JYACC FORMAKER JPL Programmer's Guide

Contents

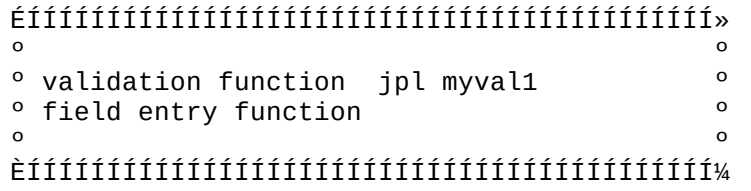
1 The JYACC Procedural Language

The JYACC Procedural Language, or JPL, is an interpreted language native to JYACC FORMAKER. It has been designed to make prototyping easier by cutting out the compile cycle associated with standard programming languages, and by providing shortcuts for operations common in JYACC FORMAKER applications. Its features include block-structured control flow, named procedures with arguments, and flexible typing of variables according to context. Fields on the screen, called occurrences in this chapter, are treated just like variables.

Every JPL command begins with a keyword, for simplicity in parsing. Each line of a JPL procedure may only contain one command; commands may be extended over many lines by ending intermediate lines with a backslash \. Lines beginning with a colon : will be treated as comments, i.e. ignored by the interpreter.

1.1 Incorporating JPL Procedures Into Your Application

JPL procedures are ordinarily stored in files, one procedure to a file, with the file name serving as the procedure name. Arguments are declared within the procedure, in the usual way. To install a JPL procedure as a standard attached function, use the special function name jpl followed by a blank, then the name of the file containing your JPL procedure, as in the figure below. Your procedure will be called with the standard argument list for attached functions.



You can also attach a nameless JPL procedure to a field, using a special edit provided for that purpose. Such a procedure is stored in the screen file rather than in a separate file, and is executed just after the field validation function, so you can actually have two JPL procedures executed during field exit.

Finally, you can call a JPL procedure from application code through the library function sm_plcall, q.v.

1.2 JPL Expressions

Expressions composed of numeric and string operations can occur within a number of JPL commands, including cat and math commands; the logical tests in if, while, and for commands; and return commands. Such an expression reduces a complex string of values and operators to a single value, which may then be tested or stored.

In the following sections, we first define what sort of component values may appear in JPL expressions. There follows a description of how to group values with operators to form expressions.

1.2.1 Values

1.2.1.1 Data Types

The type of an expression in JPL is determined by context, that is, by how the expression's value is to be used. If it is to be added to the value of another expression, it is numeric; if used to control the execution of a while loop, it is logical; and so on. All values are stored as character strings; if a different type is required, the required conversions are performed on the string.

string	Required in cat commands; assumed if the value begins with one of the quote symbols `, ', or ". Requires no conversion.
numeric	Required on the right-hand side of the equal sign in math commands and in return commands; assumed if the value begins with a digit, plus, or minus sign. Formed by collecting an optional initial sign, all digits, and a single decimal point, ignoring all other characters, and converting the result to a floating point number.
integer	Required for field numbers and for subscript and substring indices. Like numeric, except that collection stops at a decimal point, and conversion is to integer.
logical	Required in tests in if, while, and for commands. For expressions determined by other means to be numeric, nonzero means true and zero means false. For string expressions, if the first character is equal to the first character of the message file entry SM_YES, regardless of case, the value is true; anything else means false.

1.2.1.2 Constants

A numeric constant is a string beginning with a digit, a plus sign, or a minus sign.

A string constant begins and ends with one of the three quote characters quote `, apostrophe ', or double quote ". The enclosing quote character can be embedded in the string by escaping it with a backslash; other quote characters may simply be included. When using colon expansion (Section 1.3) in quoted strings, be aware that problems will arise if the items to be expanded contain quotes of the type used to delimit the string.

1.2.1.3 Variables

JPL variables are created by the vars command, which may be issued anywhere within a procedure before they are used, and disappear when the procedure exits. Their scope is dynamic (according to the rule just mentioned), limited to the procedure, and unaffected by the block structure of the procedure.

A variable name must begin with a letter, dollar sign, period, or underscore; it may be followed by any combination of letters, dollar signs, periods, underscores or numbers. It is common practice to begin variable names with an underscore or period to distinguish them from occurrences.

JPL determines the type of a variable by its context in an expression, not from its declaration; every variable's value is stored as a character string. You can define the size of that string in your declaration. Redeclaring a variable with a different size obliterates the original declaration.

Variables and occurrences are treated the same in expressions. When the name of one is mentioned, its value is substituted; no special syntax is required to dereference a variable. If a variable and an occurrence have the same name, the variable's value will be used. The scope of a variable is strictly limited to the declaring procedure, while occurrences are available to all JPL procedures; in other words, variables are local and the screen is global.

