NAME

rjestat – RJE status report and interactive status console

SYNOPSIS

`rjestat` [*host*]... [-**s***host*] [-**c***host cmd*]...

DESCRIPTION

Rjestat provides a method of determining the status of an RJE link and of simulating an IBM remote console (with UNIX features added). When invoked with no arguments, *rjestat* reports the current status of all the RJE links connected to to the UNIX system. The options are:

host	Print the status of the line to *host*. *Host* is the pseudonym for a particular IBM system. It can be any name that corresponds to one in the first column of the RJE configuration file.
-s*host*	After all the arguments have been processed, start an interactive status console to *host*.
-c*host cmd*	Interpret *cmd* as if it were entered in status console mode to *host*. See below for the proper format of *cmd*.

In status console mode, *rjestat* prompts with the host pseudonym followed by **:** whenever it is ready to accept a command. Commands are terminated with a new-line. A line that begins with ! is sent to the UNIX shell for execution. A line that begins with the letter q terminates *rjestat*. All other input lines are assumed to have the form:

ibmcmd [*redirect*]

Ibmcmd is any IBM JES or HASP command. Only the super-user or `rje` login can send commands other than display or inquiry commands. *Redirect* is a pipeline or a redirection to a file (e.g., ‘‘> file’’ or ‘‘ | grep ...’’). The IBM response is written to the pipeline or file. If *redirect* is not present, the response is written to the standard output of *rjestat*.

An interrupt signal (DEL or BREAK) will cancel the command in progress and cause *rjestat* to return to the command input mode.

EXAMPLE

The following command reports the status of all the card readers attached to host A, remote 5. JES2 is assumed.

```
rjestat -cA '$du,rmt5 | grep RD'
```

DIAGNOSTICS

The message ‘‘RJE error: ...’’ indicates that *rjestat* found an inconsistency in the RJE system. This may be transient but should be reported to the site administrator.

FILES

/usr/rje/lines	RJE configuration file
resp	host response file that exists in the RJE subsystem directory (e.g., `/usr/rje1`).

SEE ALSO

send(1C), rje(8).
OS/VS2 HASP II Version 4 Operator’s Guide, IBM SRL #GC27-6993. .}f
Operator’s Library: OS/VS2 Reference (JES2), IBM SRL GC38-0210. .}f

NAME

send, gath – gather files and/or submit RJE jobs

SYNOPSIS

`gath` [-ih] file . . .

`send` argument . . .

DESCRIPTION

Gath

Gath concatenates the named files and writes them to the standard output. Tabs are expanded into spaces according to the format specification for each file (see *fspec*(5)). The size limit and margin parameters of a format specification are also respected. Non-graphic characters other than tabs are identified by a diagnostic message and excised. The output of *gath* contains no tabs unless the -h flag is set, in which case the output is written with standard tabs (every eighth column).

Any line of any of the files which begins with ~ is interpreted by *gath* as a control line. A line beginning "~ " (tilde,space) specifies a sequence of files to be included at that point. A line beginning ~! specifies a UNIX command; that command is executed, and its output replaces the ~! line in the *gath* output.

Setting the -i flag prevents control lines from being interpreted and causes them to be output literally.

A file name of - at any point refers to standard input, and a control line consisting of ~. is a logical EOF. Keywords may be defined by specifying a replacement string which is to be substituted for each occurrence of the keyword. Input may be collected directly from the terminal, with several alternatives for prompting. In fact, all of the special arguments and flags recognized by the *send* command are also recognized and treated identically by *gath*. Several of them only make sense in the context of submitting an RJE job.

Send

Send is a command-level interface to the RJE subsystems. It allows the user to collect input from various sources in order to create a run stream consisting of card images, and submit this run stream for transmission to a host computer.

Possible sources of input to *send* are: ordinary files, standard input, the terminal, and the output of a command or shell file. Each source of input is treated as a virtual file, and no distinction is made based upon its origin. Typical input is an ASCII text file of the sort that is created by the editor *ed*(1). An optional format specification appearing in the first line of a file (see *fspec*(5)) determines the settings according to which tabs are expanded into spaces. In addition, lines that begin with ~ are normally interpreted as commands controlling the execution of *send*. They may be used to set or reset flags, to define keyword substitutions, and to open new sources of input in the midst of the current source. Other text lines are translated one-for-one into card images of the run stream.

The run stream that results from this collection is treated as one job by the RJE subsystems. *Send* prints the card count of the run stream, and the queuer that is invoked prints the name of the temporary file that holds the job while it is awaiting transmission. The initial card of a job submitted to an IBM host must have a // in the first column. The initial card of a job submitted to a UNIVAC host must begin with a "@RUN" or "‘run", etc. Any cards preceding these will be excised. If a host computer is not specified before the first card of the runstream is ready to be sent, *send* will select a reasonable default. In the case of an IBM job, all cards beginning with /*$ will be excised from the runstream, because they are HASP command cards.

The arguments that *send* accepts are described below. An argument is interpreted according to the first pattern that it matches. Preceding a character with \ causes it to loose any special meaning it might otherwise have when matching against an argument pattern.

.	Close the current source.
-	Open standard input as a new source.
+	Open the terminal as a new source.

:*spec*:	Establish a default format specification for included sources, e.g., `:m6t-12:`
:*message*	Print message on the terminal.
−:*prompt*	Open standard input and, if it is a terminal, print *prompt*.
+:*prompt*	Open the terminal and print *prompt*.
−*flags*	Set the specified flags, which are described below.
+*flags*	Reset the specified flags.
=*flags*	Restore the specified flags to their state at the previous level.
!*command*	Execute the specified UNIX *command* via the one-line shell, with input redirected to `/dev/null` as a default. Open the standard output of the command as a new source.
$*line*	Collect contiguous arguments of this form and write them as consecutive lines to a temporary file; then have the file executed by the shell. Open the standard output of the shell as a new source.
@*directory*	The current directory for the send process is changed to *directory*. The original directory will be restored at the end of the current source.
~*comment*	Ignore this argument.
?:*keyword*	Prompt for a definition of *keyword* from the terminal unless *keyword* has an existing definition.
?*keyword=xx*	Define the *keyword* as a two digit hexadecimal character code unless it already has a non null replacement.
?*keyword=string*	Define the *keyword* in terms of a replacement string unless it already has a non null replacement.
=:*keyword*	Prompt for a definition of *keyword* from the terminal.
keyword=xx	Define *keyword* as a two-digit hexadecimal character code.
keyword=string	Define *keyword* in terms of a replacement string.
host	The host machine that the job should be submitted to. It can be any name that corresponds to one in the first column of the RJE configuration file (`/usr/rje/lines`).
file−name	Open the specified file as a new source of input.

