Abstract

Name: QUASIC-2 SYSTEM V2.2-1
Category: interactive compiler and execution environment
Required hardware: DEC PDP-11 or clone or simulator. Console device and some media to start this software from. FIS is needed for floating point operations.
OS: RT-11.

Applications: Interactive manual or simple algorithmic control over custom equipment with PDP-11-based recourse-limited CPU, data acquisition.

Author: UNKNOWN. It would be nice to find out the origin of this software, source code and better manual or other versions. Please mail to azimuth55@hotmail.com if you find some relevant material.

Source: Probably it was distributed in former USSR and Eastern Europe in mid 1980s for control of CAMAC based equipment with PDP-11 clones. I've used it in 1986-88 but have only seen the executable module. I've found one old 5.25-inch diskette with this software and imported it in simulator. I've recreated the "manual" from my own experience and some tests on simulator.

Description

This is a simple programming environment with interactive mode (compiles and executes either the whole program or single line).

When run it holds in memory:
- Compiler
- Simple source editor
- Interactive monitor
- Source code of your program
- Compiled code
- Run-time library
- Certain device drivers and file system interface.

Source code can be entered within the environment or read from file system or from one of the directly supported devices using READ or NEW command. It seems there is no way to produce standalone executables.

The manual

Language syntax is similar with BASIC: each line should start with the number. Source code lines are ordered according to these numbers. Unlike BASIC the numbers are rarely referred in statements and serve nearly only for ordering lines and for editing purposes. There are only two statements that use line numbers, both for occasional use. Ordinary statements have structure with opening and closing parts. All source code should be in capital letters.

When started QUASIC brings the prompt of interactive monitor where the user can type one of the following:
- Line of source code starting with line number (it replaces the line with the existing number or is being added or inserted into the source code without syntax check at this point)
- Line of source code without line number - it is being compiled and executed but not added to the rest of the source (but can modify its context - create new variables for example).
- Monitor command with parameters.

It seems there is no way to save the state of environment so a program can behave differently from session to session or even became syntactically incorrect (due to absence of interactively created variables).

Each source line (either numbered or for immediate execution) can hold several statements separated with backslash ("\") and can end with a comment starting with exclamation ("!").

Statements

Variable declaration

<Declaration>::=<Type><Identifier["("Dimension")"][:Address]>{,<Identifier>["("Dimension")"] [:Address]}
<Type>::=BYTE | INT | REAL
Dimension and Address are non-negative integer constants. Dimension is specified as the maximal index value for zero-based array, not as the number of items.
Integer constants can be decimal or octal. Octal constant has B suffix after numerical part.

Address is an optional part to refer device hardware registers or specific memory area. It is octal number without B suffix.
Real numbers are 32-bit single precision as are used in FIS.

There is one more type called CAMAC that refers to special hardware. I have no information about its syntax. Most likely its declaration follows CAMAC addressing scheme and its references generates code that does the I/O.

All the variables are global.

Assignment

<Assignment>::= <Identifier>[%]["("<Index>")"]=<Expression>

Expression has usual syntax with some special binary and unary operators in addition to standard arithmetic +, -, *, /:

Unary:

.ADC.	add carry from the last operation
.SBC.	subtract carry from the last operation
.NEG.	negate
.COM.	complement (logical not)
.ASL.	arithmetic shift left by 1
.ASR.	arithmetic shift right by 1
.ROL.	rotate left by 1 with carry flag
.ROR.	rotate right by 1 with carry flag
.INC.	increment by 1
.DEC.	decrement by 1
.SWAB.	swap high and low bytes of word-sized operand

Binary:

.BIC.	bit clear
.BIS.	bit set
.ADD.	same as +
.SUB.	same as -

% suffix is used for treating any variable as zero-based byte array. Index 0 can be omitted.

Constants can be used in expressions but the compiler does not try to evaluate them before the statement is executed, for example, writing R=2.0/3.0 produces code that executes division each time control is passed to this statement.

Integer constants have been described. Real constants follow usual syntax like -23.45 or 4.2E-12.

Character constants are represented as strings in single or double quotation marks. In assignment only the first 1..4 bytes are taken into account according to size of the variable. See PUT and GET statements for character array input and output.

#<Identifier> is used as a constant that refers the identifier address - useful for passing it to external code (see PROC statement) or writing to address registers of bus mastering devices for I/O programming.

Runtime library has certain math functions like SIN, COS, SQRT, that can be used in expressions (see SHOW SYS command to get the complete list). There is no way to extend it using the language itself.

Runtime library also has several predefined variables that keep some status information like:
$MOD - contains the reminder of the last division
$IOSW - contains result of last IO operation.
See SHOW SYS for complete list including size and type information.

Control statements

Conditional

IF <LogicalExpression> THEN <Statements> {ELSIF <Statements>} [ELSE <Statements>] ENDI

WHILE <LogicalExpression> <Statements> ENDW

REPEAT <Statements> UNTIL <LogicalExpression>

<LogicalExpression>::= <Expression><Condition><Expression>

<Condition>::= < | <= | >= | > | = | <>

FOR <IntVar> = <InitialValue> (TO | DOWNTO) <Limit> [STEP <StepValue>] <Statements> ENDF

This is self-explanatory but note that parts of compound statements like ELSIF, ELSE, ENDI, ENDW ENDF and UNTIL are separate entities so they should start on the new line or after "\" delimiter. This also applies to other structural control statements.

