue TPUT to echo back the input                   */
  /*          - wait 10 seconds, then end passthrough mode          */
  /*                                                                */
  /******************************************************************/

  int main(void)
  {
    struct termios termios_v         = {0}; /* for setting passthrough*/
    char fmt_buf[201U+_TSO3270_LMAX] = "";  /* for echoed input       */
    size_t fmt_l                     = 0U;  /* length of echoed input */
    int tc_rc                        = 0;   /* tcxxx() R/C            */


  /*
```

```
 *   Switch into TSO/3270 passthru mode
 *   ----------------------------------
 */

  RESET_ERRNO
  tc_rc = tcgetattr(STDIN_FILENO, &termios-v);
  if (tc_rc != 0) ERROR_RC(tcgetattr, tc_rc)

  termios_v.c_cflag |= PTU3270;   /* set passthrough flag in termios*/

  RESET_ERRNO
  tc_rc = tcsetattr(STDIN_FILENO, TCSAFLUSH, &termios-v);
  if (tc_rc != 0) ERROR_RC(tcsetattr, tc_rc)

  in_passthrough_mode = 1;        /* cause passthrough reset later  */


/*
 *   Issue TPUTs to solicit user input
 *   ---------------------------------
 */

  send_request(
    _TSO3270_TPUT,
    0, 0, 0,
    strlen(MESSAGE_1), MESSAGE_1
    );

  receive_response();
  CHECK_RSP(TPUT,OK)

  send_request(
    _TSO3270_TPUT,
    _TSO3270_TPUT_HOLD, 0, 0,      /* wait until terminal gets data  */
    strlen(MESSAGE_2), MESSAGE_2
    );

  receive_response();
  CHECK_RSP(TPUT,OK)


/*
 *   Obtain user input and echo it on the screen
 *   -------------------------------------------
 */

  send_request(
    _TSO3270_TGET,
    0, 0, _TSO3270_LMAX,           /* use max-sized TGET buffer      */
    0, NULL                        /* no data for the TGET request   */
    );

  receive_response();
  CHECK_RSP(TGET,EDIT)

  fmt_l = (size_t)sprintf(
    fmt_buf,
    "Echoed data: \"%*.*s\"",
    rsp_p->__l, rsp_p->__l, rsp_p->__d
    );

  send_request(
    _TSO3270_TPUT,
    _TSO3270_TPUT_HOLD, 0, 0,      /* wait until terminal gets data  */
    fmt_l, fmt_buf                 /* formatted echoed input         */
    );
  receive_response();
  CHECK_RSP(TPUT,OK)


 /*
  *   Wait 10 seconds, then end TSO/3270 passthrough mode
  *   ---------------------------------------------------
  */

  sleep(10);
  reset_passthrough();
  return 0;
}
```

- Second example

  This program illustrates all of the programming notes in the last section. It tries to recover from all errors, and the 3270 data stream itself is fairly simple.

```
/* *********************************************************************
 *
 * 3270 Transparent Mode sample application
 * ========================================
 *
 *  The 3270 TM system service allows an application to write to
 *  and read from the terminal, via OMVS api's that are similar to
 *  the TSO api's.
 *
 *  This test application requires the following setup:
 *   -  3270 terminal using OMVS
 *
 *  How to invoke:
 *    compile and bind as a shell program
 *    run interactively from the shell on a 3270 terminal
 *    press PA1 after th flag appears
 *
 *  Program name =  I3T3270X
 *
 *  Possible error codes returned :
 *      0  = success
 *      1  = general error
 *      2  = write() returned error
 *      3  = all data not written
 *      4  = read() returned error
 *      5  = TGET     response invalid
 *      6  = TPUT     response invalid
 *      7  = SIGWINCH occurred
 *     14  = STFSMODE response invalid
 *     99  = EINTR signal received
 *    100  = freopen for stdout file failed
 *    101  = freopen for stderr file failed
 *    102  = could not get controlling terminal pathname
 *    103  = open terminal failed.
 *    104  = not started by a terminal.  isatty returned 0.
 *    105  = could not get terminal attributes.  tcgetattr failed.
 *    106  = 3270 TM packet mode is not enabled.
 *    107  = could not set terminal attributes.  tcsetattr failed.
 *    108  = set up signal handlers failed
 *    109  = could not get permission flags (stat failed)
 *
 *********************************************************************/

/**********************************************************************
 * Headers                                                            *
 *********************************************************************/
#define  _ALL_SOURCE
#define  _OPEN_SOURCE_2
#define  _OPEN_SYS_PTY_EXTENSIONS
#include <fomth32p.h>                 /* 3270 TM structure mapping   */
```

```
#include <string.h>
#include <stdio.h>
#include <limits.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <modes.h>
#include <sys/wait.h>
#include <signal.h>
#include <fcntl.h>
#include <termios.h>
#include <signal.h>
#ifndef SIGWINCH
    #define SIGWINCH 28                /* temp until CRTL support   */
#endif
#include <sys/times.h>
#include <time.h>
#include <unistd.h>
#define _BSD                           /*     to get winsize, etc. */
#include <sys/ioctl.h>

/*********************************************************************
 * Constants                                                         *
 *********************************************************************/
#define NUMVARS          2

 /* return codes */
#define   VAR_SUCCESS                 0
#define   VAR_INVALID                 1
#define   WRITE_FAILED                2
#define   WRITE_LENGTH_FAILURE        3
#define   READ_FAILED                 4
#define   INVALID_TGET_RESPONSE       5
#define   INVALID_TPUT_RESPONSE       6
#define   INVALID_STFSMODE_RESPONSE   14
#define   EINTR_RECEIVED              99


/*********************************************************************
 * Global variables                                                  *
 *********************************************************************/
   int      argcnt_0;                  /* no arguments for sig handlrs*/
   int      var_result;                /* variation result           */
   int      rc;                        /* return code                */

   struct   stat     status_info;      /* original mode from stat()   */
   mode_t   new_mode;                  /* mode with group bits off    */

   char     terminal_pathname[1025];   /* pathname of controlling term*/
   int      terminal_fd;               /* controlling terminal fd     */
   char     outbuff[1932];             /* output buffer               */
   char     inbuff[1932];              /* input buffer                */
   size_t   write_length;              /* length to write             */
   ssize_t  read_length;               /* amount read                 */
   int      tget_asis = 0;             /* 1 = do TGET ASIS            */
                                       /*   else do TGET EDIT         */
   int      keep_going;                /* !0 = keep writing & reading */
   char     a_line[] = "I3T3270X line mode. "
                       "Press PA1 to stop, "
                       "or type data and press enter.";

                                       /* ptr to request structure    */
   __tso3270_request_t
            *request_ptr = (__tso3270_request_t *)(void *) outbuff;

                                       /* ptr to response structure   */
   __tso3270_response_t
```

```
            *response_ptr = (__tso3270_response_t *)(void *) inbuff;

/*********************************************************************/
/**  prototypes                                                     **/
/*********************************************************************/
void  route (int);
void  signal_SIGWINCH_handler (int);
void  signal_SIG_TTIO_handler (int);
void  signal_SIGCONT_handler (int);
int   set_signal ();
int   send_request ();
int   receive_response ();
int   TGET_line  ();
int   TPUT_request ();
int   STFSMODE_request ();




