RPBASIC-52 PROGRAMMING GUIDE

COPYRIGHT

PRODUCT SUPPORT

If you have a question about RPBASIC-52 and cannot find the answer in this manual, call us at the number listed below during normal business hours.

When you call, please have the following at hand:

Your RPBASIC-52 programming guide
Your card hardware manual
A description of the problem

RPBASIC-52 PROGRAMMING GUIDE

TABLE OF CONTENTS

RPBASIC-52 PROGRAMMING GUIDE

RPBASIC-52 PROGRAMMING GUIDE

PREFACE

This programming guide is for Remote Processing controllers using RPBASIC-52 language. It was derived from Intel MCS-51 BASIC, V1.1. Several command extensions and features have been added to effectively speed up command execution.

● Buffered serial ports. Received characters are buffered to 256 characters. PRINT strings are put into a 256 character buffer, making it much faster.

● Multi-tasking constructs such as ON LINE, ON COM, ON COUNT, and ON KEYPAD. Lines and keypad are monitored at assembly language speed on every 5 ms tick time. This speeds up program execution because the main program no longer has to monitor these points.

● Software commands directly support hardware. DATE and TIME work with the real time clock. AIN reads a voltage while AOT outputs one.

Some cards do not have all hardware features so do not support all of the commands. Cards supported or exceptions are listed with each command. In some cases you must refer to your hardware manual for exact ranges.

A few original BASIC-52 commands have been removed. These commands were oriented around specific registers in the 8052 chip or a specific design. For example, the PROG command assumed code is stored in an EPROM. Remote Processing cards use a flash EPROM which uses a new programming algorithm. The PROG command was replaced with SAVE.

MANUAL CONVENTIONS

Information appearing on your screen is shown in a different type.

Example:
```
RPBASIC-52  V1.0
Copyright Remote Processing (1995)
Bytes free: 27434
```

Symbols and Terminology

<xxx>   Paired angle brackets are used to indicate a specific key on your keyboard. For example, <esc> means the escape key.

expr   Term meaning a number, simple variable, or mathematical expression involving variables and numbers. The following are valid *expr*:

> 45.3
> B
> CYCLE
> B*45
> C*D+54
> INT(D)

expr can be another function. Complexity of *expr* is limited by available stack memory. Usually this is 7 levels of parentheses.

For clarity, *expr* may be another name such as *position*, *channel*, and so on.

italic   Italicized variables require an expression or value. For example:

AIN(*channel*)
KEYPAD(*function*)

Ellipsis (...) follow an instruction which optionally accept more data.

DATA *data[,data][,data]...*
READ *variable[,variable]...*

Optional portions of an instruction are enclosed in brackets []:

DISPLAY *option[,option][,option]*

RPBASIC-52 PROGRAMMING GUIDE

Basic Interpreters

There are several types and levels of interpreters. A slow, very basic type of interpreter figures out what each command is supposed to do during run time. A token-based interpreter, such as this basic, is much faster. This type examines each program line as it is typed in, figures out what it should do, and converts it to a string of Basic tokens mixed with text. A token is a single character that represents a command. For example, an ASCII value of 89H represents the PRINT command.

After a line is processed, it is stored in memory. When you type the RUN command, each program line is scanned. A token causes a branch to an assembly language routine which carries out the required action.

ELEMENTS OF A BASIC PROGRAM

Commands

Commands direct or perform an output action. Examples are PRINT, SAVE, POKE, and LOAD. Commands do not return a value used for computation.

Functions

Functions return a value used for computation. Examples are AIN, PEEK, SIN, and COM$. Functions do not cause a change in an output.

Line Numbers

Program lines begin with a unique line number. Each line number may contain one or more Basic statements separated by a colon. Line numbers are in the range of 1 - 65535.

Operators

Operators act on or convert numeric or string data. These include arithmetic (+,-,*, and /), natural logarithmic (base "e"), trig (SIN, COS), relational (>, <, or <>), logical (.AND., .OR., .XOR.), and string (ASC, STR) functions. Special operators control the hardware-specific features of RPBASIC-52 such as interrupts, timers, counters, and direct read/write of I/O ports.

Tasking Statements

Tasking statements define a condition and execution location when a condition is met. Statements include ON COM$, ON LINE, ON COUNT, and ONITR. Programs executed as a result of these statements are treated as subroutines. The only difference between a tasking routine and one called by a GOSUB is the tasking can be called at any time.

Expressions

An expression is a combination of instructions, operators, data (constants, arrays or strings) and variables which, when evaluated by Basic, is equivalent to a single numerical value. Many Basic commands accept expressions as well as explicit data. Expressions which are used by commands and functions are also called arguments.

WRITING AND EDITING PROGRAMS

Program development takes place on your PC using your word processor or the RPC card. Programs from your PC are downloaded using a serial communication program.

Each program line can contain at most 79 characters. Program lines can be entered in any sequence. RPBASIC-52 properly orders line numbers.

Multiple statements on a single line are allowed when statements are separated by colon (:) and do not exceed a total of 79 characters per program line. Ending a program line with a colon may cause a program to hang.

There are two ways to write Basic programs. The first way is to directly type in the program to the card. All standard Remote Processing cards have a means of storing programs to a flash type EPROM. The second way is use a text editor and download the resulting file to the system. Just be sure to save files in DOS text format.

Downloading programs means transferring them from your PC (or MAC or terminal) to the card. Uploading means transferring them from the card back to the PC.

When uploading or downloading files, select ASCII text format. XMODEM, YMODEM, or other formats are not used. RPBASIC-52 does not know when you are typing in a program or if something

else (laptop or main frame) is sending it characters. The upload and download file does not contain any special codes; they are simply ASCII characters.

Uploading programs is simply a process of receiving an ASCII file. You or your program simply need to send "LIST" to receive the entire program.

Downloading a program requires transmitting an ASCII file. As you type in (or download) a line, RPBASIC-52 tokenizes that line. The time to do this depends upon its complexity and how many lines of code have been entered.

RPBASIC-52 must finish compiling a line before starting the next one. When a line is compiled, a ">" character is sent. This should be your terminal programs pacing character when downloading a program.

If your communications program cannot look for a pacing prompt, set it to delay transmission after each line is sent. A 100 ms delay is usually adequate, but your program may be long and complex and require more time. A result of short transmission time is missing or incomplete program lines.

A technique used to further program documentation and reduce code space is the use of comments (REM) in a downloaded file. For example, you could have the following in a file written on your editor:

    REM Check position

    REM Read output from the pot and
    REM calculate the position

    2200 a = ain(0) :REM Get position

The first 3 comments downloaded to the card are ignored. Similarly, the empty lines between comments are also ignored. Line 2200, with its comment, is a part of the program and could be listed. The major penalty by writing a program this way is increased download time.

Notice that you can write a program in lower case characters. RPBASIC-52 translates them to upper case.

Some programmers put "NEW" as the first line in the file. During debugging, it is common to insert "temporary" lines. This ensures that these lines are

gone. Downloading time is increased when the old program is still present.

If you like to write programs in separate modules, you can download them separately. Modules are assigned blocks of line numbers. Start up code might be from 1 to 999. Interrupt handling (keypad, serial ports) might be from lines 1000 to 1499. Display output might be from 1500 to 2500. The programmer must determine the number of lines required for each section.

RPBASIC-52 automatically formats a line for minimum code space. For example, you could download the following line of code:

    10 fora=0to5

When you listed this line, it would appear as:

    10 FOR A=0 TO 5

Spaces are displayed but not stored. The following line:

    10 for a =   0   to   5

would be compressed and displayed as in the second example above. Spaces are removed. However, spaces as part of a remark or PRINT are not removed.

RPBASIC-52 contains no line renumbering capability.

RPBASIC-52 contains a rudimentary line editor which allows editing a program line until a carriage return is sent. The rubout or backspace key can be used to delete characters working backwards form the current character. After a line is entered, it cannot be edited; you must enter an entire new line. Deleting an undesired line is done by typing the line number followed by a carriage return. RPBASIC-52 automatically deletes all such lines.

Uppercase/Lowercase

RPBASIC-52 is generally not case-sensitive. Program or command lines may be entered in lowercase or uppercase; however they are (with some exceptions) converted to uppercase. The case of text in remarks and strings is preserved.

RPBASIC-52 PROGRAMMING GUIDE

Variables and Constants

More than 25,000 unique variable or constant names may be defined. Names may be up to eight characters in length and must begin with a letter between A-Z (no numbers or special characters). The rest of the name may contain numbers or letters and include the underline character.

All numeric variables are floating point. Variables cannot be declared as integer or double precision. RPBASIC-52 supports eight digits plus sign and exponent. Extra digits are simply discarded. The range of valid values is ± 1E-127 to ±0.99999999E+127.

Names are identified by the first and last characters and its length. Identical length names with identical first and last characters are considered the same. PUMP_42 and PRIMER2 are considered the same. The way to correct this is to change the name length or first or last character.

Variable names longer than two characters require more time to process. Once a variable name is declared, it can only be erased by the CLEAR statement or by LOADing in a new program.

It is possible to have variable names longer than 8 characters. A problem is the name length is stored partly as a modulo 256 number. What it boils down to is a variable may or may not be recognized as unique. The Basic considers FEED_BIN_01 and FEED_BIN_11 as the same variable.

The original BASIC-52 had a bug where the variable name 'F' was erased if it was the last letter in a variable followed by a space. RPBASIC-52 corrected this.

Watch out for commands embedded in variable names. FORM_5 contains the command FOR. A BAD SYNTAX error is usually returned in these instances. The statement FORM_5=BOTTOM does not return an error but interprets it as

```
FOR M_5=BOT TO M
```

The key is to look at your statements as they are printed on the screen and make sure they are what you intended.

Valid variables names are:

```
CA5,  DA15_679,  PUMP_A, VALVE02,  A(10),
SIZE(5),  ABC_
```

Invalid variables, which may include embedded commands include:

```
4C,  C$0,  GOTOE,  FORM,  #XYZ,  _ABC
```

Constants are literal values. These are "known" values as opposed to variables which can be assigned any value, usually by a function. Constants may be numeric or string. To RPBASIC, there is no difference between the two.

Constants are expressed as integer, decimal, hexadecimal or exponential floating-point. The range of valid values are:

```
± 1E-127 to ± .99999999e+127
```

Using constants instead of a number speeds up execution by at least 5%. For example, use

```
10 CH = 5
20 A = AIN(CH)
```

instead of

```
20 A = AIN(5)
```

Variables and constants are expressed as follows:

```
A = 5               Integer format
A = 5.3             Decimal format
A = 0ACH            Hexadecimal format
A = 1.4E3           Exponential
```

RPBASIC-52 supports eight significant digits plus and exponent and truncates any extra digits. Hexadecimal constants with a leading alpha character must be preceded by a leading zero. If you fail to do this, RPBASIC-52 interprets them as variable names.

All hexadecimal constants are followed by a trailing "H" (0FFH for example). A "0" prefix is necessary when the first number is a letter (A-F).

Certain logical operators, such as .NOT., .AND., .XOR., and .OR., assume a 16-bit argument such as 0FFFFH. If you supply fewer than 16 bits, it returns a 16-bit value based on the assumption the unsupplied most significant bits are zero.

Subroutines

Use of subroutines tends to make programming more modular and easier to follow. The number of

subroutines is limited to the amount of internal stack space. Usually this is about 35 subroutines, but can go down if FOR-NEXT loops are active. This is sufficient to handle all multi-tasking (ON LINE, ON COUNT, ON KEYPAD, etc.) and several levels of subroutines.

Most complex programs tend to have a maximum of 7 nested subroutine levels. Usually the maximum is 4.

Passing Variables Between Programs

All variables in RPBASIC-52 are global. This means any routine can modify any variable at any time. When a new program is loaded using EXECUTE, variables are erased.

Values can be passed between programs using any variations of PEEK and POKE statements.

Addresses

Addresses are specified as either decimal or hexadecimal numbers. Hexadecimal addresses with a leading alpha character need a preceding zero otherwise they will be interpreted as variable names.

```
100    POKEB,1,1000H,15
110    A = PEEKB(1,1000H)
```

Memory addresses range from 0 to 0FFFFH and segments from 0 to 7. A segment represents a 64K block of memory. Programs and RPBASIC-52 variables reside in segment 0. Variables are generally stored in segment 1 and higher.
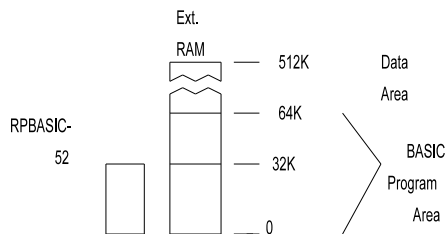
RPBASIC-52 Memory Map

Basic program area can be 32K or 64K, depending upon the amount of RAM installed.

Arrays

Arrays are single dimension and start with element 0. They are dimensioned using the DIM statement. Each variable may have up to 255 elements (0 to 254). Undimensioned arrays default to 11 elements, *variable*(0) through *variable*(10). Naming conventions used for scaler variables apply to arrays.

Strings

Memory is allocated to strings using the STRING command. There is no power up default. Up to 255 strings, identified as $(0) through $(254) are available.

To use strings, you must first determine the maximum length of any one string and then the maximum number of strings. Using the formula

(bytes/string + 1) * number of strings + 1

returns the number of bytes to allocate.

The ASC, CHR, and STR commands are used to evaluate and manipulate strings. Text assigned to a string is enclosed in double quotation marks:

```
100  STRING 1000,40
110  $(0)=">03"
```

OPERATING MODES

Command and Run Modes

RPBASIC-52 operates in two modes, Command and Run. Command mode is the direct, interactive mode accessed when RPBASIC-52 is not running a program. The Basic console prompt ">" indicates that Basic is ready for Command mode input.

Run mode is when the processor is actively executing a Basic program. Some commands (such as SAVE, LIST, LOAD) can only be executed when the processor is in command mode. Most Basic instructions can be executed in either Command or Run mode.

In Command mode, LOAD selects a Basic program from the flash. The RUN command then causes the selected program to execute. Within a Basic program, the EXECUTE instruction is used to allow the currently running program to call another stored program. A number of programs may be available to run depending upon the card and flash EPROM size installed. Refer to your hardware manual for more information.

Autorunning Programs

Programs may automatically load and run on powerup or reset when a specific jumper is removed on the card. Refer to your card's hardware manual for more information on jumper location.

When autorun is enabled, a LOAD 0, RUN sequence is performed on power up or reset. Programs are chained using the EXECUTE command.

Stopping Program Execution

<Ctrl-C> halts the execution of a program and forces the processor into Command mode (unless <Ctrl-C> has been disabled). Operation can be resumed by typing the CONT command. The STOP instruction stops a running program; execution resumes with a CONT command.

Sometimes it is desirable to not stop program execution. To disable <Ctrl-C>, execute:

```
DBY(38) = DBY(38) .OR. 1
```

X-ON and X-Off Flow Control

Serial output can be stopped with <Ctr-S> (X-OFF),
which halts output to the console serial port only; <Ctrl-Q> (X-ON) restarts it. You can use this feature to prevent screens of output data from scrolling by too quickly to read. After you type a <Ctrl-S>, Basic halts program execution if it is encountered during a PRINT command until it receives a <Ctrl-Q>. You can also reduce the serial port baud rate or use the NULL command to slow down the output of console data. Be careful of the NULL command. Some terminal programs print a space character instead.

Characters are buffered from the serial port. Therefore an additional 256 characters may continue to print after a <Ctrl-S> is sent.

WARNING: Program execution halts during a PRINT when an X-OFF is received until a X-ON is received. This means no other Basic commands are executed. Multi-tasking interrupts are recognized but not executed until after the PRINT statement is finished.

To determine if X-OFF is active (printing halted) before executing a PRINT statement, check address 38, bit 5. If high, X-OFF is active.

```
100 IF (DBY(38) .AND. 32) = 0 THEN 200
```

Normally the result of the above test is 0 (no X-OFF received) and the program branches to line 200. Of course, if X-OFF is received during a PRINT command, program execution is suspended until an X-ON is received. To clear X-OFF (due to a protocol you are using), put the following line in:

```
120 DBY(38) = DBY(38) .AND. 0DFH
```

STORING PROGRAMS

RPBASIC-52 programs are stored in non-volatile flash type EPROM. The SAVE command is used to write programs from RAM while LOAD retrieves them into RAM. Depending upon the card and the EPROM type installed, up to 8 programs can be saved and loaded. Refer to your card's hardware manual for specific programming information.