When commands are executed via $ or ! the shell environment (see *environ*(7)) will contain the values of all send keywords that begin with $ and have the syntax of a shell variable.

The flags recognized by *send* are described in terms of the special processing that occurs when they are set:

`-l`	List card images on standard output. EBCDIC characters are translated back to ASCII.
`-q`	Do not output card images.
`-f`	Do not fold lower case to upper.
`-t`	Trace progress on diagnostic output, by announcing the opening of input sources.
`-k`	Ignore the keywords that are active at the previous level and erase any keyword definitions that have been made at the current level.
`-r`	Process included sources in raw mode; pack arbitrary 8-bit bytes one per column (80 columns per card) until an EOF.
`-i`	Do not interpret control lines in included sources; treat them as text.
`-s`	Make keyword substitutions before detecting and interpreting control lines.
`-y`	Suppress error diagnostics and submit job anyway.
`-g`	Gather mode, qualifying `-l` flag; list text lines before converting them to card images.

-h Write listing with standard tabs.

-p Prompt with * when taking input from the terminal.

-m When input returns to the terminal from a lower level, repeat the prompt, if any.

-a Make -k flag propagate to included sources, thereby protecting them from keyword substitutions.

-c List control lines on diagnostic output.

-d Extend the current set of keyword definitions by adding those active at the end of included sources.

-x This flag guarantees that the job will be transmitted in the order of submission (relative to other jobs sent with this flag).

Control lines are input lines that begin with ~. In the default mode +ir, they are interpreted as commands to *send*. Normally they are detected immediately and read literally. The -s flag forces keyword substitutions to be made before control lines are intercepted and interpreted. This can lead to unexpected results if a control line uses a keyword which is defined within an immediately preceding ~$ sequence. Arguments appearing in control lines are handled exactly like the command arguments to *send*, except that they are processed at a nested level of input.

The two possible formats for a control line are: ‘‘~argument’’ and ‘‘~ argument ...’’. In the first case, where the ~ is not followed by a space, the remainder of the line is taken as a single argument to *send*. In the second case, the line is parsed to obtain a sequence of arguments delimited by spaces. In this case the quotes ′ and " may be employed to pass embedded spaces.

The interpretation of the argument **.** is chosen so that an input line consisting of ~**.** is treated as a logical EOF. The following example illustrates some of the above conventions:

```
send  -
~ argument ...
~.
```

This sequence of three lines is equivalent to the command synopsis at the beginning of this description. In fact, the - is not even required. By convention, the *send* command reads standard input if no other input source is specified. *Send* may therefore be employed as a filter with side-effects.

The execution of the *send* command is controlled at each instant by a current environment, which includes the format specification for the input source, a default format specification for included sources, the settings of the mode flags, and the active set of keyword definitions. This environment can be altered dynamically. When a control line opens a new source of input, the current environment is pushed onto a stack, to be restored when input resumes from the old source. The initial format specification for the new source is taken from the first line of the file. If none is provided, the established default is used or, in its absence, standard tabs. The initial mode settings and active keywords are copied from the old environment. Changes made while processing the new source will not affect the environment of the old source, with one exception: if -d mode is set in the old environment, the old keyword context will be augmented by those definitions that are active at the end of the new source.

When *send* first begins execution, all mode flags are reset, and the values of the shell environment variables become the initial values for keywords of the same name with a $ prefixed.

The initial reset state for all mode flags is the + state. In general, special processing associated with a mode *N* is invoked by flag -*N* and is revoked by flag +*N*. Most mode settings have an immediate effect on the processing of the current source. Exceptions to this are the -r and -i flags, which apply only to included source, causing it to be processed in an uninterpreted manner.

A keyword is an arbitrary 8-bit ASCII string for which a replacement has been defined. The replacement may be another string, or (for IBM RJE only) the hexadecimal code for a single 8-bit byte. At any instant, a given set of keyword definitions is active. Input text lines are scanned, in one pass from left to right, and longest matches are attempted between substrings of the line and the active set of keywords. Characters that do not match are output, subject to folding and the standard translation. Keywords are replaced by the specified hexadecimal code or replacement string, which is then output character by character. The expansion of tabs and length checking, according to the format specification of an input source, are delayed until substitutions have been made in a line.

All of the keywords definitions made in the current source may be deleted by setting the `-k` flag. It then becomes possible to reuse them. Setting the `-k` flag also causes keyword definitions active at the previous source level to be ignored. Setting the `+k` flag causes keywords at the previous level to be ignored but does not delete the definitions made at the current level. The `=k` argument reactivates the definitions of the previous level.

When keywords are redefined, the previous definition at the same level of source input is lost, however the definition at the previous level is only hidden, to be reactivated upon return to that level unless a `-d` flag causes the current definition to be retained.

Conditional prompts for keywords, `?:A,/p` which have already been defined at some higher level to be null or have a replacement will simply cause the definitions to be copied down to the current level; new definitions will not be solicited.

Keyword substitution is an elementary macro facility that is easily explained and that appears useful enough to warrant its inclusion in the *send* command. More complex replacements are the function of a general macro processor (*m4*(1), perhaps). To reduce the overhead of string comparison, it is recommended that keywords be chosen so that their initial characters are unusual. For example, let them all be upper case.

Send performs two types of error checking on input text lines. Firstly, only ASCII graphics and tabs are permitted in input text. Secondly, the length of a text line, after substitutions have been made, may not exceed 80 bytes for IBM, or 132 bytes for UNIVAC. The length of each line may be additionally constrained by a size parameter in the format specification for an input source. Diagnostic output provides the location of each erroneous line, by line number and input source, a description of the error, and the card image that results. Other routine errors that are announced are the inability to open or write files, and abnormal exits from the shell. Normally, the occurrence of any error causes *send*, before invoking the queuer, to prompt for positive affirmation that the suspect run stream should be submitted.