/*********************************************************************/
/************** MAIN *************************************************/
/*********************************************************************/

main (argcnt, arglst)

   int     argcnt;                   /* argument count              */
   char    *arglst[];                /* argument list               */
{
   /*****************************************************************/
   /** local data                                                  **/
   /*****************************************************************/
   int     i;                        /* variation index             */
   int     open_options =            /* open options                */
                      O_RDWR;        /*   blocking read / write     */

   struct    termios  org_termios;   /* original termios            */
   struct    termios  new_termios;   /* changed termios             */

   FILE      *stream1,               /* stream for redirected stdout*/
             *stream2;               /* stream for redirected stderr*/


   printf ("\n\n  ********* Use PA1 to stop ******** \n\n\n");

   /*****************************************************************/
   /** redirect STDOUT to a log file to prevent the                **/
   /** CRTL from writing error messages                            **/
   /*****************************************************************/
   printf ("\n STDERR and STDOUT being redirected to i3t3270a.std\n");

   stream1 = freopen ("i3t3270a.std", "a+", stdout);
   if (stream1 == NULL)
     {
      printf ("\n redirecting STDOUT failed."
            "\n    stdout=%d"
            "\n    errno=%d  errnojr=%08x ",
             stdout,
             errno,
             __errno2() );
      exit(100);
     }

   /*****************************************************************/
   /** redirect STDERR  to a log file to prevent the               **/
   /** CRTL from writing error messages                            **/
   /*****************************************************************/
   stream2 = freopen ("i3t3270a.std","a+", stderr);
```

```
if (stream2 == NULL)
  {
   printf ("\n redirecting STDERR failed."
         "\n     errno=%d  errnojr=%08x ",
          errno,
          __errno2() );
   exit(101);
  }

/*******************************************************************/
/** get the controlling terminal pathname                         **/
/*******************************************************************/
if (ctermid (terminal_pathname) == NULL)
  {
   printf ("\n Could not get controlling terminal pathname."
         "ctermid failed. \n   errno=%d  errnojr=%08x ",
          errno,
          __errno2() );
   exit(102);
  }

/*******************************************************************/
/** open the terminal for read/write, non-blocking                **/
/*******************************************************************/
if ((terminal_fd = open (terminal_pathname, open_options)) < 0)
  {
   printf ("\n open terminal failed."
         "\n   errno=%d  errnojr=%08x ",
          errno,
          __errno2() );
   exit(103);
  }

/*******************************************************************/
/** isatty (terminal_fd) to verify there is a TTY that started    **/
/** the application.  If not, close and exit                      **/
/*******************************************************************/
if (! isatty (terminal_fd))
  {
   printf ("\n not started by a terminal.  isatty returned 0."
         "\n   errno=%d  errnojr=%08x ",
          errno,
          __errno2() );
   close (terminal_fd);
   exit(104);
  }

/*******************************************************************/
/** tcgetattr (terminal_fd) to get current TTY attributes         **/
/*******************************************************************/
if ( tcgetattr (terminal_fd, &org-termios) != 0)
  {
printf ("\n  could not get terminal attributes.  tcgetattr failed."
         "\n   errno=%d  errnojr=%08x ",
          errno,
          __errno2() );
   close (terminal_fd);
   exit(105);
  }

/*******************************************************************/
/** if 3270 TM packet mode is not enabled, then close and exit    **/
/*******************************************************************/
if (( org_termios.c_cflag & PKT3270 ) != PKT3270)
  {
   printf ("\n 3270 TM packet mode is not enabled.");
   close (terminal_fd);
   exit(106);
  }

/*****************************************************************/
/** tcsetattr (terminal_fd) to turn on TSO3270 pass through      **/
/** mode and turn on TOSTOP to force SIGTTOU or SIGTTIN          **/
/** signals during write | read.  Also specify tcsaflush to      **/
/** remove all pending I/O.                                      **/
/*****************************************************************/
new_termios = org_termios;          /* save settings             */
new_termios.c_cflag = new_termios.c_cflag | PTU3270;
new_termios.c_lflag = new_termios.c_lflag | TOSTOP;

if ( tcsetattr (terminal_fd, TCSAFLUSH, &new_termios) != 0)
  {
   printf ("\n could not set terminal attributes.  tcsetattr failed."
         "\n   cflag=%08x,  lflag=%08x, "
         "\n   errno=%d  errnojr=%08x ",
          new_termios.c_cflag,
          new_termios.c_lflag,
          errno,
          __errno2() );
   close (terminal_fd);
   exit(107);
  }

/*****************************************************************/
/** set up signal handlers for :                                 **/
/**    SIGWINCH - error occurred or END3270 key pressed          **/
/**    SIGTTOU  - background interrupted a write request         **/
/**    SIGTTIN  - background interrupted a read  request         **/
/**    SIGCONT  - this process has been put back in foreground    **/
/*****************************************************************/
if ((rc = set_signal()) == -1)
  {
   close (terminal_fd);
   exit(108);
  }

/*****************************************************************/
/** chmod (term_filedes, mode) to turn off permission bits      **/
/** for group users.  This prevents overwrite of the TTY by      **/
/** another user, but does not block the super user.             **/
/*****************************************************************/
if ( stat (terminal_pathname, &status-info) != 0 )
  {
   if ( tcsetattr (terminal_fd, TCSAFLUSH, &org-termios) != 0)
     {
   printf("\n could not reset terminal attributes. tcsetattr failed."
            "\n   cflag=%08x,  errno=%d   errnojr=%08x ",
             org_termios.c_cflag,
             errno,
             __errno2() );
     }

   printf ("\n could not get status. stat failed."
         "\n   errno=%d   errnojr=%08x ",
          errno,
          __errno2() );

   close (terminal_fd);
   exit(109);
  }

new_mode = status_info.st_mode;
if ( (status_info.st_mode & 077) != 0 )  /* need to change?       */
  {
      new_mode = status_info.st_mode & 0700;
      if ( chmod (terminal_pathname, new_mode ) != 0 )
        {
         printf ("\n could not change mode.  chmod failed."
               "\n   super user is required to do this.  "
               "The test case is continuing anyway.  "
               "\n   errno=%d  errnojr=%08x ",
                errno,
                __errno2() );
        }
     }

   /*****************************************************************/
   /** Route to the proper variation.                             **/
   /*****************************************************************/
   var_result = VAR_SUCCESS;
                                 /* perform all variations
                                         unless a failure occurs   */
   for (i=1; (var_result == VAR_SUCCESS) &&; (i <= NUMVARS); i++)
        {
         route (i);
        }


   /*****************************************************************/
   /** turn off TSO3270 pass through mode by resetting original    **/
   /** attributes                                                  **/
   /*****************************************************************/
   if ( tcsetattr (terminal_fd, TCSAFLUSH, &org-termios) != 0)
     {
     printf ("\n could not reset terminal attributes. tcsetattr failed."
            "\n   cflag=%08x,  errno=%d  errnojr=%08x ",
             org_termios.c_cflag,
             errno,
             __errno2() );
     }