1.2.1.4 Occurrences

When a string beginning with a letter or pound sign appears in a JPL expression, it is interpreted as a reference to a variable or occurrence, and replaced by the value of that thing. There is a field identifier, either name or number, followed by an optional index for fields with multiple occurrences.

field number	The occurrence must be onscreen. Use a pound sign followed by the field number. If the number has a +

	or - sign, it is taken relative to the current field; if it is missing, the current field itself is used. #5 means the fifth field on the screen #-1 means the field immediately preceding the current field # means the current field
field name	Use the occurrence name as it appears in the screen. zip_code sales_tax
bracket-subscript	Append an occurrence number (not necessarily a constant) surrounded by square brackets. No blank is allowed before the left bracket. #5[2] #1[i] customers[23] customers[k]

If the name of an item with multiple occurrences appears in an expression without a subscript, the current occurrence is substituted.

1.2.1.5 Substrings

With a substring specifier, you can extract a piece of any string for use in the surrounding expression. It will be treated as a string, numeric or logical depending on the command which operates on it.

A substring specifier follows a variable or occurrence identifier; its syntax is (m,n), where m is the index of the beginning of the substring and n is its length. The indices count from 1; if n is missing, the end of the string is assumed.

The following substring expression extracts the day from a date field named today and formatted as MM/DD/YY:

```
today(4,2)
```

No blank space is permitted between the name and the left parenthesis. If the beginning index is greater than the length of the string, the value of the substring expression is the empty string; this can be useful in looping.

1.2.2 Math Expressions

JPL math expressions have a good deal in common with the math edits you can attach to screen fields using the Screen Editor. The main differences are that only JPL expressions support substrings, and that the colon form of field subscripting supported by both is inconvenient to use in JPL programs because of colon expansion (see below). Syntactically, JPL expressions bear a strong resemblance to C; but the type conversion rules are quite different.

A math expression may begin with an optional precision specifier, %m.n . Here m is the total number of characters (significant digits plus sign and decimal point) in the expression's value, and n is the number of decimal places. The rest of the expression is built up from values, unary and binary operators, and parentheses for grouping, in the usual way.

1.2.2.1 Arithmetic Operators and Subexpression Grouping

The following operators are supported in JPL expressions:

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
^	Raise to power
-	Unary negate
@date	Unary date value
@sum	Unary array sum
@abort	Test/set abort flag

If any of the first 6 are used with a string operand, an error will result. @Date converts a date field or string to a number you can then compare to other dates or perform arithmetic with. The expression

```
@date(today) + 7
```

yields a date one week from the present, while

```
@date(12/25/89) - @date(today)
```

gives the number of shopping days left till Christmas. Note that comparisons done using @date are independent of the date format, where lexical comparisons on the date fields are not.

@Sum gives the sum of all occurrences in an array or scroll; the expression

```
@sum(quantities)
```

yields a total of all the occurrences in the quantities field. @Abort, followed by a number in parentheses, calls the library function sm_isabort with the number as a parameter, causing JYACC FORMAKER to return control to the application's top level.

There are also several relational operators for comparing values, which are particularly important in logical expressions. The operators are these:

Operator	Meaning
= or ==	equal
!=	unequal
<	less than
>	greater than
<=	less or equal
>=	greater or equal
& or &&	conjunction (and)
| or ||	disjunction (or)
!	unary logical not

When two values of the same type are compared, the result is straightforward. When the types of the two items being compared are different, one of them is converted before the comparison, according to the following table and the conversion rules given in section 1.2.1.1. Note that it is an error to compare a number to a string expression.

Operand 1	Operand 2	Comparison
string	string	lexical string
	number	ERROR string
	logical	logical number
number	number	numeric number
	logical	logical logical
logical	logical	logical

The results of comparisons and unary not are always logical, while the result of an arithmetic operation is always numeric. The logical value of a string is true if the string looks like a yes (begins with the first character of SM_YES), false otherwise; a string enclosed in parentheses is a logical expression. A numeric expression is false if its value is zero, true otherwise.

1.2.3 String Expressions

String expressions occur in the cat command. All values are treated as strings; the only operation is concatenation, or splicing, of adjacent strings. Blanks between values are ignored; to get blanks in the expression's value, you must enclose them in quotes.

See the cat command for examples.

1.3 Colon Expansion

All JPL commands are colon-expanded each time they are executed. In this process, text following a colon : is interpreted as an occurrence identifier, and the colon and identifier are replaced by the value of the occurrence. The syntax of occurrence identifiers is described in Section 1.2.1.4; it allows for referring to fields by name or number, and for subscripting them. If you place a colon and asterisk :* before an occurrence identifier, it will be expanded recursively. The original occurrence will be replaced by its value; if that begins with : or :* it will in turn be replaced by its value; and so forth. This is known as double indirection.

You can escape a colon by preceding it with a backslash, or with another colon. No blanks are allowed between the colon and the following name. The colon form of occurrence subscripting (Section 1.2.1.4) will cause errors in colon expansion unless the colons are escaped; the bracket form of subscripting is strongly recommended.

The while clause in a for command is colon-expanded only at the first iteration of the loop. The test expression of a while command, on the other hand, is subject to colon expansion on every iteration of the loop.

