## 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-B microprocessor. VPM is implemented by an interpreter in the KMC 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 KMC 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**, the default) 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 **hdlc**). |
| **−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:

**atoe**(*primary*)
: Translate ASCII to EBCDIC. The returned value is the EBCDIC character that corresponds to the ASCII character represented by the value of the primary expression. The translation tables reflect the prejudices of a particular installation.

**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 $(x^{16}+x^{15}+x^2+1)$ 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.

**etoa**(*primary*)
: Translate EBCDIC to ASCII. The returned value is the ASCII character that corresponds to the EBCDIC character represented by the value of the primary expression. The translation tables reflect the prejudices of a particular installation.

**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.

**norbuf**()
Test for the availability of an empty receive buffer. The returned value is **true** (non-zero) if the queue of empty receive buffers is currently empty; otherwise the returned value is **false** (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 expression 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 expression 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 ($x^{16}+x^{12}+x^5+1$) 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 ($x^{16}+x^{12}+x^5+1$) 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. In a single-line implementation this primitive has no effect.

**snap**(*name*)
: Create a *snap* event record. Four bytes from the array specified by *name* are passed to

the driver, which prefixes a time stamp and sequence number and creates a trace event record containing the data. If minor device 1 of the trace driver is currently open, the record is placed on the read queue for that device; otherwise the event record is discarded. The information passed via the *snap* primitive can be displayed using the *vpmsnap* command (see *vpmstart*(1C)).

**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 the primary expression. 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 expression is non-zero, a software timer is loaded with the value of the primary expression 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 expression 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, which prefixes a sequence number and creates a trace event record containing the data. If minor device 0 of the trace driver is currently open, the record is placed on the read queue for that device; otherwise the event record is discarded. The information passed via the *trace* primitive can be displayed using the *vpmtrace* command (see *vpmstart*(1C)). If the second argument 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).