Unconditional

LOOP <Statements> ENDL

This is unconditional loop. Use EXIT or GOTO statement to terminate it. See also REQUEST statement for other possibility.

EXIT - terminates the innermost LOOP loop.
GOTO <LineNumber> - jumps to a specified line number (One of the rare one that use line numbers syntactically).

STOP - stops execution and invokes command monitor. Reaching end of program also stops.

The following example demonstrates direct I/O programming - it writes 1024 byte record on unit 0 of TM11 magnetic tape:

```
10 INT MTC:172522,MTCMA:172526,MTBRC:172524 ! TM11 Registers
20 BYTE BUF(1023) ! Our buffer is 1024 bytes
30 MTCMA=#BUF ! Current memory address register points to our buffer
40 MTBRC=-1024 ! Set byte count as negative
50 MTC=60005B ! Issue command: WRITE+GO unit 0
60 WHILE MTC%>0 \ ENDW ! Wait for completion (bit 7)
```

Note: % used to force BYTE access of MTC and index 0 is omitted, so MTC% is equivalent to MTC%(0).
See SHOW CODE command example to see how it is compiled.

Procedure call
<ProcedureIdentifier> [<Parameter>{,<Parameter>}]

Parameters are identifiers whose addresses are put into table pointed by R5 register. External procedures can use them. Internal ones however can't, as all variables are global.

Procedure identifier refers either to internal or external procedure defined by the PROC statement.

Procedure definitions

Internal procedures:
PROC <Identifier> <Statements> ENDP

External procedures:
PROC <Identifier>:<Address>
External procedure can access parameters by address table pointed by R5 register. It should return with RTS PC. External procedure probably could have been made resident before QUASIC execution or placed in internal buffer.

Internal procedure should use RETURN statement or just reach ENDP to exit.

Interrupt handling

ON <Vector> PSW <PSW> <Statements> ENDO
Interrupt handler is always active when the compiled program is in memory. There is no way to redefine the handler on the fly or disable it. PSW (octal number without suffix) is used to set processor priority level for processing this interrupt.

Program termination handler

ON STOP <Statements> ENDO

This handler is called in response to program termination due to voluntary stop or CTRL-P character coming from console. May be used for placing cleanup code.

RETURN statement is used to exit from internal procedures before control passes to the last statement of procedure, which also returns.

REQUEST statement defines line number where execution jumps in response to ESC char entered. Useful to interactively break lengthy or infinite processes without placing polling code. Unlike handler definition this is an executable statement, so ESC action can be dynamically redefined or cancelled. Syntax:

REQUEST [<LineNumber>]

REQUEST without parameter cancels ESC processing.
Disadvantage is that it refers the line number, so care should be taken when editing code.

Input and output statements

General form of input and output statements is

(PUT | GET) <Options> [<Channel>] [<ParameterList>]

<Options>::={(+|-)<Character>}
<Channel>::=<IntegerNumber>
<ParameterList>::=<Parameter>{,<Parameter>}
<Parameter>::=<Identifier>[:<FormatSpecifier>] | <String>

When channel is omitted console device is used. Empty parameter list useful for putting new line code.
Options can be turned on (+) or off (-).

Options for PUT:
L - put new line after output (default is +)

Options for GET:
P - put question mark prompt before waiting for input (default is +)
F - ignore I/O errors (default is -)

Format specifiers:
:I[<Length>] - Integer
:O[<Length>] - Octal
:F[<Length>.<Decimals>] - Real number without exponent (like 123456.78)
:E[<Length>.<Decimals>] - Exponential floating point (like 0.12E 13)
:A[:<Length>] - Character

Length field for output sets width and number of bytes to copy. Asterisks are printed if it is not long enough for numerical formats.
Length field for input with character format is a variable that receives the actual number of bytes read. This is useful for string processing. Input string length counts the end of line character (zero byte). For string output zero byte forces new line being printed.

Input can be cancelled by entering CTRL-J without altering variables.

End of file is treated as I/O error. Common practice for reading files is specifying +F option and examining $IOSW variable.

Channel number refers to device or file specified in OPEN statement. Its syntax is:

OPEN <Options> <Channel> <Specification>

Specification is a string constant or identifier of byte array holding zero-terminated string. It should contain either two-character name of built-in device driver or string with 'FI:' and ending with a valid file name. Valid device names are:
'TT' - console
'LP' - printer
'PR' - paper tape reader
'PP' - paper tape puncher
'NL' - null device (discards output, EOF for input)
'FI' - represents file system and needs file name after colon like ('FI:DX1:FILE.DAT').

Options for OPEN:

N - create new file (default is -)

After using channel should be closed with CLOSE statement. Its syntax is:

CLOSE <Channel>

See SHOW LUN command to see current assignments.