HARDWARE AND SOFTWARE INTERRUPTS

RPBASIC52 generates two kinds of interrupts: hardware and software. Hardware types are those generated by a voltage change and go directly to the processor. Software types require program execution and set memory flags that are read by some other program.

NOTE: Not all products support all or the same interrupts. Make sure the "Cards:" category in each command lists your card or refer to your hardware manual.

There are six interrupts in RPBASIC-52, version 1.11 and later. In the unlikely scenario that all interrupt conditions are met at exactly the same time, they would be serviced in the following order:

ONTICK	Periodic
ONITR	External line
ON COUNT	Counter
ON LINE	Line change
ON COM$	Serial input
ON KEYPAD	Keypad

Interrupt priority is based on hardware or software type. ONTICK and ONITR are considered hardware types. Should either one of these interrupts become active, ON COUNT, ON LINE, ON KEYPAD, and ON COM$ interrupts are not run until either one is finished. If an ONTICK interrupt is running, an ONITR interrupt is not serviced until ONTICK is complete. ONTICK and ONITR have the highest priority.

ON COUNT, ON LINE, and ON COM$ interrupts are serviced after ONTICK and ONITR are complete. Should any these last three interrupts occur simultaneously, ON COUNT would be executed first. However, if any of these three interrupts occur after one has started, then it would take priority.

Interrupts occur any time during program execution. The RPBASIC operating system sets appropriate flags indicating which kind of interrupt needs services. At the end of the current statement it checks these flags. The time interval between the actual interrupt and start of the interrupt routine is called latency.

Latency varies a great deal, depending upon the type of interrupt and command currently executed. A "typical" time in RPBASIC is less than 1 ms. However, it can be as short as several micro-seconds to several seconds. The reason it can take so long is due to the Basic subroutine. Suppose an ONTICK interrupt is in progress and it is written so it takes several seconds to complete. Since it is the highest priority, all other interrupts are locked out. The best way to correct this situation is to make all interrupt routines as short as possible. This is handled by setting a flag using a variable in the interrupt routine then exiting. Then at some other non-critical time, the interrupt is serviced.

WARNING: RPBASIC-52 offers a opportunity for all interrupts to occur simultaneously. It can handle all 21 interrupts simultaneously. However, it cannot handle them when they occur at a rate faster than they are serviced. Servicing all 21 interrupts requires a minimum of 21 ms. If interrupts consistently come in faster than they can be handled, the program will stop and a control stack error returned.

Whenever an ON COUNT or ON LINE multitasking command is enabled, overall program speed slows down. If all ON COUNT and ON LINE interrupts were enabled (but lines were not changing), program speed slows down by about 6%.

RPBASIC-52 PROGRAMMING GUIDE

MULTITASKING CONSTRUCTS

COUNT Multitasking

RPBASIC-52 on the RPC-3XX series of cards can count pulses while a program is running. Checking and counting is performed at assembly language speed during each system tick time (every 5 ms). This capability effectively speeds up program performance and simplifies programming.

This section describes only software counters on the card. Hardware counters are in a separate category are discussed in the hardware section of the card's manual.

Just about any valid digital I/O line can be designated as a counter input. Exceptions are interrupt inputs, keypad, and display lines. Even if a digital line is an output, it can be designated as a counter input. This is useful in situation where you may want to limit or keep track of the number of pulses to a motor, solenoid, or lamp.

Eight software counters are available. They are numbered 4-11. Counters 0-3 are reserved for any hardware ones that may or may not be on your board.

Counting is enabled as soon as a line is designated as a counter using ON COUNT. The digital line is sampled every 5 ms. When it goes from a high to low state, its counter is incremented. A line must be sampled at a high state before it can be counted again. A line must be at a high and low state for a minimum of 5 ms each to ensure detection. In theory the maximum counting rate is 100 Hz. However, due to other multitasking events (mainly serial ports), effective maximum rate is about 95 Hz assuming a perfect square wave.

There are two commands used in COUNT multitasking: COUNT and ON COUNT. Notice there are two COUNT commands. One is a function, which returns a value. This is the one used by the software counters. The other COUNT command is a statement, which writes a value to a hardware counter. This is not used by the software counters. Software counters cannot be preset.

ON COUNT declares or clears a multitasking process. There are three variations of this command. Referring to the ON COUNT command in this manual, the first syntax defines the digital line to count, number of pulses to count before executing a

subroutine. When the specified number of pulses is reached, the counter resets and a count interrupt flag is set. Should a higher priority interrupt be executing, the count subroutine is delayed until the higher one is finished. The COUNT function is not usually used in conjunction with this version.

The second syntax simply declares a line for counting. Use the COUNT function to return the number of pulses at the line. When the count reaches 65,535 it rolls over to 0. To reset or clear a count, simply re-declare the ON COUNT statement for that line.

The third syntax shuts off multitasking for that counter.

The ON COUNT command can be used to expand the number of lines used as an ON LINE command. The limitation here is an interrupt is generated only when a line goes low. Set the *count* to 1 in the ON COUNT declaration.

Serial Communication Multitasking

ON COM$ defines a program branch when either a specific character or number of characters is met. Criteria are specified in the ON COM$ statement. When the criteria is met, the incoming data is referred to as a packet.

This statement is especially useful in a networking application using the RS-485 serial port. Other devices, such as modems or scales can be used to generate an interrupt using RS-232. All serial ports can use ON COM$.

Data packets are retrieved using the COM$ function. In RS-485 networking applications, the STR(8,...) function is useful for determining its address. Two serial application programs are in this manual. The first program is a simple RS-485 network communication handler, shown in Appendix A. The second uses a modem to auto receive and is in Appendix B.

The RS-485 network handler is set up as a master-slave protocol. Slaves "do not talk unless spoken to". The host transmits to all receivers. All receivers transmitters go to the hosts receive line. The host does not transmit until it receives a response from a node or a timeout is reached.
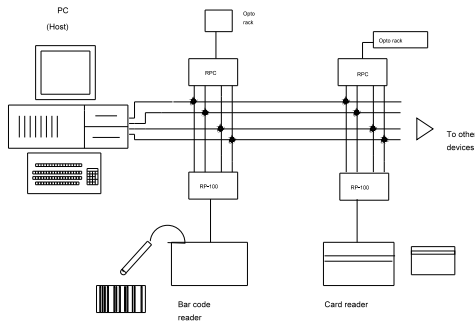
There are many communication protocols. For this example, the protocol looks something like this:

>03MB1

The protocol starts with the <cr> character. This character synchronizes all units and alerts them that the next few characters coming down are address and data. In this case, ">03" is the nodes address. Next follows a command (M). Depending upon the command, data may or may not follow. An optional checksum may follow. The figure below shows the elements in a data packet.
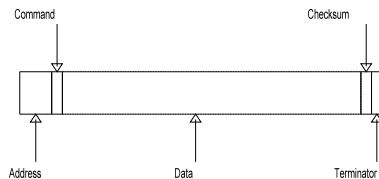
The response depends upon the nature of the command. Suppose the command M means "return door switch status". The card could read the port and respond with A1<cr>. The first letter A is an acknowledge. Data, 1, indicates a high.

Errors are returned with the letter N (negative acknowledge) followed by a number. The number identifies the general error type.

The program in Appendix A can be used on any of

the RPC-3xx series cards. Refer to this program for the following description.

The program starting at line 1000 is the network command handler. Line 1000 gets the data packet. Line 1010 determines if it is meant for this card.

Commands are sorted, or parsed out beginning at line 1020. For this example, commands are assumed to begin with the letter 'A'. By subtracting the ASCII value of A, we set up the ON GOTO structure to quickly handle each command type. This sample assumes 5 commands. If more are desired, another ON GOTO can be used. The start of the statement could read: ON OA-5 GOTO *linenumber,linenumber,linenumber...*

Command types can be broken into two groups: The first group performs an action such as setting a line, outputting to the display, or begin a complex timing process. The second group is a function, which returns data. This data can either be raw, such as a line status or voltage input, or processed. Processed data can be averages, converted values (feet/minute), operator input from a keypad, or a status report (such as OK) to determine if the board is there and functioning. The intent of these commands is to show how data is converted from string to number or number to string.

This example uses the following commands:

Command	Associated Data	Function
A	1 or 0	Set line 8
B	line, analog output 0 to 4095	Motor speed
C	0-1	Position from counter
D	String	Print to display
E	(none)	Power up acknowledge
F	(none)	General status

Command E is very useful to implement in situations where the host does not know if a unit reset (due to a power surge or something). The host may make certain assumptions about the status of a unit and continue to issue commands based on invalid assumptions. Lines that were set before may not be set.

This program is written so that no command is

processed unless the host "knows" this node has just reset. Any valid command, unless it is "E", returns a "N2" negative acknowledgement. The host recognizes this as a power up condition. Line 1220 checks for a valid power up flag.

Command F could return any number of status conditions. The way it is implemented here, data is returned to indicate the type of error. A 0 return indicates things are just fine. The type and value of data returned will depend upon the number of error conditions. If error conditions were binary weighted (1, 2, 4, 8...) then the receiver could determine exactly what errors are in the system.

A unique address in the message packet, >99, tells all units using this program to go to a 'safety' mode. It is used for emergency shut down situations. Nodes do not reply to this command. The program example does a simple return as your application determines appropriate response. The advantage to using this command is in emergency situations all units get the message in under 50 ms. It could take considerably longer, perhaps 1 second in a 20 node system, to poll all units.

A networking factor is communication time. Longer messages take longer to process. At 9600 baud, it takes about 12 ms to send out a 10 character message. This assumes the host can assemble a message string instantaneously. Add 5 ms processing time by the remote card (and 5 ms could be considered a minimum) before anything is sent out. It could be nearly 50 ms for a complete exchange. Using a simple command structure, about 20 message exchanges per second are possible.

Increasing the baud rate decreases message exchange time, but there is a point of diminishing return. Going to 19,200 baud cuts serial communication time in half. However, message processing time stays the same. At some point in time the processing power of the host and remote units is a major factor. RPC cards process commands roughly at a rate of 1/ms. To verify an address and begin carrying out a command takes about 30 ms. Any additional data processing increases this time.

The next application in Appendix B uses a modem in a receive application. This illustration uses a generic 1200 baud modem, although a higher speed modem can be used provided incoming data does not come in so fast the buffer fills and characters are lost.

COM1 is set as the receive port. The modem connects to the RPC card serial port using a VTC-9F serial cable. Most external modems have a DB-25F (female) connector for the serial port, therefore a DB-9F to DB-25M adapter is necessary. Also, since both the RPC card and modem are designed to plug into a PC, a null modem adapter must be inserted between the DB-9 connector on the VTC-9F and adapter. The connectors are shown below:

```
modem > DB9F to DB-25M > null modem > VTC-9F
```

This can be somewhat of a kludge. Another way is to make a custom cable from the RPC card 10 pin IDC connector to a DB-25M. If you choose this route, connect the pins in the following manner:

IDC	DB-25 male	Function on RPC card
3	2	Tx output
4	5	RTS input
5	3	RXD input
6	4	CTS output
9	7	Ground

Your modem may have configuration switches. Set these switches to the following conditions to use the sample program:

Force DTR lead (pin 20) true to enable modem to execute commands.

Modem responds to commands with english word result codes.

Result codes sent to the RPC card.

Echo characters while in the command state.

Modem automatically answers an incoming call.

Force CD lead (pin 8) true.

Enable modem command recognition.

You may have to set these conditions in software.

There is a certain sequence, or protocol, that is followed when answering a phone. The steps (CYCLE) follows:

CYCLE   Action

0	Wait for "RING" message. Modem auto answers.
1	Look for "CONNECT".
2	Get password. If invalid, prompt for password again.
3	Send successful login message. Prompt for command and process them.
4	Take modem off line and reset
5	Delay for a few seconds and send sign on message.

The actual program is more complicated than the steps indicate. Timeouts are used to disconnect the modem when there is inactivity. Three failed password attempts takes the modem off line. Ringing but no connect takes the modem off line. Superfluous <cr>'s are ignored.

Activity timeout is set for 10 seconds. The ONTICK routine checks for activity every second when the program cycle advances to step 1 and beyond. ONTICK could be faster if it is necessary. Keep in mind that ONTICK interrupts have the highest priority. Keep tick interrupt processing times short and as infrequent as possible. Frequent and/or long processing times take away from other program times.

ON COM $ interrupt uses a program cycle pointer (variable CYCLE) to direct the next activity on an interrupt. When a message is received, an interrupt is generated. Processing the message is handled by the appropriate routine pointed to by CYCLE. A <cr><lf> sequence is simply ignored and treated as a non-event.

After the password is accepted, the main purpose of the application takes over. There are many scenarios, or situations, possible:

1. The computer is used for data logging. Dialing in merely dumps data.
2. The computer is used for control. A dial up is for new instructions or parameters.
3. Some combination of data logging and control.
4. A computer will dial up and query and/or issue new instructions.
5. A person using a terminal will dial up the control card and query and/or issue new instructions.
6. A computer and/or person at a terminal will dial up and query and/or issue new

instructions.
7. A new program is downloaded to the card.

The number of possible applications is much too complex to even begin showing code.

Some applications use a person at a terminal to remotely query the card. In this situation, it is nice to return a character as soon as it is typed in. This can be done by setting the users terminal to local echo. However, you don't know if the card received what you send. The remote card can echo back characters as they are sent. To do this requires a program change. ON COM$ must either be disabled or changed to generate an interrupt on each character input. If ON COM$ is disabled, then the main program has to be structured so it can process incoming characters immediately.

If ON COM$ generates an interrupt on each character, then the incoming data rate should be relatively slow (1 character every 50 ms). Note this is not the baud rate. The baud rate can still be 9600. It just should not get characters more than 20 times/second. For hand typing situations, this is just fine.

When another computer is talking to the card, immediate echo may not be necessary. Instead, the incoming message can be echoed back when a <cr> is received (or when ON COM$ generates an interrupt). The cycles would merely increase based on the command. In some ways, it becomes like a RS-485 network described above. A command is received, parsed, and processed.

Scenario 7 requires some cautionary notes. It is not unusual to download programs through a modem. There is no difference between a modem down load and one directly connected to a PC. The UI and UO commands must be set to 1 when using COM 1 and before going into the command mode (executing an END or STOP statement in the program).

A problem arises when communication is lost for some reason. While the RPC cards have a watch dog timer, they are not enabled during command mode (This is true of the RPC-320 and RPC-330.) When communication is lost, usually all that is required is to redial the modem, assuming it has been set to auto answer. If communication is lost due to some external force (cell phone or network failure), the card will just sit there and not run. When the application is mission critical, an external

watchdog timer may be necessary to restart the card. Make sure call waiting is disabled.

ON LINE Multitasking

ON LINE is used to detect changes in a line. An interrupt is generated every time a line goes high or low. Use this command to detect changes in safety interlocks, level switches, or process command switches. Using this multitasking statement saves code and time because checking is done automatically in the background. A line must be high or low for a minimum of 5 ms to ensure detection to another state. Up to 8 lines can be monitored at one time.

This command is re-entrant, meaning when a routine is long enough and change interval short enough the interrupt is called twice. When there is this potential, the first part of your program should branch to routines that handle high and low line conditions. Use the LINE function to return the current status of a line.

ON COUNT can be used to expand the number of line changes. Simply specify a *count* of 1. An interrupt is generated when the line goes low.

Program execution slows down by up to 5% when all ON COUNT and ON LINE statements are enabled.

ON COUNT Multitasking

Up to 65,535 pulses can be counted on any one of eight lines. A line must be both low and high for a minimum of 5 ms to ensure counting. Maximum reliable counting rate is 95 Hz.

Counters specified in this statement are software counters only. It is not related to any hardware counters on the card.

A number of syntaxes allow simple counting to interrupt generation when a number of counts is reached.

The number of counters can be increased by using ON LINE. Counting rate must be very slow (less than 10 times/second) to effectively use this method. A counter increments when a line is low. Use the LINE function to read the status of a line.

Program execution slows down by up to 5% when all ON COUNT and ON LINE statements are enabled.

Assembly Language Interface

Assembly language programs must be placed in the RPBASIC-52 EPROM. When using the Basic, assembly language programs should start at address 6000H or higher, up to 7FFFH.

Normally a 32K EPROM is used to store RPBASIC. A 64K EPROM may be used provided a modification is performed. Refer to your hardware manual under *ASSEMBLY LANGUAGE INTERFACE* for information.

Documented assembly language interface calls listed in the Intel *MCS BASIC-52 Users Manual* will not work with RPBASIC-52. This is because RPBASIC-52 has be reassembled and code shifted around.

