NAME

tstol − TPOCC Systems Test and Operations Language (TSTOL) interpreter

SYNOPSIS

tstol [**-answer**] [**-call** [*host:*]*server*[**@***host*]] [**-debug**] [**-Debug** *debug_file*] [**-events** *mission*]
[**-FIFO** *base_name*] [**-host** *name*] [**-mission** *name*] [**-quotes1**] [**-quotes2**]
[**-server** *name*] [**-single_user**] [**-startup** *directive*] [-[**no**]**sysvar**] [**-tty**]
[**-verify** *level*] [**-vperror**] [**-@** *semaphore*]

DESCRIPTION

tstol is the command interpreter for the TPOCC Systems Test and Operations Language (TSTOL). TSTOL is derived from previous generations of the Systems Test and Operations Language (STOL) used in existing NASA satellite control centers.

TSTOL is a procedural command language consisting of a core set of generic commands, supplemented by mission-specific extensions. The generic TSTOL commands provide the following capabilities:

(1) Various data types (decimal, octal, hexadecimal, and binary integers; floating point numbers; character strings; logical constants; and date/time constants).

(2) A full complement of arithmetic, logical, and relational operators are implemented. Mixed-mode arithmetic is supported. A large number of built-in functions are available, including the standard trigonometric functions.

(3) Global and procedure-local variables. System variables (e.g., telemetry mnemonics) resident on remote systems can be referenced and assigned.

(4) Procedures (TSTOL ‘‘subroutines’’) can be defined, invoked, and controlled. Arguments can be passed to procedures. **if-then-else** blocks and looping constructs (**do**, **for**, and **while**) can be used to affect control flow within a procedure. TSTOL also includes a simple macro substitution capability.

(5) Foreign directives (i.e., mission-specific directives) can be defined and invoked. Foreign directives are programmed in TSTOL (they look just like TSTOL procedures), but they are invoked and behave as if the directives were built-in to TSTOL. Foreign directives can perform any valid TSTOL functions, including communicating with applications programs. Foreign directive definitions are usually read from an initialization file when the TSTOL server starts up.

(6) Applications programs can carry on interactive dialogs with the operator via the TSTOL parser.

(7) The TSTOL parser can wait until a specified absolute time has been reached, until a certain amount of time has elapsed, or until a given expression evaluates to TRUE.

(8) Multiple watchpoints can be set up to monitor the values of system variables (e.g., telemetry mnemonics). A requested value can be sampled every *N* seconds (asynchronous sampling) or when the value is decommutated (synchronous sampling). Whenever a value is received, a user-defined foreign directive procedure is invoked and passed the value of the system variable. Once activated, a watchpoint effectively executes in "background" mode; the operator can go on and enter other TSTOL directives, run procedures, etc.

By defining a new set of foreign directives, a project can modify or extend the language recognized by TSTOL - without changing the executable **tstol** program. Although the simplest foreign directives are structured as a keyword followed by a list of parameters, TSTOL’s support of regular expression pattern matching allows foreign directives to apply sophisticated lexical and syntactical analysis to their command lines - again, without any changes to the executable **tstol** program.

This manual is divided into the following sections and subsections:

Invocation

tstol can be run in one of three modes: as a multi-user, network-based server; as a single-user, network-based client; or as a single-user, terminal-based parser. If invoked as a server (see the **-answer** command line option), **tstol** listens at an assigned network port for connection requests from client user interface (UI) processes. When a connection request is received from a prospective client, **tstol** forks a child **tstol** process to service that connection. There may be many child **tstol** processes, each parsing and processing commands from different UI processes (or different command input windows in the same UI process). The child parsers can, in turn, establish network connections to other applications programs (e.g., a spacecraft command processor).

tstol was originally written to be a network server; subsequent circumstances, however, made it necessary for **tstol** to initiate the network connection with the UI process. When invoked with the **-call** option, **tstol**, rather than passively listening for connection requests, actively attempts to establish a network connection with a UI process functioning as a server.

If invoked with the **-tty** option, **tstol** inputs commands from and outputs messages directly to the operator's terminal. This stand-alone mode is useful for debugging and testing the **tstol** program.

The TSTOL executable is:

/home/tpocc/obj_*arch*/tstol/tstol

To invoke **tstol** as a network-based server (the standard mode of operation), enter:

```
% tstol  -mission mission
```

To invoke **tstol** as a network client, enter:

```
% tstol  -call server[@host]  -mission mission
```

To invoke **tstol** as a terminal-based parser (for stand-alone testing), enter:

```
% tstol  -tty  -mission mission
```

Stand-alone testing of the network-based **tstol** server is possible using the **-single_user** command line option (see the OPTIONS descriptions below):

```
% tstol  -single_user  -mission mission
```

Run-Time Environment

tstol expects certain items to be present in its run-time environment. The names of most of these items are based on the mission name (see the **-mission** option). **tstol** performs all case conversions needed to construct the run-time environment names, regardless of how the mission name was specified on the command line. The following items should be present when the parser is run:

1. Server name *mission*_**tstol** – determines the network port at which **tstol** listens for connection requests from display processes. The server name (with *mission* in all lower case letters) should be entered in the system **/etc/services** file; for example,

   ```
   hst_tstol          24681/tcp
   ```

   assigns server name ‘‘hst_tstol’’ to TCP port 24681.

2. Environment variable $*mission*_PROC_FILE – specifies the defaults for missing components of TSTOL procedure file names. For example, the following C Shell command specifies the default directory (‘‘/home/hst/source/procs’’) and file extension (‘‘.prc’’) for procedure files:

   ```
   setenv HST_PROC_FILE /home/hst/source/procs/.prc
   ```

   A ‘‘**start** TEST’’ directive would then read and execute the TSTOL procedure found in **/home/hst/source/procs/test.prc**. $*mission*_PROC_FILE can also contain multiple directory (and/or extension) specifications, separated by commas or spaces. For example:

   ```
   setenv HST_PROC_FILE "./.prc, /home/hst/source/procs/"
   ```

   If the operator enters a ‘‘**start** TEST’’ directive, **tstol** would first search its current working directory for **test.prc** and then search **/home/hst/source/procs**. A directory specification without an extension must have a trailing ‘‘/’’. Note that the parser’s current working directory is not necessarily the operator’s current directory, particularly if **tstol** was brought up as a network server. The **cd** directive will display and/or change your parser’s current working directory.

3. Environment variable $*mission*_SYV_FILE – specifies the system variable definition file. The information in this database file is used by **tstol** to access system variables. The **-nosysvar** command line option inhibits the loading of this file.

4. Data Services subsystem – must be up and running if system variables will be accessed. The data server should be running under the server name *mission*_**data_server** on the expected host (see the **-host** command line option and the **%liv(host)** internal variable).

5. Events subsystem (or a suitable substitute) – must be up and running. **tstol**’s interface to the event logger requires the setup of 4 environment variables:

   $*mission*_CLASS_FILE – the event classes definition file.

   $*mission*_EVENT_BASE – event message base number definitions for the various subsystems.

   $*mission*_EVENT_TEXT – event message texts.

$LOG_SUBSYS – the base server name of the event logger.

OPTIONS

–answer
Specifies that **tstol** is to function as a network server, listening for and answering connection requests from clients. This is **tstol**'s default mode of operation. The server port name at which **tstol** listens for requests from prospective clients is, by default, "*mission*_tstol", but this name may be overridden with the **-server** option. Unless **tstol** is in single-user mode (see the **-single_user** option), a new subprocess is spawned to service each new client connection.

–call [*host:*]*server*[*@host*]]
Specifies that **tstol** is to function as a network client. Instead of listening for connecting clients (see the **-answer** option), **tstol** will initiate a network contact, sending the specified server a connection request. This mode automatically puts **tstol** in single-user mode (see the **-single_user** option): rather than spawning a subprocess, **tstol** will itself service the network connection. The name of the host on which the server to be contacted is running can be specified in either of two forms shown.

–debug
Turns debug output (written to **stdout**) on.

–Debug *debug_file*
Turns debug output (written to *debug_file*) on.

–events *mission*
Specifies a mission name for the events subsystem. Normally, your project's mission name (specified using the **-mission** option) is used to construct the database file names and network server names required by the events subsystem interface. The **-events** option, which should be used in conjunction with the **-mission** option, lets you specify a different mission name for the events subsystem interface. This capability allows you, for instance, to test a new mission's TSTOL procedures while using an existing mission's event message database and event logger.

–FIFO *base_name*
Enables operator I/O through FIFOs (named pipes) and specifies the base name for the FIFOs. Two FIFOs are required: an input FIFO and an output FIFO. The parser appends ''.in'' and ''.out'' to the base name to construct the input and output FIFO names, respectively.

–host *name*
Specifies the implied host name for accessing system variables; a host can still be explicitly specified in a system variable reference. *host_name* is the name of the computer on which the system variable data server resides; if the **-host** option is not specified, *host_name* defaults to the local host. The implied host name can be changed after the parser is up by assigning a new value to the **%liv(host)** internal variable.

–mission *mission*
Specifies the mission name. This name (in lower case) is used to construct the mission-specific start-up file and server name and (in upper case) is also assigned to TSTOL global variable, MISSION.

–quotes1
Controls the delimiters for strings and quoted identifiers. If **-quotes1** is specified, string constants are delimited by single-quotes (') and identifiers containing special characters are delimited by double-quotes ("). A delimiter can be inserted in a string/identifier by entering it twice.

–quotes2
Controls the delimiters for strings and quoted identifiers. If **-quotes2** is specified (the default), string constants are delimited by double-quotes (") and identifiers containing special characters are delimited by single-quotes ('). A delimiter can be inserted in a string/identifier by entering it twice.

–server *name*
: Specifies a network server name for the TSTOL server. This name, which must be entered along with a port number in the **/etc/services** file, defaults to *mission*_**server** (where *mission* is converted to all lower case).

–single_user
: Brings the TSTOL server up in single-user mode. In this mode, the server will answer a single network connection and will service that client itself, rather than forking a child process to handle the connection. Single-user mode makes it easier to run the TSTOL server in the debugger.

–startup *directive*
: Specifies a TSTOL directive to be executed at start-up time. This will typically be a **start** directive that runs a start-up procedure.

–[**no**]**sysvar**
: Inhibits the loading of the system variable database. Although system variables are then not recognized or accessible, TSTOL server initialization is speeded up – an advantage during up-and-down testing of the parser.

–tty
: Invokes the parser as a stand-alone program that reads its operator input from **stdin**, (i.e., the TTY), rather than the network. *vperror()* message output is automatically enabled (see the **-vperror** option).

–verify *level*
: Enables *malloc*(3) heap verification. If *level* is 0 (the default), *malloc*(3) behaves normally. If *level* is 1, the system checks the arguments passed to *malloc*(3) and the heap blocks immediately affected by each call. If *level* is 2, the system verifies the integrity of the entire heap on each call to *malloc*(3). Running at level 2 is useful for detecting (presumably) inadvertent corruption of your allocated memory. This option is only supported *(i)* if the program was linked with the TPOCC **libmalloc** library, or *(ii)* under SunOS, if the program was linked with the system's **/usr/lib/debug/malloc.o** file; see the SunOS documentation on *malloc*(3) for more information.

–vperror
: Turns *vperror()* message output on. *vperror()* messages are low-level error messages generated by TPOCC library functions; normally, they are disabled. If enabled, the messages are output to **stderr**.

–@ *semaphore*
: Specifies the ID of a semaphore that **tstol** should signal upon receiving and accepting a connection request. This option is used under VMS when running **tstol** with the TPOCC PROSPER program. PROSPER spawns **tstol** in single-user, network server mode (see the **-single_user** option) to listen for connection requests from prospective clients. When a connection request is received, **tstol** closes its listening port and notifies PROSPER (by signalling the semaphore) that a new **tstol** listener can be spawned.

USAGE

The TPOCC Systems Test and Operations Language (TSTOL) is a general-purpose, programmable, operator interface language. The capabilities of TSTOL can be divided into two categories: generic capabilities and mission-specific capabilities. The following sections describe (i) the basics of the TSTOL language, (ii) the generic command set, and (iii) the mission-specific command sets.

Structure of Statements

As in prior generations of STOL, a TSTOL statement can consist of up to four possible fields:

[*label*:] *directive* [*arguments*] [; *comment*]

An alphanumeric label, if present, is the first field in the statement and is followed by a colon; the first character of a label must be alphabetic. Labels may appear on a line by themselves or may precede a directive on the same line. Reserved words (e.g., BEGIN, END) may be used as labels without causing syntax errors. *Examples:* ‘‘ADD_2_AND_2:’’, ‘‘RETRY_COMMAND:’’, etc.

A command *directive* is the keyword that identifies the command to be executed. Directive keywords are reserved words and generally should not be used as identifiers (e.g., variable names, mnemonics, etc.). Keywords for built-in directives can be used for foreign directives; typing in such a keyword then invokes the foreign directive. The built-in directive can still be executed by prefixing the keyword with a ‘‘\’’. *Examples:* ‘‘**acquire**’’, ‘‘**page**’’, ‘‘**reset**’’, etc.

Arguments to a command are separated from the directive keyword by blanks or tabs; multiple arguments are separated from each other by blanks, commas, or tabs. *Example:* ‘‘**page** *page_name*, *update_rate*’’

A semi-colon (‘‘;’’) in a statement introduces a comment. Comments can be on a line all by themselves. Blank lines and form feeds are also allowed. *Example:* ‘‘**history on** ; Turn on NASCOM block recording.’’

Commands with many arguments can be continued on successive lines by ending each line with two semi-colons (‘‘;;’’). These line continuation markers also double (no pun intended!) as comment delimiters.

Command Line History

tstol implements a primitive command line history mechanism, similar to that of the UNIX **csh**(1). The parser saves the most recent *N* lines of operator input; the currently saved set of lines can be displayed on the operator’s screen. Particular lines can be recalled upon request; a recalled line is submitted to the parser as if the operator typed in the directive again. An exclamation mark (‘‘!’’) is used to access the command line history:

!!	Recalls the previous directive and submits it to **tstol**.
!*n*	Recalls the *n*-th line of operator input.
!*string*	Recalls the most recent line of operator input that contains the specified string.
!@	Displays a list of the currently-saved input lines on the operator’s screen.

By default, **tstol** saves the 20 most recent lines of operator input. This number can be changed by setting a local internal variable (see below), **%liv(savehist)**.

Data Types and Constants

The TSTOL parser supports a number of different data types, including integers, reals, and character strings. *Integer constants* range from -2**31-1 to +2**31 and can be specified in decimal, octal, or hexadecimal using the Standard C conventions for integer constants:

Decimal Constants:	37 -1
Octal Constants:	045 07777777
Hexadecimal Constants:	0x2BAD 0xFAB4

TSTOL discourages the use of, but supports, the old STOL notations for binary (**B**), octal (**O**), and hexadecimal (**H** or **X**) numbers:

B’100101’ **O**’1234567’ **H**’DAD1’ **X**’C3D2’

Floating point numbers are stored internally as double-precision reals with a possible range of approximately -4.9E-324 to +1.8E308 (on computers that utilize the IEEE floating point representation); they can be written in a variety of ways (‘‘D’’ and ‘‘E’’ are accepted interchangeably for the exponent):

1.0   -879.5   2.25**D**03   3.6**E**-01

Character string constants should be enclosed in double-quotes; a double-quote can be inserted in a string by typing in two consecutive double-quotes:

"S/C Attitude"   "OFF"   "can""t means won""t"

Single-quotes are used to delimit identifiers containing special characters. This single- vs. double-quote convention can be reversed using the **-quotes1** command line option.

Date/time constants are expressed in the form *YY-DDD-HH:MM:SS.LLL*, where the fields are year, day of year, hour, minutes, seconds, and milliseconds, respectively. Embedded blanks are not permitted; leading zeroes are not required. One or more fields in a date/time constant may be omitted, subject to the following rules:

– A colon (‘‘:’’) must be present in order for the parser to recognize the input as a date/time constant.

– Only leading and trailing fields may be omitted.

– If, after leading or trailing fields are omitted, the form of the constant is ambiguous, TSTOL assumes the leftmost definition. For example, *XX:ZZ* is interpreted as *HH:MM*, not *MM:SS*.

– Leading and trailing delimiters are not allowed.

Missing fields are supplied by TSTOL: the year and day default to the current date; the remaining fields default to zero. In the year field of a date/time constant, ‘‘70’’ through ‘‘99’’ represent the years 1970 through 1999; ‘‘00’’ through ‘‘69’’ represent the years 2000 through 2069.

Logical constants can be written in a number of forms:

true   **false**   .TRUE.   .FALSE.   .T.   .F.

UNIX pathnames (file names) are recognized in certain instances. The standard C Shell file naming conventions are followed, although ‘‘$*var*’’ references to environment variables should be avoided (they will cause unwanted text substitution). Unfortunately, UNIX file name conventions conflict with TSTOL’s lexical conventions; for example, **˜/.login** is an acceptable pathname in TSTOL, but **/home/tpocc** looks like a badly-formed arithmetic expression. If a desired pathname might be mistaken for a variable name or for an arithmetic expression, enclose the file name in quotes (either double- or single-quotes can be used):

˜/.login   startup.prc   ’this_file’   "../that_file"

Other data types (e.g., telemetry values) are supported to some degree, but they primarily come into play when accessing system variables.

TSTOL Variables

TSTOL variables are distinguished by their name, their scope, and their data type. TSTOL follows the usual naming conventions regarding variable names: an initial alphabetic character (‘‘A’’-‘‘Z’’) followed by zero or more alphanumeric characters or underscores (‘‘_’’). TSTOL variable names are case-insensitive (e.g., ‘‘BAbCd’’ and ‘‘BaBcD’’ are equivalent identifiers) and are limited to 32 characters. Special characters can be embedded in variable names by enclosing the entire variable name in single quotes (but see the **-quotes1** command line option).

The scope of a variable defines its accessibility. *Procedure-local* variables (declared by the **local** directive) can only be referenced within a procedure and not from a parent or child procedure. Formal parameters passed by value are, for all practical purposes, procedure-local variables. Procedure-local variables disappear when the procedure exits. *Procedure-global* variables (declared using the **global** directive) are global to all procedures; they can be accessed at any level of procedure nesting or when no procedures are active. (The ‘‘console-local’’ variables, X1..X16, of MAE STOL can be emulated by declaring procedure-global variables X1 through X16.) *Process-global* variables (a.k.a. system variables) are global to all parser processes and, in a sense, to all the computers on the network.