Within JPL expressions, occurrences are replaced by their values automatically; colon expansion constitutes a second, often superfluous, level of indirection. It is useful in JPL commands that do not contain math or logical expressions, such as msg. The following are equivalent:

```
msg emsg <variable-name>
msg emsg ":<variable-name>"
```

Colon expansion is useful also for getting values inside quoted strings. The following are equivalent:

```
cat result "You have " count " widgets at " price " each."
cat result "You have :count widgets at :price each."
```

If the name of an item with multiple occurrences appears in an expression preceded by a colon but without a subscript, colon expansion causes all non-blank occurrences to be substituted, with a single blank between each.

2 JPL Commands by Name

Figure 1 summarizes the JPL commands. Some commands have additional qualifying keywords, which are described along with the command.

Command	Action
atch	execute an attached function
block	{ statement block }
break	exit prematurely from a loop
call	execute an invoked function
cat	string manipulation
else	conditionally execute following block
for	indexed loop
if	conditionally execute following block
jpl	execute a JPL routine
load	read a file of JPL routines into memory
math	numeric calculations
msg	display a message in various ways
next	skip to next iteration of loop
parms	declare parameters to a JPL routine
return	exit from JPL routine
retvar	declare variable to hold return value
system	execute a system call
unload	free up memory from loaded JPL code
vars	declare local variables
while	general loop

Figure 1: JPL commands

The pages that follow contain detailed descriptions of all the JPL commands, in alphabetical order. Each page has the following information:

- The command name, and a brief description of what it does.
- The syntax of the command. Bold text indicates required keywords. Normal text denotes parameters for which you supply values; brackets [] indicate that a parameter is optional.
- A full description of the command, explaining its parameters, outputs, and side effects.
- One or more examples, normally fragments of JPL code demonstrating the usefulness of the command in question.
- A list of other, related commands.

Parameters named expression are evaluated as JPL expressions, according to the rules defined above.

NAME

atch - execute an attached function

SYNOPSIS

atch function-name [text]

DESCRIPTION

Executes a function which is installed in the attached function list. It receives the usual four arguments. If this procedure is not attached to a field, the arguments are:

1. field number = 0
2. contents = text
3. occurrence = 0
4. flags = K_USER (invoked by user program; VALIDED and MDT bits both 0)

If this procedure is attached to a field, however, the four arguments received are those of the field.

If you have designated a return variable with the retvar command, the attached function's return value is stored there.

EXAMPLE

```
: validate state name field using an attached function

vars not_ok
retvar not_ok
atch val_state state_name
: alternate way of writing previous line: atch val_state ":state_name"
if not_ok
     msg err_reset "That state is not one of U.S."
```

NAME

block - statement blocking

SYNOPSIS

```
{
     any JPL command
     ...
}
```

DESCRIPTION

Curly braces, { and }, block together the JPL commands they enclose, causing them to be executed as a unit by an immediately preceding if, else, while, or for command. (A single JPL command following one of those need not be made into a block; but beware of the if-else ambiguity.)

The curly braces must stand alone on a line. An empty block is equivalent to a null statement; it is syntactically legal and does nothing.

There is no special processing, and no other syntactic significance, associated with a block. In particular, it does not limit the scope of variables as in C.

EXAMPLE

```
vars not_ok
retvar not_ok
atch val_state :state_name
: Multiple statements must be blocked
if not_ok
{
     msg err_reset "That state is not one of U.S."
     return
}
: Single statements need not be blocked
else
     msg d_msg "Processing :state"
```

NAME

break - exit prematurely from a loop

SYNOPSIS

break [count]

DESCRIPTION

Terminates execution of one or more enclosing while or for loops, and resumes execution at the command immediately following the last aborted loop.

Count gives the number of loops to break. It may be either a positive number or a single field value expression; if omitted, it is taken as 1.

EXAMPLE

```
vars i address
for (i = 1 while (i <= 10) step 1)
{
     cat address cities[i] "," states[i] "," zips[i]
: Second termination condition in middle of loop
     if address == ""
          break
     call do_address address
}
```

NAME

call - execute an invoked function

SYNOPSIS

```
call function-name [text]
```

DESCRIPTION

Executes a function which is installed in the invoked function list. The function receives a single argument, namely the command string from function-name onward.

If you have designated a return variable with the retvar command, the invoked function's return value is stored there.

If function-name is jpl, this command behaves just like the jpl command (q.v.).

EXAMPLE

```
vars i line
: Call a C function that stores lines of text in a file
math i = 1
while names[i] != ""
{
     cat line names[i] ", " addresses[i] ", " cities[i] ", " states[i]
     call saveline :line
     math i = i + 1
}
```

NAME

cat - string manipulation

SYNOPSIS

cat occurrence [string-expression]

DESCRIPTION

Evaluates string-expression, according to the rules given in Section 1.2, and stores it in occurrence.

Occurrence may be a JPL variable or a screen field. Note that it is used as a name for assignment; if you want to assign to an occurrence whose name is stored in another variable, you must use colon expansion.

If string-expression is missing, occurrence will be cleared.

If your destination utilizes substring notation, only that portion of the destination is affected. If you do not specify substring notation for the destination, the destination is replaced by the string expression.