Assembly language development environment

An economical way to development assembly language programs and still keep RPBASIC-52 is to use an EPROM emulator. These are available from several sources.

Parallax Inc (916) 721-8271
JDR Micro Devices (800) 538-5000

Model types frequently changes, so it is best to contact these companies for the latest information. Generally, these cards connect to the parallel port on a PC. Downloading a program is generally under 1 second.

The way programs are developed would be to remove the RPBASIC-52 EPROM and read it by an EPROM programmer. Save the file.

Install the EPROM emulator into the card. Then, load in both the RPBASIC-52 binary file and your assembly language binary file using the software provided by the emulator.

Assembly language routines are accessed using the Basic CALL command.

Another development method is to use an In-Circuit-Emulator (ICE). Which type you use depends upon the processor type and your budget.

OPERATORS

Operator categories include:

Arithmetic	=, +, *, /, **, SQR
Relational	=, <>, <, >, <=, >=
Logical	.AND., .OR., .XOR., .NOT.
Value	ABS, INT, PI, RND, SGN

Operator Precedence

The precedence of operators determines the order in which mathematical operations are executed. Basic scans an expression from left to right and performs no operations until it encounters an operator of lower or equal precedence. For instance, multiplication takes precedence over addition. Parenthetical expressions have the highest precedence.

The following list is Basic's order of precedence:

1. Operators in parenthesis
2. Exponential operators (**)
3. Negation (-)
4. Multiplication (*) and division (/)
5. Addition (+) and Subtraction (-)
6. Relational expressions (=, <>, <=, <, >=, >)
7. .AND. (logical AND)
8. .OR. (logical OR)
9. .XOR. (logical XOR)

Parenthetical expressions have the highest precedence, so their use is a good way for you to reduce ambiguity and make your programs more readable. However, parenthetical expressions use internal data memory.

ARITHMETIC OPERATORS

Arithmetic operators perform basic arithmetic functions:

+	addition
-	subtraction, not negation
*	multiplication
/	division
**	exponential

OBSOLETE and MODIFIED COMMANDS

A number of commands in the original BASIC-52 have been replaced, obsolete, or no longer functional. The following is a list of obsolete commands and are no longer available:

CLEAR I
CLOCK0
CLOCK1
FPROG through FPROG6
IP
PORT1
PROG through PROG6
RAM
RCAP2
ROM
RROM
TIMER1
TIMER2
T2CON
XFER
XTAL

The following commands have been modified with respect to name and operation:

Old	New
ONEX1	ONITR
ONTIME	ONTICK
PGM	BSAVE
RROM	EXECUTE
TIME	TIME

Some commands have been added to or otherwise enhanced:

IDLE
INPUT
PWM

The following commands are new to BASIC-52. Note that not all commands/functions are available on all cards.

AIN
BLOAD
BSAVE
CARD$
CLEAR COM
CLEAR DISPLAY
CLEAR KEYPAD
CLEAR TICK
COM
COM$
COUNT
DATE
DISPLAY
EXECUTE
KEYPAD
LINE
LOAD
ON COM$
ON COUNT
ONITR
ON KEYPAD
ON LINE
ONTICK
PEEK
POKE
SAVE
SPROM
STR
TICK
TIME
WDOG
CONFIG

COMMAND GROUPS

The Command Reference is a detailed description of each RPBASIC-52 command, function, and instruction. Note that not all cards implement all commands. Also, this list is accurate as of the date of printing. Newer cards may not make it into this programming guide.

The following is a list of commands grouped by function.

Listing and control

LIST
LOAD
NEW
RUN
STOP
SAVE

Multitasking
ON COM$
ONITR
ONTICK
ON LINE
ON COUNT
ON KEYPAD

Program flow and looping
DO-WHILE
DO-UNTIL
END
FOR-TO-NEXT
GOSUB
GOTO
ON-GOSUB
ON-GOTO
REM
RETURN
RETI

Data storage and retrieval
BLOAD
BSAVE
CBY
DBY
DATA
DIM
POKE
PEEK
READ
RESTORE
XBY

Operators
/
-
+
*
**
<
<=
<>
=
>
>=
ABS
AND
ATN
COS
EXP
INT
LOG
NOT
OR
PI
RND
SGN
SIN
TAN
XOR

The trigonometric operators SIN, COS, and TAN use a Taylor series. Results are calculated to seven significant digits. The algorithm reduces the expression to a value between zero and PI/2 and results in a loss of precision if input_expr is large.

Relational operators (=, <>, <, etc.) return a result of 65535 if the relation is true and zero if it's false. The result may be displayed or used in further calculations. Beware when comparing calculated floating-point values as rounding errors may produce unexpected results.

Logical operators perform bitwise operations on expressions which evaluate to valid positive integers between OH and OFFFFFH (65535). All non-integer values are truncated to integers. Hexadecimal values with a leading alpha character must be preceded by a leading zero or Basic will interpret your constant as a variable name. If you supply fewer than 16 bits to NOT it will return a 16-bit value based on the assumption the unsupplied bits were zeros.

Serial input/output
CONFIG BAUD
COM
COM$
GET
INPUT
ON COM$
UI
UO

Printing and formatting
CR
PRINT, P., ?
PH
SPC
USING

Hardware input/output
AIN
AOT
CARD$
COUNT
DATE
DISPLAY
KEYPAD
LINE
ON COUNT
ONITR
ON KEYPAD
ON LINE
PWM
TIME

Real time controls
EXECUTE
IDLE
TICK
WDOG

String operation

ASC
CHR
STR
STRING

Strings in RPBASIC-52 are one-dimensional arrays of characters. Strings are stored as a sequence of ASCII values terminated with a 0DH (the ASCII value of a carriage return).

Memory for strings is allocated by the STRING operator. String variables are $(0) through $(254). Strings may be any length, limited only by available memory. However, if you wish to assign a string to explicit text in quotes, it may be up to

[72-{number of digits in string identifier}]

characters in length. In other words, $(9) may be 71 characters long, but $(200) may be only 69 characters long. This is due to the BASIC-52 program line length limit of 79 characters. Longer strings must be assigned one character at a time with the ASC operator or the XBY instruction. Explicit text assigned to a string must be enclosed in double quotation marks. The ASC and CHR operators can evaluate individual characters in a string.

Interrupts

ON COM$
ON COUNT
ONITR
ON KEYPAD
ON LINE
ONTICK
RETI

Other operators
IDLE

Memory Allocation
FREE
LEN
MTOP

ABS

Syntax:	ABS(*expr*)
	Where: *expr* = any number in Basic's range
Function:	Returns the absolute value of an expression
Mode:	Command, run
Use:	PRINT ABS(C)
Cards:	All

DESCRIPTION

The absolute value of a number is always positive or zero.

AIN

Syntax:	AIN(*channel*)
	Where: *channel* = 0 to 7, is channel to convert.
Function:	Converts analog input to digital number and returns a number from 0 to 4095 (0 to 1023 for the RPC-52)
Mode:	Command, Run
Use:	B = AIN(N)
Cards:	RPC-52, RPC-320, RPC-330. RPC-52 range is 0 to 1023 (10 bit).

DESCRIPTION

AIN returns a number corresponding to the input voltage. A number from 0 to 4095 (0 to 1023 for RPC-52) is returned. The result is returned in under 2 ms. Input voltage may be 0-5V or ±2.5 volts, single ended or differential. Inputs are configured for 0-5V, single ended input on power up. Use CONFIG AIN to configure each channel's characteristics.

The RPC-52 does not have differential inputs or use CONFIG AIN. Refer to the RPC-52 hardware manual for more information. The following explaination assumes a 12 bit result (0 to 4095) is returned.

A result is scaled to obtain a result representing a physical quantity. The general equation is:

variable = K * AIN(n)

where K is a scaling constant and n is the channel number. The scaling constant is determined as follows:

K = (maximum quantity - minimum quantity) / 4096

The physical quantity can be volts, current, pressure, inches, or whatever measurement you are taking. "maximum quantity" is the number with its output at 5 volts while "minimum quantity" is the number at 0 volts. Usually, the minimum quantity is 0.

Suppose you have a 0-200 PSI pressure transducer with a 0-5V output. To compute the constant for one PSI/count, divide the pressure over the resolution:

K = 200/4096
K = 0.04828 = PSI change per count

To measure 0-5 volts, K = 0.001220703

RELATED

CONFIG AIN

ERRORS

BAD ARGUMENT	When *channel expr* > 7 or negative
BAD SYNTAX	When *channel expr* left out

ASC

Syntax:	ASC(*ASCII character*)
	ASC(*string,position expr*)
	Where: *ASCII character* = number from 0 to 255
	string = any valid string variable
	position expr = 1 to length of string
Function:	Returns or sets the integer value of an *ASCII character* or the character in *string* at *position expr*.
Mode:	Command, run
Use:	PRINT ASC(C)
	ASC($(3),1)=48H
	C = ASC($(0),P)
Cards:	All

DESCRIPTION

The ASC operator either sets or returns the value of an ASCII character. Use ASC to evaluate, change or manipulate individual characters in a string.

The first syntax returns the value of an ASCII character. If *ASCII character* were the letter 'B', a 66 is returned. Basic converts any lower case variable symbols to upper case. Lower case characters must be put into a string to be evaluated.

The second syntax, shown under Use, sets a character in a string to a specific value. This is useful when you want to manipulate individual characters in a string.

The third syntax returns a value in *string* at *position expr*. This form is useful when you want to evaluate individual characters in a string, such as generating a checksum.

The STR command, unique to RPBASIC-52, manipulates entire strings.

RELATED

CHR, STR, STRING

ERROR

SYNTAX   Attempt to convert an improper value.

EXAMPLE

The following example prints ASCII values from the string $(0). The first 3 characters are modified at lines 70 to 90. The result is then printed.

```
10    STRING 200,20
20   $(0)="abc123"
30    FOR N=1 TO 6
40    PRINT ASC($(0),N),
50    NEXT
60    PRINT
70    FOR N=1 TO 3
80   ASC($(0),N)=65+N
90    NEXT
100  PRINT $(0)

READY
>RUN
 97   98   99   49   50   51
BCD123
```

ATN

Syntax: ATN(*expr*)

              Where: *expr* = value between 0 and PI/2

Function: Returns a trigonometric arc-tangent of *expr*. Returned result is between -PI/2 and PI/2 radians.

Mode: Command, run

Use: PRINT 4*ATN(1)

Cards: All

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and PI/2. The algorithm used to reduce the value will reduce accuracy when *expr* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERRORS

ARITH. UNDERFLOW   *expr* or result is less than RPBASIC-52's smallest floating-point value of ±1E-127

ARITH. OVERFLOW   *expr* or result is greater than RPBASIC-52's largest floating-point value of ±.9999999E+127

DIVIDE BY ZERO   Attempt to take TAN(X) when COS(PI/2) = 0

EXAMPLES

```
100    PRINT SIN(PI/2),COS(10001*PI),TAN(5*PI/4)
110    PRINT ATN(TAN(PI/4))/PI

>run

1  -1  1
.24999996
```

BLOAD

Syntax: BLOAD *to RAM segment, RAM address, from EPROM segment, EPROM address, length*
            Where: *to RAM segment* = 0 to 7, is the 64K block in RAM to write to
                     RAM address = 0 to 65,535, is the address to write to
                     from EPROM segment = 0 to 7, is the 64K block in EPROM to read from
                     EPROM address = 0 to 65,535, is the address in EPROM to read from
                     length = 0 to 65,535, is the number of bytes to move from EPROM to RAM
Function: Transfers a block of binary data from flash EPROM to RAM.
Mode: Command, RUN
Use: BLOAD 1,0,5,0,1000
Cards: RPC-320, RPC-330

DESCRIPTION

BLOAD transfers a block of binary information from EPROM to RAM. BLOAD does not check to see if there is enough RAM memory to save to or if the EPROM is large enough to perform the transfer. Data is retrieved from RAM using PEEK type functions.

segment can be though of as the X0000H address of the RAM or EPROM. When a *segment* of 1 and an *address* of 4300H are used, an address equivalent to 14300H is used to access the device. When a 128K RAM or EPROM is used, *segment* is 0 or 1. A 512K RAM or EPROM can have a *segment* of 0 to 7. A 32K device only has *segment* 0.

NOTE: Avoid using RAM *segment* 0. This is where RPBASIC program and variables are used. When segment 0 must be used, transfer data to above the MTOP address location.

Data transfer rate is about 23.5 ms/1000 bytes. During BLOAD time, ONTICK and ONITR interrupts are recognized but not serviced. If these commands must be serviced quicker, transfer data in smaller blocks.

BSAVE transfers data from RAM to flash EPROM.

RELATED

BSAVE, all PEEK commands

ERROR

BAD ARGUMENT   When any parameter above is out of limits.

EXAMPLE

The following example POKEs data into segment 1 of data RAM. The data is then saved to EPROM segment 6 and loaded back to a different location in RAM. The data is then verified. A 128K RAM and 512K flash EPROM must be installed for this example to work.

```
10      RA=512
20       FOR N=0 TO 1000 STEP 2
30       POKE W1,N,N
40       NEXT
50       FOR N=2000 TO 3000 STEP 2
60       POKE W1,N,0
70       NEXT
80       BSAVE6,RA,1,0,1000
90       BLOAD1,2000,6,RA,1002
100      FOR N=0 TO 1000 STEP 2
110     B=PEEKW(1,N+2000)
120      IF B<>N THEN PRINT "Error address",N," data is",B
130      NEXT
```

BSAVE

Syntax:	BSAVE *to ROM segment,ROM address,from RAM segment, RAM address, length*
	Where: *ROM segment* = 0 to 7, the 64K byte block to write to
	ROM address = 0 to 65535, address to write to
	RAM segment = 0 to 7, a 64K byte block to read from
	RAM address = 0 to 65535, address to read from
	length = 0 to 65535, number of bytes to write
Function:	Writes raw binary data to flash EPROM from RAM.
Mode:	Command, run
Use:	BSAVE 1, ROMTO, 1,RAMPTR, 512
Cards:	RPC-320, RPC-330

DESCRIPTION

BSAVE writes a block of binary information to EPROM from RAM. Use the POKE commands to write data to RAM.

WARNING: BSAVE should be used sparingly. The flash EPROM has a limited number of write cycles (1000) to each sector.

A *length* of 0 writes 65,536 bytes.

Limited parameter checking is performed. Basic assumes RAM exists at the segment and address specified. Basic checks to make sure the ROM segment specified is within limits of the installed EPROM, but addresses and lengths are not checked.

WARNING: BSAVE can write over programs saved using the SAVE command.

A *segment* can be though of as the X0000H address of the RAM or EPROM. When a *segment* of 1 and an *address* of 4300H are used, the address equivalent to 14300H is used to access the device. When a 128K RAM or EPROM is used, *segment* is 0 or 1. A 512K RAM or EPROM can have a *segment* of 0 to 7. A 32K device only has *segment* 0 and its address is limited to 32767 decimal, or 7FFFH.

A flash EPROM is written to in sectors. A sector is 64, 128, or 512 bytes for the 32K, 128K, or 512K EPROM respectively. RPBASIC automatically detects the type of EPROM installed when it writes to it.

You must pay attention to the sector size for two reasons. First, a sector is the minimum number of bytes written. If a program requires only 35 bytes to be saved, 512 bytes are written when a 512K EPROM is installed. If the following is performed

```
1000  BSAVE 6,5,1,1000H,35
 .
 .
2000  BSAVE 6,42,1,1025H,35
```

several things happen. The data saved by line 1000 is overwritten by the data in line 2000, even though different write addresses were specified. This brings us to the second reason sector size is considered. RPBASIC forces the requested EPROM address down to an even sector address. In both cases above, data is written to the EPROM starting at address 0, not at 5 or 42.

The solution to this situation is to write data out in even sector size blocks and to write them on sector boundaries.

A program is not required to write in full sector sizes. When less than 1 sector is specified, RPBASIC writes the next data in RAM until the full sector size is reached. When a large number of bytes are written, covering

many sectors, the last written sector is filled in with more data from RAM. Note that BLOAD allows data retrieval of any length and is not affected by sector size.

The easiest way to determine an even sector address is to "AND" the EPROM address with either FFC0H, FF80H, or FE00H for 32K, 128K or 512K EPROMs respectively.

Data can be saved "above" programs. The following is a way to determine the next free sector for writing to flash.