The resolution of variable references follows the hierarchy described above. First, the variable is looked up in the procedure-local symbol table. If the variable is not found there, the procedure-global symbol table is searched. If that search fails, the system variable access table is consulted.

TSTOL variables assume the data types of the values assigned to them, usually integers, reals, or strings. More exotic data types are possible via references to system variables.

The following procedure-global variables are predefined in TSTOL:

MISSION is assigned the mission name specified by the **-mission** command line option. The value assigned to MISSION is all upper case, regardless of how it was specified on the command line. MISSION is assigned its value before the server initialization file is loaded and executed, so a multi-mission initialization procedure can test MISSION’s value and configure the TSTOL parser as needed. Of course, any other TSTOL procedure can also reference MISSION.

%status is a global status flag intended for use in programming foreign directives. **%status** is automatically set to **true** if a command completion status message received from a remote process indicates success and **false** if the message indicates failure. Foreign directives (and TSTOL procedures) can test and set **%status** as they see fit.

System Variables

Variables external to a parser process are called system variables. These variables are stored in shared memory segments on the local host computer or possibly on remote hosts. Examples of system variables include current telemetry values, processing statistics, etc. Each system variable is uniquely identified by three items: the *host* computer on which the variable is resident, the *process* that "owns" the variable, and the variable’s *mnemonic* (name).

Due to the distributed nature of system variables, TSTOL utilizes a special construct to reference system variables:

*mnemonic[***#***process][***@***host][***[***index***]***]*

A system variable can be referenced by mnemonic only, if the mnemonic uniquely identifies the system variable (i.e., no two processes share a mnemonic name) and the name has not been superseded by a procedure-local or -global variable name. Otherwise, the process name *must* be specified. If a host is not specified, the default system variable host (see the **-host** command line option and the **%liv(host)** internal variable) is used. The one-dimensional array index (1..*N*) is optional.

If used in the context of an expression (e.g, on the right hand side of a **let** directive), a system variable reference retrieves the value of a system variable:

let *local_variable = mnemonic[***#***process][***@***host][***[***index***]***]*

To store a value in a system variable, place the reference on the left hand side of a **let** directive:

let *mnemonic[***#***process][***@***host][***[***index***]***] = expr*

Recalling or storing a system variable is actually performed by a TPOCC *data server* on the *host* computer. The first reference to a system variable on *host* causes the TSTOL parser to establish a network connection to *host*’s data server. When recalling a value, the parser passes *process*, *mnemonic*, and *index* to the data server; the data server then returns the value of the variable to the parser. To store a value, the parser passes *process*, *mnemonic*, *index*, and *expr* to the data server; the data server then stores the value in its local system variable shared memory.

A system variable reference can be prefixed with the ‘‘**P@**’’ modifier in order to retrieve the processed value of the variable. If the variable contains an analog telemetry point, its processed value is the current raw telemetry count converted to engineering units (EU) - a real number. The conversion is performed by the telemetry decommutation process, not by **tstol**. If the variable is a discrete telemetry or non-telemetry system variable, then its processed value is the state name (a string) assigned to the current value; this translation is performed by **tstol**.

The telemetry decommutation process stores telemetry values in Current Value Table (CVT) entries. CVT entries can be of several types: signed and unsigned integer, single- and double-precision floating point, text, and time. A CVT entry generally consists of 3 fields: the telemetry point’s current raw value, the processed value (in engineering units), and a status word whose bits indicate the quality of the telemetry value. Normally, when you assign a value to a telemetry variable via TSTOL, you’re simply replacing the telemetry point’s raw value; the EU-converted value and the status word are not updated. It is possible, for testing purposes, to update these fields from TSTOL. To do so, just encode the 3 field’s values in a string:

let *system_variable* = "*raw_value EU_value status_word*"

The format of the different fields depends upon the data type of the CVT:

CVT_SLI (signed long integer)

CVT_ULI (unsigned long integer)

The raw value and the status word can be specified in decimal, octal, or hexadecimal, *if you follow the standard C conventions for numeric constants*; i.e., *N*, 0*N*, and 0**x***N*, respectively. The EU-converted value should be specified as a real number, following the standard C conventions for floating point numbers.

CVT_SFP (single-precision floating point)

CVT_DFP (double-precision floating point)

The raw and EU-converted values should be specified as real numbers, following the standard C conventions for floating point numbers. The status word can be specified in decimal, octal, or hexadecimal.

CVT_TEXT (text)

A CVT_TEXT entry only has 2 fields: the raw text and the status word. The raw value for CVT_TEXT entries is specified as a sequence of ASCII characters, with no embedded blanks or tabs. An arbitrary character can be included by specifying the decimal, octal, or hexadecimal equivalent of the character, preceded by ‘‘\’’. The status word can be specified in decimal, octal, or hexadecimal. Example: **let** SC_TEXT = "Hello\041 0x6D4"

CVT_TIME (time)

A CVT_TIME entry itself has 3 fields: the raw time value received in the telemetry stream, the raw value converted to UNIX time (seconds and microseconds since 1970), and the status word. Assigning a value to a CVT_TIME entry requires 5 fields: the raw value, the time in seconds, the left-over microseconds, a time status word, and the CVT status word. The raw time value is specified as a sequence of ASCII characters, the same as is done for CVT_TEXT strings. The remaining fields can be specified in decimal, octal, or hexadecimal.

Local Internal Variables

Variables internal to a parser process are called local internal variables (LIV). LIVs are used to examine and control the internal workings of the parser, primarily for debugging and testing purposes. To set the value of an internal variable, the variable is referenced on the left hand side of an assignment statement using the special **%liv** construct:

let %liv(*keyword***)** = *expr*

where *keyword* can be one of the following:

echo_network	enables or disables echoing on the operator's display of network input and output (e.g., commands to applications programs). Network I/O is normally invisible to the operator.
echo_stored	enables or disables echoing on the operator's display of internally-stored input. Stored input consists of directives that the parser submits to itself; for instance, an earlier version of the **dialog** directive waited for a response from its source by submitting and executing a **pause** directive. Stored input is normally not displayed on the operator's screen.
errno	assigns a numeric value to the C Library global variable, **errno**.
events	terminates the current event logger connection and establishes a new connection to the event logger on host *expr*.
host	changes the default host name for system variable access to *expr*.
ignore_wait	if *expr* evaluates to **true**, **wait** directives will be ignored – a real convenience during procedure testing.
lex_debug	enables or disables **lex_util** debug output (useful for monitoring what the parser is reading).
libalex_debug	enables or disables *str_dupl()* debug output (useful for tracking down memory leaks).
log_procedure	enables or disables the logging of executable procedure input. The **echo** directive controls the echoing of procedure input to the operator's screen. **%liv(log_procedure)** controls the recording of procedure input in the events log.
malloc_debug	sets the debug level for *malloc()* heap verification. If *expr* is 0 (the default), *malloc()* behaves normally. If *expr* is 1, the system checks the arguments passed to *malloc()* (and its relatives); messages are written to **stderr** if errors are detected. If *expr* is 2, the system verifies the integrity of the entire heap on each call to *malloc()* (or one of its relatives); if an error is detected, the parser aborts with a core dump. Running at level 2 is useful for detecting (presumably) inadvertent corruption of your allocated memory. This option is only supported *(i)* if the program was linked with the TPOCC **libmalloc** library, or *(ii)* under SunOS if the program was linked with the system's **/usr/lib/debug/malloc.o** file; see the SunOS documentation on *malloc*(3) for more information.
malloc_trace	enables or disables *malloc()* trace output. This debug output, written to **stdout**, traces the allocation and freeing of dynamic memory. Tracing is only possible if the program was linked with the TPOCC **libmalloc** library.
net_debug	enables or disables debug output from the TPOCC networking functions. When debug of this type is enabled, data read from or written to any of the network connections established by **tstol** is dumped to **stdout** in hexadecimal and ASCII form.
savehist	adjusts the maximum number of operator input lines saved by **tstol**'s command line history facility; the default is 20.

screen_debug	enables or disables screen debugging (e.g., with the **xstol** program). If screen debugging is enabled, **tstol** outputs file and line number information to the display interface when executing procedures. Normally, screen debugging is disabled.
text_substitution	enables or disables the application of text substitution to input lines. If text substitution is enabled, a parameter reference (‘‘$’’ followed by a number or a variable name) in the input text is replaced by the value of the parameter. Usually, if no procedures or foreign directives are active, text substitution is enabled. At the start of a procedure, text substitution is automatically enabled. At the start of a foreign directive, text substitution is automatically disabled.
timeout	sets the timeout value for **dialog**, **pause**, and **transact** directives and system variable access to *expr* seconds; the default is 60 seconds. In the case of the directives, the timeout value controls how long the parser will wait for the applications task to respond (e.g., with a status message). During an attempt to reference a system variable, the timeout value controls how long the parser will wait for the data server to return the requested value.
yydebug	enables or disables YACC debug output − you must be desperate!

Internal variables can also be referenced within an expression (e.g., on the right hand side of an assignment statement):

%liv(*keyword***)**

where different *keyword*s return different values:

help	returns a string containing the possible keywords.
echo_network	returns **true** if echoing of network I/O is enabled and **false** otherwise.
echo_stored	returns **true** if echoing of internally-stored input is enabled and **false** otherwise.
errno	returns the system error message corresponding to the current value of C Library global variable, **errno**.
events	returns the name of the host on which the current event logger is running.
host	returns the default host used when accessing system variables.
ignore_wait	returns **true** if **wait** directives will be ignored and **false** otherwise.
lex_debug	returns **true** if **lex_util** debug output is enabled and **false** otherwise.
libalex_debug	returns **true** if *str_dupl()* debug output is enabled and **false** otherwise.
localhost	returns the name of the host machine on which **tstol** is running.
log_procedure	returns **true** if logging of procedure input is enabled and **false** otherwise.
malloc_debug	returns the current *malloc()* debug level.
malloc_marker	inserts a marker string in *malloc()*’s list of allocated memory and returns a string containing the address of the marker string. The marker can be used (in a call to *malloc_dump()* or *malloc_totals()*) to monitor memory allocation activity after the marker’s creation.
malloc_verify	calls *malloc_verify()* to verify the integrity of the memory allocation heap. **true** is returned if the heap passed the test and **false** if the verification failed.
net_debug	returns **true** if debug output for the TPOCC networking functions is enabled and **false** otherwise.
privileges	returns a string containing a comma-separated list of the operator’s current

	privileges. The **access** directive must be used to set the operator's privileges.
screen_debug	returns **true** if screen debugging is enabled and **false** otherwise.
text_substitution	returns **true** if text substitution is enabled and **false** otherwise.
timeout	returns the maximum number of seconds the parser will wait for (i) an applications task to respond with a message, or (ii) a remote data server to return the value of a system variable.
version	returns the version number of the TSTOL program. This number is actually the version number of TSTOL's YACC source file, the most frequently changed file in the program.
yydebug	returns **true** if YACC debug output is enabled and **false** otherwise.
yydepth	returns the current depth of the YACC token stack.

Expressions and Assignment

TSTOL supports the normal complement of arithmetic, logical, and relational expressions.

Arithmetic operations include addition (+), subtraction (-), multiplication (∗), division (/), modulus (**mod**), remainder (**rem**), and exponentiation (∗∗). The data type of an arithmetic expression is determined by the data types of its operands. An expression consisting entirely of integer operands gives an integer result. An expression involving at least one real number generates a real value. **mod** and **rem** always return integers. A string containing a number is converted to the appropriate real or integer value; e.g., "123" + "4.56" is equivalent to 123 + 4.56.

Only addition and subtraction are defined for date/time constants. Date/time constants are represented internally as the number of seconds and microseconds since January 1, 1970 (a UNIX convention). Adding two date/time constants is allowed, but probably not meaningful. Adding or subtracting an integer or real number *X* to or from a date/time constant produces a date/time constant offset by *X* number of seconds. Subtracting a date/time constant from another date/time constant gives the difference in seconds (a floating point number) between the two times.

The *string concatenation operator* (**&**) concatenates the string values of two expressions.

Logical operators (**and**, **or**, **xor**, **not**) accept numeric or logical operands and generate logical values of **true** or **false**. In a logical expression, a zero-valued numeric operand is treated as false and a non-zero value is treated as true. The left-to-right evaluation of subexpressions is *short-circuited* as soon as the truth or falsity of a logical expression is known. (The standard FORTRAN dot notation for logical operators and values is supported, but its use is discouraged.)

Date/time constants are given special treatment in logical expressions. If a date/time constant precedes or is equal to the current GMT, it evaluates to **true** (i.e., the time is past). If the date/time constant is greater than the current GMT, it evaluates to **false** (i.e., the time is still in the future).

Relational operators (=, <>, >, <, <=, >=) accept two operands of any data type and compare them. A relational expression produces a logical value of **true** or **false** if the relational condition succeeds or fails, respectively. If the two operands are of dissimilar data types that prevent a meaningful comparison, **false** is always returned. (The standard FORTRAN dot notation for relational operators is supported, but its use is discouraged.)

Ordered from highest to lowest, TSTOL operator precedences are as follows:

1. Exponentiation (∗∗)
2. Unary minus (-), unary plus (+)
3. Multiplication (∗), division (/), modulus (**mod**), and remainder (**rem**)
4. Addition (+), subtraction (-), and string concatenation (**&**)
5. Relational operators (=, <>, <, >, <=, >=)
6. Negation (**not**)

7. Conjunction (**and**)
8. Exclusive OR (**xor**)
9. Disjunction (**or**)

Except for exponentiation, which is right-associative, operations at the same precedence level are evaluated from left to right. Parentheses may be used to alter the order of evaluation.

In TSTOL, variables may be assigned a value in one of several ways: directly using the **let** or **ask** directive, and indirectly by passing the variable by reference to a procedure.

Built-In Functions

TSTOL provides a number of built-in functions that ease the job of coding TSTOL procedures and directive definitions; these functions can be used freely wherever an expression is allowed. In the trigonometric functions, all angles are expressed in radians.

abs (*expr***)**	returns the absolute value of an expression.
nint (*expr***)**	returns the integer nearest the value of an expression.
sqrt (*expr***)**	returns the square root of an expression.
sin (*expr***)**	returns the sine of an angle.
cos (*expr***)**	returns the cosine of an angle.
tan (*expr***)**	returns the tangent of an angle.
asin (*expr***)**	returns the angle having the given sine.
acos (*expr***)**	returns the angle having the given cosine.
atan (*expr***)**	returns the angle having the given tangent.

%arg (*expr***)** returns the *i*-th argument of the currently executing procedure or foreign directive body. Arguments are numbered from 1 to the number of arguments (see the **%nargs** function).

%bin (*expr [, width]***)**
returns a string containing *expr* formatted in binary; e.g., the number 23 would be returned as ‘‘...00010111’’. The optional field *width* argument controls the number of digits in the returned value; if not specified, *width* defaults to the number of bits in a **long** integer (usually 32).

%chkgrp (*group***)**
returns **true** if the **tstol** process is a member of the specified UNIX group. *group* is case-sensitive and should be enclosed in string quotes if specified as a literal.

%chkprv (*privilege***)**
returns **true** if the operator currently has the specified privilege. *privilege* is case-sensitive and should be enclosed in string quotes if specified as a literal.

%dec (*expr [, width]***)**
returns a string containing *expr* formatted in decimal; e.g., the number 23 would be returned as ‘‘23’’. The optional field *width* argument controls the number of digits in the returned value; if not specified, *width* defaults to the number of decimal digits (plus a sign, if necessary) required to represent *expr*. Unlike the binary, hexadecimal, and octal conversion functions, **%dec** pads the field with leading blanks, not zeroes.

%default (*[expr], [default]***)**
returns *expr* if it’s defined and *default* otherwise. The default expression is not evaluated if the primary expression is defined. This function is intended for specifying defaults for missing directive arguments.

%ds (...) is used to send commands to the data server. Since this %-function can be entered as a stand-alone directive, it is documented in the section, **Generic TSTOL Commands**.

%env (*expr***)** returns the value of a UNIX environment variable. A zero-length string (‘‘’’) is returned if the environment variable is not defined.

%eval (*expr***)** returns the value of *expr*, converted to a string and evaluated as if it had been typed in. For example, given the string "COUNT+1", **%eval** would look up the value of variable COUNT and add 1 to it. **%eval** is useful for evaluating non-standard arguments in foreign directives (since, by definition, non-standard arguments are not directly evaluated by the parser).

%float (*expr***)** returns the value of *expr*, converted to a real number.

%fparse (*[file_spec [, default_file_spec [, related_file_spec]]] [, field]***)**
parses and constructs file names. **%fparse** expands *file_spec* into a full pathname. Missing components in the pathname are supplied by the default and related file specifications or, if necessary, by system defaults. If *field* is not specified, **%fparse** returns the fully-expanded pathname. If *field* is specified, **%fparse** extracts and returns the desired field from the fully-expanded pathname. Valid fields are ‘‘ALL’’, ‘‘DIRECTORY’’, ‘‘FILENAME’’, ‘‘EXTENSION’’, ‘‘VERSION’’, ‘‘FILEXTVER’’ (file name, extension, and version combined), and ‘‘NODE’’. File specifications or field names should be entered as strings in double quotes or as expressions that evaluate to strings. A file specification for a UNIX directory must have a trailing ‘‘/’’; otherwise, **%fparse** will treat the last component of the directory path as a file name. When called without any arguments, ‘‘**%fparse** ()’’ returns your current working directory.

%fsearch (*[wildcard_file_spec [, default_file_spec [, related_file_spec]]]***)**
returns, on successive calls, the name of the next file matched by the wildcard file specification; an empty string (‘‘’’) is returned when no more files are matched. The default and related file specifications are used to fill in missing components of the wildcard file specification (see **%fparse**). **%fsearch** returns the fully-expanded pathname for each file matched. Under UNIX, a sequence of calls to **%fsearch** with the same wildcard file specification can only scan a single directory; putting wildcard characters in a directory name will not effect a multiple-directory scan. Invoking the search function without any arguments (‘‘**%fsearch** ()’’) terminates the current directory scan.

%gmt returns the current GMT as a date/time constant.