   /*****************************************************************/
   /** reset permissions to original settings                     **/
   /** It may be more robust to get the current permissions, since   **/
   /** they may have changed, but it is unlikely.                  **/
   /*****************************************************************/
   if ( (status_info.st_mode & 077) != 0 )  /* need to change?      */
     {
      if ( chmod (terminal_pathname, status_info.st_mode ) != 0 )
        {
         printf ("\n could not reset mode.  chmod failed."
               "\n   super user is required to do this.  "
               "The test case is continuing anyway.  "
               "\n   errno=%d  errnojr=%08x ",
                errno,
                __errno2() );
        }
     }

   /*****************************************************************/
   /** close all files                                             **/
   /*****************************************************************/
   printf ("\n i3t3270x finished \n\n");
   close (terminal_fd);

   /*****************************************************************/
   /** end of main                                                 **/
   /*****************************************************************/
   exit(var_result);
}                                    /* end of main program        */
```

```
/*********************************************************************/
/*******  SUB PROCEDURES *********************************************/
/*********************************************************************/


/*********************************************************************
 * Route - Execute the requested variation.
 *
 * Input:  variation - variation number requested
 * Output: execution is logged
 *
 * Return codes: none
 *********************************************************************/
void route (int variation)
{

   /*****************************************************************/
   /** local data                                                 **/
   /*****************************************************************/

     /*******************************************************/
     /*              3270 datastream examples              */
     /* 27      =introducer                                */
     /* F140    =write no erase                            */
     /* F540    =erase write                               */
     /* 7E40    =erase write alt                           */
     /* 13      =insert cursor                             */
     /* 114040  =position cursor row 1 column 1            */
     /* 11C150  =position cursor row 2 column 1            */
     /* 11C1D9  =position cursor row 2 column 10           */
     /* 2841F2  =set attributes, reverse video             */
     /* 2842F5  =set attributes, color, turquoise          */
     /* 2842F1  =set attributes, color, blue               */
     /* 2842F2  =set attributes, color, red                */
     /* 2842F7  =set attributes, color, white              */
     /*******************************************************/


   char     screen_2[] =
                 "\x27\xF5\x40"
     /* row 1 */    "\x11\x40\x40\x28\x41\xF2"
                 "\x28\x42\xF1"
                 " * * * * * * * * * * "
                 "\x28\x42\xF2"
                 "                                   "
     /* row 2 */    "\x11\xC1\x50\x28\x42\xF1"
                 " * * * * * * * * * * "
                 "\x28\x42\xF7"
                 "                                   "
     /* row 3 */    "\x11\xC2\x60\x28\x42\xF1"
                 " * * * * * * * * * * "
                 "\x28\x42\xF2"
                 "                                   "
     /* row 4 */    "\x11\xC3\xF0\x28\x42\xF1"
                 " * * * * * * * * * * "
                 "\x28\x42\xF7"
                 "                                   "
     /* row 5 */    "\x11\xC5\x40\x28\x42\xF1"
                 " * * * * * * * * * * "
                 "\x28\x42\xF2"
                 "                                   "
     /* row 6 */    "\x11\xC6\x50\x28\x42\xF7"
       "                                                     "
     /* row 7 */    "\x11\xC7\x60\x28\x42\xF2"
       "                                                        "
     /* row 8 */    "\x11\xC8\xF0\x28\x42\xF7"
       "                                                        "
     /* row 9 */    "\x11\x4A\x40\x28\x42\xF2"
       "                                                        "
     /* row 10*/    "\x11\x4B\x50\x28\x42\xF7"
       "                                                      "
                 "\x13"
                 ;


   int      i;
   int      temp_result;              /* hold var_result             */

   /*****************************************************************/
   switch (variation)
     {                                /* variation switch            */

      /*****************************************************************
       *
       * Test TPUT and TGET 3270 TM line mode interfaces
       *
       *  Purpose:
       *    Write a line to the screen via TPUT.
       *    Read a line from the screen via TGET.
       *    Repeat until PA1 keyed in.
       *    If EINTR occurs, try the write / read again
       *
       *****************************************************************/
      case 2:
       {

       /*****************************************************************/
       /** Repeat the TPUT and TGET until PA1 keyed in               **/
       /** or an error occurs                                         **/
       /**                                                            **/
       /** The first TPUT will display instructions, and consequent  **/
       /** TPUTs will repeat what was keyed in.                      **/
       /**                                                            **/
       /** If EINTR occurs, start over by displaying the first line  **/
       /*****************************************************************/
       for (
            request_ptr->__p1 = 0,
            request_ptr->__l = sizeof (a_line),
            strcpy (request_ptr->__d, a_line),
            keep_going = 1
            ;
            var_result == 0  &&;  keep_going
            ;)
         {
           if ((var_result = TPUT_request()) == 0)
             if ((var_result = TGET_line()) == EINTR_RECEIVED)
               {
                var_result = 0;
                request_ptr->__p1 = 0;
                request_ptr->__l = sizeof (a_line);
                strcpy (request_ptr->__d, a_line);
               }
         }
       break;
       }                              /* end case 2                  */



      /*****************************************************************
       *
       * Test STFSMODE & TPUT full screen mode 3270 TM interface
       *
       *  Purpose:
       *  Set the mode to full screen via STFSMODE.
       *  Write a message via TPUT.
       *  Read until ATTN | PA1 via TGET.
       *  Reset to line mode via STFSMODE.
       *
       *************************************************************/
      case 1:
        {

         /**********************************************************/
         /** Set the screen mode to full via STFSMODE,            **/
         /** TPUT a message,                                      **/
         /** and wait for PA1 to terminate (via TGET line).       **/
         /**                                                      **/
         /** If EINTR occurs, keep going by setting the mode      **/
         /** again and TPUTing the message.                       **/
         /**                                                      **/
         /** Reset line mode via STFSMODE.                        **/
         /**********************************************************/
         request_ptr->__p1  = _TSO3270_STFSMODE_ON;    /* full screen */
         if ((var_result = STFSMODE_request()) == 0)
           {
            request_ptr->__p1 = _TSO3270_TPUT_FULLSCR;
            request_ptr->__l = sizeof (screen_2);
            strcpy (request_ptr->__d, screen_2);
            var_result = TPUT_request();

            for (keep_going=1; var_result == 0  &&;  keep_going;)
               {
                if ((var_result = TGET_line()) == EINTR_RECEIVED)
                  {
                   request_ptr->__p1  = _TSO3270_STFSMODE_ON;
                   if ((var_result = STFSMODE_request()) == 0)
                     {
                      request_ptr->__p1 = _TSO3270_TPUT_FULLSCR;
                      request_ptr->__l = sizeof (screen_2);
                      strcpy (request_ptr->__d, screen_2);
                      var_result = TPUT_request();
                     }
                  }
               }
            if (var_result == 0)
              {                        /* reset line mode            */
               request_ptr->__p1  = _TSO3270_STFSMODE_OFF;
               var_result = STFSMODE_request();
              }
           }

         break;
        }                              /* end case 1                 */



       /************************************************************/
       /** invalid variation number                               **/
       /************************************************************/
      default:
        {                              /* test application in error  */
         printf("\n invalid variation number specified");
         var_result = VAR_INVALID;
        }                              /* test application in error  */
```