For IBM hosts, *send* is required to translate 8-bit ASCII characters into their EBCDIC equivalents. The conversion for 8-bit ASCII characters in the octal range 040-176 is based on the character set described in ''Appendix H'' of *IBM System/370 Principles of Operation* (IBM SRL GA22-7000). Each 8-bit ASCII character in the range 040-377 possesses an EBCDIC equivalent into which it is mapped, with five exceptions: ~ into ¬, 0345 into ~, 0325 into [illegible], 0313 into |, 0177 (DEL) is illegal. In listings requested from *send* and in printed output returned by the subsystem, the reverse translation is made with the qualification that EBCDIC characters that do not have valid 8-bit ASCII equivalents are translated into ^. UNIVAC hosts, on the other hand, operate in ASCII code, and any translations between ASCII and field-data are made, in accordance with the UNIVAC standard, by the host computer.

Additional control over the translation process is afforded by the `-f` flag and hexadecimal character codes. As a default, *send* folds lower-case letters into upper case. For UNIVAC RJE it does more: the entire ASCII range 0140-0176 is folded into 0100-0136, so that ‘, for example, becomes @. In either case, setting the `-f` flag inhibits any folding. Non-standard character codes are obtained as a special case of keyword substitution.

SEE ALSO

m4(1), orjestat(1C), rjestat(1C), sh(1), fspec(5), ascii(7), hasp(8), rje(8), uvac(8).
Guide to IBM Remote Job Entry for PWB/UNIX Users by A. L. Sabsevitz and E. J. Finger.
UNIX Remote Job Entry User's Guide by K. A. Kelleman.

BUGS

Standard input is read in blocks, and unused bytes are returned via *lseek*(2). If standard input is a pipe, multiple arguments of the form - and -:*prompt* should not be used, nor should the logical EOF (~.). delim @@

NAME

vpmc − compiler for the virtual protocol machine

SYNOPSIS

`vpmc` [`-m`] [`-r`] [`-c`] [`-x`] [`-s` sfile] [`-l` lfile] [`-i` ifile] [`-o` ofile] file

DESCRIPTION

Vpmc is the compiler for a language that is used to describe communications link protocols. The output of *vpmc* is a load module for the virtual protocol machine (VPM), which is a software construct for implementing communications link protocols (e.g., BISYNC) on the DEC KMC11 microprocessor. VPM is implemented by an interpreter in the KMC11 which cooperates with a driver in the UNIX host computer to transfer data over a communications link in accordance with a specified link protocol. UNIX user processes transfer data to or from a remote terminal or computer system through VPM using normal UNIX *open*, *read*, *write*, and *close* operations. The VPM program in the KMC11 provides error control and flow control using the conventions specified in the protocol.

The language accepted by *vpmc* is essentially a subset of C; the implementation of *vpmc* uses the RATFOR preprocessor (*ratfor*(1)) as a front end; this leads to a few minor differences, mostly syntactic.

There are two versions of the interpreter. The appropriate version for a particular application is selected by means of the `-i` option. The BISYNC version (`-i bisync`) supports half-duplex, character-oriented protocols such as the various forms of BISYNC. The HDLC version (`-i hdlc`) supports full-duplex, bit-oriented protocols such as HDLC. The communications primitives used with the BISYNC version are character-oriented and blocking; the primitives used with the HDLC version are frame-oriented and non-blocking.

Options

The meanings of the command-line options are:

`-m`	Use *m4*(1) instead of *cpp* as the macro preprocessor.
`-r`	Produce RATFOR output on the standard output and suppress the remaining compiler phases.
`-c`	Compile only (suppress the assembly and linking phases).
`-x`	Retain the intermediate files used for communication between passes.
`-s` *sfile*	Save the generated VPM assembly language on file *sfile*.
`-l` *lfile*	Produce a VPM assembly-language listing on file *lfile*.
`-i` *ifile*	Use the interpreter version specified by *ifile* (default `bisync`).
`-o` *ofile*	Write the executable object file on file *ofile* (default `a.out`).

These options may be given in any order.

Programs

Input to *vpmc* consists of a (possibly null) sequence of array declarations, followed by one or more function definitions. The first defined function is invoked (on command from the UNIX VPM driver) to begin program execution.

Functions

A function definition has the following form:

> function *name*()
> *statement_list*
> end

Function arguments (formal parameters) are not allowed. The effect of a function call with arguments can be obtained by invoking the function via a macro that first assigns the value of each argument to a global variable reserved for that purpose. See *EXAMPLES* below.

A *statement_list* is a (possibly null) sequence of labeled statements. A *labeled_statement* is a statement preceded by a (possibly null) sequence of labels. A *label* is either a name followed by a colon (`:`) or a decimal integer optionally followed by a colon.

The statements that make up a statement list must be separated by semicolons (`;`). (A semicolon at the end of a line can usually be omitted; refer to the description of RATFOR for details.) Null statements are allowed.

Statement Syntax

The following types of statements are allowed:

expression
lvalue=*expression*
lvalue+=*expression*
lvalue-=*expression*
lvalue |=*expression*
lvalue&=*expression*
lvalue∧=*expression*
lvalue≪=*expression*
lvalue≫=*expression*
if(*expression*)*statement*
if(*expression*)*statement* else *statement*
while(*expression*)*statement*
for(*statement*; *expression*; *statement*)*statement*
repeat *statement*
repeat *statement* until *expression*
break
next
switch(*expression*){*case_list*}
return(*expression*)
return
goto *name*
goto *decimal_constant*
{*statement_list*}

`repeat` is equivalent to the `do` keyword in C; `next` is equivalent to `continue`.

A *case_list* is a sequence of statement lists, each of which is preceded by a label of the form:

case *constant*:

The label for the last *statement_list* in a *case_list* may be of the form:

default:

Unlike C, RATFOR supplies an automatic `break` preceding each new case label.