%hex (*expr [, width]***)**
returns a string containing *expr* formatted in hexadecimal; e.g., the number 23 would be returned as ‘‘00000017’’. The optional field *width* argument controls the number of digits in the returned value; if not specified, *width* defaults to the number of hexadecimal digits required to represent a **long** integer (usually 8).

%ident (*expr***)** instructs the TSTOL parser to treat the string value of *expr* as a keyword. This ‘‘function’’ only works in certain places, so its use is currently discouraged. Directives can be constructed ‘‘on-the-fly’’ just as easily with the **parse** directive.

%int (*expr***)** returns the value of *expr*, converted to an integer. The fractional portion of *expr* is truncated, not rounded.

%isint (*expr***)** returns **true** if *expr* can be interpreted as an integer and **false** otherwise. Integers are integers, of course, as are floating point numbers which have no fractional part. Strings can also be interpreted as integers if they follow the standard C conventions (see *strtol*(3)) for decimal, octal, and hexadecimal constants: *N*, 0*N*, and 0**x***N*, respectively.

%isnum (*expr***)** returns **true** if *expr* can be interpreted as a number (integer or floating point) and **false** otherwise. (Also see the **%isint** and **%isreal** functions.)

%isreal (*expr***)** returns **true** if *expr* can be interpreted as a real number and **false** otherwise. Floating point numbers and integers are considered real. Strings can be interpreted as reals if they follow the standard C conventions (see *strtod*(3)) for floating point numbers; e.g., 1, 2.3, 4.56E-78, etc.

%lex (*target*, *regexp [, retval0 [, retval1 [, ...]]] [, remainder]***)**
dissects a target string using the given regular expression. **%lex** supports the full functionality of *regcmp*(3), including *M*-to-*N* closure (‘‘*RE*{*[m][,[n]]*}’’) and subexpression assignment (‘‘(*RE*)$*n*’’); in addition, **lex**(1)-style alternation (‘‘*RE1* | *RE2*’’) is supported. If the regular expression match is successful, designated subexpressions are stored in the corresponding *retvalN* variables (for N = 0..9) and **%lex** returns **true**. If the match is unsuccessful, zero-length strings are assigned to the return variables and **%lex** returns **false**. In either case, *remainder* returns the text following the text matched by the regular expression. (See the subsection, **Regular Expressions**, for more information.)

%liv (*keyword***)** looks like a %-function, but it’s really a reference to an internal variable; see the earlier section, **Local Internal Variables**.

%lower (*expr***)** returns the value of *expr* as a string, with all alphabetic characters converted to lower case.

%match (*target*, *expr [, expr ...]***)**
matches the target string against a list of expressions. Matches are case-insensitive; both the target string and the match strings are converted to upper case before a match is attempted. Use **%rex** if alphabetic case is important. A string in the list of expressions can specify a fixed-length abbreviation by embedding a ‘‘#’’ in the string; e.g., ‘‘SIM#INT’’ will match **simint** and its short form, **sim**. Variable-length abbreviations are denoted by a ‘‘∗’’ embedded in the match string; e.g., ‘‘INIT∗IALIZE’’ matches **init**, **initi**, ..., **initialize**. **%match** returns the index 1..*N* (all **true** values in TSTOL) of the matching string in the list of expressions; 0 (**false** in TSTOL) is returned if no match was found.

%nargs returns the actual number of arguments passed to the currently executing procedure or foreign directive body.

%net (...**)** is used for establishing and communicating across network connections. Since this %-function can be entered as a stand-alone directive, it is documented in the section, **Generic TSTOL Commands**.

%nwords (*exp***)** returns the total number of ‘‘words’’ in a string, including null words. (See the **%word** function.)

%oct (*expr [, width]***)**
returns a string containing *expr* formatted in octal; e.g., the number 23 would be returned as ‘‘00000000027’’. The optional field *width* argument controls the number of digits in the returned value; if not specified, *width* defaults to the number of octal digits required to represent a **long** integer (usually 11).

%pick (*index*, *expr [, expr ...]***)**
returns the *i*-th expression from a list of expressions. When used in conjunction with **%match** or **%rex**, **%pick** provides an easy way to convert abbreviations to fixed keywords.

%real (*expr [, format]***)**
returns a string containing *expr* formatted as a real number; e.g., the number 2.345E-4 would be returned as ‘‘0.0002345’’. The optional *format* argument can be any standard C format string for floating point numbers; the default format is ‘‘%G’’, which lets the system choose an appropriate format, with or without an exponent, depending on the value of *expr*.

%replace (*target*, *regexp*, *replacement_text [, max_substitutions]***)**

returns a copy of the target string, with text matched by a regular expression (*regexp*) replaced by *replacement_text*. Up to *max_substitutions* are applied to the target string; if *max_substitutions* is not specified, all occurrences of text matched by *regexp* are replaced (global substitution). The text matched by *regexp* can be manipulated to some extent using special character sequences (**$***n*, **$&**, etc.) embedded in the replacement text; see the section on regular expressions for more information. To prevent **$**-character sequences from producing unwanted text substitution (if enabled), you may need to split *replacement_text* into several strings connected by the **&** string concatenation operator. (Very awkward!) Substitutions are not recursive; the search for the next *regexp* match begins following the last substitution. (See the subsection, **Regular Expressions**, for more information.)

%rest (*expr*, *index***)**
: returns the rest of a string, beginning with the *i*-th ‘‘word’’ in the string; words are delimited by blanks, commas, or tabs. Using the same scanning mechanism as **%word**, **%rest** skips arguments 1 through *i*-1 of *expr* and then returns the rest of *expr* (i.e., arguments *i* through *N*). A zero-length string (‘‘’’) is returned if *index* exceeds the number of words in the string (see the **%nwords** function). **%rest** is useful for scanning non-standard arguments.

%rex (*target*, *regexp [*, *regexp ...]***)**
: matches the target string against a list of regular expressions. The matches are case-sensitive – *target* is *not* converted to upper case before the match is attempted. If a regular expression must match the entire target string, be sure to include the ‘‘^’’ and ‘‘$’’ anchors at the beginning and end, respectively, of the regular expression. **%rex** returns the index 1..*N* (all **true** values in TSTOL) of the matching string in the list of regular expressions; 0 (**false** in TSTOL) is returned if no match was found. (See the subsection, **Regular Expressions**, for more information.)

%search (*target*, *regexp [*, *regexp ...]***)**
: returns a string containing the text matched by a regular expresssion in a target string. More than one regular expression may be specified; the first regular expression that produces a match terminates the search. A zero-length string (‘‘’’) is returned if none of the regular expressions is matched in the target string. The matches are case-sensitive – *target* is *not* converted to upper case before a match is attempted. (See the subsection, **Regular Expressions**, for more information.)

%shell (...**)** is used to execute commands in the host operating system’s shell. Since this %-function can be entered as a stand-alone directive, it is documented in the section, **Generic TSTOL Commands**.

%source (*type***)** returns the source of the current directive, where *type* is CONTEXT, LINE, or INTERACTIVE; the latter is probably the most useful. A new lexical context is pushed on the lexical stack whenever a foreign directive is executed or a procedure is started; the lexical context is popped from the stack when the directive or procedure completes. **%source (**CONTEXT**)** returns the source of the current lexical context; i.e., the keyword for a foreign directive and the file name for a TSTOL procedure. A context source is fixed throughout the lifetime of its lexical context. Because TSTOL has multiple input sources, a directive may be received from a source other than the current lexical context’s source. The dynamic source of the current directive is returned by **%source (**LINE**)**. For example, if a line of input is read from a network source during the execution of a TSTOL procedure, the context source will be the procedure and the line source will be the logical name of the network source. If the network input is itself a foreign directive, a new lexical context for the directive is pushed on the lexical stack; the new context source and line source will then be the foreign directive's keyword. The original source of the executing directive has not been lost, however. **%source (**INTERACTIVE**)** scans the lexical stack to determine the ''interactive'' source of the current directive; i.e., the source of the network or operator input that resulted in the execution of the current directive. For example, the interactive source of a foreign directive executing within a procedure started by the operator is the operator; **%source** scans past the foreign directive's lexical context and the procedure's lexical context until it encounters the interative source, the operator's lexical context. A zero-length string (‘‘’’) is returned if the requested source type is the operator.

%status looks like a %-function, but it's really a reference to TSTOL's global status variable, **%status**; see the earlier section, **TSTOL Variables**.

%time (*expr***)** returns the value of *expr*, converted to a date/time constant. If *expr* is a string in the format of a date/time constant, the string is converted to that date/time constant (useful when parsing non-standard directive arguments). Otherwise, *expr* is treated as the number of seconds since January 1, 1970 and is converted to the equivalent date/time constant.

%upper (*expr***)** returns the value of *expr* as a string, with all alphabetic characters converted to upper case.

%word (*expr*, *index***)**
returns the *i*-th ''word'' from a string, where words are delimited by blanks, commas, or tabs. Null words in a string can be specified with consecutive commas (with intervening white space allowed). A zero-length string (‘‘’’) is returned if *index* specifies a null word or if it exceeds the number of words in the string (see the **%nwords** function). **%word** is useful for scanning non-standard arguments.

Procedure Definition and Control

Automatic sequencing and execution of directives is possible using TSTOL procedure files. Once a sequence of directives is stored in a procedure file, the procedure can be called up at a later time and the directives will be automatically executed, one after another.

TSTOL procedures files are normal UNIX text files that can be created and modified using the standard system text editors. Full- and in-line comments, form feeds, and blank lines can be used to document procedures and improve their readability.

A procedure definition has the following form:

```
proc name (formal_parameters)
    ... body of procedure ...
endproc
```

name is the name assigned to the procedure. The formal parameters are place holders for the arguments being passed into the procedure. The specification of formal parameters can take one of two forms. The traditional STOL approach specifies the number of arguments that will be passed into the procedure. In the body of the procedure, the arguments are referenced by number: $1, $2, ..., $N (see Text Substitution in the next section). For example:

```
proc ADD_NUMBERS (2)
    X1 = $1 + $2
    write "Sum = ", X1
endproc
```

TSTOL also supports named procedure parameters. Arguments are then referenced by the corresponding names in the formal parameter list. As an example:

```
proc ADD_NUMBERS (FIRST, SECOND)
    X1 = FIRST + SECOND
    write "Sum = ", X1
endproc
```

When a **start** directive is executed, the file containing the procedure is loaded into memory; the memory is deallocated when the procedure returns. There are no limits on the size of a procedure file, as long as the parser doesn't run out of memory – a remote possibility thanks to virtual memory.

When a procedure is invoked, each line in the body of the procedure is echoed on the operator's screen and then executed. Various control flow directives (e.g., **goto**, **return**, **killproc**, etc.) are available to alter the sequential execution of statements. These directives can appear in the procedure itself or may be manually entered by the operator as the procedure executes.

The execution of a procedure continues until: (i) a **return** directive is executed, (ii) the **endproc** directive terminates the procedure, or (iii) the operator aborts the procedure (see the **killproc** directive).

In the event of a syntax error or a processing error, **tstol** displays an informative error message on the operator's screen and halts procedure execution in an unconditional wait state. The operator can then enter zero or more directives to correct the error, followed by a **go** or **goto** directive to exit the wait state and continue execution. (Another option, of course, is to just kill the procedure.)

The following list summarizes the TSTOL directives that affect procedure control flow:

ask	prompts the operator for input and waits until something is entered.
break	breaks out of the current **do** loop.
continue	continues with the next iteration of the current **do** loop.
do-enddo	repeatedly executes a block of statements – an unconditional **do** loop.
for-do-enddo	executes a block of statements a certain number of times – a counted **do** loop.
go	causes a procedure to exit wait mode and continue execution.
goto	unconditionally shifts execution to the specified line number or label.
if	provides for the conditional execution of a single directive.
if-then-elseif-else-endif	provides for the conditional execution of blocks of statements.
killproc	aborts the currently active procedure, as if a **return** directive was executed.
position	causes procedure execution to branch to a specified line number or label and to enter an unconditional wait state, pending operator input.
return	exits the currently active procedure and returns control to the next higher level procedure.
start	starts a procedure up and passes it the specified arguments.
step	allows the operator to control the rate of procedure execution. If timed step mode is selected, the parser pauses for the specified amount of time between each statement executed. If manual step mode is selected, the parser pauses after each statement executed; the operator must enter **go** before execution can continue with the next directive in sequence.
wait	suspends procedure execution. There are four types of wait states: unconditional (wait for operator direction), conditional (wait until an expression evaluates to

true), delta timed (wait for a certain amount of time), and absolute timed (wait until the specified GMT is reached).

while-do-enddo
executes a block of statements as long as an expression evaluates to **true** – a conditional **do** loop.

Since TSTOL is a block-structured language, there are restrictions on the destinations of **goto**, **position**, and **start-at** directives. A *block* is a group of consecutive TSTOL directives, bracketed by certain language constructs. TSTOL blocks include **if-then** blocks, **elseif** blocks, **else** blocks, and **do** loops. Performing a **goto** from outside of a block into the middle of a **do** loop, for instance, is generally meaningless. Consequently, TSTOL prohibits such transfers of control, as the following example shows:

```
goto Label_2                        ; Illegal jump.

if (expr1) then
        ... block 1 ...
        goto Label_5                ; Legal jump.
elseif (expr2) then
        Label_2:
        ... block 2 ...
else
        ... block 3 ...
        goto Label_4                ; Illegal jump.
endif

for I = 1 to N do
        Label_4:
        ... block 4 ...
enddo

Label_5:
        ...
```

Argument Passing Mechanisms

In MAE and most earlier STOLs, there was only one means of passing arguments to procedures: text substitution. In text substitution, the text of an argument string is substituted in the procedure text wherever the argument is referenced (via ‘‘$1’’, ‘‘$2’’, etc.) This substitution occurs before the affected directive is parsed. Text substitution is essentially macro expansion, a form of *call-by-name* argument passing.

TSTOL supports a more general form of text substitution, applicable to any variable (procedure-local or procedure-global), not just procedure arguments. Text substitution is initiated using the ‘‘$*n*’’ or ‘‘$*variable*’’ constructs; parentheses can be used to separate *n* or *variable* from surrounding text. For example, the following code executes the **killproc all** directive:

```
let COMMAND_OF_THE_DAY = "killproc all"
$COMMAND_OF_THE_DAY                         ; Text substitution.
```

Text substitution can be turned on and off by setting or resetting the local internal variable, **%liv(text_substitution)**. Text substitution is initially enabled when TSTOL comes up. When a procedure starts up, text substitution is automatically *enabled*; the procedure can explicitly disable text substitution if need be. When a foreign directive is invoked, text substitution is automatically *disabled*; if necessary, text substitution can be explicitly enabled in the directive’s definition. The current text substitution mode is saved and restored as procedures or directives are called and return. If procedure *A* disables text substitution and then calls procedure *B*, text substitution is automatically enabled in *B* and disabled when *B* returns to *A*. Foreign directives manipulate the substitution mode in a similar fashion.

Text substitution is applied to an input line before the line is passed to the parser's lexical analyzer. Consequently, substitutions are performed without regard to the presence of string delimiters or non-standard argument designations. If invoked by ‘‘**start** BROKE ("an elephant")’’, the following procedure:

```
proc BROKE (1)
        write "I have $1 in my pocket!"
endproc
```

will display ‘‘I have an elephant in my pocket!’’, probably not what was intended. A different formulation of the **write** directive, that avoids text substitution, would be:

```
write "I have $" & "1 in my pocket!"
```

As the command-of-the-day example illustrates, the syntax of a TSTOL statement involving text substitution cannot be determined until the text substitution actually takes place. Compiling TSTOL procedures into a fast-executing, intermediate form (a future possibility) would be hampered by the use of text substitution. Consequently, its use in procedures is discouraged. The advantages of text substitution can still be achieved in other ways in TSTOL. The **%arg(***number***)** function provides the ability to reference arguments by number. The **parse** directive allows the construction and execution of an arbitrary directive. The following procedure emulates the example code above:

```
proc EXECUTE_COD (COMMAND_OF_THE_DAY)
        parse %arg(1)           ; Equivalent to $(COMMAND_OF_THE_DAY)
endproc
```

In addition to text substitution, TSTOL also supports *call-by-value* and *call-by-reference* mechanisms for passing arguments to procedures. If an argument is passed by value (the default mode), TSTOL makes a copy of the argument's value; the called procedure is free to manipulate the argument as it pleases. If an argument is passed by reference (similar to FORTRAN-style argument passing), the address of the argument is passed to the called procedure. Any changes to the argument in the called procedure are reflected back outside of the called procedure. Call-by-reference arguments thus provide a means of returning values from procedures.

How are arguments specified in a **start** directive? Since TSTOL is primarily a command language, not a programming language, a simple identifier in an argument list is interpreted as a text string, not a variable name. To treat the identifier as a variable reference, enclose it in parentheses or explicitly request call-by-value or call-by-reference argument passing:

start XYZ (ABC, *arg2*, ...)
: passes the text string ‘‘ABC’’ to procedure XYZ.

start XYZ ((ABC), *arg2*, ...)
: looks up the value of variable ABC and passes that value to procedure XYZ.

start XYZ (**%val** (ABC), *arg2*, ...)
: passes the value of variable ABC to procedure XYZ.

start XYZ (**%ref** (ABC), *arg2*, ...)
: passes the address of variable ABC to procedure XYZ. An assignment (via a **let** directive, for instance) to the corresponding formal parameter in procedure XYZ will store a new value in variable ABC.

Numerical and string expressions can be specified as arguments without surrounding parentheses, but avoid statements such as ‘‘**start** XYZ (ABC+2, ...)’’ – the parser tries to add 2 to the string ‘‘ABC’’!

If the number of actual arguments *M* in a **start** directive is less than the number of formal parameters *N* specified in the **proc** declaration, the missing parameters (*M*+1 to *N*) are treated as null arguments. If the number of arguments exceeds the number of expected parameters (*M*>*N*), the extra arguments are retained and can be referenced within the procedure using the **%arg**(*number*) function.