EXAMPLE

```
: This is faster than math i = 1
cat i "1"

: This combines some field data with constants. Note that cat
: does NOT automatically leave blanks between items!
cat sons_name first " " last ", Jr."
: This does the same thing
cat sons_name ":first :last, Jr."

:This tacks on a four-digit zip code extension
cat zip(6,5) "-" zip_extension

: To append something to a field or variable, you can use it
: as both a source and destination
cat zip zip "-" zip_extension
: or
cat zip ":zip-:zip_extension"
: This last example works because hyphens are not acceptable
: characters in variable names
```

NAME

else - conditionally execute following block

SYNOPSIS

```
else
single command or block

else if
single command or block
```

DESCRIPTION

This command is only valid immediately after an if. The body of the else (the command or block of commands following) will be executed only when the condition following the if is false.

When you want to check for a number of possible conditions, you can use an "else-if chain," like the one in the example. This is the only circumstance in which two JPL commands may appear on a single line.

EXAMPLE

```
: Figure out a person's sex, based on his or her personal title.
if title = "MR"
     cat sex "Male"
else if title = "MS"
     cat sex "Female"
else if title = "MRS"
     cat sex "Female"
else if title = "MISS"
     cat sex "Female"
else
{
     cat sex "Unknown"
     msg err_reset "Please supply a title"
}


: Beware of misplaced braces and ambiguous "elses"
: Examples #1 and #2 give the same results,
: which are different from #3

: Example #1
if x = 1
if y = 2
     cat fld3 "yes"
else
     cat fld4 "no"

: Example #2
if x = 1
{
if y = 2
     cat fld3 "yes"
else
     cat fld4 "no"
}

: Example #3
if x = 1
{
if y = 2
     cat fld3 "yes"
}
else
     cat fld4 "no"
```

SEE ALSO

block
if

NAME

for - indexed loop

SYNOPSIS

```
for index-var = value while ( condition )  \
step [-] increment
single command or block
```

DESCRIPTION

This command provides an indexed loop. It has three clauses, the initial step, loop condition, and index step, which control the repeated execution of the loop's body (the following statement or block). The three clauses (respectively) set an index variable to an initial value, test it against a limiting condition, and increment it, as when accessing all occurrences of an onscreen array in turn.

The initial-step assigns value, which must be a numeric constant or a single variable, to index-var, which must be a JPL variable (and not an occurrence). It is executed once, at entry to the loop.

The increment is a numeric constant or single variable which is added to index-var on each iteration, after the body of the loop but before evaluation of the condition. It may be positive or negative.

The condition can be any JPL expression; its value is treated as logical. It is evaluated on each iteration, before the body of the loop; if it is false after the initial step, the loop body will never be executed.

EXAMPLE

```
vars i
: Change each element of an array to its absolute value
for i = 1 while (i < 10) step 1
{
     if amounts[i] = ""
          cat amounts[i] "0"
     else if amounts[i] < 0
          math amounts[i] = -amounts[i]
}

: Compute the length of a string variable
for i = 1 while (string(i) != "") step 1
{
}
```

SEE ALSO

block
while

NAME

if - conditionally execute following block

SYNOPSIS

```
if condition
single statement or block
[else single statement or block]
```

DESCRIPTION

This command provides for the conditional execution of other JPL commands. Condition may be any JPL expression; if its logical value is true, the following statement or block (called the body of the if) will be executed. If the condition is false, the body will not be executed; if there is an else clause (q.v.), its body will be executed instead.

EXAMPLE

```
: Supply a default value for an empty field
if amount = ""
     cat amount "N/A"

: Condition can test a numeric variable
vars x
math x=#5 - #4
if x
     cat recfld srcfld

:Condition can test a string
vars more
msg query "Would you like to see another?" more
if more
     return 0
else
     return 1
```

SEE ALSO

block
else

NAME

jpl - execute a JPL routine

SYNOPSIS

jpl routine [argument ...]

DESCRIPTION

Calls another JPL routine, optionally with arguments. The file named routine is loaded into memory, if necessary, and the commands therein are executed. Control returns to the command following the jpl command when the called routine executes a return command.

The length and lexical content of routine names are subject to the operating system's file naming conventions. JYACC FORMAKER searches for the named file in (1) memory (for routines read in with the "load" command), (2) the memory-resident form list, (3) the current directory, and (4) the directories listed in the SMPATH setup variable. It does not append a default extension.

This command enables you to code commonly performed tasks in subroutines, which can be called from many places. Commonly used subroutines can be pre-loaded, using the load command, for greater efficiency. Note that parameters are passed by value.