1) Save the program. Note the number of bytes saved.
2) Add the sector size (based on flash EPROM type) plus 64 bytes to the number of bytes saved. (64 bytes is for program overhead). For example, suppose the program is 28145 bytes long and a 512K (29C040) EPROM is installed. 28145 + 512 + 64 = 28721
3) At the terminal, print the following:
```
print 28721 .AND. 0FE00H
```
the response is
```
28672
```
What you have done is told the computer to print the length of the program + 512 bytes (for the sector) + 64 bytes (for program overhead) and 'and' it with FE00H. Notice the address, 28672, is higher than the number of bytes saved and less than the number we figured for sector size and overhead.

4) BSAVE can be used starting at this address (28672, or 7000H).

This method will work regardless of the number of programs saved or segment number.

Writing takes about 35 ms/1000 bytes. During BSAVE time, ONTICK and ONITR interrupts are recognized but not serviced. If these commands must be serviced quicker, write data in smaller blocks.

RELATED
BLOAD, POKE commands

ERRORS
BAD ARGUMENT   When any parameter is out of range or EPROM does not work properly.
HARDWARE   When verify to EPROM is bad

EXAMPLE
The following example POKEs data into segment 1 of data RAM. The data is then saved to EPROM segment 6 and loaded back to a different location in RAM. The data is then verified. A 128K RAM and 512K flash EPROM must be installed for this example to work.

```
10     RA=512
20      FOR N=0 TO 1000 STEP 2
30      POKE W1,N,N
40      NEXT
50      FOR N=2000 TO 3000 STEP 2
60      POKE W1,N,0
70      NEXT
80      BSAVE6,RA,1,0,1000
90      BLOAD1,2000,6,RA,1002
100     FOR N=0 TO 1000 STEP 2
110    B=PEEKW(1,N+2000)
120     IF B<>N THEN PRINT "Error address",N," data is",B
130     NEXT
```

CALL

Syntax: CALL *address*
              Where: *address* = address of assembly language program from 0 to 65535
Function: Calls an assembly language program in external Program Memory
Mode: Command, Run
Use: CALL 16
Cards: All

DESCRIPTION

CALL instruction invokes an assembly language program. To return to Basic, you must execute a RET instruction in the assembly language program. Original BASIC-52 code to multiply *address* by two and add 4100H was removed.

Expressions and variables are not allowed for *address*; it must be an explicit number. The assembly language program must reside in external program memory. RPBASIC-52 occupies internal program memory locations 0 through 6FFFH.

RELATED

EXAMPLE

CALL 0   Performs soft power up reset

CARD$

Syntax:	CARD$(*expr*)
	Where: *expr* = 0 to 3, is the card reader to scan.
Function:	Checks card reader for data. If present, returns the site code and card number. If no data is present, an error code described below is returned. All data is returned in a string format.
Mode:	Run
Use:	$(0) = CARD$(N)
Cards:	RPC-52, RPC-320, RPC-330

DESCRIPTION

CARD$ returns either the site code and card number or an error code. The site code and card number is returned in the following format:

"SSS-NNNNN"

Site codes and card numbers have leading 0's. The '-' character is used as a separator.

There are 4 different kinds of error returns possible. These errors are always in a 2 character "-X" format. 'X' is a number with the following meanings:

1   No card number present
2   Hardware error - both data bits down
3   Parity error
4   Timeout error - some data received

A "-1" return is the most common. It indicates no card was swiped. A "-3" error indicates the card was improperly swiped.

This command was designed to work with Sensor Engineering Co. (Phone 203 777 7444) model no 31503. Cards are in a 26 bit format: 2 check sum, 8 site code, 16 data.

Up to 4 card readers may be connected to the digital port at J3. Ports A and B are used to read and control the readers. Port C may be used for additional opto or digital I/O. Port A must be configured as an input and port B an output using the CONFIG LINE 100. . . statement. Port C may be input or output as required. The high current driver, U12, must also be installed. Each card reader is connected to digital port J3 as follows:

Card Number		J3 pins		
	Hold	D1	D0	LED
0	8	21	19	10
1	6	25	23	4
2	3	22	24	1
3	7	18	20	5

The green LED on the reader may be controlled using the LINE# command. A '0' forces the LED to green and a '1' forces it to red. A yellow LED indicates a card has been swiped and the reader is ready to send the information.

NOTE: This command takes approximately 27 ms to process. This is because the reader sends a bit of information every 1 ms. Serial and timing interrupts are processed at the hardware level. However, commands such as ONTICK and ONITR are delayed until CARD$ is finished processing the data.

RELATED
CONFIG LINE

ERROR
BAD ARGUMENT   When *expr* > 3 or negative

EXAMPLE
The following example reads the card. CONFIG LINE is performed only once. The error code is returned in B if no card was swiped.

```
CONFIG LINE 100,12,0,255,0,0

10 STRING 200,10
100 GOSUB 1000
110 IF B = 1 THEN 100
120 PRINT "Card number: ",$(0)
130 GOTO 100

1000 $(0) = CARD$(0)
1010 IF ASC($(0),1)= 45 THEN 1040:REM See if '-'
1020 B=0
1030 RETURN
1040 B = ASC($(0),2)-48:REM Return error number
1050 $(0)= ""
1060 RETURN
```

CBY

Syntax: CBY(*expr*)
            Where: *expr* = address from 0 to 65535
Function: Reads internal program code
Mode: Command, run
Use: PRINT CBY(1000H)
Cards: All

DESCRIPTION

The CBY instruction reads data from program memory space in the 8052. *expr* must evaluate to a valid integer address of 00H through 0FFFFH (65535). Code memory is read-only.

RELATED

DBY, XBY, PEEK, POKE

ERROR

BAD ARGUMENT   *expr* must be a valid integer (0 through 65535).

EXAMPLE

```
10   FOR N=0 TO 10
20   PRINT CBY(N),
30   NEXT

>RUN
97   203   255   210 22   50   2   39   110   255   255
```

CHR

Syntax: CHR(*expr*)

              CHR(*string,position*)

              Where: *expr* = number from 0 to 255

                    string = string variable

                    position = 1 to length of string

Function: Converts *expr* to ASCII character or prints *string* at *position*

Mode: Command, run

Use: PRINT CHR(65)

              PRINT CHR($(0),1)

Cards: All

DESCRIPTION

CHR is a dual use operator, similar to ASC. One version converts a numeric expression to an ASCII character, allowing a variety of string manipulation operations. The second version uses CHR to print individual characters in an ASCII string. *expr* is a decimal number and truncates numbers from 0 through 65535. There must be no space between CHR and the left parentheses or an ARRAY SIZE error results. Although *expr* can be any integer, printable ASCII characters range from 20H through 7EH (32 through 127).

The STR function may be used to manipulate and print longer portions of strings.

RELATED

ASC, STR, STRING

ERRORS

BAD ARGUMENT   *expr* can't be truncated to an integer (0 through 65535)

ARRAY SIZE   space between CHR and left parentheses

EXAMPLE

```
10      STRING 200,20
20    $(1)="1234567890"
30     FOR N=64 TO 80
40     PRINT CHR(N),
50     NEXT
60     PRINT
70     FOR N=1 TO 9
80     PRINT CHR($(1),N),
90     NEXT

RUN
@ABCDEFGHIJKLMNOP
1234567890
```

CLEAR
CLEAR S

Syntax: CLEAR
             CLEAR S
Function: Sets variables to zero, clears stacks
Mode: Command, run
Use: CLEAR
             CLEAR S
Cards: All

DESCRIPTION

The CLEAR instruction sets all variables to 0 and resets all Basic stacks. ONERR is cleared. Error trapping must be redeclared after a CLEAR. CLEAR is generally used to clear all variables. CLEAR does not de-allocate memory allocated to strings by the STRING instruction. It does clear the contents of the strings. Data put to the stack by PUSH is cleared. CLEAR also resets any FOR-NEXT loops. A C-STACK error is returned when a NEXT is performed after a CLEAR. CLEAR also resets any GOSUB return addresses.

Use CLEAR to perform a soft reset of a program. Keep in mind that multi-tasking routines are not cleared or reset using this command. However, if CLEAR is used as part of a multi-tasking program (ON COM$, ON LINE, etc.), a RETURN will cause a C-STACK error.

CLEAR S resets the control stack (C-STACK) only. This stack is used in loops and subroutines to tell it where to return to. Use this command to branch (GOTO) out of FOR-NEXT, GOSUB-RETURN, DO-UNTIL type structures. It can be used in emergency stop situations where nesting of loop structures is not known. Variables are not cleared using CLEAR S.

RELATED none

EXAMPLE

```
10    CLEAR TICK(0)
20    ONTICK 1,1000
25    ONERR 500
30    IF TICK(0)<2.5 THEN 30
40   A=TICK(0)/0
50    IF TICK(0) < 3.3 THEN 50
60    CLEAR
70    PRINT "CLEARED"
80    GOTO 80
500   PRINT "IN ERROR"
510   ONERR 500
520   GOTO 50
1000  PRINT TICK(0),A
1010 A=A+1
1020  RETI

>RUN
 1   0
 2   1
IN ERROR
 3   2
 4   0
 5   1
 6   2
```

The above example shows that ONTICK continues to run after a CLEAR statement but variables are cleared. If a program error were generated after the clear, the program would stop because ONERR was cleared.

The next example demonstrates how CLEAR S can be used in a FOR-NEXT loop. A C-STACK error is returned if the CLEAR S is not in line 20.

```
10   FOR N=0 TO 10
20   IF N=5 THEN CLEAR S : GOTO 10
30   PRINT N
40   NEXT

>RUN
 1
 2
 3
 4
 0
 1
```

CLEAR COM

Syntax: CLEAR COM(*port*)

              Where: *port* = 0 or 1, the serial communication port. *port* may be larger. Check your cards manual.

Function: Clears received characters in specified serial port buffer.

Mode: Run

Use: CLEAR COM(0)

Cards: All

DESCRIPTION

Received characters in the specified serial port are cleared. Characters in the transmit buffer are not affected.

RELATED

COM, COM$, GET

ERRORS

BAD SYNTAX   Any parameters left out

BAD ARGUMENT When *port* > 1 or card limit or negative

EXAMPLE

```
100     CLEAR COM(1)
```

CLEAR DISPLAY

Syntax: CLEAR DISPLAY   Clears character and, if available, graphics displays.
       CLEAR DISPLAY LINE Clears character line
       CLEAR DISPLAY LINE(x1,y1)-(x2,y2)   Clears graphics line
       CLEAR DISPLAY P(x,y)   Clears a point on a graphics screen
       CLEAR DISPLAY C   Clears characters only on graphics screen
       CLEAR DISPLAY G   Clears graphics only on graphics screen

Function: Clears display as directed by its options
Mode: Command, Run
Use: CLEAR DISPLAY   Clears entire display and homes cursor
Cards: All

DESCRIPTION

Character displays may use only CLEAR DISPLAY and CLEAR DISPLAY LINE.

Character displays require several milli-seconds to clear. After CLEAR DISPLAY statement, it is best to execute several other RPBASIC-52 commands before using the DISPLAY command again. This will allow the display to "catch up" to the program. Failure to do so may result in an incomplete screen clear or missing characters/data.

NOTE: CLEAR DISPLAY LINE requires several milli-seconds to execute. LCD displays require up to 10 ms while the VF display requires 20 ms. Processing other RPBASIC-52 interrupts are delayed by this amount of time.

The x and y graphic coordinates are the same as those specified in the DISPLAY LINE and DISPLAY P commands.

RELATED

DISPLAY

ERROR

BAD SYNTAX   When wrong option is used with a display.

CLEAR TICK
CLEAR KEYPAD

Syntax: CLEAR TICK(*timer*)
             Where: *timer* = 0 to 3
        CLEAR KEYPAD
Function: Resets specified tick timer or clears keypad buffer.
Mode: Command, Run
Use: CLEAR TICK(1)

DIFFERENCES FROM BASIC-52

The TICK function replaced TIME as a process clock. See TICK function for more information. KEYPAD has no equivalent function in BASIC-52.

DESCRIPTION

There are four independent tick timers that can be cleared independently of each other. This statement resets any one of the four tick timers to 0.

CLEAR KEYPAD clears the keypad buffer.

RELATED

TICK, KEYPAD

ERRORS

BAD SYNTAX   Any parameters left out
BAD ARGUMENT   When *timer* > 3 or negative

COM

Syntax:	COM(*port*)
	Where: *port* = 0 or 1, the serial communication port. *port* may be larger. Check your hardware manual.
Function:	Returns the number of characters received in the specified serial port buffer.
Mode:	Run
Use:	A = COM(0)
Cards:	All

DESCRIPTION

Use this function in conjunction with GET and COM$ to determine the number of characters to extract from the serial buffer. A GET 0 data value can be processed with the knowledge that it is a valid character and not an indication of an empty buffer.

RELATED

COM$, GET

ERRORS

BAD SYNTAX   Any parameters left out
BAD ARGUMENT   When *port* > 1 or card limit or negative

COM$

Syntax:	$(n) = COM$(*port*)
	Where: *port* = 0 or 1, the serial communication port. *port* may be larger. Check your hardware manual.
Function:	Return either all characters or up to a <CR> in specified serial port.
Mode:	Run
Use:	$(0) = COM$(0)
Cards:	All

DESCRIPTION

Characters in the specified communications port buffer are put into the string (on the left side of the =) until one of three conditions occur: 1) There are no more characters to extract. 2) A <CR> character is encountered. 3) The maximum number of characters specified in the STRING statement is reached.

This statement is useful when the application cannot risk using an INPUT statement. The INPUT statement waits until a <CR> is returned before continuing execution.

Unlike the INPUT statement, the value of all characters, except a <CR> (ASCII 0DH) are returned. All control characters and characters with ASCII values above 128 are returned.

NOTE: COM$ works only when it is assigning another string variable. A BAD SYNTAX error is returned when it is part of a PRINT, IF-THEN, ASC, or other command or function. Use this function only as shown in SYNTAX above.

RELATED

GET, INPUT, ON COM$

ERRORS

BAD SYNTAX	Any parameters left out
BAD ARGUMENT	When *port* > 1 or card limit or negative

EXAMPLE

The following example prints the number of characters in the buffer as they are entered. When 10 characters have been received, the string is printed.

```
10      STRING 100,20 : CLEAR COM(0)
15      PRINT "Enter characters."
20      A=COM(0)
30      B=COM(0)
40      IF A=B THEN 30
50      PRINT "Number of characters in buffer:",B, CR ,
55      A=B
60      IF B<10 THEN 30
70      $(0)=COM$(0)
75      PRINT
80      PRINT "Received string =",$(0)
100     PRINT "Characters left in buffer=",COM(0)
110     GOTO 20
```

When you enter a <CR> before the 10th character, the string to the <CR> is returned. Note that there are still some characters left in the buffer. When 10 characters are entered without a <CR>, characters are put into the string until the buffer is emptied or the maximum number of string characters set by STRING is reached. To see how this works, change line 60 to IF B<25 THEN 30. The number of characters left in the buffer will always be 5, unless a <CR> was entered.

CONT

Syntax: CONT
Function: Continue program execution after a STOP or Command-C
Mode: Command
Use: CONT
Cards: All

DESCRIPTION

CONT resumes program execution following a <Ctrl-C> or STOP instruction. You can display or modify variables while the program is stopped, but you cannot continue a program that is modified.

RELATED

STOP, GOTO, RUN

ERROR

CAN'T CONTINUE When program was modified.

COS

Syntax: COS(*expr*)

             Where: *expr* = numeric value up to ±200,000

Function: Returns the trigonometric cosine of *expr* which is in radians.

Mode: Command, run

Use: PRINT COS(PI)

Cards: All

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and PI/2. the algorithm used to reduce the value will reduce accuracy when *value* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERROR

ARITH. UNDERFLOW   *value* or result is less than RPBASIC-52's smallest floating-point value of ±1E-127

ARITH. OVERFLOW   *value* or result is greater than RPBASIC-52's largest floating-point value of ±.9999999E+127

DIVIDE BY ZERO   Attempt to take TAN(X) when COS(PI/2) = 0

EXAMPLES

```
10   PRINT SIN(PI/2),COS(10*PI),TAN(8*PI/4)
20   PRINT ATN(PI)

>run

 1  1  0
 1.2626272
```

CR

Syntax: PRINT CR,
Function: Used with PRINT. Sends a carriage return without a line feed.
Mode: Command, run
Use: PRINT CR,
Cards: All

DESCRIPTION

Used to update a line on a serial console device. A comma is necessary to prevent the usual line feed from terminating the PRINT instruction.