Foreign Directives

tstol is a programmable parser that provides the user with the ability to define new directives at run-time using ‘‘internal’’ procedures. With this approach, similar to that used in extensible text editors (UNIX Emacs, VMS TPU, and assorted MS-DOS editors), a directive is defined in terms of other TSTOL directives. ‘‘Foreign’’ directive definitions are structured as follows:

```
directive name (formal_parameters) is
    ... directive attributes ...
begin
    ... body of directive definition ...
end
```

name specifies the directive keyword and can incorporate abbreviated short forms. Directive arguments can be specified using the argument count (MAE STOL-style) or by listing the named arguments (TSTOL-style). Directive attributes include keyword aliases (short forms), operations classes, if the directive is built-in (generic TSTOL?), and if the directive arguments should be interpreted (non-standard arguments?). The body of the directive definition specifies the procedural implementation of the directive and is not required for built-in directives. Any TSTOL directive is allowed in the body of the procedure, including **do** loops, procedure calls, and foreign commands. For example, the ICE/IMP **orbit** directive is defined as follows:

```
directive 'ORB#IT' (ORBIT_NUMBER) is
    class CC
begin
    ORBIT_NUMBER = %int (ORBIT_NUMBER)
    if ((ORBIT_NUMBER < 1) or (ORBIT_NUMBER > 99999)) then
        write "Enter an orbit number between 1 and 99999."
    else
        transact STATE_MANAGER "[XQ] ORBIT ", ORBIT_NUMBER
    endif
end
```

The ‘‘#’’ in the keyword indicates a fixed-length abbreviation (**orb** is an abbreviation of **orbit**); ‘‘*’’ can be used for variable-length abbreviations. Within the directive body, the orbit number is range-checked and an ORBIT command is forwarded to the State Manager subsystem.

tstol reads the directive definitions (from the server initialization file, for instance) and stores the internal procedures in memory for fast access. When the operator enters a ‘‘foreign’’ keyword, the parser collects up the remaining arguments on the command line and passes them to the internal procedure defined for the keyword. The effect is the same as if the foreign directive were defined as a procedure in a TSTOL procedure file and the operator entered a **start** directive. Text substitution is automatically disabled when a foreign directive is invoked; it can be explicitly enabled within the body of the directive by setting local internal variable, **%liv(text_substitution)**.

The arguments to some directives cannot be parsed correctly by **tstol**. For example, the parser has trouble with ICE/IMP’s ‘‘**simint init** E-SCI’’ directive; ‘‘E-SCI’’ looks too much like an arithmetic expression. Directives such as this are marked as **not standard** in the directive definition. When the operator enters the keyword for a non-standard directive, the parser simply reads the rest of the command line into a string and passes it as a single argument to the keyword’s internal procedure. The directive definition (somewhat abbreviated) for **simint** looks as follows:

```
directive 'SIM#INT' (REST_OF_LINE) is
    class CC
    not standard
begin
    local  ACTION, FMT

    ACTION = %word (REST_OF_LINE, 1)
    ACTION = %pick (%match (ACTION, "INIT*IALIZE", "CHANGE",  ;;
                                       "CHG", "TERM*INATE"),  ;;
                     "INIT", "CHG", "CHG", "TERM")
    if (ACTION = "") then
        ... invalid action ...
    elseif (ACTION = "INIT") then
        let FMT = %upper (%word (REST_OF_LINE, 2))
        if (%lex (FMT, "^[E-J]-(SCI | ENG | VA[A-D])$")) then
            transact STATE_MANAGER "[XQ] SIMINT INIT ", FMT
        else
            ... invalid format ...
        endif
    else
        ... other actions ...
    endif
end
```

%word is a built-in function that returns the *i*-th ‘‘word’’ from a string (the remainder of the command line, in this case), where words are delimited by blanks or commas. The **%pick-%match** expression validates the action keyword. Regular expressions provide a more concise way of matching the 30 possible telemetry formats, so a **%lex** expression is used to validate the telemetry format.

The ubiquitous ‘‘**shoval** *value format*’’ directive illustrates the use of some other built-in functions to process non-standard arguments:

```
directive SHOVAL (REST_OF_LINE) is
    not standard
begin
    local  FMT, VALUE

    VALUE = %word (REST_OF_LINE, 1)
    FMT = %upper (%word (REST_OF_LINE, 2))
    if (FMT = "B") then
        write "Value = B'", %bin (%eval (VALUE)), "'"
    elseif (FMT = "E") then
        write "Value = ", %real (%eval (VALUE), "%E")
    elseif (FMT = "H") then
        write "Value = H'", %hex (%eval (VALUE)), "'"
    elseif (FMT = "I") then
        write "Value = ", %dec (%eval (VALUE))
    elseif (FMT = "O") then
        write "Value = O'", %oct (%eval (VALUE)), "'"
    elseif (FMT = "R") then
        write "Value = ", %real (%eval (VALUE), "%f")
    elseif (FMT = "") then
```

```
                write "Value = ", %eval (VALUE)              ; Strings, etc.
            else
                write "Invalid SHOVAL format: ", FMT
                error "Valid formats are B, E, H, I, O, R, or none."
            endif
        end
```

%eval evaluates the contents of a string as if they had been typed directly into the parser. For example, ‘‘**shoval** 2**12’’ passes the string ‘‘2**12’’ to **shoval**; **%eval**ing the string produces the number 4096, which is then displayed. (Note that **%word** doesn't allow embedded blanks or commas in *value*.) **%bin**, **%dec**, **%hex**, **%oct**, and **%real** convert the number to the desired output format.

tstol attempts to make the invocation and execution of a foreign directive indistinguishable from the invocation and execution of a hypothetical, built-in directive that performs the same function. Doing so has introduced some subtleties into the process of resolving variable references, subtleties that affect foreign directives which pass non-standard arguments to the **parse** directive or the **%eval** function. Imagine a procedure called LOOP that simply displays an incrementing counter:

```
        proc LOOP (N)
            local COUNTER
            for COUNTER = 1 to N do
                shoval COUNTER
            enddo
        endproc
```

When the LOOP procedure is run, a new block containing LOOP's local variables is pushed on the symbol table stack; when LOOP returns, its block of local symbols is popped from the stack. When the **shoval** directive is executed, the string ‘‘COUNTER’’ is passed as a non-standard argument to the body of **shoval**. In earlier versions of **tstol**, a new local symbol block was created for foreign directives. **%eval**ing ‘‘COUNTER’’ then produced a *symbol not found* error, since COUNTER would not be found in the current (**shoval**) block of local symbols or in the global symbols.

Later releases of **tstol** corrected this problem by implementing *local symbol subblocks*. When a TSTOL procedure is run, a local symbol block is pushed on the symbol table stack. A local symbol block itself is a stack of local symbol subblocks. The base subblock contains the procedure's local symbols. When a foreign directive is invoked, a subblock for the directive's local symbols is pushed on the subblock stack of the current local symbol block; when a foreign directive completes, its subblock is popped from the stack. A picture is worth a thousand words (the symbol stack is growing down towards the bottom of the page):

```
MISSION          <-- global symbol block
%status
N                <-- LOOP's local symbol block (and base subblock)
COUNTER
REST_OF_LINE     <-- shoval's local symbol subblock
FMT
VALUE
```

When no procedure is active, the global symbol block functions as the current, ‘‘local’’ symbol block. Foreign directive subblocks are pushed and popped as needed, but the base subblock containing the global symbols is never popped from the symbol table stack.

Subblocks are considered part of the current local symbol block as far as the symbol resolution process is concerned. The search for a symbol in a local symbol block begins at the top of the subblock stack and descends through parent subblocks (if any) down to the base subblock. **%eval** (COUNTER) in **shoval** now returns the value of LOOP's variable. If **shoval** also declared a local variable called

COUNTER, then its declaration would hide, within the body of **shoval**, LOOP's COUNTER variable.

Normally, the details of symbol table subblocks are of no concern. Foreign directives that make use of **parse** or **%eval**, however, should be aware of the implications of evaluating a string containing a variable name: the variable might have been declared local to the directive, local to a parent foreign directive, local to the base subblock's procedure, or global. Of particular concern are nested foreign directives, since a symbol table search will scan each of the nested subblocks. To guard against problems, declare all local variables and use unique names. For example, ''**shoval** FMT'' and ''**shoval** VALUE'' will probably display meaningless numbers, given the definition of **shoval** above (FMT and VALUE are local to **shoval**!).

Watchpoints

Watchpoints provide a means of monitoring the values of system variables (e.g., telemetry mnemonics). Multiple watchpoints can be set up and simultaneously active. A requested value can be sampled every *N* seconds (*asynchronous* sampling) or when the value is decommutated (*synchronous* sampling). Whenever a value is received, a user-defined foreign directive procedure is invoked and passed the value of the system variable. Once activated, a watchpoint effectively executes in ''background'' mode; the operator can go on and enter other TSTOL directives, run procedures, etc. (Currently, watchpoints cannot ''make themselves heard'' if TSTOL is stopped because of a syntax error.)

Watchpoints are defined and activated by the **watch** directive. The **watch** directive can be entered with or without a body:

```
watch variable [ sampling_type [ rate ] ] is
begin
        ... watch body ...
end

watch variable [ sampling_type [ rate ] ]
```

The sampling type is **asynchronous** (the default) or **synchronous**. The watch body defines and is stored as a temporary, foreign directive (called a *watchpoint* directive) that will be executed whenever a value is received for the specified system variable. If no watch body is defined in the **watch** directive, a default watchpoint directive is substituted that just displays the variable's value on the operator's screen.

After storing the foreign directive, the TSTOL parser requests a stream of values for the system variable from the data server. If an **asynchronous** data stream is requested, the data server will sample and send the system variable's value every *rate* seconds. If a **synchronous** data stream is requested, the data server sends every *rate*-th occurrence of a system variable's value; the data stream is synchronous with respect to when the values are generated (e.g., when they are decommutated). Synchronous data streams are serviced by mission-specific applications, not the data server, so this mode is not necessarily supported for all system variables.

When a system variable's value is received, the watchpoint's foreign directive is invoked with a single argument, the variable's value encoded in ASCII. Watchpoint directives for telemetry points (data type: CVT_*type*) will receive a string containing several values: the raw value, the EU or processed value (if applicable), and the status flags (use **%word** to access the individual values). If the ''**P@**'' modifier is specified in the **watch** directive, the EU-converted value or the translated state name, whichever is applicable, is passed to the watchpoint's directive.

The foreign directive temporarily defined for a watchpoint is marked as non-standard; i.e., the rest of the command line following the keyword is packaged as a single string and passed to the directive. Specifying directive attributes in the watchpoint definition seems to serve no useful purpose.

A ‘‘**watch** MNF_COUNT’’ directive would simply display the satellite’s MNF counter every 5 seconds. The following, more complex example monitors the setting (ON or OFF) of the master switch in the satellite’s power subsystem:

```
watch P@MASTER_SWITCH#POWER@orion synchronous is
begin
        if (%arg (1) = "OFF") then
                write "Who turned out the lights?"
        endif
end
```

The **show watch** directive displays a list of active watchpoints on the operator’s screen. The **stop watch** directive cancels an active watchpoint:

stop watch *watchpoint*

watchpoint is the index (1..*N*) in the active watchpoints list of the watchpoint to be cancelled. The following example illustrates the normal procedure for stopping a watchpoint:

```
show watch              ; Display list of active watchpoints.
stop watch 2            ; Cancel #2 on the list.
```

Operations Privileges

TSTOL execution privileges limit the set of directives available to an operator. For example, an operator must typically have Command Controller privilege (‘‘CC’’) in order to issue spacecraft commands. The privileges required to execute a directive are specified by the ’’**class** *privilege(s)*’’ clause in the foreign directive definition for the directive. For example, **/cmd** is defined as follows:

```
directive ’/CMD’ (REST_OF_LINE) is
        class CC        ; Must be Command Controller
        not standard
begin
        ...
end
```

At first, the operator has no privileges; he or she can only execute those directives which have no privileges defined for them. Execution privileges can be changed and examined using the following directives:

access add *privilege [, privilege ...]*
: attempts to add the requested privileges to the operator’s current privileges.

access delete *privilege [, privilege ...]*
: deletes the specified privileges from the operator’s current privileges.

shoacc *[keyword]*
: displays the operator’s current privileges. If a keyword is given, **shoacc** displays the privileges needed to execute that directive.

Prior to executing a directive, TSTOL checks the privileges assigned to the directive’s keyword against the operator’s current privileges. If the operator has at least one of the keyword’s privileges (or if no privileges are required), the directive is executed. Otherwise, an error message is displayed on the operator’s screen; if a procedure was active, the procedure is halted.

Although privileges can be assigned to built-in directives, TSTOL currently checks privileges on foreign directives only. At this time, it doesn’t appear that privileges are needed for the built-in directives.

Because of the way execution privileges are implemented (described below), TSTOL doesn't know or care what operations classes are defined for a mission. TSTOL doesn't mind that ICE/IMP only has 2 classes (CC and FC) while GRO has 9 (MC, CC, PROC, FC, AN, EXR, EXL, OP, or DOCS).

How does TSTOL know that some untrustworthy operator can't become Master Controller (''MC'') and take over the world? Well, TSTOL doesn't know. The **access add** directive internally calls a foreign directive, ''_ _ACCESS'', and passes it the list of requested privileges (minus any that the operator already has).

_ _**access** is a mission-supplied foreign directive that is responsible for determining if the operator can have the requested privileges. In a typical TPOCC-based system that includes TSTOL, display, and state manager subsystems, the _ _**access** directive must communicate with Display and State Manager to verify the operator's new privileges. The following definition of the _ _**access** directive does just that:

```
directive _ _ACCESS (REST_OF_LINE) is
        not standard
begin
        local ACTION, PRIVILEGES

        %status = false
        ACTION = %word (REST_OF_LINE, 1)
        PRIVILEGES = %rest (REST_OF_LINE, 2)
        if (%match (ACTION, "ADD")) then
                transact OPIO "[XQ] ACCESS ADD ", PRIVILEGES
                if (%status)  transact STATE_MANAGER  ;;
                                          "[XQ] ACCESS ADD ", PRIVILEGES
        elseif (%match (ACTION, "DEL#ETE")) then
                transact STATE_MANAGER "[XQ] ACCESS DEL ", PRIVILEGES
        else
                write "Invalid action, ", ACTION, ", passed to _ _ACCESS directive."
        endif
end
```

If the operator already had Flight Controller (''FC'') privilege, an ''**access add** MC, CC'' directive would result in the following exchange of messages:

```
TSTOL  ---->   "[XQ] ACCESS ADD MC, CC"   ---->   Display
TSTOL  <----   "[ST] 0"                   <----   Display
TSTOL  ---->   "[XQ] ACCESS ADD MC, CC"   ---->   State Manager
TSTOL  <----   "[ST] 0"                   <----   State Manager
TSTOL  ---->   "[CL] FC, MC, CC"          ---->   Display
```

If a bad status message (non-zero status code) is received from either Display or State Manager, the **access add** directive fails and none of the requested privileges are granted. The ''[CL]'' message is generated internally by the TSTOL parser, not by the _ _**access** foreign directive. It lets the display subsystem know what the operator's current privileges are.

If _ _**access** is not defined, then TSTOL will automatically check if the operator is a member of the UNIX groups corresponding to the requested privileges. Actually, TSTOL checks if the person who started up the TSTOL program is a member of the specified groups. *That person may or may not be the operator*. After the privileges are verified, a ''[CL]'' message is output to the display subsystem.

The **access delete** directive deletes the specified privileges from the operator's current privileges, calls the _ _**access** foreign directive, and then writes a ''[CL]'' message to Display. When deleting privileges, the _ _**access** code should send an ''ACCESS DEL'' message to the state manager:

```
TSTOL  ---->   "[XQ] ACCESS DEL CC"       ---->   State Manager
```

TSTOL	<----	"[ST] 0"	<----	*State Manager*
TSTOL	---->	"[CL] FC, MC"	---->	*Display*

The display subsystem must handle two types of message, the ‘‘ACCESS ADD’’ command and the ‘‘[CL]’’ screen update. The state manager must handle both the ‘‘ACCESS ADD *privilege(s)*’’ message and the ‘‘ACCESS DEL *privilege(s)*’’ message. In both cases, an ‘‘[ST]’’ status message is expected in return. The bad ADD status message will abort the original **access** directive. A bad DEL status message is too late to do any good!

Regular Expressions

Regular expressions are a means of specifying text patterns. Built-in TSTOL functions **%rex** and **%lex** provide foreign directive and TSTOL procedure programmers with a means of applying regular expressions to arbitrary strings (e.g., directive arguments or operator input). **%rex** allows you to verify that a text string matches a given pattern. **%lex** not only lets you verify that a string matches a pattern, it also lets you extract fields from within the string based on the pattern.

The following components of regular expressions are supported by TSTOL:

c matches one instance of the specified character. For example, regular expression ‘‘abc’’ matches the string **abc**. To match a special character, such as ‘‘.’’ or ‘‘$’’, escape it with a forward slash: ‘‘\c’’.

. matches one instance of any character. For example, regular expression ‘‘a.c’’ matches the text strings **aac**, **abc**, **acc**, etc.

ˆ at the beginning of a regular expression, anchors the left edge of the match at the beginning of the target string. If a regular expression is not anchored, **%rex** and **%lex** will scan the entire text string looking for a match. For example, regular expression ‘‘abc’’ would match both **abc** and **xyzabc**. Anchored regular expression ‘‘ˆabc’’, on the other hand, will only match **abc**, not **xyzabc**.

$ at the end of a regular expression, anchors the right edge of the match to the end of the target string. For example, regular expression ‘‘abc$’’ matches **abc** and **xyzabc**, but not **abcxyz**. A regular expression anchored at both ends (‘‘ˆ...$’’) must consume the entire target string in order for the match to succeed.

[*characters*]
matches one instance of a character in the specified set of characters. Character sets can be specified literally (e.g., ‘‘[abc]’’ matches **a**, **b**, or **c**) or as a range of characters (e.g., ‘‘[A-Za-z0-9]’’ matches any upper case letter, lower case letter, or digit.

[ˆ*characters*]
matches one instance of a character *not* in the specified set of characters.

[:*class*:]
matches one instance of any character that belongs to the specified class of characters. The possible classes are **alpha**, **upper**, **lower**, **digit**, **xdigit**, **alnum**, **space**, **punct**, **print**, **cntrl**, and **graph**. Class names are not case-sensitive. Although the meanings of some of the classes are obvious, check the UNIX documentation for **ctype**(3) before using this regular expression construct.

[ˆ:*class*:]
matches one instance of any character that does *not* belong to the specified class of characters.