EXAMPLE

```
vars i r
retvar r
: Loop through a group of parallel arrays, calling a JPL subroutine
: to assemble an address from each "line," and a C subroutine to
: store the result in a file.
for i = 1 while (i < 10) step 1
{
     jpl getaddr.jpl whole :name[i] :street[i] :city[i]\
          :state[i] :zip[i]
     if r > 0
          call store whole
}

: In the file "getaddr.jpl"
: Note the use of colon expansion for the first parameter,
: which is the name of an occurrence in which to store
: the result of this routine.
vars .result_name .name .street .city .state .zip
parms .result_name .name .street .city .state .zip
cat :.result_name ""
if .name = "" | .street = "" | .city = "" | .state = ""
     return 0
cat :.result_name .name ", " .street ", " .city ", "\
     .state ", " .zip
return 1
```

SEE ALSO

load
parms
return
retvar

NAME

load - read a JPL routine into memory

SYNOPSIS

load routine [routine ...]

DESCRIPTION

Reads a file of JPL statements into memory. Pre-loading routines that are frequently called with the jpl command can make them execute much more quickly. The memory used to hold them can later be released, using the unload command.

A routine is executed in exactly the same way, whether it is pre-loaded or read from disk. Note that if you are debugging a JPL procedure it is best not to pre-load it.

EXAMPLE

```
: Load three subroutines into memory for future use.
load validname.jpl defaultname.jpl blank.jpl

: Example of a loop that calls one of these subroutines
for i=1 while (i<10) step 1
     jpl validname.jpl name[i]
```

SEE ALSO

unload

NAME

math - numeric calculations

SYNOPSIS

math [%precision] occurrence = expression

DESCRIPTION

Evaluates a JPL expression, and assigns its value to a variable or occurrence. See Section 1.2 for a long discussion of JPL expressions; the expression's value is treated here as numeric.

Occurrence may be a JPL variable, a screen field or an LDB entry. Note that it is used as a name for assignment; if you want to assign to an occurrence whose name is stored in another variable, you must use colon expansion.

The optional precision controls the number of digits and decimal places in the result. Its format is %n.m, where n is the total number of digits in the result and m is the number of decimal places. If precision is omitted, the default is unlimited width and two decimal places; however, if occurrence is a entry with a real data type or currency format edit, that default will be used instead.

Note that string operations, such as substring, are available in math expressions.

EXAMPLE

```
: Simple initialization
math k = 0
math %9.4 total = @sum (subtotals)
: Computing the cost of an item
vars cost
math cost = (price * (1 - discount)) * (1 + tax_rate)
```

SEE ALSO

cat

NAME

msg - display a message in various ways

SYNOPSIS

```
msg mode text [!] [response-var]
where mode is one of
     d_msg
     emsg
     err_reset
     query
     qui_msg
     quiet
     setbkstat
```

DESCRIPTION

Displays text on the terminal's status line, in one of several modes. The modes correspond to a number of JYACC FORMAKER library routines, and are explained briefly here; see the Programmer's Guide for more details. The optional response-var is a JPL variable or occurrence, and is allowed only for mode query.

d_msg	Displays text on the status line and leaves it there, until cleared or replaced by another message. It may be temporarily replaced by a msg command with another mode (except setbkstat).
emsg	Displays text as an error message, until you acknowledge it with a keystroke. Acknowledgement is controlled by the SMEROPTIONS setup variable, or the library routine sm_er_options. Cursor is not forced to be turned on.
err_reset	Like emsg, but forces the cursor to be turned on at its current position.
query	Displays text as a question, and sets response-var to true if the answer is yes, or false otherwise. If response-var is preceded by an exclamation point, that logic is reversed (true = no). If there is no response-var and the answer is no, the JPL procedure exits immediately; a lone exclamation point reverses that too.
qui_msg	Displays text as an error message until it is acknowledged. The text will be preceded by the SM_ERROR string from the message file, which is normally "ERROR:". Cursor is not forced to be turned on.
quiet	Like qui_msg, but forces the cursor to be turned on at its current position.
setbkstat	Installs text as the background status line. It will be displayed when no other message is active.

EXAMPLE

```
: Indicate that the field called state is invalid
msg err_reset ":state is not one of U.S."

: Indicate that the current entry is being processed
: Note that d_msg overrides delayed write and flushes text
: to the screen immediately
msg d_msg "Processing :name"

: Ask whether the user wants to quit the current screen
vars quit
msg query "Are you ready to quit?" quit
if quit
     return EXIT
```

NAME

next - skip to next iteration of loop

SYNOPSIS

next

DESCRIPTION

This command is valid only within the body of a for or while loop. It causes commands between itself and the end of the loop body to be skipped, so that the next things that happen are the increment step (in a for) and the loop condition test. The next command applies only to the innermost enclosing loop.

Next is more similar to the continue statement in C than to the next statement in BASIC.

EXAMPLE

```
vars k
: Process all the males in a list of people
for k = 1 while (sex[k] != "") step 1
{
     if sex[k] != "Male"
          next
: Print mailing label for sports car brochure, or whatever...
}
```

SEE ALSO

for
while

NAME

parms - declare parameters in a called JPL routine

SYNOPSIS

parms variable [variable ...]

DESCRIPTION

Associates variable names with the arguments to a JPL routine supplied in a jpl command. The variables must already have been declared, using the vars command.

If you declare more parameters than were actually passed, the excess variables will be uninitialized. If you declare fewer, the undeclared parameters will be inaccessible.