```
     }                                  /* variation switch            */
   return;
}                                       /* end of route subroutine     */



/*********************************************************************/
/**  signal handler for SIGWINCH                                    **/
/**                                                                 **/
/**    Indicates an error occurred or END3270 key was pressed       **/
/**                                                                 **/
/**    Reset tty permission bits and close all files.               **/
/**    Exit to end the program.                                     **/
/**                                                                 **/
/**    Note that it is not necessary to reset 3270 pass through mode **/
/**    bit because OMVS does this while issuing SIGWINCH, but       **/
/**    another process could send SIGWINCH.  So, a truly robust     **/
/**    application should reset the 3270 pass through mode bit.     **/
/**                                                                 **/
/*********************************************************************/
void  signal_SIGWINCH_handler (int signal_value)
{

   printf ("\n SIGWINCH occurred");
   if (status_info.st_mode != new_mode)  /* mode ever changed?        */
     {
      printf ("\n resetting permission bits");
      if ( chmod (terminal_pathname, status_info.st_mode ) != 0 )
        {
         printf ("\n chmod failed in signal_SIGWINCH_handler."
               "\n   errno=%d  errnojr=%08x ",
                errno,
                __errno2() );
        }
     }

   close (terminal_fd);

   exit(7);
}



/*********************************************************************/
/**  signal handler for SIGTTOU and SIGTTIN                         **/
/**                                                                 **/
/**    Indicates background has interrupted a write() or read().    **/
/**                                                                 **/
/**    Simply return and wait for the SIGCONT to occur              **/
/**                                                                 **/
/*********************************************************************/
void  signal_SIG_TTIO_handler (int signal_value)
{
   printf ("\n SIGTTOU or SIGTTIN occurred. value = %d", signal_value);
   return;
}



/*********************************************************************/
/**  signal handler for SIGCONT                                     **/
/**                                                                 **/
/**    Indicates the application process has been placed back in    **/
/**    foreground mode after being interrupted by SIGTTOU or SIGTTIN.**/
/**                                                                 **/
/**    Restart 3270 tm mode by getting attributes, setting TSO3270   **/
/**    pass through mode on, and return, so the application resumes  **/
/**    at the failing write() or read() with EINTR.                  **/
/**                                                                  **/
/**    When an error occurs here, the terminal_fd is closed, so      **/
/**    consequent write() or read() will fail and the application    **/
/**    will go through normal termination, which includes resetting  **/
/**    tty permission bits and appropriate Roast calls.               **/
/**                                                                  **/
/*********************************************************************/
void  signal_SIGCONT_handler (int signal_value)
{
   struct    termios  sig_termios;     /* attributes                 */

   printf ("\n SIGCONT occurred");

   /******************************************************************/
   /** tcgetattr (terminal_fd) to get current TTY attributes        **/
   /******************************************************************/
   if ( tcgetattr (terminal_fd, &sig-termios) != 0)
     {
      printf ("\n tcgetattr failed in signal_SIGCONT_handler."
            "\n   errno=%d  errnojr=%08x ",
             errno,
             __errno2() );
      close (terminal_fd);
      return;
     }

   /******************************************************************/
   /** tcsetattr (terminal_fd) to turn on TSO3270 pass through      **/
   /** mode and turn on TOSTOP to force SIGTTOU or SIGTTIN          **/
   /** signals during write | read.  Also specify tcsaflush to      **/
   /** remove all pending I/O.                                      **/
   /******************************************************************/
   if ((sig_termios.c_cflag & PTU3270) == PTU3270)
     {
      printf ("\n PTU3270 still set in signal_SIGNCONT_handler");
      return;
     }

   sig_termios.c_cflag = sig_termios.c_cflag | PTU3270;
   sig_termios.c_lflag = sig_termios.c_lflag | TOSTOP;

   if ( tcsetattr (terminal_fd, TCSAFLUSH, &sig-termios) != 0)
     {
      printf ("\n tcsetattr failed in signal_SIGCONT_handler."
            "\n   cflag=%08x,  lflag=%08x, "
            "\n   errno=%d  errnojr=%08x ",
             sig_termios.c_cflag,
             sig_termios.c_lflag,
             errno,
             __errno2() );
      close (terminal_fd);
      return;
     }

   /******************************************************************/
   printf ("\n attributes reset after SIGCONT");
   return;
}



/*********************************************************************/
/**  set signal handler                                             **/
/**                                                                  **/
/**    sigaction() to set the signal handler for the specified signal**/
/**    No signals are masked.                                        **/
/**    Log any error                                                 **/
/**                                                                  **/
/**********************************************************************/
int  set_signal ()
{
   struct  sigaction  sigact;          /* sigaction interface         */
   int     ret_code;
   int     i;
   int     signal_value[] =            /* signals to handle           */
                        { SIGWINCH,
                          SIGTTOU,
                          SIGTTIN,
                          SIGCONT };

   for (i=0, ret_code=0; i<4 &&; ret_code==0; i++)
     {
      sigemptyset (&(sigact;sa_mask));    /* no signals masked        */
      sigact.sa_flags = 0;
      switch (i)
       {
        case 0:
          { sigact.sa_handler = &signal-SIGWINCH-handler; break;}
        case 1:
          { sigact.sa_handler = &signal-SIG-TTIO-handler; break;}
        case 2:
          { sigact.sa_handler = &signal-SIG-TTIO-handler; break;}
        default:
          { sigact.sa_handler = &signal-SIGCONT-handler;  }
       }

      if ((ret_code = sigaction (signal_value[i], &sigact, NULL)) == -1)
        {
         printf ("\n sigaction() for signal value %d failed."
               "\n   errno=%d  errnojr=%08x ",
                signal_value[i],
                errno,
                __errno2() );
        }
     }
   return (ret_code);
}




/**********************************************************************/
/**                                                                  **/
/** send_request                                                     **/
/**    Sends a TSO request to OMVS via write()                       **/
/**    Logs any errors                                               **/
/**                                                                  **/
/**    input:                                                        **/
/**     outbuff - buffer to be written                               **/
/**     write_length - amount of bytes in outbuff to be written      **/
/**     terminal_fd - terminal's file descripter                     **/
/**                                                                  **/
/**    return values:                                                **/
/**     0 - success                                                  **/
/**     2 - write() failed                                           **/
/**     3 - write() did not write all requested data                 **/
/**    99 - EINTR occurred, repeat request if desired                **/
/**                                                                  **/
/**********************************************************************/
int  send_request ()
{
   ssize_t  wrote_length;             /* amount written              */
   int      send_rc;