RE∗ matches zero or more instances of regular expression *RE*. For example, ‘‘a∗b’’ matches **b**, **ab**, **aab**, **aaab**, etc.

RE+ matches one or more instances of regular expression *RE*. For example, ‘‘a+b’’ matches

ab, **aab**, **aaab**, etc.

RE? matches zero or one instance of regular expression *RE*. For example, ‘‘a?b’’ matches **b** and **ab** only.

RE{*[m][,[n]]*}
: matches *m* through *n* instances of regular expression *RE*. If not specified, *m* defaults to 0 and *n* defaults to *m* (‘‘*RE*{*m*}’’) or to a very large number (‘‘*RE*{*m*,}’’). ‘‘*RE*∗’’ is equivalent to ‘‘*RE*{0,}’’. ‘‘*RE*+’’ is equivalent to ‘‘*RE*{1,}’’. ‘‘*RE*?’’ is equivalent to ‘‘*RE*{0,1}’’.

(*RE*) matches regular expression *RE*. The parentheses allow you to group regular expressions. For example, regular expression ‘‘(ab)∗cd’’ matches **cd**, **abcd**, **ababcd**, etc.

(*RE*)$*n*
: matches regular expression *RE* and assigns the text matched by *RE* to the *n*th variable (where *n* is a single digit in the range 0 through 9). This capability is only available in the **%lex** function; when used in a call to **%rex**, ‘‘(*RE*)$*n*’’ is equivalent to ‘‘(*RE*)’’. For example, if regular expression ‘‘((ab)∗)$0(cd)$1’’ matches **ababcd** in a call to **%lex**, then **abab** is stored in *retval0* and **cd** in *retval1*.

RE1 RE2
: matches regular expression *RE1* followed immediately by regular expression *RE2*; there should be no intervening spaces in the composite regular expression or in the target string. For example, regular expression ‘‘a∗b’’ is the concatenation of regular expressions ‘‘a∗’’ and ‘‘b’’.

RE1 | *RE2*
: matches either *RE1* or *RE2*. For example, regular expression ‘‘((abc)|(def))ghi’’ matches **abcghi** and **defghi**.

Regular expressions add powerful lexical analysis capabilities to foreign directives and TSTOL procedures, capabilities that do not need to be hard-coded into the underlying **tstol** program. The following regular expression, for instance, recognizes ICE serial magnitude spacecraft commands of the form ‘‘*mnemonic*/*X*:77.777.777.777.777’’, where *X* is command decoder A or B:

[:**alnum**:]+-?[:**alnum**:]∗/[AB]:[0-7]{2}(\ .[0-7]{3}){4}

(Example ICE command mnemonics include ‘‘137’’, ‘‘ANXMEM’’, and ‘‘BAM-ION’’; hence, the double alphanumeric pattern for the mnemonic.) Another example is the directive from the original ICE POCC that is used for changing values in a simulated, telemetry data stream:

simint chg=s*xxxyyyzzz-n*,*m*/*value*

where *xxx* is the word location, *yyy* is the minor frame, *zzz* is the major frame, *n* is the start bit, *m* is the stop bit, and *value* is the value to be inserted in the data stream. **simint** is defined as a foreign directive with non-standard arguments; consequently, **simint** receives a single argument called REST_OF_LINE. A call to **%lex** validates the format of the user input and extracts the individual ‘‘arguments’’. The regular expression used to parse the command line is:

ˆCHG=S([:**digit**:]{3})$0([:**digit**:]{3}?)$1([:**digit**:]{3}?)$2-([0-7])$3,([0-7])$4/([:**digit**:]+)$5$

And the actual directive definition:

```
directive 'SIM#INT' (REST_OF_LINE) is
    not standard
begin
    local  MJF, MNF, RE, START_BIT, STOP_BIT, VALUE, WORD

    RE = "ˆCHG=S([:digit:]{3})$0([:digit:]{3}?)$1" &  ;;
           "([:digit:]{3}?)$2-([0-7])$3,([0-7])$4/([:digit:]+)$5$"
```

```
        if (%lex (%upper (REST_OF_LINE), RE,  ;;
                WORD, MNF, MJF, START_BIT, STOP_BIT, VALUE)) then
            ... Range-check the arguments and
            ... forward them to the internal simulator.
        else
            ... invalid argument ...
        endif
    end
```

Two of TSTOL's lexical functions, **%search** and **%replace**, provide powerful search-and-replace capabilities based on regular expressions. **%search** returns the text matched by a regular expression in a target string. For example,

%search ("MB012345.678", "[**:digit:**]{3,3}")

returns "012", the day field from an IMP block history file name. (Actually, **%lex** is more suited to field extraction, but I can't think of a good example off the top of my head!)

%replace provides TSTOL with string editing abilities and is invoked as follows:

%replace (*target*, *regexp*, *replacement_text [*, *max_substitutions]*)

%replace scans *target*, looking for a substring matched by *regexp*; if found, the matched substring is replaced in a copy of *target* by *replacement_text*. Up to *max_substitutions* are made in *target*; the default is to replace all occurrences of the matched substring by *replacement_text*. The following character sequences embedded in the replacement text provide additional editing capabilities:

$*n*	Insert the subexpression (0..9) matched by ‘‘(*RE*)$*n*’’ in the regular expression.
$&	Insert the text matched by the full regular expression. For example, ‘‘$&$&’’ replaces the matched text by two copies of itself.
$l*n*	Insert the matched subexpression (0..9), converted to lower case.
$l&	Insert the text matched by the full regular expression, converted to lower case.
$u*n*	Insert the matched subexpression (0..9), converted to upper case.
$u&	Insert the text matched by the full regular expression, converted to upper case.
c	Insert character *c*; e.g., ‘‘\$’’ inserts a dollar sign in the replacement text.

To illustrate, the following invocation of **%replace**

%replace ("** abcdef **", "(abc)$1(def)$2", "$2$1 is $& split in half and reversed")

returns the following string:

‘‘defabc is abcdef split in half and reversed’’

A more useful application of **replace** is found in the definition of directive LIST_PAGES, a foreign directive which lists the names of pages the operator can bring up with the **page** command. Environment variable $UIDPATH is used by TPOCC's display subsystem to locate page definition files. The value of UIDPATH is a colon-separated list of UNIX pathnames, as in the following example:

/home/*mission*/display/uil/%U.uid:/home/*mission*/database/dandr/uil/%U.uid

When processing a **page** *name* directive, the display subsystem substitutes *name* for ‘‘%U’’ in the UIDPATH string and scans the list of directories, looking for the ‘‘*name*.uid’’ page definition file. The LIST_PAGES directive, on the other hand, is not looking for a specific file, but for a whole set of files. LIST_PAGES uses **%replace** to convert the UIDPATH to a comma-separated list of **csh**(1)-style wildcard file names:

/home/*mission*/display/uil/*.uid, /home/*mission*/database/dandr/uil/*.uid

A list of all file names matching these wildcard pathnames is displayed on the operator's console. The complete directive definition for LIST_PAGES is shown here:

```
directive LIST_PAGES is
begin
        local  FILE_NAME, I, UIDPATH

        UIDPATH = %default (%env ("UIDPATH"), "./%U.uid")
        UIDPATH = %replace (UIDPATH, ":", ",")
        UIDPATH = %replace (UIDPATH, "%[:alpha:]", "*")
        for I = 1 to %nwords (UIDPATH) do
                do
                        FILE_NAME = %fsearch (%word (UIDPATH, I))
                        break if (FILE_NAME = "")
                        write %upper (%fparse (FILE_NAME, "FILENAME"))
                enddo
        enddo
end
```

The first **%replace** substitutes commas for colons in UIDPATH; the second **%replaces** converts ''%*X*''s to asterisks.

Generic TSTOL Commands

The following TSTOL commands are available in all TSTOL languages, regardless of their mission dependencies. Sample definitions of aliases and operations classes for the built-in, generic commands can be found in **/home/tpocc/source/generic.prc**. A ''**start** GENERIC'' statement in your server initialization file will process these definitions, although the TSTOL parser still works without them. See the **directive** directive for more information on assigning directive attributes.

access add *privilege [, ...]*
access delete *privilege [, ...]*

changes the operator's current privileges for executing directives. The **access add** directive attempts to add the requested privileges to the operator's current privileges. If any one of the requested privileges is refused, the directive fails and none of the requested privileges are added. The **access delete** directive deletes the specified privileges from the operator's current privileges. The **shoacc** directive displays the operator's current privileges.

Abbreviations: **acc**
del

Examples: **access add** MC, CC, DOC ; Rule the world.
access del CC ; Relinquish command controller status.

ask *prompt [variable]*

issues a prompt to the operator and stores the next line of input from the operator into *variable*. If *variable* is not specified, the input is stored in local variable ANSWER. **abort** (or **ab**) can be entered to terminate the **ask** prompt with an error.

Example: **ask** "Host name of front-end computer? " FRONT_END

break *[* **if** *expr]*

breaks out of the current **do** loop if *expr* evaluates to a non-zero value. **break** by itself is an unconditional break.

Example: (*see* **do-enddo**)

cd *[pathname]*

changes the parser's current working directory to *pathname*; if a pathname is not specified, **cd** simply displays the current directory. This directive is useful if you're running a network-based parser server, you're editing and testing TSTOL procedures in *pathname*, and the parser's $*mission*_PROC_FILE environment variable was set so the parser can search *pathname* for TSTOL procedures. Difficult-to-parse pathnames (e.g., VMS directory specifications) should be enclosed in string quotes.

Examples: **cd** ˜/test_procs ; On UNIX.
cd "[operations.test_procs]" ; On VMS.

continue *[* **if** *expr]*

skips the remaining code in the current **do** loop and continues with the next iteration of the loop. **continue if** continues if *expr* evaluates to a non-zero value. **continue** by itself unconditionally continues the loop.

Example: (*see* **for-do-enddo**)

debug *[* **on** | **off** *]*

turns debug on and off. If debug is on, the parser outputs to **stdout** the lexical and syntactical rules that are matched during the parsing of input. If **on** or **off** is not specified, the current debug state is displayed.

dialog manual *[... message,] prompt [* **[***tag***]** *]*
dialog prompt *[... message,] prompt [* **[***tag***]** *]*

performs a dialog mode transaction. The **dialog** directive should only be submitted by applications programs communicating with the parser over a network connection (established via the **remote** directive). When a **dialog manual** directive is received, zero or more *message*s are displayed in the operator output window and the operator is prompted for input by *prompt*; the next line entered is returned to the *source* of the **dialog** directive. The parser then waits for another message from *source*; e.g., another **dialog** directive or a status message. The effect of **dialog manual** is basically equivalent to the following sequence of directives:

write *message*
... other messages ...
ask *prompt*
let ANSWER = **%upper** (ANSWER)
transact *source* "[DR*tag*] ", ANSWER

If no procedure file is active, **dialog prompt** functions just like **dialog manual**. If a procedure is active, the dialog input will be read from the procedure file, not the operator. **dialog manual** is typically used for dialogs such as critical command acknowledgements; **dialog prompt** is intended for dialogs such as spacecraft command entry.

Before a dialog response is returned to the source of a **dialog** directive, text substitution is applied to the response, leading and trailing blanks are deleted, and the entire line is converted to upper case.

Both **dialog** directives allow an optional *tag* that will be returned to *source* along with the input. A tag is a string of arbitrary characters enclosed in square brackets that the applications task can use to keep track of **dialog** directives and responses.

directive *keyword [* (*formal1, ..., formalN*) *]* **is**

    ... directive attributes ...
begin
    ... directive body ...
end

directive *keyword* **is**
    built_in
    ... other directive attributes ...
end

defines a ‘‘foreign’’ directive. The directive definition specifies the keyword that invokes the foreign directive, assigns attributes (aliases, operations classes, etc.) to the keyword, and provides a procedural definition of the directive’s execution. Directive attributes include:

alias *alias [, ...]*
defines aliases for the directive keyword. Fixed- and variable-length abbreviations can be specified by enclosing the alias in single quotes (assuming the **-quotes2** command line option) and embedding one of the special characters, ‘‘#’’ or ‘‘*’’. ‘‘#’’ denotes a fixed-length abbreviation; e.g., ’P#AGE’ matches **p** and **page**. ‘‘*’’ denotes a variable-length abbreviation; e.g., ’P*AGE’ matches **p**, **pa**, **pag**, and **page**.

built_in
indicates that the directive is built-in, i.e., the generic TSTOL directives that are hard-coded in the parser. In this case, no directive body is defined; the foreign directive definition is used only to assign attributes to the built-in directive.

class *class [, ...]*
assigns operations classes to the directive. In order to execute the directive, the operator must possess one of the corresponding capabilities.

[**not** *]* **standard**
controls the interpretation of the directive arguments. Arguments to standard directives are interpreted in the same fashion as arguments in a **start** directive. Arguments to non-standard directives are not interpreted; the parser treats the remainder of the command line following the directive keyword as a single argument. Non-standard directives should be declared as expecting a single argument; e.g., ‘‘**directive** *name* (REST_OF_LINE) **is** ...’’. The procedure body is responsible for parsing and interpreting the contents of REST_OF_LINE, not a difficult task if you use the **%**-functions available in TSTOL.

The directive body is essentially a TSTOL procedure − rather than being input and executed from a file upon demand, the directive body is input at start-up time and stored in memory. During the processing of a directive definition and after the **begin** keyword has been parsed, the directive body is input line-by-line − no parsing and no text substitution − until a line whose first word is **end** is encountered. Consequently, directive definitions should not be nested within other directive definitions. Directive definitions can appear in **if**-**then**-**else** blocks, however.

do *[* **until** *expr]*
    ...
enddo

repeats a group of commands over and over. **do until** loops until *expr* evaluates to **true** (a non-zero value). **do** by itself loops forever. **break** terminates a loop; **continue** skips to the next iteration. (The **do if** and **do while** directives available in an earlier version of TSTOL are still supported, but the use of **while-do** is preferred.)

Example:
```
do
    ask "Finished (y/n)? "
    break if (ANSWER = "y")
    ... continue processing ...
enddo
```

[status =] **%ds (***command***)**

sends an ASCII command string to the primary data server (designated by the **%liv(host)** internal variable). **%ds** returns **true** if the command was successfully sent and **false** otherwise. Currently, the data server only responds to ‘‘DEBUG ON’’ and ‘‘DEBUG OFF’’ commands.

Example:
```
%ds ("debug on")                ; Enable data server debug.
```

echo *[* **on** | **off** *]*
echo save *[* **on** | **off** *]*
echo restore

turns command input echo on and off. By default, operator and procedure input are echoed to the operator’s display. If **on** or **off** is not specified, the current echo state is displayed. **echo save** and **echo restore** push and pop settings on an echo state stack, allowing procedures to turn echo on and off without affecting the echo settings of parent procedures.

Example:
```
proc CLASSIFIED
    echo save off
    ... secret stuff ...
    echo restore
endproc
```

error *[expr [, expr ...]]*

simulates an error condition. The optional expressions are formatted, concatenated, and displayed on the operator’s screen as a single message and execution is suspended until further direction is received from the operator. **error** is useful in procedures for signalling user-detected error conditions.

Example:
```
proc CONNECT (HOST)
    remote STATE_MANAGER is MISSION & "_stmgr_parser" on HOST
    if (%status)  error "Unable to contact the state manager on " & HOST & "."
endproc
```

exec *[* **on** | **off** *]*

sets the internal execution mode of the parser. These modes include *nop*, *execute*, *proc_search*, *proc_exit*, *wait*, etc. If the parser gets lost executing your commands, **exec on** will unconditionally return the parser to *execute* mode. **exec** by itself displays the current execution mode.

Examples:
```
exec
... Modes: Execution = ‘‘proc_search’’ is displayed on the screen ...
exec on
... Modes: Execution = ‘‘execute’’ is displayed on the screen ...
```

exit *[prompt]*

terminates the parser. If *prompt* is specified, the operator is first asked if he/she really wants to exit; if the answer is yes, the program exits. This directive only terminates the parser associated with the current TSTOL input window; the directive does not bring down parsers attached to other windows or the parent server process.

Examples: **exit**
exit "The pass is still in progress; do you really want to exit (Y/N)? "

for *variable = expr [***down***]* **to** *expr [* **step** *expr]* **do**
...
enddo

repeats a group of commands a specified number of times. In an incrementing loop:

for *counter = lower* **to** *upper [* **step** *increment]* **do**

the counter variable is initialized to the lower bound; the loop is executed as long as the counter's value is less than the upper bound (the loop may not be executed at all). After each loop iteration, the step-value is added to counter. In a decrementing loop:

for *counter = upper* **down to** *lower [* **step** *decrement]* **do**

the counter variable is initialized to the upper bound; the loop is executed as long as the counter's value is greater than the lower bound (zero iterations are possible). After each loop iteration, the step-value is *added* to counter, so be sure and specify a negative increment. In all cases, **break** will prematurely exit a loop; **continue** will skip to the next iteration.

Example:
```
for I = 1 to NUM_ITEMS do
    ask "Process item " & I & " (y/n)? "
    continue if %match (ANSWER, "N#O")
    ... process item i ...
enddo
```

global *symbol [, ...]*

declares one or more global variables. The **global** command can be used anywhere and anytime, but, once entered, global variables remain in the symbol table forever. Global variables are accessible at any procedure level, unlike local variables (see the **local** command) which can only be referenced at the level in which they were declared.

Example:
```
global MISSION, PI
let MISSION = "HST"
let PI = 3.14
```

go

terminates a wait condition (see the **wait** command) or single-steps through a procedure (see the **step** command).

Abbreviations: **g**

goto *label*
goto *line*

causes the parser to branch to a specified line number or label in the current procedure. Branching into the middle of an **if-then-else** block or a **do** loop is forbidden; see the subsection, **Procedure Definition and Control**, for more information about this restriction.

Examples:
```
goto ICE_SKATE
...
ICE_SKATE:
...
```

if *expr command*

conditionally executes a single command. The command following **if** is executed if *expr* evaluates to a non-zero value. (Also see the block-structured **if-then-else** command.)