EXAMPLE

```
: This routine returns the value 1 if the given array contains
: a certain string, and 0 otherwise.

vars array_name pattern k
parms array_name pattern

for k = 1 while (:array_name[1][k]) step 1
{
     if :array_name[1][k] == pattern
          return 1
}

: Note that we need the [1] because we want to colon-expand
: array_name and access the kth element of the expanded value.
: If we specified only :array_name[k], it would try to colon-
: expand the kth element of array_name, and not the first.

return 0
```

SEE ALSO

jpl
vars

NAME

return - exit from JPL routine

SYNOPSIS

return [value]

DESCRIPTION

This command causes a JPL procedure to exit. Control is returned to the procedure that called it, if any, or to the JYACC FORMAKER run-time system.

If the optional value is supplied and the calling procedure has established a return variable with the retvar command, that variable is set to value. The return value must be a numeric constant or a single variable or occurrence.

If the routine was called from other than a JPL procedure, the returned value must be an integer.

Return is also accomplished automatically by coming to the end of a JPL file. This is dangerous with memory-resident procedures, however, because there is no way to tell when they end if they have no null terminators!

EXAMPLE

see parms

SEE ALSO

retvar

NAME

retvar - establish a variable to hold return values

SYNOPSIS

retvar [variable]

DESCRIPTION

Variable is the name of a JPL variable, which will be set to the return value of the called function in subsequent call, atch, and jpl commands. The variable must previously have been created with the vars command. (However, it could be a global variable instead -- i.e., a field or LDB entry.)

If variable is omitted, the return values of called functions are unavailable.

EXAMPLE

```
vars r
retvar r
call validname :name
if !r
     return
: Process the validated name...


: A return variable can also be colon-expanded, if it
: contains the name of a variable into which the return
: value is to be placed

vars r
cat r "name"
retvar :r
call getname
...
```

SEE ALSO

atch
call
jpl

NAME

system - execute a system call

SYNOPSIS

system text

DESCRIPTION

Text is sent to the operating system as a program to be executed. The screen is cleared, and the output of the program (if any) is displayed; when it exits, the JYACC FORMAKER screen is refreshed and screen processing resumes.

If you have established a return value variable with the retvar command, the program's exit status is available there.

If you want text to contain the values of occurrences, you must colon-expand them.

EXAMPLE

```
: On a UNIX system, check whether a file exists
vars status
retvar status
system test -f :filename
if !status
     return
: process the file...
```

SEE ALSO

retvar

NAME

unload - free up memory from loaded JPL code

SYNOPSIS

unload procedure

DESCRIPTION

Releases the memory used to hold a JPL procedure loaded by a previous load command. If the procedure is subsequently called again, it will be read in from disk.

EXAMPLE

```
: unload three subroutines
unload validname.jpl defaultname.jpl blank.jpl
```

SEE ALSO

jpl
load

NAME

vars - declare local variables

SYNOPSIS

vars variable [variable ...]

DESCRIPTION

Creates a variable or variables local to the current JPL procedure. The names so created will not be visible to any other JPL procedure. If a variable has the same name as an occurrence, it will hide the occurrence. For this reason it is sometimes useful to establish a naming convention to prevent conflicts; beginning variable names with a period, underscore, or dollar sign will work, since field names must begin with a letter.

Vars may occur anywhere within a JPL procedure. It is an executed statement -- i.e., the declaration occurs when the statement is executed. If a variable name is redeclared with a different size, it erases the old declaration and value; hence it is not a good idea to place a vars command inside a loop. The initial value of a newly declared variable is the empty string, zero, or false, depending on context.

Variables may have any number of occurrences, which you place after the name, enclosed in square brackets []. You may also specify the size in bytes of a variable, placing it after the name (and occurrences if present), enclosed in parentheses. If a variable is to be used for a string parameter, it is best not to specify a size. No blanks may occur between the name and following left bracket or parenthesis.

EXAMPLE

```
vars name(50) flag(1) widget
vars address[3](50) abbrevs[10]
```

NAME

     while - general loop

SYNOPSIS

```
     while condition
     single command or block
```

DESCRIPTION

The while command provides for repeated execution of the commands within its body (the following command or block). The body is executed as long as condition, which is an arbitrary JPL expression treated as logical, remains TRUE.

Condition is evaluated before every iteration of the loop, so that if it is initially false, the body will never be executed.