Expression Syntax

A *primary_expression* (abbreviated *primary*) is an lvalue or a constant. An *lvalue* is one of the following:

name
name[*constant*]

A *unary_expression* (abbreviated *unary*) is one of the following:

primary
name()
system_call
++*lvalue*
--*lvalue*
(*expression*)
!*unary*
~*unary*

The following types of expressions are allowed:

unary
unary+*primary*
unary-*primary*
unary |*primary*
unary&*primary*
unary&~*primary*
unary∧*primary*
unary≪*primary*

unary>> *primary*
unary== *primary*
unary!= *primary*
unary> *primary*
unary< *primary*
unary>= *primary*
unary<= *primary*

Note that the right operand of a binary operator can only be a constant, a name, or a name with a constant subscript.

System Calls

A VPM program interacts with a communications device and a driver in the host computer by means of system calls (primitives).

The following primitives are available only in the BISYNC version of the interpreter:

`crc16(`*primary*`)`
: The value of the primary expression is combined with the cyclic redundancy check-sum at the location passed by a previous `crcloc` system call. The CRC-16 polynomial @(x sup 16 + x sup 15 + x sup 2 + 1)@ is used for the check-sum calculation.

`crcloc(`*name*`)`
: The two-byte array starting at the location specified by *name* is cleared. The address of the array is recorded as the location to be updated by subsequent `crc16` system calls.

`get(`*lvalue*`)`
: Get a byte from the current *transmit* buffer. The next available byte, if any, is copied into the location specified by *lvalue*. The returned value is zero if a byte was obtained, otherwise it is non-zero.

`getrbuf(`*name*`)`
: Get (open) a *receive* buffer. The returned value is zero if a buffer is available, otherwise it is non-zero. If a buffer is obtained, the buffer parameters are copied into the array specified by *name*. The array should be large enough to hold at least three bytes. The meaning of the buffer parameters is driver-dependent. If a receive buffer has previously been opened via a `getrbuf` call but has not yet been closed via a call to `rtnrbuf`, that buffer is reinitialized and remains the current buffer.

`getxbuf(`*name*`)`
: Get (open) a *transmit* buffer. The returned value is zero if a buffer is available, otherwise it is non-zero. If a buffer is obtained, the buffer parameters are copied into the array specified by *name*. The array should be large enough to hold at least three bytes. The meaning of the buffer parameters is driver-dependent. If a transmit buffer has previously been opened via a `getxbuf` call but has not yet been closed via a call to `rtnxbuf`, that buffer is reinitialized and remains the current buffer.

`put(`*primary*`)`
: Put a byte into the current *receive* buffer. The value of the primary expression is inserted into the next available position, if any, in the current receive buffer. The returned value is zero if a byte was transferred, otherwise it is non-zero.

`rcv(`*lvalue*`)`
: *Receive* a character. The process delays until a character is available in the input silo. The character is then moved to the location specified by *lvalue* and the process is reactivated.

`rsom(`*constant*`)`
: Skip to the beginning of a new *receive* frame. The receiver hardware is cleared and the value of *constant* is stored as the receive sync character. This call is used to synchronize the local receiver and remote transmitter when the process is ready to accept a new receive frame.

`rtnrbuf(`*name*`)`
: Return a *receive* buffer. The original values of the buffer parameters for the current receive buffer are replaced with values from the array specified by *name*. The current receive buffer is then released to the driver.

`rtnxbuf(`*name*`)`
: Return a *transmit* buffer. The original values of the buffer parameters for the current transmit buffer are replaced with values from the array specified by *name*. The current transmit buffer is then released to the driver.

`xeom(`*constant*`)`
: Transmit end-of-message. The value of the constant is transmitted, then the transmitter is shut down.

`xmt(`*primary*`)`
: Transmit a character. The value of the primary expression is transmitted over the communications line. If the output silo is full, the process waits until there is room in the silo.

`xsom(`*constant*`)`
: Transmit start-of-message. The transmitter is cleared, then the value of *constant* is transmitted six times. This call is used to synchronize the local transmitter and the remote receiver at the beginning of a frame.

The following primitives are available only with the HDLC version of the interpreter:

`abtxfrm( )`
: The current transmission, if any, is aborted, if possible, by sending a frame-abort sequence (seven one bits, followed immediately by a terminating flag). This operation is not feasible with some hardware interfaces, in which case this primitive is a no-operation.

`getxfrm(`*primary*`)`
: Get a transmit buffer. If the transmit-buffer queue is *not* empty, the buffer at the head of the queue is removed from the queue and attached to the sequence number specified by the value of the *primary* expression. If the sequence number is greater than seven or the sequence number already has a buffer attached, the process is terminated in error. The returned value is zero if a buffer was obtained, otherwise non-zero.

`rcvfrm(`*name*`)`
: Get a completed receive frame. If the queue of completed receive frames is non-empty, the frame at the head of the queue is removed and becomes the current receive frame. If a frame is obtained, the first five bytes of the frame are copied into the array specified by *name*. The returned value is `true` (non-zero) if a frame was obtained; otherwise, it is `false` (zero). The rightmost four bits of the returned value indicate the frame length as follows: if the value of the rightmost four bits is equal to fifteen, the frame length is greater than or equal to 15; otherwise the frame length is equal to the value of the rightmost four bits. The frame length includes the two CRC bytes at the end of the frame and any control information at the beginning of the frame. Bytes following the first two bytes of the frame, but not including the two CRC bytes, are copied into a receive buffer, if one is available at the time the frame is received. Bit 020 of the returned value is zero if a receive buffer was available, otherwise non-zero. The values of the leftmost three bits of the returned value are currently unspecified. If a frame was obtained, the first five bytes of the frame are copied into the array specified by *name*. Frames with errors are discarded; a count is kept for each type of error. Frames may be discarded for any of the following reasons: (1) CRC error, (2) frame too short (less than four bytes), (3) frame too long (buffer size exceeded), or (4) no receive buffer available. If a frame with a buffer attached was previously obtained with `rcvfrm`, but the buffer has not been released to the driver with `rtnrfrm`, that buffer is returned to the queue of empty receive buffers. At most one receive frame with no buffer attached is retained by the interpreter; if a new frame arrives before the frame with no buffer attached has been obtained with `rcvfrm`, the new frame is discarded.