RELATED

PRINT

EXAMPLE

```
100    PRINT TICK(0),CR,
110    GOTO 10

>run
 3.242
```

The number is continuously printed at the same position.

COUNT (statement)

Syntax: COUNT *counter,data*
                Where: *counter* = 0 or 1
                        data = 0 to 16777215
Function: Writes *data* to specified up/down counter.
Mode: Command, Run
Use: COUNT 0,A
Cards: RPC-320, RPC-330

DESCRIPTION

Use this command to write 3 data bytes to the preset register (PR) in the LSI 7166 counter. This command does not transfer PR to the counter (CNTR). To do this, execute:

```
LINEB 6,X,8
```

Where: X = 1 for counter 0, 3 for counter 1.

NOTE: The sign of *data* is ignored. It can be a positive or negative number. When negative, *data* is simply treated as a positive number.

Decimal portion of *data* is ignored. For example, if *data* = 100.99999, 100 is loaded into the counter.

See your hardware manual for more information about using the LSI 7166 chip.

Software counters 4 - 11 cannot be set.

RELATED

COUNT (function)

ERROR

BAD ARGUMENT   When *counter* <> 0 or *data* out of range.

EXAMPLE

```
10   COUNT 0,124735
```

COUNT (function)

Syntax: A = COUNT(*counter*)
              Where: *counter* = 0 - 1, or 4 - 11
Function: Reads a multimode hardware or software counter
Mode: Command, Run
Use: A = COUNT(0)
Cards: RPC-320, RPC-330

DESCRIPTION

RPBASIC-52 recognizes a hardware and software counter. The hardware counter is 24 bits wide from a LSI 7166 chip. (Your board may use a different kind. Please check your hardware manual.) The RPC-320 has one of these and the RPC-330 has two. Additionally, there are 8 software counters on all cards.

counter 0 and 1 retrieve a 24 bit (3 byte) number from the LSI 7166 multimode counter IC. A number from 0 to 16777215 is returned. See your hardware manual for more information about using the LSI 7166 chip.

Eight software counters, set by the ON COUNT command, return a count from 0 to 65535. Software *counter* is 4 to 11. A software count is incremented when a line goes low.

RELATED

COUNT (statement), ON COUNT

ERROR

BAD ARGUMENT   When *counter* is out of range

EXAMPLE

The following example sets up line 3 as a software counter input. A count is printed once a second. A count is incremented by bringing line 3 low momentarily.

```
10 ON COUNT4,3
20 ONTICK 1,1000
30 IDLE
40 GOTO 30

1000 PRINT COUNT(4)
1010 RETI
```

ON COUNT can be configured to generate an interrupt when a specified number of counts is reached. See *COUNT MULTITASKING* under MULTITASKING CONSTRUCTS at the beginning of this manual.

DATA

Syntax: DATA *expr* [,*expr*,...]
                Where: *expr* = numeric data.
Function: It is an expression list used by READ.
Mode: Run
Use: DATA 23.4,17,3.2,PI*3
Cards: All

DESCRIPTION

Elements of a DATA statement are sequentially retrieved by the READ instruction. Multiple DATA expressions on a single program line must be separated by commas. There must be no spaces between *expr* and the commas.

See RESTORE for more information and examples.

RELATED

READ, RESTORE

DATE (function)

Syntax: A = DATE(*n*)

             Where: *n* = 0 to 3

                    0 = year (last two digits)

                    1 = month

                    2 = day

                    3 = day of week

Function: Returns the month, day, day of week, or year from the optional real time clock

Mode: Command, Run

Use: A=DATE(2)   Returns day of month

Cards: All. Note exceptions for RPC-52.

DESCRIPTION

A DS1216DM must be in the RAM socket. Consult your hardware manual for location. A numerical value of the month, day, or year is returned. The program under the TIME function is used to convert numerical date to a string. Substitute DATE for TIME in the program. STR function 10 also converts a number to a string.

A HARDWARE error is returned if the RTC is missing or bad. Use the ONERR construct to trap a defective DS1216DM. Hardware error code at address 101H is 50.

Day of week is returned only on cards which use a DS1216DM clock module. (This excludes the RPC-52.)

RELATED

DATE (command), TIME

ERRORS

BAD ARGUMENT When *n* out of range or negative

HARDWARE   RTC module missing or bad

EXAMPLE

```
100    PRINT "Time: ",
110    FOR N=0 TO 2
120    PRINT TIME(N),
130    NEXT
140    PRINT " Date: ",
150    FOR N=0 TO 3
160    PRINT DATE(N),
170    NEXT
180    PRINT CR,
190    GOTO 100

run

Time:  13   24   12  Date:  94  11  14   3
```

DATE (statement)

Syntax: DATE *year,month,day[,day of week]*
                Where: *year* = 0 to 99
                      month = 1 to 12
                      day = 1 to 31
                      day of week = 1 to 7
Function: Sets the date to the real time clock
Mode: Command, Run
Use: DATE 96,11,17   Sets date to November 17, 1996
Cards: All

DESCRIPTION

Leap year is automatically set. Tests for *day* check limits of 1 to 31. It does not check for a valid day in a month. You could set 2-31-96 as a valid date.

This command must be executed first to turn on the clock module. DATE and TIME functions or the TIME command will not work otherwise.

day of week can only be set on cards using a DS1216DM type clock module. (This excludes the RPC-52.)

RELATED

DATE (function), TIME

ERRORS

BAD ARGUMENT When *month, day, or year* is out of range.
HARDWARE   Clock module missing or bad.

DBY

Syntax: A=DBY(*expr*)

  DBY(*expr*)=*variable*

  Where: *expr* = 0 to 255

    variable = 0 to 255

Function: Read/write internal data memory.

Mode: Command, run

Use: DBY(0F0H) = 45H

  A=DBY(100)

Cards: All

DESCRIPTION

The DBY instruction retrieves or assigns a value to the 8052 internal data memory. *expr* and *variable* must both must be between 0 and 255 since there are only 256 internal memory locations and one byte can only be between 0 and 255.

RPBASIC-52 uses many internal memory locations for its own use. Change internal memory with caution or Basic may malfunction. Locations 1BH through 21H may be used in any way you wish.

RELATED

CBY, XBY

ERROR

BAD ARGUMENT Invalid *expr* value, such as DBY(256) or attempt to assign an invalid value to a DBY(*expr*), such as DBY(18H)=1000.

EXAMPLE

```
100   DBY(1EH) = 234
110   PRINT DBY(1EH)

>run

 234
```

DIM

Syntax: DIM *name*(*size*)[,*name*(*size*)...]
               Where: *name* = Any valid variable name
                      size = 1 to 255 elements
Function: Reserves storage for single-dimension array.
Mode: Command, run
Use: DIM FLOW(200) : REM dimensions a 200 element array called FLOW
Cards: All

DESCRIPTION

The maximum number of array elements is 255, accessed as *name*(0) through *name*(254). CLEAR, NEW, or RUN commands de-allocate all array storage. The default size of undeclared arrays is 10 (i.e. 11 elements). An array cannot be redimensioned after it has been dimensioned. Memory required for an array is ((integer size + 1) * 6). Array A(99) requires 600 bytes of memory. Available memory typically limits the size and number of dimensioned arrays.

RELATED

STRING, CLEAR

ERROR

ARRAY SIZE   When *size* >255

EXAMPLE

```
10    DIM FLOW(200), LEVEL(200)
20    ONTICK 1,1000
30    IF PTR < 200 THEN 30
40    ONTICK 0,1000
50    FOR N=0 TO 199
60    PRINT FLOW(N),LEVEL(N)
70    NEXT
80    END
1000 FLOW(PTR)=AIN(0)
1010 LEVEL(PTR)=AIN(1)
1020 PTR=PTR+1
1030  RETI
```

DISPLAY

Syntax: DISPLAY *option[,option][,option]*
            Where: *option* is one or more of the following
              "string"   Prints to display
              $(n)   Prints to display
              (row,col[,cursor])   Positions cursor and turns it on or off
              data   Puts *data* values to display
              CR   Prints a carriage return to the display
              LINE   Puts a line to a graphics display
              P   Puts a point to a graphics display
              ON [G,C]   Enables character, graphic, or both displays
              OFF [G,C]   Turns off character, graphic, or both displays

Function: Writes information to display.
Mode: Command, Run
Use: DISPLAY (1,2,OFF),28,"Name: ",$(0)
Cards: All

DESCRIPTION

DISPLAY has many options, some of which cannot be used with all displays. Graphics commands (LINE, P, C, and G) are only valid with the LCD-5003. An error is returned when they are used with character only displays.

Strings and cursor positioning may be placed in any order on the command line with the exception of *data*. The following example shows how some options can be combined in a program line.

```
100 DISPLAY (1,0,ON),"Batch no.: ",$(0),(2,0),"Enter process no.:"
```

The cursor is positioned at line 1, first position (0) and the cursor is turned on. The string "Batch no.: " is printed. The string in $(0) is then printed. The cursor is then re-positioned to line 2 (third line down), first position. The string "Enter process no.:" is then printed. The cursor is positioned just after the ':' character.

DISPLAY does not format text like PRINT. SPC, TAB, and USING commands return an error. Use STR function 10 to format numbers.

NOTE: Unlike the PRINT command and serial ports, DISPLAY does not buffer sending data to the display. Due to display speed limitations, it may take up to 1 ms to write 1 character or data point to a screen. Long strings or lines may take several milli-seconds. Time sensitive interrupts, such as ONTICK, can be "missed" if printing is long and the tick interval is very short. In these situations, it is best to break up any DISPLAY command into smaller sizes.

The following paragraphs explain each display option.

"*string*" is any quoted text used in PRINT statements.

DISPLAY "Hello world"

$(n) is any string array. Variable numbers must be printed from this array. The program in TIME function shows how to convert a number into a string.

DISPLAY $(0)

(*row,col,[,cursor*]) positions the cursor and, optionally, turns it ON or OFF. This option affects the character cursor position only. The row and collum always start at 0,0, which is the upper left corner of the screen. If *row* or *col* exceed the displays limits, a BAD ARGUMENT error is returned. The optional *cursor* is turned on or off using ON or OFF.

```
DISPLAY (1,5)
DISPLAY (2,0,OFF)
```

data is byte information written to the display. Functionally, it is equivalent to CHR$(n) found in other Basics. *data* can be used to control additional features of a display not normally available. For example, the vacuum florescent display brightness can be dimmed to minimum by executing DISPLAY 28.

NOTE: data does not update cursor position. The display may act 'unusual' when printing characters or strings. The best way to solve this problem is to position the cursor before resuming string displaying.

NOTE: data should not be used with the graphics display. Character values are offset by 20H. For example, the ASCII value for 'A' is 41H. The software subtracts 20H from this number before sending it to the display.

CR simply positions the cursor at the beginning of the current line.

```
DISPLAY CR
```

The following options are valid on the LCD5003 display only.

LINE draws a line on a graphics display. Its syntax is:

```
DISPLAY LINE (x1,y1)-(x2,y2)
Where:  x1,x2 = 0 to 159
     y1,y2 = 0 to 127
```

The LINE option is optimized for high speed. However, nearly vertical lines will take much longer to draw. A line is erased using the CLEAR DISPLAY LINE (x1,y1)-(x2,y2) command.

P puts a single point to a graphics display. Its syntax is:

```
DISPLAY P(x,y)
Where:  x = 0 to 159
     y = 0 to 127
     These values are valid for LCD5003 display only.
```

A line is erased using the CLEAR DISPLAY P(x,y) command.

ON enables character, graphics, or both displays. Three syntaxes possible are:

```
DISPLAY ON          Turns on both character and graphics displays.
DISPLAY ON G        Turns on graphic display only.
DISPLAY ON C        Turns on character display only.
```

Power on default is both graphics and character display ON. Turning on character or graphic does not affect the other. In other words, you could turn the character display ON and OFF without affecting the graphics display.

OFF disables character, graphics or both displays. Three syntaxes possible are:

DISPLAY OFF   Turns off both character and graphics displays.
DISPLAY OFF G Turns off graphic display only.
DISPLAY OFF C Turns off character display only.

Turning off the character display does not turn off graphics.

Using DISPLAY ON/OFF [option] allows you to switch between character and graphics displays. It is possible to update both graphics and character screens even if they are off.

RELATED
  CONFIG DISPLAY

ERROR
  BAD SYNTAX When *option* is invalid

DO-UNTIL

Syntax:	DO
	{program statements}
	UNTIL *relational expr*
	Where: *relational expr* is any logical evaluation such as =, <, >, etc.
Function:	Executes a number of program statements a relational expression is true.
Mode:	Run
Use:	100 A=0 : DO : A=A+1 : PRINT A : UNTIL A=4 : PRINT "Done"
Cards:	All

DESCRIPTION

This statement always executes at least once. DO-UNTIL loops may be nested. This loop may be exited without meeting *relational expr* by executing a CLEAR or CLEAR S statement.

This statement always executes to UNTIL once. When *relational expr* is evaluated and if it is false, program flow branches back to DO. If true, program resumes at the next statement after UNTIL.

When there are no {program statements} between DO and UNTIL, and {relational *expr*} is false, the "loop" will repeat forever, or until a <ctrl-c> is typed at the console.

DO-UNTIL and DO-WHILE loops can be nested.

RELATED

DO-WHILE, FOR-TO-NEXT-STEP

ERROR

BAD SYNTAX   When relational *expr* is omitted

EXAMPLE

The following program stays in a DO-UNTIL loop until a line has changed.

```
10    ON LINE 0,0,500
20    DO
30    UNTIL C=1
40    PRINT "Line 0 changed.  Is now a",line(0)
50    C=0
60    GOTO 20
500   C=1
510   RETURN

>run
Line 0 changed.  Is now a 0
Line 0 changed.  Is now a 1
Line 0 changed.  Is now a 0
```

DO-WHILE

Syntax:	DO
	{program statements}
	WHILE {relational *expr*}
Function:	Executes {program statements} while {relational *expr*} is true.
Mode:	Run
Use:	100 CLEAR TICK(0) : DO : PRINT TICK(0) : WHILE TICK(0)<10
Cards:	All

DESCRIPTION

The {program statements} between DO and WHILE are executed once, regardless of the {relational *expr*} result. At WHILE the {relational *expr*} is evaluated. If true, all {program statements} are executed again, and the test is repeated. If false, execution continues at the program statement after WHILE. DO-WHILE and DO-UNTIL loops can be nested.

RELATED

DO-UNTIL, FOR-TO-STEP-NEXT

EXAMPLE

The following program stays in a DO-UNTIL loop until a line has changed.

```
10      ON LINE 0,0,500
20      DO
30      WHILE C=0
40      PRINT "Line 0 changed.  Is now a",line(0)
50      C=0
60      GOTO 20
500     C=1
510     RETURN

>run
Line 0 changed.  Is now a 0
Line 0 changed.  Is now a 1
Line 0 changed.  Is now a 0
```

END

Syntax: END
Function: Terminates program execution and returns to command mode.
Mode: Run
Use: 65000 END
Cards: All

DESCRIPTION

The END instruction terminates Basic program execution. If no END instruction is used at the end of a program, the last instruction automatically terminates the program. Use END after the body of your program and prior to any subroutines.

Without an END after the main body of your Basic program and prior to any subroutine program lines, RPBASIC-52 will attempt to execute any subroutines at the end of your program as if they were a continuation of the main program. This will generate a C-STACK error whenever a RETURN is encountered.

RELATED

CONT, STOP, GOSUB, ON-GOSUB

ERROR

CAN'T CONTINUE The CONT instruction cannot follow an END instruction.

EXAMPLE

```
10      GOSUB 100
20      END
100     PRINT PI
110     RETURN

>run

 3.1415926
```

If you remove line 20, a C-Stack error is returned.

EXECUTE

Syntax: EXECUTE *[segment]*
            Where: *segment* = program to execute
Function: Loads and runs program specified by *segment.*
Mode: COMMAND, RUN
Use: EXECUTE n
Cards: RPC-320, RPC-330

DESCRIPTION

Command gets a program from the flash EPROM and executes it. *segment* species the program to execute. The program saved by the SAVE n command is executed. The range of *segment* depends upon the flash EPROM size. See the SAVE command for more information.

The effect of EXECUTE is the same as typing LOAD n, then RUN. The difference is EXECUTE is part of a program.

NOTE: Every time EXECUTE is run, all variables and strings are reset. Variables and strings *CANNOT* be passed from one program to another except through peeking and poking to RAM. ONTICK and ONITR interrupts are cleared as is ONERR.