EXAMPLE

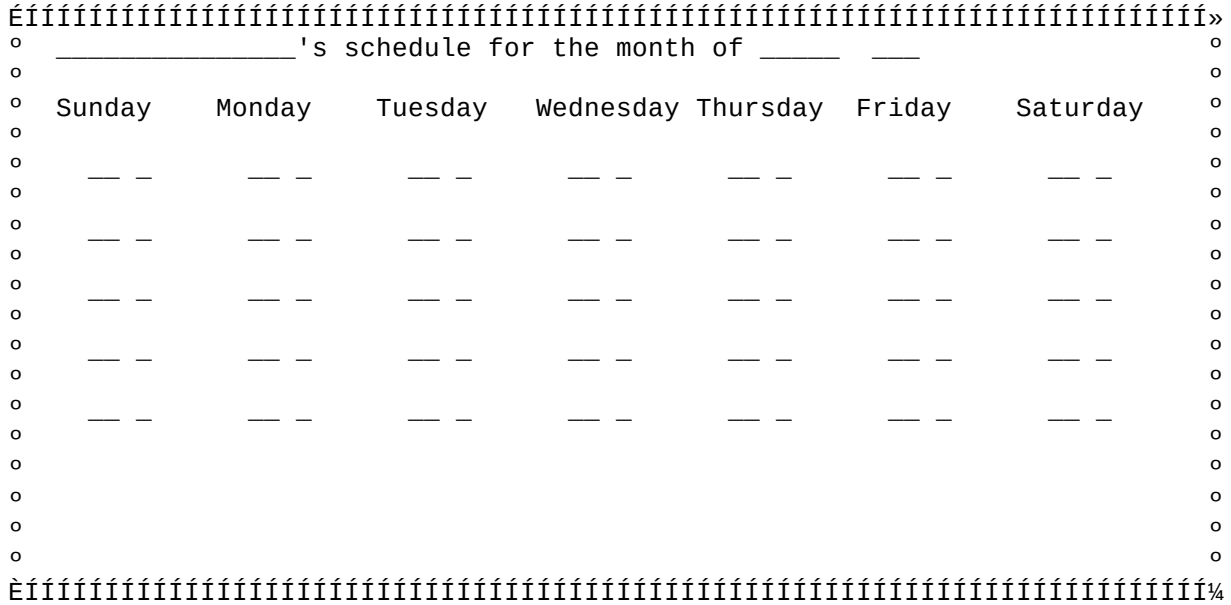
```
vars k another
cat k "1"
cat another "a"
while k
{
     msg query "Do you want to do :another widget?" k
     if !k
          return
     jpl do_widget

     cat another "another"
}
```

3 JPL Examples

3.1 Example Screen

The examples in this section will make use of a calendar screen, whose picture and screen listing follow. The purpose of the code is to paint a month's calendar on the screen and to accept a code letter for each day describing someone's whereabouts.



FORM 'calendar'

FIELD DATA:

```
Field number        : 1   (line 2, column 6, length = 15)
Field name          : who
Display attribute   : UNDERLINED HIGHLIGHTED WHITE
Field edits         : RIGHT-JUSTIFIED; DATA-REQUIRED; CLEAR-ON-INPUT;


Field number        : 2   (line 2, column 50, length = 5)
Field name          : today
Display attribute   : UNDERLINED HIGHLIGHTED WHITE
Date field data     : SYSTEM DATE; FORMAT = MM/YY


Field number        : 3   (line 6, column 8, length = 2)
Field name          : sunday_num
Vertical array      : 5 elements; distance between
                      elements = 2
Array field numbers :  3 17 31 45 59
Display attribute   : WHITE
Field edits         : RIGHT-JUSTIFIED; PROTECTED FROM:  ENTRY OF DATA;
                       TABBING INTO;  CLEARING;  VALIDATION;
Amount field data   : RIGHT-JUST; 0 DEC. PLACES; DON'T APPLY IF EMPTY;
Data type           : INT


Field number        : 4   (line 6, column 11, length = 1)
Field name          : sunday
Vertical array      : 5 elements; distance between
                      elements = 2
```

```
Array field numbers :  4 18 32 46 60
Display attribute   : UNDERLINED HIGHLIGHTED WHITE
Field edits         : UPPER_CASE;
Validation func.    : 'jpl calval.jpl'

...analogous fields for Monday through Friday have been omitted...

Field number        : 15   (line 6, column 68, length = 2)
Field name          : saturday_num
Vertical array      : 5 elements; distance between
                      elements = 2
Array field numbers : 15 29 43 57 71
Display attribute   : WHITE
Field edits         : RIGHT-JUSTIFIED; PROTECTED FROM:  ENTRY OF DATA;
                       TABBING INTO;  CLEARING;  VALIDATION;
Amount field data   : RIGHT-JUST; 0 DEC. PLACES; DON'T APPLY IF EMPTY;


Field number        : 16   (line 6, column 71, length = 1)
Field name          : saturday
Vertical array      : 5 elements; distance between
                      elements = 2
Array field numbers : 16 30 44 58 72
Display attribute   : UNDERLINED HIGHLIGHTED WHITE
Field edits         : UPPER_CASE;
Validation func.    : 'jpl calval.jpl'


Field number        : 73   (line 17, column 1, length = 9)
Field name          : dow
Horizontal array    : 7 elements; distance between
                      elements = 1
Array field numbers : 73 74 75 76 77 78 79
Display attribute   : NON-DISPLAY WHITE


Field number        : 80   (line 18, column 1, length = 2)
Field name          : month_length
Horizontal array    : 12 elements; distance between
                      elements = 1
Array field numbers : 80 81 82 83 84 85 86 87 88 89 90 91
Display attribute   : NON-DISPLAY WHITE


Field number        : 92   (line 19, column 1, length = 35)
Field name          : schedule
Display attribute   : NON-DISPLAY WHITE


Field number        : 93   (line 20, column 1, length = 10)
Field name          : firstname
Display attribute   : NON-DISPLAY WHITE
Date field data     : NO SYSTEM DATE; FORMAT = dow
```