`rtnrfrm( )`
: Return a receive buffer. The current receive buffer (the one obtained by the most recent `rcvfrm` primitive) is returned to the driver. If there is no current receive buffer, the process is terminated in error.

`rsxmtq( )`
: Reset the transmit-buffer queue. The sequence number assignment is removed from all transmit buffers. If a transmission is currently in progress, the transmission is aborted, if possible.

`rtnxfrm(`*primary*`)`
: Return a transmit buffer. The transmit buffer currently attached to the sequence number specified by the value of the *primary* is returned to the driver and the sequence number assignment is removed from that buffer. If the specified sequence number does not have a buffer attached, the process is terminated in error. Transmit buffers must be returned in the same sequence in which they were obtained, otherwise the process is terminated in error.

`setctl(`*name*`,` *primary*`)`
: Specify transmit-control information. The number of bytes specified by the *primary* are copied from the array specified by *name* and saved for use with subsequent `xmtfrm` or `xmtctl` primitives. If the transmitter is currently busy, the process is terminated in error.

`xmtbusy( )`
: Test for transmitter busy. If a frame is currently being transmitted, the returned value is `true` (non-zero); otherwise the returned value is `false` (zero).

`xmtctl( )`
: Transmit a control frame. If a transmission is not already in progress, a new transmission is initiated. The transmitted frame will contain the control information specified by the most recent `setctl` primitive, followed by a two-byte CRC. The CRC-CCITT polynomial @(x sup 16 + x sup 12 + x sup 5 + 1)@ is used for the CRC calculation. The returned value is zero if a new transmission was initiated, otherwise non-zero.

`xmtfrm(`*primary*`)`
: Transmit an information frame. If a transmission is not already in progress, a new transmission is initiated. The transmitted frame will contain the control information specified by the most recent `setctl` primitive, followed by the contents of the buffer which is currently attached to the sequence number specified by the value of the *primary* expression, followed by a two-byte CRC. The CRC-CCITT polynomial @(x sup 16 + x sup 12 + x sup 5 + 1)@ is used for the CRC calculation. The returned value is zero if a new transmission was initiated, otherwise non-zero. If the sequence number is greater than seven or the sequence number does not have a buffer attached, the process is terminated in error.

The following primitives are available with all versions of the interpreter:

`dsrwait( )`
: Wait for modem-ready and then set modem-ready mode. The process delays until the modem-ready signal from the modem interface is asserted. If the modem-ready signal subsequently drops, the process is terminated. If `dsrwait` is never invoked, the modem-ready signal is ignored.

`exit(`*primary*`)`
: Terminate execution. The process is halted and the value of the primary expression is passed to the driver.

`getcmd(`*name*`)`
: Get a command from the driver. If a command has been received from the driver since the last call to `getcmd`, four bytes of command information are copied into the array specified by *name* and a value of `true` (non-zero) is returned. If no command is available, the returned value is `false` (zero).

`pause( )`
: Return control to the dispatcher. This primitive informs the dispatcher that the virtual process may be suspended until the next occurrence of an event that might affect the state of the protocol for this line. Examples of such events are: (1) completion of an output transfer, (2) completion of an input transfer, (3) timer expiration, and (4) a buffer-in command from the driver. In a multi-line implementation, the `pause` primitive allows the process for a given line to give up control to allow the processor to service another line.

`rtnrpt(`*name*`)`
: Return a report to the driver. Four bytes from the array specified by *name* are transferred to the driver. The process delays until the transfer is complete.

`testop(`*primary*`)`
: Test for odd parity. The returned value is `true` (non-zero) if the value of the primary

expression has odd parity, otherwise the returned value is `false` (zero).

`timeout(`*primary*`)`
: Schedule or cancel a timer interrupt. If the value of the *primary* expression is non-zero, the current values of the program counter and stack pointer are saved and a timer is loaded with the value of *primary*. The system call then returns immediately with a value of `false` (zero) as the returned value. The timer is decremented each tenth of a second thereafter. If the timer is decremented to zero, the saved values of the program counter and stack pointer are restored and the system call returns with a value of `true` (non-zero). The effect of the timer interrupt is to return control to the code immediately following the `timeout` system call, at which point a non-zero return value indicates that the timer has expired. The `timeout` system call with a non-zero argument is normally written as the condition part of an `if` statement. A `timeout` system call with a zero argument value cancels all previous `timeout` requests, as does a `return` from the function in which the `timeout` system call was made. A `timeout` system call with a non-zero argument value overrides all previous `timeout` requests. The maximum permissible value for the argument is 255, which gives a timeout period of 25.5 seconds.

`timer(`*primary*`)`
: Start a timer or test for timer expiration. If the value of the *primary* is non-zero, a software timer is loaded with the value of the *primary* and a value of `true` (non-zero) is returned. The timer is decremented each tenth of a second thereafter until it reaches zero. If the value of the *primary* is zero, the returned value is the current value of the timer; this will be `true` (non-zero) if the value of the timer is currently non-zero, otherwise `false` (zero). The timer used by this primitive is different from the timer used by the `timeout` primitive.

`trace(`*primary*`[,`*primary*`])`
: The values of the two primary expressions and the current value of the script location counter are passed to the driver. If the second *primary* is omitted, a zero is used instead. The process delays until the values have been accepted by the host computer.

Constants

A *constant* is a decimal, octal, or hexadecimal integer, or a single character enclosed in single quotes. A token consisting of a string of digits is taken to be an octal integer if the first digit is a zero, otherwise the string is interpreted as a decimal integer. If a token begins with `0x` or `0X`, the remainder of the token is interpreted as a hexadecimal integer. The hexadecimal digits include `a` through `f` or, equivalently, `A` through `F`.

Variables

Variable names may be used without having been previously declared. All names are global. All values are treated as 8-bit unsigned integers.

Arrays of contiguous storage may be allocated using the `array` declaration:

array *name*[*constant*]

where *constant* is a decimal integer. Elements of arrays can be referenced using constant subscripts:

name[*constant*]

Indexing of arrays assumes that the first element has an index of zero.

Names

A *name* is a sequence of letters and digits; the first character must be a letter. Upper- and lower-case letters are considered to be distinct. Names longer than 31 characters are truncated to 31 characters. The underscore (_) may be used within a name to improve readability, but is discarded by RATFOR.