String and numeric data can be saved for use by other programs using any of the POKE and PEEK statements. Data can be POKEd in to space above MTOP (7E00H in a 32K RAM system) or into memory segment 1 (128K RAM) or 1-7 (512K RAM).

Some parameters are not cleared by running EXECUTE. These are the tick timers (TICK), serial communication buffers, and data saved by POKEing. No hardware conditions are reset. No parameters set by any CONFIG statement are reset.

Loading and executing time depend upon program length. 0.22 seconds is required for clearing variables and resetting Basic. Add to this time the actual transfer time. Transfer time is at a rate of 50,000 bytes/second. A 20K program requires about 0.4 seconds to begin running after the EXECUTE statement is finished.

RELATED

LOAD, SAVE

ERROR

BAD ARGUMENT when *segment* is out of range.

EXAMPLE

The first lines were saved to program segment 0. The second set to 1.

```
10     PRINT "Program number 0"
20     EXECUTE 1

>save 0
10     PRINT "Program number 1"
20     EXECUTE 0

>save 1
>run
Program number 0
Program number 1
Program number 0
```

EXP

Syntax: EXP(*expr*)
Function: Raises "e" (2.71828) to the power of *expr*
Mode: Command, run
Use: PRINT EXP(COS(1))
Cards: All

DESCRIPTION

This function returns the result of the number *e* (2.718282) raised to the power given by *expr*. This function is very computation time intensive. Small values of *expr* take about 5 milli-seconds to calculate while larger ones (near 250) require nearly 0.2 seconds. Avoid using this function in tight control or time intensive applications.

ERROR

BAD ARGUMENT   When result of *expr* > 256

FOR-TO-STEP-NEXT

Syntax: FOR *variable=initial index expr* TO *index limit expr* [STEP *step expr*]
            program statements
            NEXT [*variable*]
            Where: *variable* = any valid variable symbol
                    initial index expr = starting value assigned to *variable*
                    index limit expr = ending value of *variable*
                    step expr = optional increment or decrement to *variable* when repeating a loop
Function: Looping structure useful for executing a sequence of instructions a number of times.
Mode: Run, command
Use: FOR A=0 to 4000 STEP 200 : AOT 0,A : NEXT
Cards: All

DESCRIPTION

The FOR-TO-STEP-NEXT instruction is a loop structure common to many high level languages. It is used to perform *program statements* a number of times.

variable is a loop counter initialized to *initial index expr* at the start of the loop. A number of *program statements* are executed until NEXT is encountered. At this point the value of *step expr* is added to the value of *variable*. The resulting new *variable* value is compared to the value of *index limit expr*. If the new value of *variable* value is less than or equal to the value of *index limit expr*, all *program statements* are executed again, and the test is repeated.

program statements are always executed at least once. If *step expr* is larger than *index limit expr*, the loop executes only once.

STEP is optional. When omitted, it defaults to 1. The value of *step expr* may be positive or negative.

FOR-NEXT loops may be inside other FOR-NEXT loops. *variable* following NEXT is optional.

There are two ways to break out of a for next loop and still maintain the control stack. The first is to execute a CLEAR S command. This command also clears any subroutine return locations and DO-WHILE, DO-UNTIL loops. Another is to set *variable* to a high value within *program statements*. When a program continuously breaks out of a FOR-NEXT loop and re-declares a new loop, a C-Stack error is eventually returned.

RELATED

DO-UNTIL, DO-WHILE

ERROR

C-STACK   NEXT without a corresponding FOR. This error can also appear if a number of FOR-NEXT loops were set up but were illegally branched out of or re-declared.

EXAMPLE

The following example gets characters from the receive buffer and generates a checksum. A string of 10 characters is entered at com port 0.

```
10     STRING 200,20
20      PRINT "Type in 10 characters.  Characters are not echoed"
30      IF COM(0) < 10 THEN 30
40     $(0) = com$(0)
50     CKSUM = 0
60      FOR N = 1 to STR(0,$(0))
70     CKSUM = CKSUM + ASC($(0),N)
80      NEXT
90      PRINT "Checksum of incoming string:",CKSUM

>run

Type in 10 characters.  Characters are not echoed
(1234567890 are entered at the keyboard)
Checksum of incoming string: 525
```

FREE

Syntax: FREE
Function: Returns the bytes of available in program RAM
Mode: Command, run
Use: PRINT FREE
Cards: All

DESCRIPTION

FREE returns how many bytes of RAM are available to the program and Basic variables. It does not return the amount of expanded RAM in 128K or 512K RAM systems. The amount of free memory is determined by the following formula:

FREE = MTOP - LEN - system memory

"system memory" on cards with two serial ports is 1791. Add 512 bytes for any additional serial ports on a card.

RELATED

LEN

ERROR

BAD SYNTAX Attempt to assign a value to FREE

FREQ (Function)

Syntax: FREQ(*channel*)
            Where : *channel* = 0 or 1, depending upon card.
Function: Returns a counter value
Mode: Command, run
Use: PRINT FREQ(0)
Cards: RPC-210, RPC-320, RPC-330 (RPC-210 and -320 are *channel* 0 only)

DESCRIPTION

This command returns a frequency, or number of pulses over a period of time. FREQ returns the latest value from the hardware counter. FREQ does not actually read from the counter. The operating system reads the counter at set intervals defined by CONFIG FREQ and stores them for retrieval by this function. This function is used to read analog input modules made by Greyhill, Dutec, and others. Equivalent 15+ bit analog input readings are theoretically possible.

FREQ function returns 0 until set up by CONFIG FREQ.

The latest FREQ value remains in memory until updated by the RPBASIC-52 operating system. The update interval is determined by the CONFIG FREQ command.

Avoid using COUNT(n) when using this command. It is possible values returned by COUNT(n) could be wrong.

Hardware counters (LSI 7166) are used to count pulses. CONFIG FREQ defines the time interval between readings. The operating system reads and resets the counters every time interval. Thus, you can measure a frequency in 1/10 second. The result is multiplied by 10 to obtain the "true" frequency. Errors in this case are also multiplied by 10. The best rule is to set the time interval in CONFIG FREQ for as long of a period as possible (up to 1.275 seconds) to get the most stable and accurate readings. Shorter intervals make counts appear less stable.

Best resolution is below 80 Khz at a measurement interval of 1/2 second. Between 80 Khz and about 190 Khz counts can easily vary by ±2. From 190 Khz to about 1 Mhz, counts vary by up to ±10. Above 1 Mhz, counts vary much more. Counting to 20 Mhz is possible.

You will have to play with the measurement interval, based on the input frequency and desired stability. Averaging the counts helps stabilize the readings.

There are several sources of errors and instability. The time interval between counter readings is based on the system tick timer, which is based on the crystal. Accuracy is usually better than 0.01%. The error is very noticeable at higher (> 200Khz) frequencies. Another potential source of error is the program or functions you may be executing. Some functions, such as AIN, turn off all interrupts for a "short" period of time (50 micro-seconds). What this means is, if it is time for the operating system to read the counters, the reading will be delayed by up to 50 micro-seconds. If the frequency is very high, additional counts are read. No counts are missed, so averaging readings helps to reduce errors.

Counts are missed when the frequency is above 1 Mhz (500 Khz on the RPC-210). This is because of CPU processing time between latching and reseting the counter (about 1 - 2 micro-seconds). Increasing the time interval between readings helps to reduce errors but does not eliminate them.

RELATED
CONFIG FREQ

ERROR

BAD DATA   When channel is out of range for a card.

EXAMPLE

The following example sets up frequency multitasking and prints the counts received in a time interval.

```
10 LINEB 6,1,32
20 LINEB 6,1,72 : REM Reset counter and enable inputs
30 CONFIG FREQ 0,200 : REM Get count every second (5 ms * 200)
40 C = 5/56000 : REM Constant using Greyhill module. 5V / 56000Hz = V/Hz
50 CLEAR TICK(0)
60 IF TICK(0) < 1 THEN 40  : REM Wait for a second
70 A = FREQ(0) : REM Get frequency
80 V = (A-14400) * C : REM Multiply by constant to get Voltage
90 PRINT "Voltage = ",V, "Frequency =",A
100 GOTO 50
```

You may need to add a 1K ohm pull up resistor from the output of the Greyhill module to the input of the counter. The input rise time should be 1 micro-second or faster.

GET

Syntax: A = GET
Function: Gets character from buffer.
Mode: Run
Use: A = GET
Cards: All

DESCRIPTION

GET is similar to INKEY$ in other Basic languages. GET returns the ASCII value of the character rather than the string. This feature makes it useful when receiving binary information.

To receive a control-C value (3), set bit 1, address 26H.

    DBY(38) = DBY(38) .OR. 1

This disables program breaks when a <Ctrl-C> is received.

GET can extract characters from COM 0 or COM 1. The UI 0 or UI 1 command is executed to get characters from an alternate serial port.

The ASCII value 0 is a valid number. Unfortunately, this value can indicate that there are no characters available. If your application program expects to receive ASCII 0's, the following program will wait until if there are characters in the buffer to ensure a value of 0 is indeed valid.

```
100    IF COM(0) = 0 THEN 100
110    A = GET
```

Line 100 loops until there is a character in the buffer. Line 110 extracts the character from the buffer. When A = 0, zero is the ASCII value.

RELATED

COM, COM$, INPUT, UI 1, UI 0

EXAMPLE

The following program takes characters one at a time from the buffer and puts them into expanded memory. 128K or more of RAM is needed.

```
100    IF COM(0) = 0 THEN 100
110    A=GET
120    POKEB 1,X,A
130    X=X+1
140    GOTO 100
```

GOSUB

Syntax: GOSUB *line number*

            ...
            line number program statements
            RETURN

Function: Transfers program control to the specified *line number*. The RETURN causes execution to resume at the program statement after GOSUB.

Mode: Run

Use: 100 FOR A=1 to 20 : GOSUB 200 : NEXT A : END
            200 PRINT A, SQR(A) : RETURN

Cards: All

DESCRIPTION

GOSUB provides subroutine capability within RPBASIC-52 programs. A subroutine may be called from within another subroutine.

GOSUB saves the location of the program statement after GOSUB on the C-Stack and immediately transfers program control to *line number*. When a RETURN is encountered, program execution resumes at program statement after GOSUB.

GOSUBs can be nested. The number nesting is limited by available C-Stack RAM, but is usually enough for at least 30 routines.

RELATED

GOTO, ON-GOTO, ON-GOSUB

ERROR

C-STACK   An unexpected RETURN is encountered or the number of subroutines executed was excessive.

EXAMPLE

```
10     GOSUB 100
20    PRINT "Back from routine"
30     END
100    PRINT "In subroutine"
110    RETURN

>run

In subroutine
Back from routine
```

GOTO

Syntax: GOTO *line number*
Function: Routes program execution to *line number*
Mode: Command, run
Use: GOTO 100
Cards: All

DESCRIPTION

When *line number* is the line number of an executable statement, that statement and those following are executed. GOTO can be used in the command mode to re-enter a program at a desired point.

RELATED

GOSUB, ON-GOTO, ON-GOSUB, RUN

ERROR

INVALID LINE NUMBER   Specified line number does not exist.

EXAMPLE

```
100    PRINT "At line 100"
200    GOTO 100
```

IDLE

Syntax:	IDLE [*option*]
	Where: *option* specifies a card dependent mode.
Function:	Suspends program execution and waits for an interrupt.
Mode:	RUN
Use:	IDLE
Cards:	All. Variations are card dependent.

DESCRIPTION

Different cards have a variety of parameters. Refer to your hardware manual for more information.

Use this command to suspend program execution and wait for an interrupt. An interrupt is from an ONTICK, ONITR, ON COUNT, ON COM$, ON LINE, or ON KEYPAD command.

RELATED none

ERRORS none

EXAMPLE

```
10      ONITR 0,1000
.
.                  Other initialization
.
200     IDLE               Wait for interrupt
.
.                  On exit from idle, continue program
.
1000    RETI              Simply exit
```

IF THEN ELSE

Syntax:	IF *expr* [THEN] *statement(s)* [ELSE *statement(s)*]
	Where: *expr* = any logical evaluation or variable
	statement(s) = any number of Basic statements
Function:	When *expr* is TRUE (not zero), the instruction following THEN is executed, otherwise the instruction following ELSE is executed.
Mode:	Run
Use:	10 IF A<>B THEN PRINT "A=B" ELSE PRINT "A<>B"
Cards:	All

DESCRIPTION

THEN is implied by IF. You may omit THEN. ELSE is optional. It is included when an "either - or" situation is encountered.

In the case of multiple statements per line following an IF-THEN-ELSE, Basic executes the following statements only if *expr* was true. This enables you to conditionally execute multiple statements with a single *expr* test. Remember this applies only to Basic statements separated by the {:} delimiter and on the same program line.

expr can be either a logical evaluation (=, <, >, <>, .AND., .OR., .XOR., or .NOT.) or a variable. Using a simple variable as a flag can speed up program execution. The following examples illustrate different execution speeds.

```
10   A = 1000
20   CLEAR TICK(0)
30   IF A<>0 THEN A=A-1 : GOTO 30
40   PRINT TICK(0)
```

The above program takes about 1 second to execute, which translates to about 1 ms/ line for this example. If line 30 were re-written as:

```
30   IF A THEN A=A-1 : GOTO 30
```

Execution time is reduced by about 20% by taking away the "<>0" evaluation.

RELATED none

ERRORS   none

EXAMPLE

```
10    A = 1
20    IF A=0 THEN PRINT "A is 0"  ELSE PRINT "A is non-zero"

>run

Is non-zero
```

INPUT

Syntax: INPUT ["*prompt text*"] [,] [,*variable*...]
            Where: *prompt text* = optional text
                     variable = list of variables to assign
Function: Program pauses to receive data entered from the console input.
Mode: Run
Use: 100 INPUT "Enter batch number",$(0)
Cards: All

DESCRIPTION

INPUT brings in numeric and string data from the console serial port during execution. Variables are string, numeric, or both. Variables are separated by a comma. Optional *prompt text* must be enclosed in quotes.

When an optional comma precedes the first *variable,* the question mark prompt character is suppressed and data entry is on the same line as *prompt text.*

Multiple numeric data may be entered by separating individual values with commas and using <cr> on the last one. Or, each data entry may be entered using a <cr>.

Strings must be entered with a carriage return.

If you do not enter enough data or the correct type, Basic sends the message TRY AGAIN and *prompt text* after which you must enter **all** the data. If you enter too many characters for the size of allocated STRING memory, or more numeric values than were requested, Basic discards the extra data, emits the message EXTRA IGNORED, and continues execution.

There are two major differences between RPBASIC-52 and BASIC-52 while using INPUT. Input characters are buffered. The operator or device may "type ahead" into the buffer and INPUT will respond just that much quicker. The back-space character (ASCII value 08) is recognized in the same way as the delete key was. This makes editing programs more convenient.

RELATED
COM$, GET, STRING

ERRORS   none

EXAMPLE

```
10      STRING 200,20
20      INPUT "Enter a number, string, and 2 more numbers: ",A,$(0),B,C
30      PRINT "String:",$(0)
40      PRINT "Numbers:",A,B,C

>run

Enter a number, string, and 2 more numbers: 4,Bob
?7,9
String:Bob
4   7   9
```

INT

Syntax: INT(*expr*)
Function: Returns an integer portion of *expr*
Mode: Command, run
Use: PRINT INT(PI)
Cards: All

DESCRIPTION

    The integer portion is stored as a floating point number.

RELATED none

ERRORS   none

EXAMPLE

```
print int(45.67)
 45

print int(-16.9999)
-16
```

    To produce true rounding to the closest whole number, use the following formula:

```
A = INT(B+0.5)
```

KEYPAD

Syntax: A = KEYPAD(*function*)

           Where: *function* = 0 or 1

                0 = return keypad position pressed from buffer

                1 = returns number of keys in buffer

Function: Returns keypad pressed position or number of keys in keypad buffer.

Mode: Command, Run

Use: A = KEYPAD(0) Returns a keypad position

Cards: All

DESCRIPTION

The keypad is automatically scanned, debounced, and placed in an 8 position buffer in the background. Key presses are buffered until retrieved by the KEYPAD(0) function. Keypad positions are returned as a number from 1 to 24. When a 0 is returned, there are no more keys in the buffer.

Position numbers 1 - 4 correspond to the top row while positions 12 - 16 are the bottom row of keys on the KP-1 and KP-3 keypads. Thus, the letter 'B' on the KP-1 corresponds to position 8.

Use CLEAR KEYPAD to remove all characters from the buffer.