There is also a short driver program, which brings up the screen, reads the keyboard, and dispatches to the appropriate function below.

```
#include "smdefs.h"
#include "smkeys.h"

/* Driver program for the calendar JPL example.
 * This just brings up the screen, then loops reading keys.
 * It recognizes TRANSMIT, PF1, and EXIT.
 */

main()
{
     int key;

     sm_initcrt ();
     if (sm_r_form ("calendar") < 0)
          exit (-1);

     while ((key = sm_openkeybd ()) != EXIT)
     {
          switch (key)
          {
          case XMIT:
               sm_plcall ("calstore.jpl");
               break;
          case PF1:
               sm_plcall ("caldefs.jpl");
               break;
          default:
               sm_bel ();
               break;
          }
     }

     sm_resetcrt ();
     exit (0);
}
```

3.2 Screen Entry Function

Here is the JPL screen entry function. It determines what day of the week the first of the month falls upon, then fills in dates in the appropriate slots. (At present, no provision is made for leap Februaries.)

File calup.jpl

```
: This is the screen entry function.
: It figures out what day of the week the first of the month
: falls on, then writes the appropriate dates into the columns
: 'sunday_num', 'monday_num', etc.

vars _scr _j _k _day(2) _fld
vars _firstday _lastday

: Get ordinal of first day.
: Make sure the field "today" contains the date.
if today = ""
     cat today ""
cat _scr today(1,2) "/1/" today(4,2)
math firstname = @date (_scr)
for _firstday = 1 while (_firstday <= 7) step 1
{
     if (dow[:_firstday](1,3) == firstname)
          break
}
```

```
: Get ending limit.
cat _scr today(1,2)
cat _lastday month_length[_scr]

: Using the limits, display the month's dates.
math %2.0 _day = 1
for _k = 1 while (_k <= 5) step 1
{
     for _j = 1 while (_j <= 7) step 1
     {
          if _k == 1 && _j < _firstday
               next
          if _day > _lastday
               next
          cat _fld dow[_j] "_num[:_k(1,1)]"
          math :_fld = _day
          math _day = _day + 1
     }
}

: Present a prompt
msg d_msg "Press %KPF1 for default data."
```

3.3 Field Validation Function

File calval.jpl

```
: This is a field validation function for the enterable
: columns 'sunday', 'monday', etc. It blanks invalid entries
: and reminds you that help is available.

: Standard parameter list: field number, contents, occurrence #,
: and validation information.

vars _num _content _occur _bits
parms _num _content _occur _bits

if (_content != "H" && _content != "V" && _content != "W" &&\
    _content != "")
{
     msg err_reset "Please enter H, V, or W; press %KHELP for help."
     cat #:_num ""
     return -1
}
return 0
```

3.4 Screen Completion Function

The function below is bound to the TRANSMIT key. It make use of a subroutine, listed after it.

File calstore.jpl

```
: This function is bound to the TRANSMIT key. It just
: calls a subroutine to store and display the packed schedule.
jpl calpack.jpl schedule
```

File calpack.jpl

```
: This function takes the name of a variable in which to store
: the packed schedule, and stores it there (one character per day).
: It then displays the result.
```

```
vars _packname
parms _packname
vars _packed
vars _day _week _i _fld

math _i = 1
: Loop through the screen
for _week = 1 while (_week <= 5) step 1
{
     for _day = 1 while (_day <= 7) step 1
     {
          cat _fld dow[_day] "[:_week(1,1)]"
          cat _packed(_i,1) :_fld
          math _i = _i + 1
     }
}

cat :_packname _packed
msg err_reset "Schedule stored as -> :_packed <-"
```

3.5 Default Values Function

File caldefs.jpl

```
: This function is bound to the PF1 key.
: It installs default values in the LDB, and calls a subroutine
: to expand the default schedule on the screen.

cat who "President Fred"
cat today ""
cat schedule "HWWWWWHHWWWWWHHWWWWWHHWWWWWHHWWWWWH"
jpl calunp.jpl :schedule
```

File calunp.jpl

```
: This function takes a packed schedule as argument, and
: unpacks each character into the appropriate screen field.

vars _packed
parms _packed
vars _day _week _i _fld

math _i = 1
for _week = 1 while (_week <= 5) step 1
{
     for _day = 1 while (_day <= 7) step 1
     {
          cat _fld dow[_day] "[:_week(1,1)]"
          cat :_fld _packed(_i,1)
          math _i = _i + 1
     }
}
```

Index

In this Index, library functions are displayed in boldface, without the prefixes specific to the language interface. Video and setup file entries appear in ELITE CAPS, while utility programs and JPL commands are in elite lower-case. Function key names are in ROMAN CAPS.