Preprocessor Commands

If the `-m` option is omitted, comments, macro definitions, and file inclusion statements are written as in C. Otherwise, the following rules apply:

1. If the character # appears in an input line, the remainder of the line is treated as a comment.

2. A statement of the form:

define(*name*, *text*)

causes every subsequent appearance of *name* to be replaced by *text*. The defining text includes everything after the comma up to the balancing right parenthesis; multi-line definitions are allowed. Macros may have arguments. Any occurrence of $ *n* within the replacement text for a macro will be replaced by the *n*th actual argument when the macro is invoked.

3. A statement of the form:

   include(*file*)

   inserts the contents of *file* in place of the `include` command. The contents of the included file is often a set of definitions.

EXAMPLES

These examples require the use of the `-m` option.

```
# The function defined below transmits a frame in transparent BISYNC.
# A transmit buffer must be obtained with getxbuf before the function
# is invoked.
#
# Define symbolic constants:
#
define(DLE,0x10)
define(ETB,0x26)
define(PAD,0xff)
define(STX,0x02)
define(SYNC,0x32)
#
# Define a macro with an argument:
#
define(xmtcrc,{crc16($1); xmt($1);})
#
# Declare an array:
#
array crc[2];
#
# Define the function:
#
function xmtblk()
        crcloc(crc);
        xsom(SYNC);
        xmt(DLE);
        xmt(STX);
        while(get(byte)==0){
                if(byte == DLE)
                        xmt(DLE);
                xmtcrc(byte);
        }
        xmt(DLE);
        xmtcrc(ETB);
        xmt(crc[0]);
        xmt(crc[1]);
        xeom(PAD);
end
#
# The following example illustrates the use of macros to simulate a
# function call with arguments.
#
# The macro definition:
#
define(xmtctl,{c=$1;d=$2;xmtctl1()})
#
```

```
# The function definition:
#
function xmtctl1()
        xsom(SYNC);
        xmt(c);
        if(d!=0)
                xmt(d);
        xeom(PAD);
end
#
# Sample invocation:
#
function test()
        xmtctl(DLE,0x70);
end
```

FILES

sas_temp∗	temporaries
/tmp/sas_ta??	temporary
/tmp/sas_tb??	temporary
/usr/lib/vpm/pass∗	compiler phases
/usr/lib/vpm/pl	compiler phase
/usr/lib/vpm/vratfor	compiler phase
/lib/cpp	preprocessor
/usr/bin/m4	preprocessor
/bin/kasb	KMC11−B assembler
/usr/lib/vpm/bisync/∗	interpreter source for the BISYNC interpreter
/usr/lib/vpm/hdlc/∗	interpreter source for the HDLC interpreter

SEE ALSO

m4(1), ratfor(1), vpmstart(1C), vpm(4).
C Reference Manual by D. M. Ritchie.
RATFOR−A Preprocessor for a Rational Fortran by B. W. Kernighan.
The M4 Macro Processor by B. W. Kernighan and D. M. Ritchie.
Software Tools by B. W. Kernighan and P. J. Plauger (pp. 28−30).

NAME

vpm – The Virtual Protocol Machine

DESCRIPTION

This entry describes a particular kind of special file and gives an introduction to the Virtual Protocol Machine (VPM).

The VPM is a software construct for implementing link protocols on the KMC11 in a high-level language. This is accomplished by a compiler that runs on UNIX and that translates a high-level language description of a protocol into an intermediate language that is interpreted by an interpreter running in the KMC11.

The VPM driver is functionally split into two parts: a top VPM device and a bottom VPM device. The top device may be modified or replaced to suit particular applications; the bottom device interfaces with the VPM interpreter using the KMC driver. When using the *mknod* command to make a directory entry and corresponding i-node for a VPM special file, the minor device number identifies the top, bottom, and physical KMC devices to be used for this special file. The two most significant bits of the minor device number denote the physical KMC device; the next two bits denote the VPM bottom device; the four least significant bits denote the VPM top device. For example, if top device 1 is to be used with bottom device 2, which in turn is to be used with KMC device 3, the minor device number would be 0341(octal).

UNIX user processes transfer data to or from a remote terminal or computer system through VPM using normal *open*, *read*, *write*, and *close* operations. Flow control and error recovery are provided by the protocol description residing in the KMC11.

The VPM software consists of six components:

1. *vpmc*(1C): compiler for the protocol description language; it runs on UNIX.
2. VPM interpreter: a KMC11 program that controls the overall operation of the KMC11 and interprets the protocol script.
3. *vpm.c*: a UNIX driver that provides the interface to the VPM.
4. *vpmstart*(1C): a UNIX command that copies a load module into the KMC11 and starts it.
5. *vpmsnap*(1C): a UNIX command that prints a time-stamped event trace while the protocol is running.
6. *vpmtrace*(1C): a UNIX command that prints an event trace for debugging purposes while the protocol is running.

The VPM *open* for reading-and-writing is exclusive; *open*s for reading-only or writing-only are not. The VPM *open* checks that the correct interpreter is running in the KMC11, then sends a RUN command to the interpreter (causing it to start interpreting the protocol script), and supplies a 512-byte receive buffer to the interpreter.

The VPM *read* returns either the number of bytes requested or the number remaining in the current receive buffer, whichever is less. Bytes remaining in a receive buffer are used to satisfy subsequent reads. The VPM *write* copies the user data into 512-byte system buffers and passes them to the VPM interpreter in the KMC11 for transmission.

The VPM *close* arranges for the return of system buffers and for a general cleanup when the last transmit buffer has been returned by the interpreter.