Examples:
```
if (VALUE > LIMIT)  write "Value ", VALUE, " exceeds limit."
if (ERROR)  goto ERROR_EXIT
```

if *expr* **then**
        ...
elseif *expr* **then**
        ...
else
        ...
endif

conditionally executes blocks of commands. The commands following **if** are executed if the **if**'s *expr* evaluates to a non-zero value. Commands following **elseif**s are executed if the previous **if**-**elseif**s failed and the new **elseif**'s *expr* evaluates to a non-zero value. Commands following **else** are executed if all else fails. **elseif** and **else** blocks are optional. **if**-**then**-**else** blocks can be nested.

Example:

```
do
    ask "Choice (1 = first choice, 2 = second choice, 3 = third choice)? "
    if (ANSWER = 1) then
        goto CHOICE_1
    elseif (ANSWER = 2) then
        goto CHOICE_2
    elseif (ANSWER = 3) then
        goto CHOICE_3
    else
        write "Invalid choice: ", ANSWER
    endif
enddo
```

killproc *[* **all** *]*

aborts the currently executing procedure and resumes execution of the parent procedure. If **all** is specified, all levels of procedure execution are aborted.

Abbreviations: **kp**
killp

[**let**] *variable* = *expr*
[**let**] *system_variable* = *expr*

assigns a value to a variable. *variable* can be any local or global variable (see the **global** and **local** commands); if not yet declared, *variable* is automatically declared as a local variable. System variables on both the left and right hand sides of the equals sign can be referenced as:

*mnemonic[***#***process][***@***host][***[***index***]***]*

The parser automatically establishes a network connection with the data server on the remote host and transmits the commands needed to store or recall a system variable's value.

Examples:

```
global GVAR
local LVAR
...
let GVAR = 123.456
LVAR = "abcdef"
; System variables are on both sides in the following assignment:
FORMAT_STR[2] = FORMAT_STR#ICE_DECOM@space[3]
```

local *symbol [, ...]*

declares one or more local variables. Local variables are local to the currently executing procedure and can be declared anywhere within a **proc**-**endproc** definition. A procedure's local variables disappear when the procedure exits (via **return**, **endproc**, or **killproc**). If no procedure is running, a **local** declaration is equivalent to a **global** declaration.

Example: (*see* **proc-endproc**)

memory

displays the amount of memory allocated via *str_dupl()*. This command is useful for detecting memory leaks during debug and test of the parser.

[status =] **%net (answer**, *name [, options]***)**
[status =] **%net (call**, *name [, options]***)**
[status =] **%net (close**, *name***)**
[is_connected =] **%net (connected**, *name***)**
[input_pending =] **%net (poll**, *name***)**
[message =] **%net (read**, *name***)**
[status =] **%net (write**, *name, message***)**

are used to establish and communicate across network connections; messages are exchanged as ASCII strings in XDR format. These functions can be used as stand-alone directives or embedded in expressions. If a **%net** function is entered interactively as a directive, its return value is displayed on the operator's screen.

The **answer** function listens for and accepts a connection request from a client. **true** is returned if the operation was successful; **false** indicates an error. *name* identifies the connection in subsequent **%net** calls (e.g., the **read** and **write** functions). *options* is a string containing one or more of the following, UNIX-style command line options:

–server *name*
specifies the name of the server port at which to listen for connection requests. If this option is not specified, the server name defaults to the connection name (the argument following the **answer** keyword).

–connect *directive*
specifies a TSTOL directive that is to be executed when the network connection is established. A connect directive is useful in conjunction with the **-nowait** option (see below).

–error *directive*
specifies a TSTOL directive that is to be executed when an error is detected on the connection. This error can be assumed to be a broken connection.

–input *directive*
specifies a TSTOL directive that is to be executed when input is detected on the connection. The directive must read one or more lines of input from the connection using the **%net (read**, ...**)** function. If the **-input** option is not specified, then input on this connection will be treated as standard, TSTOL network input; i.e., each line of input is read and executed as if it were a directive or status message received from a remote application.

–*[***no***]***wait**
controls whether or not **%net (answer**, ...**)** waits for a connection request to be received; **-wait** is the default. If **-nowait** is specified, the **answer** function creates a server listening socket, but does not wait for a client connection. When a connection request is eventually received, it is accepted in ‘‘background’’ mode and the **-connect** directive, if specified, is executed.

The **call** function establishes a connection with a (possibly remote) network server. **true** is returned if the connection was successfully established and **false** otherwise. *name* identifies the connection in subsequent **%net** calls (e.g., the **read** and **write** functions). *options* is a string containing one or more of the following, UNIX-style command line options:

–host *name*

specifies the host name of the computer on which the to-be-contacted server is running. The default is the local host; i.e., the computer on which **tstol** is running.

–server *name*

specifies the name of the server being contacted. If this option is not specified, the server name defaults to the connection name (the argument following the **call** keyword).

–connect *directive*

–error *directive*

–input *directive*

are described under the **answer** function above. There is no counterpart to the **-nowait** option for the **call** function, so the **-connect** directive will be executed immediately if the connection attempt is successful.

The **read** function inputs the next message from the designated network connection. The text of the message is returned as **read**'s function value; a zero-length string (‘‘’’) is returned in case of an error.

The **write** function outputs a message to the designated network connection. *message* can be an arbitrary TSTOL expression that is evaluated, formatted as a string, and output to the connection. The **write** function returns **true** if the message was succesfully written to the connection and **false** if not.

The **poll** function returns **true** if there is input pending on the specified network connection and **false** otherwise.

The **connected** function returns **true** if the specified connection has been established and **false** if not. This function is intended for checking to see if a connection request has been received and answered on a **-nowait** server port (see the **answer** function).

The **close** function closes the specified connection.

Examples: ; Connect to the state manager.

```
    %net (call, STATE_MANAGER, "-host host" &  ;;
            "-server mission_stmgr_parser" &  ;;
            "-error {%net (close, STATE_MANAGER)}")
                                          ; Send it a WORKSTATION command.
    transact STATE_MANAGER "[XQ] WORKSTATION host"
```

parse *expr [, expr ...]*

constructs and executes a directive. The expressions are formatted and concatenated to produce a single directive string, which then replaces the current line of input (i.e., the **parse** directive itself). **parse** is more powerful than text substitution and, consequently, useful when text substitution isThis capability is useful when

Examples:

```
write "Connecting to spacecraft command server ..."
write "Your menu choice of ", I, " is invalid.  Please enter it again."
```

pause *[process] [pause_attributes]*

pauses execution until input is received from the designated process or until the operator enters a command. A network connection must be established to *process* (using the **remote** directive) before pausing. If *process* is not specified, **pause** waits for input from any remote process or from the operator. **pause** only waits for input to become available - it does not actually read the input. The optional *pause_attributes* include

[**not** *]* **inline**

affects when the input received from *process* is read. The default behavior, **inline**, guarantees that the next line read by the parser will be the input from *process*. If **not inline** is specified, the input from *process* must compete with other active input sources for the parser's attention. Since the execution of a foreign directive normally locks out other input, these attributes are most meaningful in foreign directives.

timeout *seconds*

specifies how long (in seconds) **pause** will wait for input from *process*; timing out is treated as a directive error. The timeout period is initially set to 60 seconds, but it can be changed by assigning a new value to **%liv(timeout)** (see Local Internal Variables).

pause is useful in foreign directive definitions for waiting on status messages from remote processes (although the **transact** directive is preferred).

Example:
```
remote STATE_MANAGER is ...
...
tell STATE_MANAGER "[XQ] ACQUIRE ON"     ; Send command.
pause STATE_MANAGER                      ; Wait for completion status.
```

pms

prints memory statistics for *malloc*(3)'ed memory to **stdout**.

position *label* | *line*

causes procedure execution to branch to a specified line number or label and to enter an unconditional wait state, pending operator input. Branching into the middle of an **if-then-else** block or a **do** loop is forbidden; see the subsection, **Procedure Definition and Control**, for more information about this restriction.

Abbreviations: **pos**

Example: **position** START_OF_TEST

proc *name [* (*formal1, ..., formalN*) *]*
... body of procedure ...
endproc

defines a TSTOL procedure called *name* which expects arguments *formal1* through *formalN*. Ideally, procedure *name* should be stored alone in a file (see the **start** directive for file naming conventions); then, a simple

start *name* (*arg1, ..., argN*)

will call the procedure. Multiple procedures can be defined in a single file, but the **start** directive must then indicate where the procedure can be found:

start *name* (*arg1, ..., argN*) **in** *filename*

The parser will search the file at run-time for the called procedure; the procedure files do not need to be pre-processed to build a cross-reference for the parser.

Example:
```
proc SLOW_SQUARE (X, SQUARE)            ; Pass SQUARE by %ref.
   SQUARE = 0
   for I = 1 to X do
         SQUARE = SQUARE + X
   enddo
endproc
```

remote *logical_name [* **is** *server [* **on** *host]]*

establishes a network connection with *server* on the computer *host*; global variable **%status** is set **true** if the connection is successfully established and **false** otherwise. An internal table associates *logical_name* with the connection. *logical_name* can be used in the **tell** command to send a message to *server*; any input from *server* is interpreted as a normal TSTOL command. Some mission-specific directives assume connections have been established under pre-defined logical names, so the appropriate **remote** commands should be issued before executing these directives. Logical names are case-insensitive. Host and server names are case-insensitive, too; the parser converts these to all lower case before using them. Host and server names can be arbitrary expressions that are resolved into strings; remember to enclose them in quotes, when necessary. An unspecified *host* defaults to ‘‘localhost’’ (the machine you’re running on).

If **remote** *logical_name* is entered without a server specification, a dummy server entry is created in the parser’s internal table. Output to *logical_name* is not written out to the network, although event messages and so forth are generated as if it were. This feature is useful during stand-alone testing of the parser.

Example:
```
let FRONT_END = "mvme-147"
do
   remote NBP is "nbp_server" on FRONT_END
   break if (%status)
   write "Attempting to connect to Nascom Block Processor on ", FRONT_END
enddo
```

reset

attempts to reset the execution state of the parser. Use **reset** if your new TSTOL procedure has hopelessly confused the parser.

respond error *expr [, expr ...]*
respond success

return a status message to the source of the current directive. If the source of the directive is the operator (OPIO), **respond error** displays an error message consisting of the concatenation of the *expr*’s; **respond success** displays a ‘‘directive complete’’ message on the operator’s screen. If the source of the directive is a remote application on the network, **respond error** returns an ‘‘[ST] 1 *text*’’ message to the remote task, where *text* is the concatenation of the *expr*’s; **respond success** returns an ‘‘[ST] 0’’ message to the remote process.

Example: **respond error** "Invalid parameter: ", **%word** (REST_OF_LINE, 1)

return

returns control from the currently executing procedure to its parent.

Example: (*see* **proc–endproc**)

see *table*

dumps the contents of an internal parser table to **stdout**. *table* can be one of the following:

echo	dumps the echo state stack (see **echo_util.c**).
file	lists the current procedure file (see **fish_util.c**).
foreign	dumps the foreign directive table (see **fig_util.c**).
hash	dumps the foreign directive hash table (see **fig_util.c**).
help	displays a list of the tables that **see** knows about.
lex	dumps the LEX input context stack (see **lex_util.c**).
net	dumps the network client/server table (see **clash_util.c**). This table maps logical names to network connections (see the **remote** command).
reserved	dumps the reserver keyword list (see **res_util.c**).

stack	dumps the procedure file stack (see **fish_util.c**).
symbol	dumps the symbol table (see **sym_util.c**).

[status =] **%shell (open** *[, shell]***)**
[status =] **%shell (close)**
[is_connected =] **%shell (connected)**
[input_pending =] **%shell (poll)**
[input =] **%shell (read)**
[status =] **%shell (write**, *command***)**

are used to access a shell execution stream through which the TSTOL interpreter can submit commands to the host operating system's command language shell (e.g., **csh**(1) under UNIX, DCL under VMS).

The **open** function spawns a subprocess to execute the shell commands. Under VMS, this subprocess always runs the VMS Command Language Interpreter (CLI); under UNIX, you can specify a shell other than the default **/usr/bin/csh**. **open** returns **true** if the shell execution stream was successfully opened and **false** otherwise.

The **write** function is used to submit a command to the shell subprocess. The command can be an arbitrary TSTOL expression; the expression is evaluated and sent as a string to the shell subprocess. **write** returns **true** if the command was successfully written to the shell execution stream and **false** otherwise.

The **read** function is used to read the results of a command executed by the shell subprocess. **read** reads and returns a single line of text from the shell execution stream; a zero-length string (‘‘’’) is returned if there was an error reading from the stream.

The **poll** function checks to see if output from a previously-submitted command is ready for reading. **true** is returned if there is input pending from the shell execution stream and **false** if not.

The **connected** function returns **true** if the shell execution stream has been opened and **false** if not.

The **close** function closes the shell execution stream and deletes the shell subprocess. **close** returns **true** on a successful close and **false** on an error.

Examples:
```
%shell (open)                ; Display current directory (VMS).
%shell (write, "SHOW DEFAULT")
write "Current directory is " & %shell (read)
%shell (close)
```

shoacc *[directive]*

displays the operator's current privileges for directive execution. If a directive keyword is specified, **shoacc** displays the privileges required in order to execute that particular directive.

Abbreviations: **sa**

Examples:
```
shoacc                       ; Display current privileges.
shoacc SIMINT                ; Displays "MC, CC", for instance.
```

show *table*

displays the contents of an internal parser table on the operator's screen. *table* can be one of the following:

help	displays a list of the tables that **show** knows about.
syv	lists the names of the system variables known to TSTOL. **show syv** by itself lists all of the system variables. A more selective listing is obtained by specifying a wildcard variable name: ‘‘**show syv** *wildcard*’’. For

example, ‘‘**show syv** ∗VOLT∗’’ lists all variables with ‘‘VOLT’’ in their name.

watch displays a list of the active watchpoints (see the **watch** directive).

start *name [* (*arg1, ..., argN*) *]*
start *name [* (*arg1, ..., argN*) *]* **at** *label* | *line*

calls a TSTOL procedure. Arguments *arg1...argN* are bound to formal parameters *formal1...formalN* in the procedure. *name* (converted to all lower case) is assumed to be the name of the file containing the procedure; environment variable $*mission*_PROC_FILE supplies missing pathname components (e.g., directory, extension, etc.). When the procedure returns, execution will resume with the statement following the **start** command.

Entering a **start**-**at** directive causes execution of the procedure to begin at the specified line or label within the procedure. Starting in the middle of an **if-then-else** block or a **do** loop is forbidden; see the subsection, **Procedure Definition and Control**, for more information about this restriction.

Text substitution is automatically enabled at the beginning of a procedure; the previous text substitution mode is restored when the procedure exits.

Examples: **start** SLOW_SQUARE (23, **%ref** (RESULT))
start EVERYWHERE (Here, There) **at** SOMEWHERE

step *[* **on** | **off** *]*
step *time*

turns single-step mode on and off. In single-step mode, the parser pauses before each input directive and prompts the operator to enter **go** before the parser executes the next directive. **step** or **step on** activate single-step mode; **step off** turns it off. **step** *time* puts the parser in timed single-step mode – the parser pauses for *time* seconds between each input directive.

Examples:

step	; Equivalent to **step on**
step off	
step 5.0	; Execute a command every 5 seconds.

stop watch *watchpoint*

cancels an active watchpoint (see the **watch** directive). **watchpoint** is the index (1..*N*) in the active watchpoints list of the watchpoint to be cancelled; the **show watch** directive displays the list of active watchpoints.

Example:

show watch	; Display list of active watchpoints.
stop watch 2	; Cancel #2 on the list.

tell *logical_name expr [, expr ...]*

sends a message over the network to a remote server process; global variable **%status** is set **true** if the message is successfully sent and **false** otherwise. The server process is identified by its logical name (see the **remote** directive). The *expr*s are converted to ASCII, concatenated, and output as a single string. ‘‘OPIO’’ is a special logical name that designates the operator’s display.

Examples: **tell** OPIO "[XQ] JUMP"
tell STATE_MANAGER "[XQ] ACQUIRE ON"

transact *process expr [, expr ...] [pause_attributes]*

sends a message to the designated process and then waits for a response; global variable **%status** is set **true** if the transaction is successfully completed and **false** otherwise. **transact** is equivalent to an indivisible sequence of the following directives:

tell *process expr [, expr ...]*
pause *process [pause_attributes]*

The indivisibility of **transact** is important. If the non-atomic **tell-pause** were used instead of **transact**, the response from *process* might be read and interpreted before **pause** is executed. This behavior has, in fact, been seen in TSTOL procedures − the **pause** directive waits in vain. The problem does not occur in foreign directives, since network and operator input are normally locked out during the execution of a foreign directive. See the **tell** directive for more information about process names; see the **pause** directive for more information about *pause_attributes*.

Example:
```
remote STATE_MANAGER is ...
...
transact STATE_MANAGER "[XQ] SPIN SATELLITE"     ; Send command.
```

wait *[expr]*
wait until *date_time*
wait until *expr [* **timeout** *seconds]*

stalls the parser for a designated amount of time. **wait** by itself causes an indefinite wait. **wait** *expr* stops the parser for *expr* seconds. **wait until** *date_time* suspends execution until the specified GMT is reached. **wait until** *expr* re-evaluates *expr* once a second until *expr* evaluates to **true**, at which point execution continues with the next directive. The operator can terminate any type of wait state by entering **go** or a superseding **wait** command.

Abbreviations: **w**

Examples:
```
wait                          ; Wait forever.
wait 60.0                     ; Wait a minute.
wait until 12:00:00           ; Wait until lunchtime.
wait until (BATTEMP > 100)    ; Conditional wait.
```

watch events*[@host] [filtering_parameters]*

watch events*[@host] [filtering_parameters]* **is**
    ... directive attributes ...
begin
    ... watch body ...
end

sets a watch on event messages from *host*. If no host is specified, the default is the current event logging host (see the **%liv(events)** internal variable); active watches on event messages, however, do not follow changes to the event logging host. The watch body defines and is stored as a temporary, foreign directive that will be executed whenever an event message is received from the specified host. If no watch body is defined in the **watch** directive, a default watchpoint directive is substituted that displays the event message on the operator's screen.

The filtering parameters specify the types of event messages that are to be received. The parameters are specified as UNIX-style command line options encoded in a string. Valid options are:

-class *class* specifies a class of event messages that are to be received. Multiple classes can be requested by entering this option more than once in the filter string.

-critical specifies that critical commands only are to be received.

-number *start[/end]*
specifies that event numbers in the range *start...end* are to be received.