ON KEYPAD branches to a subroutine when a key is pressed. (check card for availability)

RELATED

CLEAR KEYPAD, ON KEYPAD

ERROR

BAD DATA   When *function* is out of range.

EXAMPLE

The following program prints out the keypad position as a key is pressed.

```
10 CLEAR KEYPAD
20 DO
30 UNTIL KEYPAD(1) = 1
40 PRINT KEYPAD(0)
50 GOTO 20
```

LD@

Syntax: LD@ *expr*
            Where: *expr* = valid integer address of 00H through 0FFFFH (65535)
Function: Retrieves a floating-point number previously saved with ST@
Mode: Command, run
Use: LD@3000
Cards: All

DESCRIPTION

LD@ is used in conjunction with PUSH, POP, and ST@. Use these commands to save and retrieve floating point numbers to program RAM.

NOTE: LD@ and ST@ cannot use extended RAM. Only segment 0 RAM (used for running Basic programs) is used. Use PEEKF and POKEF commands to access this memory.

WARNING: When 128K and 512K RAM are installed, all of memory is cleared on power up and reset. Do not use LD@ or ST@ to save floating point numbers in segment 0. Use POKE and PEEK type commands instead.

32K RAM systems have address 7E00H set as MTOP. This location up to 7FFFH may be used to store variables.

expr is the address in RAM of where a number is stored.

Each floating-point number requires six bytes of memory. *expr* in the ST@ and LD@ instructions specify the high address. A number is stored at locations *expr* through *expr-6*.

RELATED

ST@, PUSH, POP, PEEKF POKEF

ERROR

BAD ARGUMENT when *expr* > 65535

EXAMPLE

```
100     A=AIN(0)*.234
110     PUSH A
120     ST@7F00H
    .
    .
300     LD@7F00H
310     POP B
320     PRINT "Analog value retrieved=",B

>run

Analog value retrieved=",B
```

LEN

Syntax: LEN
Function: Returns length of the current program in RAM
Mode: Command
Use: PRINT LEN
Cards: All

DESCRIPTION

The LEN function tells you the length of the program in RAM. LEN returns a value of 1 when no program is in RAM memory (1 is the length of the end-of-program marker).

RELATED

FREE

ERROR   BAD SYNTAX   Attempt to assign a value to LEN

LINE (Function)

Syntax: A = LINE(*line*)

     Where: *line* = 0-9 or 100 to 123 (Line ranges may vary. Check your hardware manual.)

Function: Returns status of a line at on-card lines L0-7 or interrupt port.

Mode: Command,Run

Use: A = LINE(2)   Reads line 2.

Cards: Basic function available on all cards. Ranges vary from card to card. See hardware manual.

DESCRIPTION

LINE returns a 0 or a 1. A '0' corresponds to a low while a '1' is a high. LINE returns the status of an external opto rack line or on card lines 0-7. *line* number corresponds to a position on an external opto rack. For on card lines, the range is 0 to 9. For an off card rack connected to the digital I/O port, it is numbered 100 to 123. 100 is simply added to the opto position number to specify a position.

When using LINE to return the status of an opto output line, a '0' means the module is ON while a '1' indicates it is OFF. This is in contrast to the LINE statement which turns on a module with a '1'. When reading an opto input module, a '0' indicates there is no voltage applied to the inputs.

LINE returns true logic for L0-L7. A "0" is a logic low while a "1" is a logic high. Line 8 returns the status of INT0 and/or ISOA/B input. Line 9 returns the status of INT 1.

LINE(n) and LINE#(n) may be used interchangeably in a program. For example, you may have an external 8 position opto rack and use some of the non opto digital lines for switch inputs.

RELATED

LINE#, LINEB functions, LINE, LINE#, LINEB statements, CONFIG LINE

ERRORS

BAD SYNTAX   When '(' or ')' are missing

BAD DATA   When *line* is out of range for a port.

EXAMPLE

The following example show how LINE and LINE# may be used

```
10 CONFIG LINE 100,12,0,0,1  Configure I/O port
20 PRINT LINE(104)            Read external opto rack position 4
30 PRINT LINE#(119)           Read digital I/O port line 19 (Port A.0)
40 LINE 100,1                 Turns on opto module at external rack position 0
50 LINE#110,1                 Turns on high current output at I/O port line 10.
```

LINE# (Function)

Syntax:	A = LINE#(*line*)
	Where: *line* = connector number from 101 to 125 (Line ranges may vary. Check your hardware manual.)
Function:	Returns status of a line at the digital I/O connector.
Mode:	Command,Run
Use:	A = LINE#(103) Reads level from digital I/O port connector number 3.
Card:	Function available on all cards. Ranges will vary from card to card. See hardware manual.

DESCRIPTION

The '#' modifier to LINE specifies the actual line number at the digital I/O port connector. *line* must range from 101 to 125 or else a BAD ARGUMENT is returned. Line 102 is also not valid. LINE# cannot be used for the on card opto rack (0 - 3). The line number is computed by simply adding 100 to the connector pin number.

LINE# returns a '0' or a '1', which correspond directly to the logic level at the connector. When using LINE# to return the status of an opto output line, a '0' means the module is ON while a '1' indicates it is OFF. This is in contrast to the LINE statement which turns on a module with a '1'. When reading an opto input module, a '0' indicates there is no voltage applied to the inputs.

The following example returns the status at digital I/O connector J3, pin 19 (82C55 port A, bit 0);

    A = LINE#(119)

See LINE function for more program examples.

RELATED

LINE, LINEB functions, LINE, LINE#, LINEB statements, CONFIG LINE

ERRORS

BAD SYNTAX	When # is used for on card positions.
BAD DATA	When *line* is out of range for a port.

LINEB (Function)

Syntax: A = LINEB(*i/o bank,address*)

          Where: *i/o bank* = 0 to 7. Specific functions are card dependent. Refer to your hardware manual.

              address = device dependent. Usually it is 0 to 3.

Function: Reads a byte from an I/O device.

Mode: Command, Run

Use: A = LINEB(3,0) Reads port A of 8255 at digital port.

Cards: All. *i/o bank* is unique to each card.

DESCRIPTION

This function is equivalent to INP in other BASICs. Data is read 8 bits at a time in contrast to other LINE functions which return 1 bit at a time. The *i/o bank* selects a particular I/O device listed in your hardware manual.

Use this command to read devices and obtain data not otherwise available using RPBASIC-52.

RELATED

LINE, LINE# (function), LINE, LINE#, LINEB (statement), CONFIG LINE

ERROR

BAD ARGUMENT   *i/o bank* > 7

EXAMPLE

The following example reads all 8 lines at port A on the digital I/O port.

    100 A = LINEB(3,0)

LINE (Statement)

Syntax:	LINE *line,data*
	Where: *line* = 0 to 8 or 100 to 123 (Line ranges may vary. Check your hardware manual.)
	data = 0, 1, ON, or OFF. See text below.
Function:	Turns a external opto module or lines L0-L8 on or off.
Mode:	Command, Run
Use:	LINE 0,1
Cards:	Basic statement available for all cards. *line* ranges are card dependent. See hardware manual.

DESCRIPTION

LINE is used to control an external output opto module or on card lines 0-8. On board opto positions are numbered 0-3. Off card opto racks using the digital I/O port are numbered 100 to 123. 100 is simply added to the opto position to identify the external rack. For example,

    LINE 105,0

turns external opto rack position number 5 off.

data is ON, OFF, 0, or 1. ON is equivalent to 1 while OFF is 0. A '0' value turns off a module while a '1' turns it on. These values are in contrast to the LINE# statement, which has the opposite meaning. For lines 0-7, "ON" sets a line to a 1 while "OFF" sets it to 0.

LINE 8,0 turns off the high current port. LINE 8,1 turns it on.

Using ON or OFF instead of numbers or variables speeds up this statement by 20%.

LINE and LINE # may be used interchangeably in a program.

RELATED

LINE, LINE#, LINEB (function), LINE#, LINEB (statement), CONFIG LINE

ERROR

BAD ARGUMENT   When *line* is out of range

EXAMPLE

The following example shows how different data is returned.

```
10   LINE 118,OFF   Turns off external opto module 18.
20   LINE #118,0    Sets digital I/O connector line 18 to 0.
30   PRINT LINE(118),LINE#(118)

run

 1   0
```

The function LINE(118) returns a 1 because that is the necessary condition to turn off a module. Also notice that LINE(118) returns the status at opto port position 18 while LINE#(118) returns the condition at the digital I/O port connector pin 18.

Use the CONFIG LINE statement to configure lines as inputs and outputs. Refer to the Digital I/O lines section in the manual and CONFIG LINE statement for more information.

LINE# (Statement)

Syntax:	LINE# *line,data*
	Where: *line* = 101 to 125, is the digital I/O line connector number. (Line ranges may vary. Check your hardware manual.)
	data = ON, OFF, 0, or 1. See text below.
Function:	Sets a specified line at the digital I/O connector high or low.
Mode:	Command, Run
Use:	LINE #102,0
Card:	Basic command available for all cards. *line* ranges are card dependent. Refer to hardware manual.

DESCRIPTION

LINE # addresses the digital I/O connector pins. *line* must be between 101 and 125. Line 102 is not valid (it is the +5 V supply).

data is either ON, OFF, 0 or 1. ON is the same as a 1 while OFF is a 0. '0' sets the line low while a '1' sets it high. This is the opposite of the LINE command. Opto modules require a low, or '0' level to turn on. LINE inverts *data* while LINE # does not. Using ON and OFF speeds up statement execution by about 20%.

LINE and LINE # may be used interchangeably in a program.

RELATED

LINE, LINEB (function), LINE, LINEB (statement), CONFIG LINE

ERRORS

BAD ARGUMENT   When *line* is out of range
BAD SYNTAX   When # is used for on card opto rack

EXAMPLE

The following example shows how different data is returned.

```
10  LINE 118,OFF   Turns off external opto module 18.
20  LINE #118,0    Sets digital I/O connector line 18 to 0.
30  PRINT LINE(118),LINE#(118)

run

 1  0
```

The function LINE(118) returns a 1 because that is the condition to turn off a module.

LINEB (Statement)

Syntax:	LINEB *i/o bank,address,data*
	Where: *i/o bank* = 0 to 7. Specific functions are card dependent. Refer to your hardware manual.
	address = device dependent. Usually it is 0 to 3.
	data = 0 to 255, data to output.
Function:	Writes a byte to an I/O device.
Mode:	Command, Run
Use:	LINEB3,0,A Writes value in A to port A of 8255 at digital port.
Card:	Basic command available for all cards. Device/Function changes slightly for each card. Refer to the hardware manual.

DESCRIPTION

This statement is equivalent to OUT in other BASICs. Data is written 8 bits at a time. LINE and LINE # write 1 bit at a time. The *i/o bank* selects a particular I/O device listed in your hardware manual.

Use this command to access or program devices into modes not directly supported by RPBASIC-52.

RELATED

LINE, LINE#, LINEB (function), LINE, LINE# (statement), CONFIG LINE

ERROR

BAD ARGUMENT *i/o bank* > 7, *data* > 255 or negative

EXAMPLE

The following example writes the value in variable 'C' to port B on the digital I/O connector.

```
100  LINE B3,1,C
```

LIST

Syntax: LIST
                LIST *line number*
                LIST *line number - line number*
                Where: *line number* is a program line number
Function: Prints all or some of a program to the console.
Mode: Command
Use: LIST 10-100
Card: All

DESCRIPTION

The LIST command prints the program in RAM to the console device. LIST inserts spaces after the line number and before and after instructions. Program listings are terminated with a <Ctrl-C>.

LIST *line number* lists the program line number to the end of the program. LIST *line number-line number* lists the program from the first line number to the second line number.

RELATED

LIST#

LIST#

Syntax: LIST# *port*
            LIST# *port,line number*
            LIST# *port,line number-line number*
            Where: *port* = 0 or 1 or number of serial ports on your card.
                    line number = program line number
Function: Outputs the currently selected program to the serial printer port.
Mode: Command
Use: LIST#0
Cards: All. *port* limit is card dependent

DESCRIPTION

The LIST# command outputs all or some of the currently program in RAM to the specified serial port. *port* 0 is the console port.

LIST# inserts spaces after the line number and before and after instructions. LIST#*port, line number* lists the program from the *line number* to the end of the program. LIST#*port,line number - line number* lists the program from the first line number to the second line number. These line numbers must be separated by a dash(-).

RELATED

LIST

LOAD

Syntax: LOAD *[segment]*
            Where: *segment* = 0 to 7, see table below.
Function: Loads a program from EPROM
Mode: Command
Use: LOAD 1   Loads program from memory segment 1
Card: All. Maximum *segment* is card dependent. Refer to your cards hardware manual.

DESCRIPTION

Up to 8 different programs can be saved and loaded from flash EPROM. The maximum number depends upon the EPROM size. When no *segment* is specified, 0 is assumed.

Use LOAD to retrieve programs for editing. LOAD overwrites and replaces the previous program. You cannot merge programs. Programs are saved to flash EPROM using the SAVE n command.

For more information on *segment*s and EPROM sizes, see the SAVE command.

RELATED

SAVE, EXECUTE

ERROR

BAD ARGUMENT   *segment* > 7

LOG

Syntax: LOG (*expr*)

Function: Returns the natural logarithm (base "e") of *expr* which must evaluate to greater than zero. Calculated to seven significant digits.

Mode: Command, run

Use: PRINT LOG(COS(0))

Cards: All

ERRORS

ARITH. UNDERFLOW   *expr* or result is less than RPBASIC-52's smallest floating-point value of ± 1E-127

ARITH. OVERFLOW   *expr* or result is greater than RPBASIC-52's largest floating point value of ± .99999999E+127

BAD ARGUMENT   Attempt to take LOG() of zero

EXAMPLE

```
100    PRINT EXP(-200), LOG(1.383901E-87)

>run

 1.383901 E-87-200
```

MTOP

Syntax: MTOP
            MTOP = *last valid RAM address*
Function: Reads or assigns the top of external data memory which will be used by Basic for variable, string,
            and RAM program storage
Mode: Command, run
Use: MTOP=30000
            PRINT MTOP
Cards: All. Command is limiting on cards with 128K or more of RAM.

DESCRIPTIONS

The MTOP system control value is the maximum external data memory address which RPBASIC-52 will use
for RAM program space and variable and string storage. MTOP is not necessarily the top of available
external data memory. On cards with 32K of RAM, MTOP is automatically set to 7E00H on power up. On
cards with 128K or more of RAM, MTOP is set to 0FFFFH on power up.

RELATED

ST@, LD@

ERROR

MEMORY ALLOCATION   MTOP has been assigned a value greater than top of external data memory.

EXAMPLE

```
? MTOP
 65535
```

NEW

Syntax: NEW
Function   Erases current program in RAM. All variables and strings are cleared.
Mode: Command
Use: NEW
Cards: All

DESCRIPTION

The NEW command deletes the program currently in RAM, sets all variables equal to zero, and clears all strings and multi-tasking interrupts. NEW does not effect the real-time clock or string allocation.

RELATED

CLEAR

NULL

Syntax:	NULL *integer*
	Where: *integer* = 0 -255
Function:	Sets number of NULL characters output to user after a carriage return
Mode:	Command
Use:	NULL 100
Cards:	All

DESCRIPTION

The NULL command controls how many NULL characters (00H) are output following a carriage return. After a reset, NULL = 0. Because this is a command mode command, it cannot be used as part of a program. The NULL count is stored at external data memory location 15H. Change the value of NULL in a program using the DBY(21)=*expr* instruction, where *expr* is any value between 0 and 255. No error is returned if it is greater than 255.

NULL is generally needed only if you have a slow printer connected to the serial printer port. Note that NULL affects all serial ports.

Some terminal programs will advance the cursor when a null character is received. This may result in an strange looking display.

RELATED

LIST, PRINT

ERROR

BAD SYNTAX When *integer* is negative.

ON COM$

Syntax: ON COM$*port,length,terminator,program line*
            ON COM$*port*
            Where: *port* = the com port number 0 or 1
                    length = number of received characters for an interrupt
                    terminator = character to cause an interrupt
                    program line = executes subroutine when *length* or *terminator* is met.
Function: Branches to a subroutine when *length* or *terminator* criteria is met.
Mode: Run
Use: ON COM$0,5,13,1000   Executes subroutine at line 1000 when either 5 characters or a <CR> is
                                    received.
Cards: RPC-320, RPC-330

DESCRIPTION

ON COM$ is a multitasking statement. *length* and *terminator* parameters are checked on every received character in the background. If either parameter is met, the program branches to the *program line* designated.