   /************************************************************/
   /** send request via write()                              **/
   /************************************************************/
   send_rc = 0;
   if ((wrote_length = write (terminal_fd, outbuff, write_length)
       ) == -1 )
     {
      /************************************************************/
      /** if EINTR, then try the write again.                   **/
      /** This indicates that the application was put in         **/
      /** background and then returned to foreground             **/
      /** (SIGTTOU occurred followed by a SIGCONT)               **/
      /************************************************************/
      if (errno == EINTR)
        {
         printf ("\n EINTR occurred during write");
         send_rc = EINTR_RECEIVED;
        }
      else
       /************************************************************/
       /** else error - stop the variation                        **/
       /************************************************************/
        {
         printf ("\n write failed.  \n   output buffer = %s"
               "\n   write length = %d"
               "\n   errno=%d  errnojr=%08x ",
                outbuff, write_length,
                errno,
                __errno2() );

         send_rc = WRITE_FAILED;
        }
     }
   else
    /************************************************************/
    /** check for all data written                              **/
    /************************************************************/
    {
     if (wrote_length != write_length)
       {
        printf ("\n all data not written.  %d of %d bytes written."
              "\n   outbuff = %s",
               wrote_length, write_length, outbuff);

        send_rc = WRITE_LENGTH_FAILURE;
       }
    }

   return (send_rc);
}



/*******************************************************************/
/**                                                              **/
/** receive_response                                             **/
/**    Reads a TSO response from OMVS via read()                 **/
/**    Logs any errors                                           **/
/**                                                              **/
/**    input:                                                    **/
/**     inbuff - buffer to be written to                         **/
/**     terminal_fd - terminal's file descripter                 **/
/**                                                              **/
/**    return values:                                            **/
/**     0 - success                                              **/
/**     4 - read() failed                                        **/
/**    99 - EINTR occurred, repeat request if desired            **/
/**                                                              **/
/*********************************************************************/
int  receive_response ()
{
   int    receive_rc;

   /**************************************************************/
   /** read response                                           **/
   /**************************************************************/
   receive_rc = 0;
   if ((read_length = read (terminal_fd, inbuff, sizeof (inbuff) ))
                    == -1 )
     {
      /*************************************************************/
      /** if EINTR, then try the write again.                    **/
      /** This indicates that the application was put in         **/
      /** background and then returned to foreground              **/
      /** (SIGTTIN occurred followed by a SIGCONT)               **/
      /*************************************************************/
      if (errno == EINTR)
        {
         printf ("\n EINTR occurred during read");
         receive_rc = EINTR_RECEIVED;
        }

      else
       /*************************************************************/
       /** else error - stop the variation                        **/
       /*************************************************************/
        {
         printf ("\n read response failed."
               "\n   errno=%d  errnojr=%08x ",
                errno,
                __errno2() );

         receive_rc = READ_FAILED;
        }
     }
   return (receive_rc);
}



/*********************************************************************/
/**                                                              **/
/**  TGET_line                                                   **/
/**    read a line from the terminal via TGET                    **/
/**    send the request and receive the response                 **/
/**    validate the response                                     **/
/**                                                              **/
/**  input:                                                      **/
/**    request_ptr                                               **/
/**    response_ptr                                              **/
/**    tget_asis  -  0 = do TGET EDIT                            **/
/**                  1 = do TGET ASIS                            **/
/**                                                              **/
/**  output:                                                     **/
/**    keep_going  !0 = user entered more data, keep writing &  **/
/**                     reading lines                            **/
/**                 0 = user entered PA1 to stop                 **/
/**                                                              **/
/**  return values:                                              **/
/**     0 - successful TGET                                     **/
/**     2 - write() failed                                       **/
/**     3 - write() did not write all requested data               **/
/**     4 - read() failed                                          **/
/**     5 - invalid TGET response                                  **/
/**    99 - EINTR occurred, repeat request if desired              **/
/**                                                                **/
/********************************************************************/
int  TGET_line  ()
{
   int      tget_rc;

   /*****************************************************/
   /** read a line from the terminal via TGET          **/
   /** encode TGET request using fomth32p structure    **/
   /** use defaults unless ASIS specified              **/
   /*****************************************************/
   request_ptr->__ff  = _TSO3270_FF;
   request_ptr->__fcn = _TSO3270_TGET;
   request_ptr->__p1  = 0;
   if (tget_asis)
      request_ptr->__p1  = _TSO3270_TGET_ASIS;
   else
      request_ptr->__p1  = 0;
   request_ptr->__p2  = 0;
   request_ptr->__p3  = sizeof (inbuff) - _TSO3270_REQH_L;
   request_ptr->__l   = 0;
   write_length = _TSO3270_REQH_L;

   /*****************************************************/
   /** send the TGET request to read a line            **/
   /*****************************************************/
   tget_rc = send_request();
   if (tget_rc == 0)
     {
      /*****************************************************/
      /** successful send - read the response            **/
      /*****************************************************/
      printf ("\n successful TGET line request sent");

      tget_rc = receive_response();
      if (tget_rc == 0)
        {
         /***********************************************/
         /** successful receive - verify response      **/
         /***********************************************/
         if (   read_length < _TSO3270_RSPH_L
             || response_ptr->__fe    != _TSO3270_FE
             || response_ptr->__fcn   != _TSO3270_TGET
             || response_ptr->__error != 0
             || response_ptr->__r0    != 0
             || response_ptr->__rc    != 0
            )
              {
               /***********************************************/
               /** if ATTN key pressed, then finished        **/
               /** else bad response                         **/
               /***********************************************/
               if ( response_ptr->__rc == _TSO3270_TGET_RC_ATTN )
                 {
                  printf ("\n ATTN key received.");
                  keep_going = 0;       /* stop writing and reading   */
                  tget_rc = 0;
                 }

               else
                 {
                  printf ("\n TGET response is incorrect."
                        "\n   read length = %d"
```

```
                         "   fe=%x   fc=%x   error=%x"
                         "   r0=%x   rc=%x   l=%d",
                          read_length,
                          response_ptr->__fe,
                          response_ptr->__fcn,
                          response_ptr->__error,
                          response_ptr->__r0,
                          response_ptr->__rc,
                          response_ptr->__l );

                   tget_rc = INVALID_TGET_RESPONSE;
                 }
               }
         else
               {
                /*********************************************/
                /** good response                           **/
                /** if no data was returned,                **/
                /**   then write initial line again         **/
                /** else write the same data back to the    **/
                /**   terminal                              **/
                /*********************************************/
                printf ("\n successful TGET line response received"
                      "  %d bytes of data", response_ptr->__l);

                if (response_ptr->__l == 0)
                    {
                     request_ptr->__l = sizeof (a_line);
                     strcpy (request_ptr->__d, a_line);
                    }
                else
                    {
                     request_ptr->__l = response_ptr->__l;
                     if (request_ptr->__l >
                             (sizeof (outbuff) - _TSO3270_REQH_L))
                         request_ptr->__l =
                             (sizeof (outbuff) - _TSO3270_REQH_L);

                     strncpy (request_ptr->__d, response_ptr->__d,
                              request_ptr->__l);
                    }