The user command *vpmtrace*(1C) reads the trace driver and prints event records. While this command is executing, the VPM driver will generate a number of event records, allowing the activity of the VPM driver and protocol script to be monitored for debugging purposes. The system functions *vpmopen*, *vpmread*, *vpmwrite*, and *vpmclose* generate event records (identified respectively by o, r, w, and c). Calls to the *vpmc*(1C) primitive *trace*(*arg1*,*arg2*) cause the VPM interpreter to pass *arg1* and *arg2* along with the current value of the script location counter to the VPM driver, which generates an event record identified by a T. Each event record is structured as follows:

```
struct event {
        short   e_seqn;             /*sequence number*/
        char    e_type;         /*record identifier*/
        char    e_dev;          /*minor device number*/
        short   e_short1;       /*data*/
```

```
                short   e_short2;       /*data*/
        }
```

When the script terminates for any reason, the driver is notified and generates an event record identified by an E. This record also contains the minor device number, the script location counter, and a termination code defined as follows:

0	Normal termination; the interpreter received a *halt* command from the driver.
1	Undefined virtual-machine operation code.
2	Script program counter out of bounds.
3	Interpreter stack overflow or underflow.
4	Jump address not even.
5	UNIBUS error.
6	Transmit buffer has an odd address; the driver tried to give the interpreter too many transmit buffers; or a *get* or *rtnxbuf* was executed while no transmit buffer was open, i.e., no *getxbuf* was executed prior to the *get* or *rtnxbuf*.
7	Receive buffer has an odd address; the driver tried to give the interpreter too many receive buffers; or a *put* or *rtnrbuf* was executed while no receive buffer was open, i.e., no *getrbuf* was executed prior to the *get* or *rtnxbuf*.
8	The script executed an *exit*.
9	A *crc16* was executed without a preceding *crcloc* execution.
10	Interpreter detected loss of modem-ready signal.
11	Transmit-buffer sequence-number error.
12	Command error; an invalid command or an improper sequence of commands was received from the driver.
13	Not used.
14	Invalid transmit state.
15	Invalid receive state.
16	Not used.
17	*Xmtctl* or *setctl* attempted while transmitter was still busy.
18	Not used.
19	Same as error code 6.
20	Same as error code 7.
21	Script to large.
22	Used for debugging the interpreter.
23	The driver's OK-check has timed out.

SEE ALSO

vpmc(1C), vpmstart(1C), trace(4).

NAME

pnch – file format for card images

DESCRIPTION

The PNCH format is a convenient representation for files consisting of card images in an arbitrary code.

A PNCH file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0–80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

NAME

rje – RJE (Remote Job Entry) to IBM

SYNOPSIS

```
/usr/rje/rjeinit
/usr/rje/rjehalt
```

DESCRIPTION

RJE is the communal name for a collection of programs and a file organization that allows a UNIX system, equipped with a KMC11-B, KMC11 driver, and associated Virtual Protocol Machine (VPM) software, to communicate with IBM's Job Entry Subsystems by mimicking an IBM 360 remote multileaving work station.

Implementation.

RJE is initiated by the command *rjeinit* and is terminated gracefully by the command *rjehalt*. While active, RJE runs in the background and requires no human supervision. It quietly transmits, to the IBM system, jobs that have been queued by the *send*(1C) command, and operator requests that have been entered by the *rjestat*(1C) command. It receives, from the IBM system, print and punch data sets and message output. It enters the data sets into the proper UNIX directory and notifies the appropriate user of their arrival. It scans the message output to maintain a record on each of its jobs. It also makes these messages available for public inspection, so that *rjestat*(1C), in particular, may extract responses.

Unless otherwise specified, all files and commands described below reside in directory `/usr/rje` (first exceptions: *send* and *rjestat*).

There are two sources of data to be transmitted by RJE from UNIX to an IBM System/370. In both cases, the data is organized as files in the `/usr/rje/squeue` directory. The first are files named `co`∗ which are created by the enquiry command *rjestat*(1C). The second source, containing the bulk of the data, are files named `rd`∗ or `sq`∗ which have been created by *send* and queued by the program *rjeqer*. On completion of processing *send* invokes *rjeqer*. *Rjeqer* and *rjestat* inform the program *rjexmit* that a file has been queued via the file `joblog`. Upon successful transmission of the data to the IBM machine, *rjexmit* removes the queued file. As files are transmitted and received, the program *rjedisp* writes an entry containing the date, time, file name, logname, and number of records in the file `acctlog`, if it exists. This file can be used for local logging or accounting information, but is not used elsewhere by RJE. The use of this information is up to the RJE administrator.

Each time *rjeinit* is invoked, the `joblog` file is truncated and recreated from the contents of the `/usr/rje/squeue` directory. During this time, *rjeinit* prevents simultaneous updating of the `joblog` file.

Output from the IBM system is classified as either a print data set, a punch data set, or message output. Print output is converted to an ASCII text file, with standard tabs. Form feeds are suppressed, but the last line of each page is distinguished by the presence of an extraneous trailing space. Punch output is converted to *pnch*(5) format. This classification and both conversions occur as the output is received. Files are moved or copied into the appropriate user's directory and assigned the name `prnt`∗ or `pnch`∗, respectively, or placed into user directories under user-specified names, or used as input to programs to be automatically executed, as specified by the user. This process is driven by the ''usr=...'' specification. RJE retains ownership of these files and permits read-only access to them. Message output is digested by RJE immediately and is not retained.

A record is maintained for each job that passes through RJE. Identifying information is extracted contextually from files transmitted to and received from the IBM system. This information is stored and used by the *rjedisp* program for IBM job acknowledgements and delivery of output files.

The IBM system automatically returns an acknowledgement message for each job it receives. Other status messages are returned in response to enquiries entered by users. All messages received by RJE are appended to the `resp` file. The `resp` file is automatically truncated when it reaches 70,000 bytes. Each enquiry is preceded and followed by an identification card image of the form ''$UX<*process id*>''. The IBM system will echo this back as an illegal command. The appearance of process ids in the response stream permits responses to be passed on to the proper users.

While it is active, RJE occupies at least the three process slots that are appropriated by *rjeinit*. These slots are used to run *rjexmit*, the transmitter, *rjerecv*, the receiver, and *rjedisp*, the dispatcher. These three processes are connected by pipes. The function of each is as follows:

rjexmit
: Cycles repetitively, looking for data to transmit to the IBM system. After transmission, *rjexmit* passes an event notice to *rjedisp*. If *rjexmit* encounters a `stop` file, (created by *rjehalt*), it exits normally. In the case of error termination, *rjexmit* reboots RJE by executing *rjeinit*.

rjerecv
: Cycles repetitively, looking for data returning from the IBM machine. Upon receipt of data, *rjerecv* notifies either *rjexmit* or *rjedisp* of the event (transfer information is sometimes passed to *rjexmit*). *Rjerecv* exits normally at the first appropriate moment when it encounters the file `stop`, or exits reluctantly when it encounters a run of errors.

rjedisp
: Follows up event notices by directing output files, updating records, and notifying users. *Rjedisp* references the system files `/etc/passwd` and `/etc/utmp` to correlate user names, numeric ids, and terminals. Termination of *rjerecv* causes *rjedisp* to exit also.