The following example prompts for file name, creates it and put its name in it without new line as there is -L option specified and byte counter is decreased to avoid EOL characters printed:

```
10 BYTE S(40) ! Buffer to hold file name
20 INT I ! String length will be put here
30 GET S:A:I ! Now S holds ASCIIZ string and I is its length
40 OPEN +N 1 S ! Create new file
50 PUT -L 1 S:A:.DEC.I ! Write the name in it
60 CLOSE 1
```

Monitor commands

LIST [<Specification>] [<LineNumber>[,<LineNumber>]]
Outputs formatted listing of source code for the specified line(s) to the specified device or file (see OPEN statement). Default is listing the whole source to console. Formatting means adding leading spaces in front of line numbers. So the listing can not be read back into environment. Use WRITE command to save the source code in readable form.

COMP [<Options>] [<Specification>]
Compiles source code into executable image in internal memory area. Specification is for listing output. Default is 'NL' (no listing). Listing will include error messages and locators ("^") under corresponding lines.
Options for COMP
L - list source code lines (default is +)

WRITE <Specification>
Saves source code to the specified device as text. The result can be read by READ or NEW command.

READ <Specification>
Reads source code from the specified device. Source should be text with lines starting with numbers and separated with CR and LF characters (CTRL-M and CTRL-J). The lines being read are added to source code, or replace lines with the same numbers like console input. Use NEW command to completely replace existing program.

NEW <Specification>
Deletes existing source and compiled code and clears list of variables and reads source from the specified location. Use NEW 'NL' to completely reset environment without any source.

RUN
Compiles the source code into executable form if not yet compiled and passes control to it. The monitor is brought back when:

- The program terminates with STOP statement or reaches the last line.
- The user enters CTRL-P character.
- Error condition is found (such as floating-point overflow or underflow or I/O error).
- Processor exception is trapped (invalid operation code or non-existing memory access).

If error or exception occurs monitor prints the reason for termination in form of:
[I/O] ERROR <Number> [AT LINE <Number> PC=<PC>]
or
TRAP TO <Number> [PC=<PC>]
or
^P PC=<PC>
Upon normal termination monitor prints
STOP AT LINE <Number> PC=<PC>
It seems that these messages are printed before the ON STOP handler is called.
Error messages include line number when error takes place in the running program. Line number is 0 for interactively executed statement. Line number is negative if it originates from runtime library. Line number is absent if monitor can not locate it. Original manual had explanation for error codes. You can experiment with compiler and your code to find out what they can mean.

SHOW [<Qualifier>[<Parameters>]]
Displays status information

When qualifier is omitted the version is printed.
Valid qualifiers are:
SYS - lists predefined variables and functions with type, size (both in elements and bytes) and address information.
MEM - list sizes and addresses of internal memory areas for runtime library, source code, compiled code, variable storage, free and unused areas.
LUN - lists I/O channel number assignments. Blank lines means unassigned channel.
CODE [<StartLine>[,<EndLine>]] - displays compiled codes for specified range of lines and compiled instruction addresses.

This is sample output of SHOW CODE command:

```
LINE=     10 SOURCE=  51520 OBJECT=  52234
   10 INT MTC:172522,MTCMA:172526,MTBRC:172524 ! TM11 Registers

LINE=     20 SOURCE=  51620 OBJECT=  52234
   20 BYTE BUF(1023) ! Our buffer is 1024 bytes

LINE=     30 SOURCE=  51700 OBJECT=  52234
   30 MTCMA=#BUF ! Current memory address register points to our buffer
    12737   131510   172526

LINE=     40 SOURCE=  52010 OBJECT=  52242
   40 MTBRC=-1024 ! Set byte count as negative
    12737   176000   172524

LINE=     50 SOURCE=  52070 OBJECT=  52250
   50 MTC=60005B ! Issue command: WRITE+GO unit 0
    12737    60005   172522

LINE=     60 SOURCE=  52152 OBJECT=  52256
   60 WHILE MTC%>0 \ ENDW ! Wait for completion (bit 7)
   113700   172522     5700     3002      167        2      771
```

And this is how corresponding memory area looks when disassembled by simulator:

```
52234:  MOV #131510,@#172526
52242:  MOV #176000,@#172524
52250:  MOV #60005,@#172522
52256:  MOVB @#172522,R0
52262:  TST R0
52264:  BGT 52272
52266:  JMP 52274
52272:  BR 52256
```

EDIT <StartLine>[,<EndLine>]
Invokes simple editor for each line in the specified range. Lines are edited in sequence. Current line is printed. The user enters editing commands for current line in form:
<Delimiter><SearchForText><Delimiter><ReplaceWithText>
The result of replacement is printed. Entering empty command moves to the next line.

SET <Parameter> <Value>
Sets various environmental options. Valid parameters are (only one is known):
TERM - set console terminal type (0 - backspace not supported, 2 - backspace supported).
SET TERM 0 forces printing deleted characters surrounded by "\" character during editing.
SET TERM 2 forces normal editing with backspace.

DEL <LineNumber>[,<LineNumber>]
Deletes source lines within specified range.

BYE
Terminates session.