-substitute */RE/text/[***g***]*
specifies that regular expression text substitution is to be applied to each

incoming event message (on this **watch events** stream). This capability allows for the arbitrary insertion, deletion, or rearrangement of fields in fixed-format event messages. If the **g** flag is not present, the first match of regular expression *RE* in the event message is replaced by *text*; if the **g** flag is present, the replacement is applied globally to all matches of *RE* in the event message. Regular expressions and replacement texts are described in more detail in another subsection, **Regular Expressions**.

If no filtering parameters are specified, all event messages generated on the given host are received.

The **show watch** directive displays a list of active watchpoints on the operator's screen. The **stop watch** directive cancels an active watchpoint.

For more information on watchpoints, see the subsection, **Watchpoints**, that appears earlier in this manual.

Abbreviations: **event**

Examples:	**watch events** "-class telemetry -critical"	; Monitor critical telemetry events.
	watch events@dsn	; Monitor events from DSN.

watch *system_variable [sampling_type [rate]]*

watch *system_variable [sampling_type [rate]]* **is**
    ... *directive attributes* ...
begin
    ... *watch body* ...
end

sets a watch on a system variable. *sampling_type* is one of the following:

asynchronous	specifies that the data is to sampled every *rate* seconds (default = 5.0); the sampling is asynchronous with respect to when the values are generated.
synchronous	requests every *rate*-th value (default = 1) of the system variable; the stream of data is synchronous with respect to when the values are generated.

The watch body defines and is stored as a temporary, foreign directive that will be executed whenever a value is received for the specified system variable. If no watch body is defined in the **watch** directive, a default watchpoint directive is substituted that just displays the variable's value on the operator's screen.

The **show watch** directive displays a list of active watchpoints on the operator's screen. The **stop watch** directive cancels an active watchpoint.

For more information on watchpoints, see the subsection, **Watchpoints**, that appears earlier in this manual.

Abbreviations: **async**
sync

Example: (*see the subsection on* **Watchpoints**)

while *expr* **do**
    ...
enddo

conditionally executes a group of directives over and over, while *expr* evaluates to **true** (a non-zero value). **break** and **continue** provide further control of execution inside the loop.

Example:
```
while (BATTVOLT < 220) do      ; Poll telemetry point every 5 seconds.
    wait 5
enddo
```

write *expr [, expr ...]*

constructs a string of text and displays it on the operator's screen. Numeric expressions are formatted in ASCII.

Examples:
```
write "Connecting to spacecraft command server ..."
write "Your menu choice of ", I, " is invalid.  Please enter it again."
```

Sample TSTOL Server Procedure

Running **tstol** in **-answer** mode creates a TSTOL server process that listens for connection requests from operator interface tasks. When a network connection request is received, a TSTOL interpreter subprocess is forked to service the connection (i.e., to interpret directives sent by the operator interface task). To minimize the delay in bringing up an interpreter subprocess, TSTOL initialization is divided into two phases. First, the TSTOL server process reads and executes a *server* procedure, after which it listens for connection requests. Second, each forked intepreter subprocess reads and executes a *startup* procedure. (When running **tstol** in **-call**, **-single_user**, or **-tty** mode, or under an operating system that doesn't support **fork**(3), the server and startup procedures will both be read by the same process.)

The server procedure is typically a large file containing definitions for the mission-specific TSTOL directives. The startup procedure, on the other hand, is a small file containing initialization activities that must await the connection to the operator interface task: establishing network connections with remote applications, bringing up display pages, etc.

TSTOL server procedures are generally layed out as follows:

```
;*********************************************************************
;
;          TSTOL Server Initialization Procedure
;
;*********************************************************************

proc mission_SERVER
        echo save off

        local LOG_STATE                    ; Disable procedure logging.
        LOG_STATE = %liv (log_procedure)
        %liv (log_procedure) = false

        ... global variable declarations and initialization ...

        ...
        ... directive definitions ...
        ...

        write MISSION, " TSTOL Server initialization complete."

        %liv (log_procedure) = LOG_STATE
        echo restore
endproc
```

Procedure statements are normally logged as they're executed. Since a server initialization procedure may be a 1000 or more lines in length, none of which is of any interest to the user, procedure logging is disabled at the beginning of the server procedure and then restored to its prior state at the end of the procedure.

Global TSTOL variables are primarily used to hold constants. In the following example from the server initialization procedure for the X-Ray Synthetic Aperture Radar (X-SAR) mission, a number of regular expressions used to validate directive arguments are defined:

```
;*******************************************************************
;
;    Global variables:
;
;         FORMAT_RE - is a regular expression that defines the legal
;                   telemetry formats for the mission.
;
;         GEOMETRY_RE - is a regular expression that defines the
;                   format of X Windows geometry specifications for
;                   the TPOCC display program (see the PAGE directive).
;
;         STATIONS_RE - is a regular expression that defines the
;                   mission's legal station IDs.
;
;         SUBSYSTEM - is the subsystem ("TS" or "CS") in which TSTOL
;                   is running.  This variable is set by the startup
;                   procedure, XSAR_STARTUP.
;
;*******************************************************************

     global  FORMAT_RE, GEOMETRY_RE, STATIONS_RE, SUBSYSTEM

     FORMAT_RE = "^(SCI | ENG | MAN | CON)$"
     GEOMETRY_RE = "^([0-9]+[xX][0-9]+)?([-+][0-9]+[-+][0-9]+)?$"
     STATIONS_RE = "(JS1) | (JS2)"
     SUBSYSTEM = "TS"              ; "TS" or "CS", set in "xsar_startup.prc".
```

The simplest directives simply forward themselves to another task for processing. For example, X-SAR's **static** directive instructs the report generator to bring up a static display page (animated by archival telemetry data):

```
;*******************************************************************
;         STATIC <page_name>
;*******************************************************************

     directive STATIC (PAGE_NAME) is
     begin
          %status = false
          if (PAGE_NAME = "") then
               error "Enter ""STATIC <page_name>""."
          else
               transact REPORTS "[XQ] STATIC ", PAGE_NAME
          endif
     end
```

The **transact** statement sends the **static** directive to the report generator and waits for a status message in return; the status message will automatically set the global **%status** variable.

The **/cmd** directive, used to send spacecraft commands to the spacecraft command processor, is a special case (in many ways!). If **/cmd** is entered without any arguments, the spacecraft command processor enters into a dialog with **tstol**. Since **tstol** must exit **/cmd**'s directive body before it can carry on an open-ended conversation with the command processor, the **transact** directive is qualified with the **not inline** attribute. If **/cmd** is entered with arguments (e.g., an immediately-uplinked spacecraft command), the directive is executed as a normal exchange of directive and status messages with the spacecraft command processor. In the following definition of **/cmd**, typical for the missions on which this directive has been used, **tstol** communicates with the spacecraft command processor via the state manager:

```
;*********************************************************************
;          /CMD Spacecraft Commands
;*********************************************************************

        directive '/CMD' (REST_OF_LINE) is
                alias '/'
                class CC
                not standard
        begin
                %status = false
                if (REST_OF_LINE = "") then
                        transact STATE_MANAGER "[XQ] /CMD" not inline
                else
                        transact STATE_MANAGER "[XQ] /CMD ", REST_OF_LINE
                endif
        end
```

X-SAR's **database** directive causes the Database Interface (DBIF) task to open up a window on the screen for viewing or editing database tables. In the definition below, the **%match** function is used to validate the action keyword; **%pick** provides a simple means of converting the different abbreviations of ''EDIT'' to the fixed keyword expected by the Database Interface task:

```
;*********************************************************************
;          DATABASE VIEW | EDIT
;*********************************************************************

        directive DATABASE (ACTION) is
                alias DB
        begin
                %status = false
                ACTION = %pick (%match (ACTION, "ED*IT", "VIEW"), "EDIT", "VIEW")
                if (ACTION = "") then
                        error "Enter ""DATABASE <VIEW | EDIT>""."
                elseif (%net (connected, DBIF)) then
                        transact DBIF "DATABASE ", ACTION
                else
                        error "Unable to communicate with the Database Interface task."
                endif
        end
```

The definition of the **page** directive illustrates how to parse a non-standard command line using TSTOL's built-in lexical functions. The directive is entered as follows:

page *page_name [, [interval] [, [geometry] [,* **protect***]]]*

Because **page** was defined as a non-standard directive, the rest of the command line, from *page_name* on, is passed without interpretation to the directive body as a single string, REST_OF_LINE. Each of the four arguments is extracted from REST_OF_LINE using the **%word** function. The page name must be present. The update interval, a number, is checked with the **%isnum** function. The page geometry is a single argument specifying the dimensions and location of the page:

*[width***x***height][+ | -x_coord+ | -y_coord]*

This argument is most easily handled by a regular expression: if **%lex** is unable to match the user-specified page geometry against regular expression GEOMETRY_RE (declared as a global TSTOL variable above), an error message is displayed. A simple **%match** suffices for the fourth and final argument, the page protection flag:

```
;************************************************************************
;          PAGE <page_name>, [<interval>], [<geometry>], [PROTECT]
;************************************************************************

        directive PAGE (REST_OF_LINE) is
            alias P
            not standard
        begin
            local PAGE_NAME, UPDATE_INTERVAL, GEOMETRY, PROTECTION

            %status = false
            PAGE_NAME = %word (REST_OF_LINE, 1)
            UPDATE_INTERVAL = %word (REST_OF_LINE, 2)
            GEOMETRY = %word (REST_OF_LINE, 3)
            PROTECTION = %word (REST_OF_LINE, 4)
            if (PAGE_NAME = "") then
                write "No page name specified in PAGE directive."
                error "Enter ""PAGE <page_name>, [<update_interval>], " &        ;;
                      "[<geometry>], [PROTECT]""."
            elseif ((UPDATE_INTERVAL <> "") and                          ;;
                  not %isnum (UPDATE_INTERVAL)) then
                write "Invalid update interval (", UPDATE_INTERVAL,             ;;
                      ") in PAGE directive."
                write "Enter ""PAGE <page_name>, [<update_interval>], " &        ;;
                      "[<geometry>], [PROTECT]"", where"
                error "<interval> is the interval in seconds between page updates."
            elseif ((GEOMETRY <> "") and                          ;;
                  not %lex (GEOMETRY, GEOMETRY_RE)) then
                write "Invalid page geometry """, GEOMETRY, """ in PAGE directive."
                write "Enter ""PAGE <page_name>, [<update_interval>], " &        ;;
                      "[<geometry>], [PROTECT]"", where"
                error "<geometry> is formatted as " &                          ;;
                      """[<width>x<height>][+/-<x_coord>+/-<y_coord>]""."
            elseif ((PROTECTION <> "") and
                  not %match (PROTECTION, "P*ROTECT")) then
                write "Invalid protection argument """, PROTECTION,             ;;
                      """ in PAGE directive."
                error "Enter ""PAGE <page_name>, [<update_interval>], " &        ;;
```

```
                        "[<geometry>], [PROTECT]""."
          else
               if (PROTECTION <> "")  PROTECTION = "PROTECT"
               transact OPIO "[XQ] PAGE ", PAGE_NAME, ", ",          ;;
                        UPDATE_INTERVAL, ", ", GEOMETRY, ", ", PROTECTION
          endif
     end
```

The X-SAR **view** directive brings up a separate editor window in which the operator can view a file. The **view** directive uses **%shell** functions to fork a subprocess and issue an editor command to it. In the directive definition below, a VAX/VMS DCL subprocess is created by **%shell(open)** and a SPAWN command is submitted to the DCL intepreter to spawn an editor subprocess in the background. (In a UNIX environment, you would use **&** to put the editor subprocess in the background.) A WRITE SYS$OUTPUT command is queued up after the SPAWN command to return a ‘‘-- Done --’’ message to the TSTOL interpreter. Since the SPAWN command must complete before the WRITE SYS$OUTPUT command is executed by the DCL interpreter, this message acts as a ‘‘caboose’’ to the ‘‘train’’ of output from the preceding commands (i.e., the SPAWN directive). This is a commonly-used technique in directives that interact with the shell subprocess. (In a UNIX environment, you would use **echo**(1) instead of WRITE SYS$OUTPUT.)

```
;**********************************************************************
;          VIEW <file_name>
;**********************************************************************

     directive VIEW (REST_OF_LINE) is
          not standard
     begin
          local DCL_OUTPUT, FILE_NAME, VIEW_COMMAND

          %status = false

          FILE_NAME = %word (REST_OF_LINE, 1)
          if (FILE_NAME = "") then
               error "Enter ""VIEW <file_name>""."
               return
          endif
          FILE_NAME = %fparse (FILE_NAME, MISSION & "$REPORTS:.DMP")

          VIEW_COMMAND = "SPAWN/NOWAIT/INPUT=NL:/OUTPUT=NL: "       ;;
              & "EDIT/TPU/READ_ONLY/INTERFACE=DECWINDOWS "           ;;
              & FILE_NAME
                                                  ; Has the shell been opened?
          if (not %shell (connected) and not %shell (open)) then
               error "Error opening a DCL subprocess: " & %liv (errno)
               return
          endif

          if (%shell (write, "ON ERROR THEN CONTINUE") and   ;;
              %shell (write, VIEW_COMMAND) and               ;;
              %shell (write, "WRITE SYS$OUTPUT ""-- Done --""")) then
               do                                 ; Read DCL output until done.
                    DCL_OUTPUT = %shell (read)
                    break if (DCL_OUTPUT = "-- Done --")
                    write DCL_OUTPUT
```

```
                enddo
            else
                error "Error issuing """ & VIEW_COMMAND &  ;;
                    """ to DCL: " & %liv (errno)
                return
            endif

            %status = true

        end
```

Sample TSTOL Startup Procedure

A mission's *server* initialization procedure is primarily used to define foreign directives. The mission's *startup* procedure, effectively executed when the operator ‘‘logs on’’ to TSTOL, is more flexible and can make use of user-unique or time-dependent information to dynamically customize its initialization actions.

For example, the X-SAR startup procedure performs the following functions:

- Stores the subsystem name, ‘‘TS’’ (telemetry subsystem) or ‘‘CS’’ (command subsystem), in a global TSTOL variable.
- Instructs the Dynamic Display process to display any initial pages (e.g., the events page and the TS or CS status page) on the screen.
- Listens for and establishes a network connection with the User Interface process (the menu system).
- Listens for and establishes a network connection with the Database Interface process.
- Establishes a network connection with the State Manager/
- Establishes a network connection with the Report Controller (TS only).
- Issues the appropriate initialization commands to the above tasks.

TSTOL startup procedures are generally layed out as follows:

```
;**********************************************************************
;
;           TSTOL Startup Procedure
;
;**********************************************************************

proc mission_STARTUP (arguments)
        echo save off

        ... local variable declarations and initialization ...

        ...
        ... startup processing ...
        ...

        write "********** " & MISSION & " (" & SUBSYSTEM & ") TSTOL **********"

        echo restore
endproc
```

The X-SAR control center is a distributed system consisting of a telemetry host computer running the real-time telemetry software (e.g., telemetry reception and decommutation), a command host running the real-time command software (e.g., command packaging and uplink), and a number of workstations running the operator interface software (e.g., TSTOL and the graphical user interface). Each instance of TSTOL must establish network connections with applications running on the real-time hosts and on its own workstation. Consequently, X-SAR's TSTOL startup procedure expects two arguments: the subsystem name (TS or CS) and the name of the subsystem's real-time host. The subsystem name is stored in a global TSTOL variable (declared in the server initialization procedure) and the host name is converted to lower-case:

```
proc XSAR_STARTUP (TS_OR_CS, REALTIME_HOST)
    echo save off

    local  LOCAL_HOST, MESSAGE, SERVER_NAME

    SUBSYSTEM = %uppper (%default (TS_OR_CS, "TS"))
    LOCAL_HOST = %lower (%liv (localhost))
    REALTIME_HOST = %lower (%default (FRONT_END, LOCAL_HOST))

    ...
```

The startup procedure then initiates the display of the local (TPOCC) events page and, depending on which subsysetm is being accessed, the TS or CS status page:

```
page LOCAL_EVENT, , , +0+830, PROTECT
if (SUBSYSTEM = "CS") then
    page CS_STATUS, , , +945+740, PROTECT
else
    page TS_STATUS, , , +735+830, PROTECT
endif
```

Each X-SAR TSTOL process has network connections with 5 other applications (not including the data server and the events subsystem): the dynamic (TPOCC) display task, the menu interface task, the database interface task, the report generator, and the state manager. The state manager runs on the real-time host computer; the other programs run on the operator's workstation. The dynamic display task is the first to connect to TSTOL, thereby ‘‘logging’’ the operator on to TSTOL. The X-SAR startup procedure is responsible for establishing the remaining 4 connections. TSTOL is a server with respect to the menu interface task, so it listens for a connection request from that client:

```
if (SUBSYSTEM = "CS") then                  ; Command subsystem?
    SERVER_NAME = %lower (MISSION) & "_csm"
else                                        ; Telemetry subsystem.
    SERVER_NAME = %lower (MISSION) & "_tsm"
endif

%status = %net (answer, UIF, "-server " & SERVER_NAME & " " &  ;;
                "-error {%net (close, UIF)} " & "-nowait")
if (%status) then
    write "Listening for a connection request on " & SERVER_NAME & "."
else
    write "Error listening for a connection request on " &  ;;
              SERVER_NAME & ": " & %liv (errno)
```

```
        endif
```

The **-nowait** option in the **%net(answer)** call puts the ‘‘listen’’ in the background and allows TSTOL to continue processing. When a connection request is finally received from the user interface task, the network connection is automatically established without further intervention by TSTOL. TSTOL is also a server to the database interface task, so a similar piece of code listens for connection requests from that task.

TSTOL is a client of the report generator and the state manager, so these tasks listen for connection requests from TSTOL. Since TSTOL may be up and running before the report generator and the state manager, it periodically ‘‘calls’’ them until they ‘‘answer’’. The following extract from X-SAR’s startup procedure establishes a connection to the state manager and sends it a message identifying TSTOL by its workstation name:

```
SERVER_NAME = %lower (MISSION) & "_stmgr_parser"

while (not %net (connected, STATE_MANAGER)) do
    %status = %net (call, STATE_MANAGER, "-host " & REALTIME_HOST &     ;;
                " -server " & SERVER_NAME &;;
                " -error {%net (close, STATE_MANAGER)}")
    if (not %status) then
        MESSAGE = "Error establishing a network connection with "     ;;
                & SERVER_NAME & "@" & REALTIME_HOST & ": "   ;;
                & %liv (errno)
        write MESSAGE
        wait 10         ; Try again in 10 seconds.
    endif
enddo

write "Established a network connection with " &  ;;
      SERVER_NAME & "@" & REALTIME_HOST & "."
transact STATE_MANAGER "[XQ] WORKSTATION " & LOCAL_HOST
```

FILES

/home/tpocc/obj_*arch***/tstol/tstol**
Executable for TSTOL server/processor.