Rjeinit has the capability of *dialing* any remote IBM system with the proper hardware and software configuration.

Most RJE files and directories are protected from unauthorized tampering. The exception is the `spool` directory. It is used by *send*(1C) to create temporary files in the correct file system. *Rjeqer* and *rjestat*(1C), the user's interfaces to RJE, operate in *setuid* mode to contribute the necessary permission modes.

Administration.

Some minimal oversight of each RJE subsystem is required. The RJE mailbox should be inspected and cleaned out periodically. The `job` directory should also be checked. The only files placed there are output files whose destination file systems are out of space. Users should be given a short period of time (say, a day or two), and then these files should be removed.

The configuration table `/usr/rje/lines` is accessed by all components of RJE. Each line of the table (maximum of 8) defines an RJE connection. Its seven columns may be labeled *host*, *system*, *directory*, *prefix*, *device*, *peripherals* and *parameters*. These columns are described as follows:

`host`
: The name of a remote IBM computer (e.g., A B C). This string can be up to 5 characters.

`system`
: The name of a UNIX system. This name should be the same as the system name from *uname*(1).

`directory`
: This is the directory name of the servicing RJE subsystem (e.g., `/usr/rje1`).

`prefix`
: This is the string prefixed (redundantly) to several crucial files and programs in `directory` (e.g., `rje1`, `rje2`, `rje3`).

`device`
: This is the name of the controlling VPM device, with `/dev/` excised.

`peripherals`
: This field contains information on the logical devices (readers, printers, punches) used by RJE. Each subfield is separated by `:`, and is described as follows:

(1) Number of logical readers.
(2) Number of logical printers.
(3) Number of logical punches.

Note: the number of peripherals specified for an RJE subsystem `must` agree with the number of peripherals which have been described on the remote machine for that line.

`parameters`

This field contains information on the type of connection to make. Each subfield is separated by **:**. Any or all fields may be omitted; however, the fields are positional. All but trailing delimiters must be present. For example, in

```
1200:512:::9-555-1212
```

subfields 3 and 4 are missing, but the delimiters are present. Each subfield is defined as follows:

(1) `space`

This subfield specifies the amount of space (*S*) in blocks that RJE tries to maintain on file systems it touches. The default is 0 blocks. *Send* will not submit jobs and *rjeinit* issues a warning when less than 1.5*S* blocks are available; *rjerecv* stops accepting output from the host when the capacity falls to *S* blocks; RJE becomes dormant, until conditions improve. If the space on the file system specified by the user on the ‘‘usr=’’ card would be depleted to a point below *S*, the file will be put in the `job` subdirectory of the connection’s home directory, rather than in the place that the user requested.

(2) `size`

This subfield specifies the size in blocks of the largest file that can be accepted from the host without truncation taking place. The default is no truncation.

(3) `badjobs`

This subfield specifies what to do with undeliverable returning jobs. If an output file is undeliverable for any reason other than file system space limitations (e.g., missing or invalid ‘‘usr=’’ card) and this subfield contains the letter **y**, the output will be retained in the `job` subdirectory of the home directory, and login `rje` is notified. If this subfield contains an `n` or has any other value, undeliverable output will be discarded. The default is **n**.

(4) `console`

This subfield specifies the status of the interactive status terminal for this line. If the subfield contains an **i**, all console status facilities are inhibited (e.g., *rjestat*(1C) will not behave like a status terminal). In all cases, the normal non-interactive uses of *rjestat*(1C) will continue to function. The default is **y**.

(5) `dial-up`

This subfield contains a telephone number to be used to call a host machine. The telephone number may contain the digits 0 thru 9 and the character - which denotes a pause. If the telephone number is not present, no dialing is attempted and a leased line is assumed.

Sign-on is controlled by the existence of a `signon` file in the home directory. If this file is present, its contents are sent as a sign-on message to the host system. If this file does not exist, a blank card is sent. Sign-off is controlled in the same way, except that the `signoff` file is sent by *rjehalt* if it exists. If the `signoff` file does not exist, a ‘‘`/*signoff`’’ card is sent. These files should be ASCII text and no more than 80 characters.

Send(1C) and *rjestat*(1C) select an available connection by indexing on the `host` field of the configuration table. RJE programs index on the `prefix` field. A subordinate directory, `sque`, exists in `/usr/rje` for use by *rjedisp* and *shqer* programs. This directory holds those output files that have been designated as standard input to some executable file. This designation is done via the ‘‘usr=**...**’’ specification. *Rjedisp* places the output files here and updates the file `log` to specify the order of execution, arguments to be passed, etc. *Shqer* executes the appropriate files.

All RJE programs are shared text; therefore, if more than one RJE is to be run on a given UNIX system, simply link (via *ln*(1)) RJE2 program names to RJE names in **/usr**.

SEE ALSO

rjestat(1C), send(1C), vpm(4), pnch(5), mk(8).
UNIX Remote Job Entry User's Guide by K. A. Kelleman.
UNIX Remote Job Entry Administrative Guide by M. J. Fitton.
Setting Up UNIX.

DIAGNOSTICS

Rjeinit provides brief error messages describing obstacles encountered while bringing up RJE. They can best be understood in the context of the RJE source code. The most frequently occurring one is ''cannot open /dev/vpm?''. This may occur if the VPM script has not been started, or if another process already has the VPM device open.

Once RJE has been started, users should assist in monitoring its performance, and should notify operations personnel of any perceived need for remedial action. *Rjestat*(1C) will aid in diagnosing the current state of RJE. It can detect, with some reliability, when the far end of the communications line has gone dead, and will report in this case that the host computer is not responding to RJE. It will also attempt to reboot RJE if it detects a prolonged period of inactivity on the KMC-11B.