               }
        }
     }
   return (tget_rc);
}



/*********************************************************************/
/**                                                                 **/
/**  TPUT_request                                                   **/
/**    write a line to the terminal via TPUT                        **/
/**    send the request and receive the response                    **/
/**    validate the response                                        **/
/**                                                                 **/
/**  input:                                                         **/
/**    response_ptr pointing to input buffer                        **/
/**    request_ptr  pointing to the output buffer                   **/
/**    request_ptr->__p1= mode required (EDIT | FULLSCREEN | etc.)  **/
/**    request_ptr->__l = length of data to be sent                 **/
/**    request_ptr->__d = data to be sent                           **/
/**        Note: when full screen mode is specified, the data       **/
/**              must be in 3270 datastream format                  **/
/**                                                                 **/
/**  return values:                                                 **/
/**     0 - successful TPUT                                         **/
/**     2 - write() failed                                          **/
/**     3 - write() did not write all requested data                **/
/**     4 - read() failed                                           **/
/**     6 - invalid TPUT response                                   **/
/**    99 - EINTR occurred, repeat request if desired               **/
/**                                                                 **/
/*********************************************************************/
int  TPUT_request ()
{
   int      tput_rc;

   /**********************************************************/
   /** encode TPUT request using fomth32p structure        **/
   /**********************************************************/
   request_ptr->__ff = _TSO3270_FF;
   request_ptr->__fcn = _TSO3270_TPUT;
   request_ptr->__p2 = 0;
   request_ptr->__p3 = 0;
   write_length = _TSO3270_REQH_L  + request_ptr->__l;

   /**********************************************************/
   /** send TPUT request to write a line or screen         **/
   /**********************************************************/
   tput_rc = send_request();
   if (tput_rc == 0)
     {
      /**********************************************************/
      /** successful send - receive the response              **/
      /**********************************************************/
      printf ("\n successful TPUT request sent");

      tput_rc = receive_response();
      if (tput_rc == 0)
        {
         /*****************************************************/
         /** succesfull receive - verify response            **/
         /*****************************************************/
         if (   read_length < _TSO3270_RSPH_L
             || response_ptr->__fe    != _TSO3270_FE
             || response_ptr->__fcn   != _TSO3270_TPUT
             || response_ptr->__error != 0
             || response_ptr->__r0    != 0
             || response_ptr->__rc    != 0
             || response_ptr->__l     != 0
            )
              {
               printf ("\n TPUT response is incorrect."
                     "\n   read length = %d"
                     "\n   fe=%x   fc=%x   error=%x"
                     "\n   r0=%x   rc=%x   l=%d",
                      read_length,
                      response_ptr->__fe,
                      response_ptr->__fcn,
                      response_ptr->__error,
                      response_ptr->__r0,
                      response_ptr->__rc,
                      response_ptr->__l );

               tput_rc = INVALID_TPUT_RESPONSE;
              }
         else
               printf ("\n successful TPUT response received"
                     "  %d bytes of data", response_ptr->__l);
        }
     }
   return (tput_rc);
```

```
}




/*********************************************************************/
/**                                                                 **/
/**  STFSMODE request                                               **/
/**    send the STFSMODE request                                    **/
/**    receive the response                                         **/
/**    validate the response                                        **/
/**                                                                 **/
/**  input:                                                         **/
/**    request_ptr  pointing to the output buffer                   **/
/**    request_ptr->p1 = _TSO3270_STFSMODE_ON | _TSO3270_STFSMODE_OFF**/
/**                                                                 **/
/**  return values:                                                 **/
/**    14 - invalid STFSMODE response                               **/
/**                                                                 **/
/*********************************************************************/
int  STFSMODE_request()
{
   int      stfsmode_rc;

   stfsmode_rc = VAR_SUCCESS;
   /*************************************************************/
   /** build the STFSMODE request                              **/
   /*************************************************************/
   request_ptr->__ff  = _TSO3270_FF;
   request_ptr->__fcn = _TSO3270_STFSMODE;
   request_ptr->__p2  = 0;
   request_ptr->__p3  = 0;
   request_ptr->__l   = 0;
   write_length = _TSO3270_REQH_L;

   /*************************************************************/
   /** send the STFSMODE request                               **/
   /*************************************************************/
   stfsmode_rc  = send_request();
   if (stfsmode_rc == 0)
     {
      /****************************************************/
      /** successful send - read the response            **/
      /****************************************************/
      printf ("\n successful STFSMODE request sent");

      stfsmode_rc  = receive_response();
      if (stfsmode_rc  == 0)
        {
         /***********************************************/
         /** successful receive - verify response      **/
         /***********************************************/
         if (   read_length < _TSO3270_RSPH_L
             || response_ptr->__fe    != _TSO3270_FE
             || response_ptr->__fcn   != _TSO3270_STFSMODE
             || response_ptr->__error != 0
             || response_ptr->__r0    != 0
             || response_ptr->__rc    != 0
             || response_ptr->__l     != 0
            )
           {
            printf ("\n STFSMODE response is invalid."
                  "\n   read length = %d"
                  "\n   fe=%x   fc=%x   error=%x"
                  "\n   r0=%x   rc=%x   l=%d",
                   read_length,
                   response_ptr->__fe,
                   response_ptr->__fcn,
                   response_ptr->__error,
                   response_ptr->__r0,
                   response_ptr->__rc,
                   response_ptr->__l );

            stfsmode_rc = INVALID_STFSMODE_RESPONSE;
           }
         else
           {
            printf ("\n successful STFSMODE response received");
           }
        }
     }
   return (stfsmode_rc);
}
```

[END]

Appendix B. Message Facility Overview

To facilitate translation of messages into various languages and make them available to a program based on a user's locale, it is necessary to keep messages separate from the program by putting them in message catalogs that the program can access at run time. z/OS UNIX provides commands and subroutines for this purpose.

The programmer uses these tools to create message source files that contain application program messages, and convert those files to message catalogs. The application uses these catalogs to retrieve and display messages as needed. Thus it is not necessary to change and recompile a program to translate message source files into other languages.

Creating a Message Source File

z/OS UNIX provides commands and subroutines to retrieve and display program messages located in externalized message catalogs. The **gencat** command is used to convert a message source file containing application messages into a message catalog. The **mkcatdefs** command can be used to preprocess a message source file into a format that can be passed to the **gencat** command. **mkcatdefs** processing is only needed if you wish to use symbolic names for messages. To create a message-text source file, open a file using any text editor. Enter a message identification number or symbolic identifier. Then enter the message text as shown below:

```
1 message-text
2 message-text
OUTMSG  message-text
4 message-text
```

The following usage rules apply:

- There must be one blank character between the message ID number or identifier and the message text.
- A symbolic identifier must begin with an alphabetic character and can contain only alphanumeric characters (letters of the alphabet, decimal digits, and underscores).
- The first character of a symbolic identifier cannot be a digit.
- The maximum length of a symbolic identifier is 255 bytes.
- Message ID numbers must be assigned in ascending order within a single message set, but need not be contiguous. 0 (zero) is not a valid message ID number. Message IDs in a gencat input file can be in the range 1 - NL_MSGMAX.
- Message ID numbers must be assigned as if intervening symbolic identifiers are also numbered. If the lines in the previous example had been numbered 1, 2, OUTMSG, and 3, this would be an error. This is because the **mkcatdefs** command also assigns numbers to symbolic identifiers and would have assigned 3 to the **OUTMSG** symbolic identifier.

Continuing Messages on the Next Line

All text following the blank after the message number is included as message text, up to the end of the line. Use the escape character \ (backslash) to continue message text to the following line. The backslash must be the last character on the line, as in the following example:

```
5 This is the text associated with \
message number 5
```

Special Characters in the Message Text

The \ (backslash) can be used to insert the following special characters in the message text:

\n New-line

\t Horizontal tab

\v Vertical tab

\b Backspace

\r Carriage return

\f Form feed

\\ Backslash (\)

\ddd Single-byte character associated with the octal value represented. One, two, or three octal digits may be specified. However, you must include a leading zero if the characters following the octal digits are also valid octal digits. For example, the octal value for $ (dollar sign) is 44. To display $5.00, specify \0445.00, not \445.00, or the 5 is parsed as part of the octal value. In general, when you are porting a message catalog (either to or from z/OS UNIX) you should:

- Run iconv to convert it to the code page of the system you will run it on.
- Look for and change any octal or hex codes in the catalog.

Defining a Character to Delimit Message Text

You can use the **$quote** directive in a message source file to define a character for delimiting message text. The format is:

```
$quote [character]   [comment]
```

Use the specified character before and after the message text as shown. In this example, the **$quote** directive sets the quote character to _ (underscore) and then disables it before the last message, which contains the quote character.

```
$quote _           Use an underscore to delimit message text
$set MSFAC         Message facility - symbolic identifiers
SYM_FORM _Symbolic identifiers can contain alphanumeric \
characters or the \_ (underscore character) \n_
5  _You can mix symbolic identifiers and numbers \n_
$quote
MSG_H  Remember to include the _msg_h_ file in your program\n
```

The last **$quote** directive in the previous example disables the underscore character.

In the following example, the **$quote** directive defines " (double quotation marks) as the quote character. The quote character must be the first nonblank character following the message number. Any text following the next occurrence of the quote character is ignored.

```
$quote "        Use a double quote to delimit message text
$set 10              Message facility - quote command messages
1 "Use the $quote directive to define a character \
for delimiting message text\n"
2 "You can include the \"quote\" character in \
a message by placing a \\ in front of it\n"
$quote
3   You can disable the "quote" mechanism by \
using the $quote directive without a character \
after it\n
```

This example illustrates two ways the quote character can be included in message text:

- Place a \ (backslash) in front of the quote character.
- Disable the quote mechanism by using the $quote directive without a character after it. Define the message, then define the quote character again.

This example also shows the following:

- A \ (backslash) is still required to split a quoted message across lines.
- To display a \ (backslash) in a message, place another \ in front of it.
- You can format a message with a newline character by using \n.
- Using the **$quote** directive with no character argument disables the quote mechanism.

Assigning Message Set Numbers and Message ID Numbers

All message sets require a set number or symbolic identifier. Use the **$set** directive in a source file to assign a number or identifier to a group of messages:

```
$set n [comment]
```

The message set number is specified by the value of **n**, a number between 1 and **NL_SETMAX**. Instead of a number, you can use a symbolic identifier. All messages following the **$set** directive are assigned to that set number until the next occurrence of a **$set** directive. The default set number is 1. Set numbers must be assigned in ascending order but need not be in series. Empty sets are created for skipped numbers. However, large gaps in the number sequence decrease efficiency and performance.

You can also include a comment in the **$set** directive, as follows:

```
$set 10       Communication error messages

$set OUTMSGS   Output error messages
```

Creating a Message Catalog

To create a message catalog, use the **mkcatdefs** command or the **gencat** command to process your completed message source file:

- Use the **gencat** command to process a message source file containing set numbers, message ID numbers, and message text. Message source files containing symbolic identifiers cannot be processed by the **gencat** command.
- Use the **mkcatdefs** command to preprocess a message source file containing symbolic identifiers. The resulting file can then be used as input to the **gencat**

command. The **mkcatdefs** command produces a *SymbolName.h* file containing definition statements. These statements equate symbolic identifiers with set numbers and message ID numbers assigned by the **mkcatdefs** command. The *SymbolName.h* file should be included in programs that use these symbolic identifiers.

- Use the **runcat** command to automatically process a source file containing symbolic identifiers. The **runcat** command invokes the **mkcatdefs** command and pipes its output to the **gencat** command.

If a message catalog with the name specified by the *CatalogFile* parameter exists, the **gencat** command modifies the catalog according to the statements in the message source files. If a message catalog does not exist, the **gencat** command creates a catalog file with the name specified by the *CatalogFile* parameter.

You can specify any number of message text source files. Multiple files are processed in the sequence specified. Each successive source file modifies the catalog. If you do not specify a source file, the **gencat** command accepts message source data from standard input.

Catalog Sizing

A message catalog can be virtually any size. The maximum number of sets in a catalog, messages in a catalog, and bytes in a message are defined in the **limits.h** file by the following macros:

NL_SETMAX

Specifies the maximum number of set numbers (up to 255) that can be specified by the **$set** directive. If the **NL_SETMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.

NL_MSGMAX

Specifies the maximum number of message ID numbers (up to 32767) allowed by the system. If the **NL_MSGMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.

NL_TEXTMAX

Specifies the maximum number of bytes (up to 2048) that a message can contain. If the **NL_TEXTMAX** limit is exceeded, the **gencat** command issues an error message and does not create or update the message catalog.

Removing Messages from a Catalog

The **$delset** directive removes all the messages of a specified set from an existing catalog:

```
$delset  n  [comment]
```

The message set is specified by **n**. The **$delset** directive must be placed in the proper set-number order with respect to any **$set** directives in the same source file. You can also include a comment in the **$delset** directive.

Examples

This example shows how to create a message catalog from a source file that contains message identification numbers. The following is the text of the **hello.msg** message source file:

```
$ file:  hello.msg
$set 1   prompts
1 Please enter your name
2 Hello,  %s \n
$ end of file: hello.msg
```

To create the **hello.cat** message catalog from the **hello.msg** source file, enter:

```
gencat  hello.cat hello.msg
```

The following example shows how to create a message catalog from a source file with symbolic references. The following is the text of the **hello.msg** message source file that contains symbolic references to the message set and the messages:

```
$ file:  hello.msg
$quote "
$set PROMPTS
PLEASE "Please enter your name"
HELLO "Hello, %s \n"
$ end of file: hello.msg
```

The following is the text of the **msgerrs.msg** message source file that contains error messages that can be referenced by their symbolic IDs:

```
$ file: msgerr.msg
$quote "
$set CAT_ERRORS
MAXOPEN "Cannot open message catalog %s \n \
Maximum number of catalogs already open"
NOT_EX "File %s not executable \n"
$set MSG_ERRORS
NOT_FOUND "Message %1$d, Set %2$d not found \n"
$ end of file: msgerr.msg
```

To process the **hello.msg** and **msgerrs** message source files, enter:

```
runcat hello hello.msg
runcat msgerrs msgerrs.msg  /usr/lib/nls/msg/$LANG/msgerrs.cat
```

The **runcat** command invokes the **mkcatdefs** and **gencat** commands. The first call to the **runcat** command takes the **hello.msg** source file and uses the second parameter, **hello**, to produce the **hello.cat** message catalog and the **hello.h** definition file.

The **hello.h** definition file contains symbolic names for the message catalog and symbolic IDs for the messages and sets. The symbolic name for the **hello.cat** message catalog is **MF_HELLO**. The name is produced automatically by the **mkcatdefs** command.

The second call to the **runcat** command takes the **msgerrs.msg** source file and uses the first parameter, **msgerrs**, to produce the **msgerrs.h** definition file. Because the third parameter, usr/lib/nls/msg/$LANG/msgerrs.cat, is present, the

runcat command uses this parameter for the catalog file name. This parameter is an absolute path name that specifies exactly where the **runcat** command must put the file. The symbolic name for the **msgerrs.cat** catalog is **MF_MSGERRS**.

Displaying Messages with an Application Program

You must include the following items to retrieve messages in your application program:

- The *catalog file***.h** definition file created by the **mkcatdefs** or **runcat** command if you used symbolic identifiers in the message source file, or the **limits.h** and **nl_types.h** files if you did not use symbolic identifiers.
- A call to initialize the locale environment.
- A call to open a catalog.
- A call to read a message.
- A call to display a message.
- A call to close the catalog.

The following run-time library functions provide the services necessary to display program messages:

setlocale
Sets the locale. Specify the **LC_ALL** or **LC_MESSAGES** environment variable in the call to the **setlocale** subroutine for the preferred message catalog language.

catopen
Opens a specified message catalog and returns a catalog descriptor, which you use to retrieve messages from a catalog.

catgets
Retrieves a message from a catalog after a successful call to the **catopen** subroutine.

printf Converts, formats, and writes to the standard output stream.

catclose
Closes a specified message catalog.

The following C program, **hello**, illustrates opening the **hello.cat** catalog with the **catopen** subroutine, retrieving messages from the catalog with the **catgets** subroutine, displaying the messages with the **printf** subroutine, and closing the catalog with the **catclose** subroutine.

```
/* program:  hello  */
#include <nl_types.h>
#include <locale.h>
nl_catd catd;
main()
{
/*  initialize the locale  */
setlocale (LC_ALL, "");
/* open the catalog */
catd=catopen("hello.cat",0);
printf(catgets(catd,1,1,"Hello, World!"));
catclose(catd);               /*  close the catalog  */
exit(0);
}
```

In the previous example, the **catopen** subroutine refers to the **hello.cat** message catalog only by file name. Therefore, you must make sure that the **NLSPATH** environment variable is set correctly. If the message catalog is successfully opened by the **catopen** subroutine, the **catgets** subroutine returns a pointer to the specified message in the **hello.cat** catalog. If the message catalog is not found or the message does not exist in the catalog, the **catgets** subroutine returns the **"Hello World"** default string.

Understanding the NLSPATH Environment Variable

The **NLSPATH** environment variable specifies the directories to search for message catalogs. The **catopen** subroutine searches these directories in the order specified when called to locate and open a message catalog. If it cannot find the catalog while searching the directories in **NLSPATH**, **catopen** uses a system default value of /usr/lib/nls/msg/%L/%N. This **NLSPATH** value is always searched after any user-specified **NLSPATH** value. If the message catalog is not found, **catgets** returns the program-supplied default message.

The **%L** and **%N** special variables are defined as follows:

%L Holds the locale-specific directory containing message catalogs. Depending on how you coded the **catopen()** call, the current value of the **LANG** or **LC_MESSAGES** environment variable is used.

%N Holds the name of the message catalog to be opened. This is the name passed as the first parameter of the **catopen()** call.

References

Refer to *z/OS UNIX System Services Command Reference* for descriptions of the following commands and subroutines:

- **dspcat** command
- **dspmsg** command
- **gencat** command
- **mkcatdefs** command
- **runcat** command

The following functions are described in *z/OS C/C++ Run-Time Library Reference*:

- **catgets** subroutine
- **catclose** subroutine
- **catopen** subroutine
- **printf** subroutine

Appendix C. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Acknowledgments

lex, **yacc**, and **make** are InterOpen source code products licensed from Mortice Kern Systems (MKS) Inc. of Waterloo, Ontario, Canada. These utilities complement the InterOpen/POSIX Shell and Utilities source code product providing POSIX.2 functionality to the z/OS UNIX services.

Source Code Policy

The source code created by MKS LEX and YACC may be freely distributed, provided that the copyright notices are not removed. The source code for the library routines used by MKS LEX and YACC is similarly protected. The user is free to sell or distribute programs that were created using MKS LEX and YACC, provided that this procedure is followed.

The z/OS UNIX **lex** utility is based on a similar program written by Charles Forsyth at the University of Waterloo (in Ontario, Canada) and described in an unpublished paper, “A Lexical Analyzer Generator” (1978). The implementation is loosely based on the description and suggestions in the book *Compilers, Principles, Techniques, and Tools*, by A. V. Aho, Ravi Sethi, and J. D. Ullman (Addison-Wesley, 1986).

This **lex** utility was inspired by a processor of the same name at Bell Labs, which also runs under z/OS UNIX, and, more distantly, on AED-0. z/OS UNIX **lex** is described in the paper “Lex — A Lexical Analyser Generator”, by M. E. Lesk, *Computer Science Technical Report* 39, Bell Labs (October 1975). AED-0 is described in “Automatic Generation of Efficient Lexical Analysers using Finite State Techniques”, by W. L. Johnson, appearing in the *Communications of the ACM* 11 (no. 12, 1968): 805–13.

z/OS UNIX **yacc** is input compatible with UNIX YACC (Yet Another Compiler-Compiler), written by S. C. Johnson of Bell Telephone Laboratories, Murray Hill, N.J. The LALR(1) version of MKS YACC was written by K. W. Lalonde of the Software Development Group of the University of Waterloo, Ontario, Canada.

The parsing algorithm used by **yacc** is derived from the article “Methods for Computing LALR(k) Look-ahead” by B. B. Kristensen and O. L. Madsen, *ACM Transactions on Programming Languages and Systems* 3 (no. 1, January 1981): 60–82. Those interested in reading this article should first have a good grasp of parsing theory principles.

The information contained in the glossary section and tagged by the word [POSIX] is copyrighted information of the Institute of Electrical and Electronics Engineers, Inc., extracted from IEEE Std 1003.1-1990, IEEE P1003.0, and IEEE P1003.2. This information was written within the context of these documents in their entirety. The IEEE takes no responsibility or liability for and will assume no liability for any damages resulting from the reader’s misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

Index

Special Characters

Readers' Comments — We'd Like to Hear from You

z/OS
UNIX System Services
Programming Tools

Publication No. SA22-7805-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SA22-7805-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY
12601-5400

Fold and Tape **Please do not staple** Fold and Tape

SA22-7805-00

Cut or Fold
Along Line

Program Number: 5694-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SA22-7805-00