/home/*mission***/source/procs/***mission***_server.prc**
Procedure file read and executed by the TSTOL server at initialization time; the file should contain a procedure named *mission*_SERVER. The TSTOL server is the parent process that listens for connection requests from the display subsystem and forks child TSTOL processors to service accepted connections. Perform time-consuming start-up tasks that are common to all child processors (e.g., loading directive definitions) in the server initialization file; by doing so, the start-up time for the child processes is reduced considerably.

/home/*mission***/source/procs/***mission***_startup.prc**
By convention, the procedure file read and executed by the TSTOL processor at initialization time (see the **-startup** command line option). The file should contain a procedure named *mission*_STARTUP. To keep processor start-up time to a minimum, make a sensible division of responsibilities between the server initialization file and the processor initialization file.

/home/tpocc/obj_*arch***/tools/maple**
Executable for Multiple APpLications Emulator. This program emulates a typical state manager or applications task and is useful for stand-alone testing of the parser without having to bring up the remote applications tasks with which a mission-specific parser must

communicate. Directives sent by the parser to the emulator produce a successful status message in response. In addition, spacecraft command dialogs are supported; a ''/'' directive sent by the parser initiates a chain of **dialog prompt** directives from the emulator, which continues until **abort** or **end** is received from the parser. Testing the X-SAR TSTOL provides a good example of using **maple**. The X-SAR parser must establish connections with and talk to the REPORTS and STATE_MANAGER subsystems. The following command starts up an emulator for both of these subsystems, as well as a data ''sink'' for event messages:

```
% maple mission_reports_server mission_stmgr_parser -z mission_logger_subsys
% ... run TSTOL ...
```

The non-option arguments on **maple**'s command line are the network server names of the REPORTS and STATE_MANAGER tasks; the **-z** option specifies the event logger's server name.

/home/tpocc/obj_*arch*/tools/prosper

Executable for PROcess SPawnER. **prosper** is used to spawn a program when needed. **prosper** was written for use under operating systems which don't support UNIX-style *fork*(2)s. **prosper** simply spawns a specified program and waits for the program to signal a semaphore. When the subprocess signals the semaphore, **prosper** spawns another copy of the program and waits for it to signal the semaphore. And so on and so on.

When **prosper** spawns a program, it inserts a ''**-@** *semaphore*'' option in the subprocess's argument list. *semaphore* is the integer identifier of the semaphore that the subprocess should signal when it is time for **prosper** to spawn another copy of the program.

prosper was specifically written to run TPOCC server processes under VMS (although **prosper** will compile, link, and run under UNIX). Some TPOCC programs such as TSTOL and the Report Server execute as network servers. When a network connection request is received, the server process forks a subprocess to handle the new connection. Under UNIX, the new connection's socket can be passed from the server process to the child process. Not so under VMS! With **prosper**, the server process must act as both the server and the child process. Initially, the server opens a listening port socket and waits for a connection request. When a request is received, the server process closes its listening port socket, signals the **prosper** semaphore, and goes off to service the new connection. **prosper**, upon being signalled, starts up a new server process, which creates a listening port socket, waits for a connection request, etc., etc.

TSTOL's **-single_user** option, originally intended for testing purposes only, turned out to be just the ticket for the **prosper** mode of operation. Under VMS, the following command sequence:

```
$ tstol :== $disk:[directory]tstol.exe
$ prosper tstol -mission mission -single_user ... other TSTOL options ...
```

runs TSTOL as a single-user network server: when the server process receives a connection request from an operator interface task, another server process is spawned to listen for the next connection request.

DIAGNOSTICS

Meaningful error messages are generated.

LIMITATIONS

Compile-time limits are defined in header file **max.h**:

- Operator input lines are restricted to 1024 characters, as are procedure input lines. Continuation lines can be used for longer commands, although the total length of a **wait until** directive should not exceed 1024 characters.
- Variable names are limited to 32 characters.
- Echo state saves can only be nested to a depth of 32.

- 32 connections at most can be established with the **remote** directive.

There are no practical limits on:

- the length of a character string.
- the number of lines in a procedure file.
- the number of consecutive continuation lines in a procedure file.
- the number of arguments to a directive or procedure.
- the number of foreign directive definitions.
- the number of local or global symbols.

Currently, there is no controlled limit to the depth of nesting for **if-then-else** statements, **do** loops, TSTOL procedures, and foreign directives. The size of the YACC token stack *does* limit the nesting depth, but that limit may vary depending on what is being parsed. For example, while the stack may support 8 levels of **do** loops or 8 levels of **if** statements, it may not support a combination of the two. Some simple tests showed that the following numbers of tokens are required for each level of nesting: 6 tokens for an **if-then** block, 14 tokens for a **for-do** loop, 26 tokens for a TSTOL procedure, and 21 tokens for a foreign directive.

The YACC token stack is sized at compile-time by setting a C preprocessor macro called YYMAXDEPTH. The current setting is 512, which means that a recursive factorial procedure is able to compute 19!, but not 20!. If the YACC token stack does overflow, an error message is output to the operator and the parser restarts itself. Restarting TSTOL is essentially equivalent to entering **killproc all** followed by **reset**; the values of global variables are preserved. (When **tstol** is built under HP/UX, macro __RUNTIME_YYMAXDEPTH is defined, which enables dynamic sizing of the YACC stack at run-time. Since the stack can now grow as needed, stack overflow is an unlikely event.)

(The following figures have not been updated recently.) Some rough benchmark testing of ICE/IMP TSTOL was performed using the **mgrsim** state manager simulator (a precursor to **maple**); **tstol** was running in its terminal interface mode with procedure echoing turned off. The first test procedure consisted of a **for** loop that output 5 spacecraft commands per iteration (and waited for status responses from **mgrsim** for each command output). The result: 1000 spacecraft commands output at a rate of 50 commands per second. The second test procedure initiated a spacecraft command dialog and then fed 1000 dialog-mode commands to the state manager (and waited for dialog mode responses from **mgrsim** for each line of dialog input). The result: 1000 spacecraft commands output at a rate of 130 commands per second. The tests were performed on a Sun SPARCstation 1 workstation, with both **tstol** and **mgrsim** running on the same machine. Both test procedures were also performed with **tstol** running on the SPARCstation and **mgrsim** running on a Sun-3/80: slower, but still a respectable 40 commands per second for the first test and 95 commands per second for the second test.

BUGS

Not too many, I hope!

Dialog-mode sequences in procedures are problematic. The parsing of the dialog responses are the responsibility of the application that requested them. TSTOL is unable to parse them and, in particular, to determine where the sequence ends. As a result, dialog-mode sequences can't be embedded in **if-then-else** constructs and nor can you **goto** around them. An awkward work-around is to store each dialog-mode sequence in a separate procedure file and to conditionally execute the appropriate **start** directives in a higher-level, driver procedure.

Branching to an undefined label or line number in a procedure effectively aborts the procedure. TSTOL scans all the way through to the **endproc** looking for the label or line number. By the time the **endproc** is reached and TSTOL has realized the destination is invalid, the procedure has exited. Conceivably, the procedure could be halted at the **endproc** and not allowed to exit, but it would definitely be impossible to restore the parse context at the time the offending **goto** directive (or whatever) was parsed. This is primarily a problem when TSTOL is in a wait state (due to an error or a **wait** directive) and the operator interactively enters a **goto** or **position** directive to change the control flow of the procedure. Many, but not all, of these cases could be eliminated by letting the procedure itself jump around in response to errors or other conditions.

A potential problem exists for watchpoints. When a system variable is referenced in an expression, **tstol** sends a one-shot data request to the data server and waits until that variable's value is received. If two or more values for a single watchpoint are received between the time the one-shot request goes out and the requested value comes in, then each new watchpoint value overwrites the preceding one. This is unlikely to be a problem unless you are dealing with a high-rate synchronous data stream.

The lexical analyzer can be built by the standard LEX processor or by GNU's super-LEX processor, FLEX. The FLEX-generated lexer did have some problems scanning quoted strings with embedded quotes. I forget the details (it's been a year and a half), but the lexer apparently was trying to back-track into a previous buffer of input. I didn't delve into the problem too deeply (we had LEX after all), so use FLEX at your own risk!

The shift/reduce conflicts (203 at last count) reported by YACC when it generates the parser are not of serious concern. The one ''true'' shift/reduce conflict is caused by the spacecraft command ''/'' mark. If ''/'' is encountered during the scan of an ''**if** *expr command*'' directive, the parser can't tell if ''/'' is a division operator embedded in *expr* or the start of a spacecraft command directive in *command*. The parser assumes the former interpretation.

The list of attributes in a **pause** directive accounts for 14 of the shift/reduce conflicts. When faced with one of the 3 possible attributes (**inline**, **not inline**, and **timeout**) at the beginning of an attribute list, the parser could (but doesn't) insert an empty attribute into the list. Likewise, in the middle of or at the end of the list, the parser could (but doesn't) insert an empty attribute when faced with one of the 3 possible attributes or the end-of-line separator. A duplicate set of 7 shift/reduce conflicts occurs for the **transact** directive, which also makes use of the **pause** attributes.

The other shift/reduce conflicts (188 so far) are all related to the fact that TSTOL allows either blanks or commas as argument separators. The TSTOL parser does not actually see blanks in the input; the grammar rule for two consecutive arguments separated by blanks or commas is simply ''*arg_list* => *arg1 field_sep arg2*'', where ''*field_sep* => *<empty>* | *comma*''. The shift/reduce conflicts include the following:

60 conflicts -	When parsing input according to the rule for starting TSTOL procedures, ''*directive* => **start** *procedure* (*arg1, arg2, ...*)'', the parser could (but doesn't) insert an empty *arg0* between the left parenthesis and the first argument. (The parser *will* recognize an empty argument if the first token after the parenthesis is a comma.)
60 conflicts -	Foreign directives are basically **start** directives without the parentheses. Consequently, the grammar rule for foreign directive invocation, ''*foreign_directive* => *keyword arg1, arg2, ...*'', suffers the same shift/reduce conflicts as the rule for **start** directives. In this case, the conflict occurs between the keyword and the first argument.
60 conflicts -	Once into the argument list, the parser can still run into problems. After moving beyond the comma separating *arg1* from *arg2*, the parser could (but doesn't) insert an empty *arg1.5*.
6 conflicts -	Immediately after processing an argument, the parser could recognize an empty (blank) field separator, no matter what input follows. This is the correct interpretation in most cases, except when the next input token is a non-empty (comma) field separator (in the middle of an argument list), an end-of-line separator (which terminates a foreign directive's argument list), or a right parenthesis (which terminates a **start** directive's argument list). The parser resolves each of these conflicts correctly.
2 conflicts -	The two unary operators, ''+'' and ''–'', are another source of confusion for

the parser. Faced with ‘‘2+3’’ in the middle of an argument list, the parser could (but doesn’t) parse the input as two arguments, 2 and +3. Instead the input is interpreted as a single argument whose value is 5.

Although formal parameter lists in procedure and foreign directive definitions look like argument lists, they do not cause shift/reduce conflicts. This is because an argument list can contain empty arguments, but a parameter list cannot contain empty parameters.

In the 3 groups of 60s, a conflict is generated for each possible token that could lead off a new expression (i.e., the next argument): numbers, strings, **%**-functions, **sin**, **cos**, etc. Adding a new data type or function will automatically increase the number of shift/reduce conflicts by three.

By only allowing commas to separate arguments, all the shift/reduce conflicts above (except the spacecraft command one) were eliminated. Having the lexical analyzer explicitly detect ‘‘blank’’ field separators also eliminated the conflicts, but proved too restrictive on TSTOL’s free-format input. Since YACC assumes the desired interpretation in each case, the shift/reduce conflicts are an aesthetic problem, not a functional problem.

VERSIONS

I left the TPOCC project in February of 1992 and went to work on the X-SAR project, on which we ported TPOCC to VAX/VMS. The last release of TPOCC that we received was 7.0, although the port of TSTOL is actually the Release 6.1 version of TSTOL, ported and upgraded with most of the functionality added in Release 7.0’s version of TSTOL. This manual documents the X-SAR version of TSTOL. Most of the manual also applies to TPOCC Release 7.0’s version of TSTOL; exceptions are noted below.

The X-SAR version of TSTOL:

– can be built under either UNIX or VMS by ANSI or non-ANSI C compilers.

– utilizes ANSI C function prototypes for the TSTOL functions and for the TPOCC library functions.

– uses ANSI C header files whenever possible.

– has been compiled with a compiler (VAX C) that performs the argument type checking made possible by function prototypes.

– works with either a Bison- or a YACC-generated parser.

– works with either a FLEX- or a LEX-generated scanner.

The X-SAR version of TSTOL has been enhanced with the following capabilities, none of which are mission- or operating system-specific:

– X-SAR’s TSTOL can function either as a client or as a server of the operator interface task (see the **-call** and **-answer** command line options). TPOCC’s TSTOL only supports the server mode.

– X-SAR’s TSTOL recognizes full-word command line options (e.g., **-mission**) as well as TPOCC’s single-letter options (e.g., **-m**).

– The ‘‘**-@** *semaphore*’’ command line option makes it possible for TSTOL to be run as a multi-process network server under operating systems that don’t support the UNIX *fork*(2) call (e.g., VMS). (See the description of **prosper** in the **FILES** section.)

– Numeric exceptions (e.g., overflow, underflow, divide-by-zero, etc.) are trapped using the portable UNIX *signal*(3) and *setjmp*(3)/*longjmp*(3) mechanisms.

– Several new local internal variables (**%liv**) have been added: **errno**, **lex_debug**, and **net_debug**.

– New **%** functions include **%ds**, **%ident**, **%isint**, **%net**, **%shell**, and **%source**.

- The **%ds** function is used to send ASCII commands to the Data Server. This capability is currently used to dynamically turn Data Server debug on and off and to display memory statistics.
- The **%net** functions provide a flexible, powerful, and straightforward means of establishing and communicating over network connections. These functions feature client- and server-mode connections; background answering of server-mode connections; and the ability to bind TSTOL directives to a connection's answer, input, and error events. TPOCC's **remote** and **transact** directives are still supported.
- The **%shell** functions provide access to the operating system's command language interpreter (e.g, **csh**(1) under UNIX and DCL under VMS). Shell commands are written to a single subprocess spawned to execute these commands; the output from such commands can, in turn, be read by TSTOL.
- The **watch events** directive (and the X-SAR Event Server) recognizes full-word filtering parameters (e.g., **-critical**) as well as TPOCC's single-letter parameters (e.g., **-c**). A new **-substitute** option allows for the arbitrary cutting and pasting of event message texts (implemented in the Event Server).
- In most places, time intervals can be specified as real numbers; e.g., 1.234 seconds instead of TPOCC's more limited choice of 1 or 2 seconds.
- When a status message is received from a remote process, TPOCC's TSTOL forwards the message to the operator interface task (OPIO) in the form of a ''Directive Complete'' message or, if there was an error, an error message. X-SAR's TSTOL, in contrast, makes an informed guess as to the source of the original directive and sends the completion or error message to that source. For example, X-SAR's menu interface, a remote process to TSTOL, is our main source of directives and it expects status messages to be returned to it, not to the ''operator interface'' task (TPOCC's **xtpdsp** program).
- The handling of syntax and similar errors is, I believe, more robust and general in X-SAR's TSTOL.

Areas of incompatibility between X-SAR's TSTOL and TPOCC's TSTOL (as of TPOCC Release 7.0) are few:

- X-SAR's TSTOL provides limited support for STOL's FORTRAN-inspired numeric constants, e.g., **X**'1234', **O**'765', etc. Standard C's notations for numeric constants are preferred and fully supported.
- X-SAR's **%default** function differs slightly - the default expression is not evaluated if the primary expression is defined.
- X-SAR's TSTOL provides 3 functions for testing if a string can be interpreted as a number: **%isint**, **%isreal**, and the generic **%isnum**. TPOCC's TSTOL has **%isreal** and **%isnum**; the latter only tests for integers.
- On X-SAR, text substitution is not applied to directives received on non-OPIO, network connections.
- TPOCC's TSTOL was modified to look up interactively-entered variable names in the global symbol table only. X-SAR's TSTOL retains the old local-first, then-global scheme.
- There are slight differences in the generation of ''Directive Complete'' messages for built-in directives.

So, which would I choose, the TPOCC version or the X-SAR version of TSTOL? X-SAR's TSTOL version is more portable and more powerful; TPOCC's TSTOL is more closely integrated with the TPOCC environment.

CONCLUSIONS

TSTOL ought to be eliminated! In the TPOCC environment, TSTOL is primarily used as a point of contact between the distributed graphical user interface (GUI) tasks and the centralized State Manager task. The intelligence that the GUI should have has been fobbed off on TSTOL; likewise, many of the functions previously handled by MAE's STOL have been pushed off to the State Manager.

My recommendation? The GUI should have an embedded interpreter for a command language such as John Osterhout's Tool Command Language, Tcl. For example, my own TWERP extensions to Tcl (derived from my Motif-based WIMP extensions!) provide access to X, Motif, the TPOCC real-time widgets, the TPOCC data server, and message-based network communications. A GUI incorporating TWERP could load and animate TPOCC display screens, send messages to and receive messages from remote processes (e.g., the State Manager or other applications), and anything else it might need to do – all via user-programmable, Tcl scripts!

The State Manager, too, should have an embedded Tcl interpreter. For example, my own PICL extensions to Tcl (derived from my non-X-based NICL extensions) provide access to the TPOCC data server and to message-based network communications. With the appropriate Tcl extensions, a generic State Manager could be written that provided mission-specific functionality through mission-specific Tcl scripts.

...

AUTHOR

Alex Measday, *Integral Systems, Inc. (301) 731-4233 alexm@vlsi.gsfc.nasa.gov*