The first syntax enables ON COM$ while the second one turns it off.

When *terminator* is 0, then character values are not checked. Only a *length* criteria will cause an interrupt.

Review **HARDWARE AND SOFTWARE INTERRUPTS** in the first part of this manual for interrupt handling and multitasking information. A far more extensive example is shown earlier in this manual under *Serial Multitasking*.

RELATED

COM$

ERROR

BAD ARGUMENT when *length* or *terminator* > 255.

EXAMPLE

The following example executes a program at line 1000 when either 5 characters or the <CR> character is received. The received string is transferred to $(0) minus the <CR> character.

```
10 STRING 200,20
20 ON COM$0,5,13,1000
100 IDLE
200 GOTO 100
1000 $(0)=COM$(0)
1010 PRINT "COM string:",$(0)
1020 RETURN
```

ON COUNT

Syntax: ON COUNT *number, line number, count, program line*

              ON COUNT *number, line number*

              ON COUNT *number*

              Where: *number* is 4 to 11. It represents a counter number.

                    line number is 0-7 or 100-123 and is the digital I/O line number.

                    count is 1 to 65535. It is the number of pulses needed for an interrupt.

                    program line is the subroutine to execute when *count* is reached.

Function: Enables count multi-tasking at a specific I/O line. Optionally generates a software interrupt when the specified number of counts at an I/O line is reached.

Mode: Run

Use: ON COUNT 10,7,200,5000   Executes a subroutine at line 5000 when 200 counts are reached at I/O line 7.

Cards: RPC-320, RPC-330. *line number* is card dependent. Refer to your hardware manual.

DESCRIPTION

This command enables software counting. This command is not related to any hardware counters on the card.

The three syntaxes control counting as follows: The first syntax with all parameters generates a softwIare interrupt when *count* is reached. The second syntax simply enables counting at the *line number*. The third syntax turns off count multi-tasking for that *number* only.

A pulse is counted on a high to low transition. A line must be high and low for a minimum of 5 ms to ensure detection. The RPBASIC-52 operating system scans the specified lines every 5 ms. Thus, maximum counting frequency is 100 Hz. In practice, maximum is 95 Hz using a perfect square wave.

The current number of pulses at a counter *number* is read using the COUNT function. To reset or zero a count value, re-execute ON COUNT again for that particular *number*.

number is from 4 to 11 to distinguish it from the other hardware counters on board.

Review **HARDWARE AND SOFTWARE INTERRUPTS** in the first part of this manual for interrupt handling and multitasking information. Read *COUNT MULTITASKING* earlier in this manual for a summary of operation.

RELATED

COUNT function

ERROR

BAD ARGUMENT when *number* is out of range.

EXAMPLE

The following example sets line 0 as a counter and branches to a subroutine when this line is brought low 10 times

```
10 ON COUNT 4,0,10,1000
20 IDLE
30 GOTO 20
   .
   .
   .
1000 PRINT "Counter 4 interrupt"
1010 RETURN
```

This example makes line 3 a counter only input. Its value is printed every second using COUNT function.

```
10 ON COUNT 10,3
20 ONTICK 1,1000
30 GOTO 30
    .
    .
    .
1000 PRINT COUNT(10)
1010 RETURN
```

ONERR

Syntax: ONERR *line number*
Function: Goes to *line number* on arithmetic error, bad argument, and hardware errors.
Mode: Run
Use: ONERR 1000
Cards: All

DESCRIPTION

The ONERR instruction traps arithmetic errors and hardware problems, transferring control to *line number*. ONERR can be used to handle errors generated due to bad user input from and INPUT instruction. ONERR is a GOTO, not a GOSUB. Consequently, there is no easy way to resume program execution. The control and argument stacks are cleared so all GOSUB's, FOR-NEXT loops, etc. are cleared.

Error codes are stored at external memory location 257 (101H) and are accessed using the XBY instruction.

Code	Error
0AH (10)	DIVIDE BY ZERO
14H (20)	ARITH OVERFLOW
1EH (30)	ARITH UNDERFLOW
28H (40)	BAD ARGUMENT
32H (50)	HARDWARE

EXAMPLE

```
100    ONERR 1000
110    A=1/0
1000   PRINT "Error code:",XBY(257)

>run
Error code: 10
```

ON GOSUB

Syntax: ON *expr* GOSUB *line0[,line1[,line2...]]*
              Where: *expr* = 0 to number of subroutines after GOSUB
                     *line*n = subroutine line number to execute
Function: Calls subroutine based on value of *expr*.
Mode: Run
Use: ON A GOSUB 100, 200, 500
Cards: All

DESCRIPTION

The ON-GOSUB instruction conditionally branches to one of several possible subroutines depending on the value of *expr*. *expr* must evaluate to greater than or equal to zero. If *expr* evaluates to zero, execution branches to *line0*. When *expr* evaluates to one, execution branches to *line1*, etc. If necessary, *expr* is truncated to an integer.

ON-GOSUB saves the location of the program statement after ON-GOSUB on the control stack and immediately transfers program control to the selected subroutine. The subroutine is then executed. When Basic encounters the RETURN instruction, program execution resumes at the program statement after ON-GOSUB. ON-GOSUB instructions can be nested.

One or more of *line*n may be the same, to execute the same subroutine with different *expr* values. At least one *line*n must be provided. *line*n can be in any order.

RELATED

ON GOTO, GOSUB, RETURN

ERRORS

BAD ARGUMENT   The value of *expr* is less than 0
BAD SYNTAX   The *expr* value is larger than the number of subroutine locations provided, or commas
               were omitted between {subr n line#} values, or no subroutine locations were given.
C-STACK   Attempted recursion caused control stack overflow

EXAMPLE

```
10    P=2
20   ON P GOSUB 1000, 3000, 2000
30   END
1000 PRINT "Line 1000"
1010 RETURN
2000 PRINT "Line 2000"
2010 RETURN
3000 PRINT "Line 3000:
3010 RETURN

>run
Line 3000
```

ON GOTO

Syntax: ON *expr* GOTO *line0[,line1[line2…]]*
Function: Branches to a program line based on *expr* value.
valuate to greater than or equal to zero; if *expr* evaluates to zero, execution branches to {0th line#}; if *expr*
evaluates to one, execution branches to {1st line#}, etc. Commas shown are required.
Mode: Run
Use: ON A/5 GOTO 100, 200, 500
Cards: All

DESCRIPTION

The ON-GOTO instruction conditionally branches to *line*n where 'n' is the value of *expr*. The *expr* must
evaluate to greater than or equal to zero. When *expr* evaluates to zero, execution branches to *line0*. When
expr evaluates to one, execution branches to *line1*, etc. If necessary, *expr* is truncated to an integer.

One or more of the program lines may be the same, to GOTO the same location with different *expr* values.
At least one program line must be provided. Program lines may occur in any order, for example, ON A
GOTO 500,700,600.

RELATED

GOTO, GOSUB, ON-GOSUB

ERRORS

BAD ARGUMENT   The value of *expr* is less than 0.
BAD SYNTAX   The *expr* value is greater than the number of {"nth" line#} numbers provided, or
               commas were omitted between {line#} values, or no line numbers were provided after
               the ON-GOTO.

EXAMPLE

```
10    P=2
20    ON P GOTO 1000,2000,3000
30    END
1000  PRINT "Line 1000"
1010  END
2000  PRINT "Line 2000"
2010  END
3000  PRINT "Line 3000"
3010  END

>run
Line 3000
```

ONITR

Syntax:	ONITR *number,line number*
	ONITR *number*
	ONITR *line number*
	ONITR
	Where: *number* = interrupt line. This is card dependent. Refer to your hardware manual.
	line number = Subroutine line number to go.
Function:	Branches to a service subroutine on an external or counter interrupt.
Mode:	Run
Use:	ONITR 0,5000 Executes a subroutine at line 5000 on hardware interrupt 0.
Cards:	Basic command available for all cards. *number* may or may not be used. Refer to your hardware manual for more information.

DESCRIPTION

ONITR provides a way to respond to hardware interrupts. It replaces ONEX1 in BASIC-52. Interrupts can be external through the opto isolator, external TTL, or any number of card dependent sources. The number of interrupts available depend upon the card type. Refer to your hardware manual for specific information.

The first two syntaxes are for the RPC-330. The second two are for the RPC-320 and RPC-52.

Hardware interrupts are edge sensitive and latched. When the current RPBASIC program statement is completed, execution branches to the subroutine specified by *line number*. Interrupt latency is determined by the current program statement. The IDLE command provides the fastest response to an interrupt.

You must exit an ONITR using the RETI statement. Failure to do so prevents other ONITR and ONTICK interrupts.

To turn off ONITR, refer to the card's hardware manual.

ONITR can be interrupted only by an ONTICK interrupt. Also, ONITR can interrupt any other multi-tasking statement (ON LINE, ON COM$, ON KEYPAD, etc.) but cannot be interrupted by them. An interrupt pulse to the card must be at least 1 micro-second long, low level.

RELATED
RETI

ERRORS

EXAMPLE

The following example responds to an external interrupt on the RPC-330.

```
10     ONITR 1,1000    Declare interrupt
.
.                   Other program initialization
.
200    IDLE            Wait for interrupt
210    IF F = 0 THEN 200  If not done
.
.                   Program continues
.
990    END
1000   PRINT "In interrupt  Print something
1010 C=C+1             Increment counter
1020   IF C=5 THEN F=1   Set flag on 5 times
1030   RETI
```

ON KEYPAD

Syntax: ON KEYPAD *subroutine line*
            ON KEYPAD
            Where: *subroutine line* = program to execute
Function: Branches to a subroutine when a keypad switch is pressed.
Mode: RUN
Use: ON KEYPAD 1000
Cards: RPC-320, RPC-330

DESCRIPTION

Program branches when any key is pressed on the keypad. Use the routine below to build a string.

Review **HARDWARE AND SOFTWARE INTERRUPTS** in the first part of this manual for interrupt handling and multitasking information.

RELATED

KEYPAD, CLEAR KEYPAD

ERRORS

EXAMPLE

The following program sets up a string array and keypad multi-tasking. When the enter key is pressed, the string is printed. Keypad position 16 is designated as enter while 12 is clear.

```
10 STRING 200,20              Initialize string area
20 $(0) = "123A456B789C*0#D"  Initialize keypad string
30 P = 1                      String position pointer
40 ON KEYPAD 500              Declare interrupt
50 PRINT "Enter a number from the keypad",
REM Rest of program continues
REM Scan keypad flag
210 IF PF = 0 THEN 210        Check flag.  Prints string
220 PRINT                     when 'enter' is pressed.
230 PRINT "Entered string is: ",$(2)
240 PF = 0
250 GOTO 210
500 A = KEYPAD(0)             Get keypad character
520 IF A = 12 THEN 600 : REM Process clear   Add other traps as needed
530 IF A = 16 then 700 : REM process enter
540 A=ASC($(0),A)            Get ASCII equivalent
550 PRINT CHR(A),
560 ASC($(2),P) = A          Put into keypad input $
570 P = P + 1                Update position pointer
580 ASC($(2),P) = 13         Set CR as end of string
590 RETURN
600 REM Clear input string
610 $(2) = ""
620 P = 1
630 RETURN
700 REM 'Enter' processing
710 P = 1
720 PF = 1
730 RETURN
```

ON LINE

Syntax: ON LINE *number,I/O line,subroutine line*
             ON LINE *number*
             ON LINE *ON/OFF [,CLEAR]*
             Where: *number* = 0 to 7, is the interrupt reference number
                  I/O line = 0 to 7 or 100 to 123. Line number range is card dependent.
                  subroutine line = program subroutine to execute on line change
                  ON/OFF = enable / suspend ON LINE checking
                  CLEAR = clears all line change flags
Function: Branches to a service subroutine when an I/O line changes state.
Mode: RUN
Use: ON LINE 3,7,5000   Executes a subroutine at line 5000 when line 7 changes.
Cards: RPC-320, RPC-330

DESCRIPTION

Up to 8 digital I/O lines can be monitored for changes in state. Lines are monitored by the operating system every 5 ms. When a line changed from the last monitored state, a flag is set. This flag is checked at the end of the current Basic statement. Thus, an interrupt is generated when a line goes low or high. Unless an ONTICK or ONITR subroutine is currently executing, the *subroutine line* is then executed.

number is from 0 to 7. It acts, to some extent, as a priority arbitrator. It does not have any relationship to *I/O line* or *subroutine line* except to number interrupts. More information later.

An ON LINE interrupt is turned off by specifying *number* only. ON LINE interrupts can be turned off any time in a program.

I/O line numbers 100-123 correspond to opto rack positions. Use the table in the *DIGITAL I/O* chapter to make the correspondence between an opto position and actual digital I/O line. Lines 0-7 are designated L0-L7 on the card.

ON LINE ON/OFF enables/suspends line interrupts. Lines are still checked every 5 ms by the operating system. If a line did change, it is flagged. *ON LINE OFF* suspends interrupts while *ON LINE ON* resumes this type of interrupt. Use *ON LINE OFF* when an I/O line interrupt cannot be preempted by any other line interrupt. *ON LINE ON* resumes interrupts. When this command is executed, any changed lines cause an interrupt. To cancel or clear interrupts, use the *CLEAR* parameter shown above. All line change flags are reset and no interrupts are generated until a line changes state.

When two lines change between the 5 ms sampling time, the higher numbered interrupt takes priority. However, if the same or another line changes in the next sample period, its subroutine will take priority.

For an interrupt to occur, a line must be stable for at least 5 ms. When a line changes faster than this, one or both of the following scenarios happen: Since lines are sampled every 5 ms, a pulsed signal can be missed. Use one of the ONITR interrupts to capture this kind of signal. The second scenario is more of a problem.

ON LINE generates subroutines. When a line change is detected, a subroutine is generated. When the subroutine is long and a line change quick enough, these routines become nested. When too many routines are stacked, program execution is terminated and a control-stack error is returned. Maximum nesting level depends upon other control structures currently running. 30 levels is a reasonable number. However, if a number of FOR-NEXT loops are running, this number is diminished.

There are two ways to take care of this program. First, make the service routine very short - less than 3 commands. Second, is to execute the ON LINE OFF command. This shuts off all ON LINE execution.

The overall speed of RPBASIC-52 slows down by about 3% when all ON LINE tasks are enabled.

Review **HARDWARE AND SOFTWARE INTERRUPTS** in the first part of this manual for interrupt handling and multitasking information.

RELATED none

ERRORS
BAD ARGUMENT when *number* > 7 or *I/O line* is not between 0-7 or 100-123.

EXAMPLE
The following example sets up several interrupts.

```
10  ON LINE 0,1,1000
20  ON LINE 5,2,2000
30  ON LINE 3,3,3000
    .
    .
    .
1000 PRINT "In LINE 0 interrupt"
1100 RETURN

2000 PRINT "In LINE 5 interrupt."
2010 PRINT "Suspending other line interrupts."
2020 ON LINE OFF
    .
    .
    .
2300 PRINT "Resuming line interrupts."
2310 ON LINE ON , CLEAR
2320 RETURN

3000 PRINT "In LINE 3 interrupt."
3010 RETURN
```

Lines 10-30 set up ON LINE interrupts for lines 1, 2, and 3. For this example, line 5 cannot be interrupted by any other line changes. Line 2020 suspends interrupts. The program continues to process this subroutine and lines are still checked for changes. Line 2310 resumes line interrupts but it also clears out previous changes.

ONTICK

Syntax:	ONTICK *time,line number*
	Where: *time* = time interval from 0.01 to 327
	line number = line to branch
Function:	Calls subroutine at *line number* every *time* interval.
Mode:	Run
Use:	ONTICK 1.25,500
Cards:	All

DESCRIPTION

ONTICK calls a subroutine every *time* interval. *time* ranges from 0.010 seconds to 327.7 seconds (approximately 5.5 minutes). *time* can be specified in increments as small as 0.005 seconds. ONTICK interrupts are turned off when *time* = 0. A line number must still be provided even though it is not used.

The interval period can be reset at any time in a program. When an ONTICK statement is executed, an interrupt will occur in *time* seconds. Time accumulated since the last interrupt is discarded.

NOTE: Use the RETI command to exit this subroutine. Failure to do so prevents future ONTICK interrupts.

Make sure your ONTICK subroutine can finish before the next interrupt. If the program is in the subroutine longer than the specified time interval, the next one will be missed.

This interrupt has the highest priority of any others. ONITR can interrupt any other routine, but no other interrupt can take over this one.

RELATED

RETI

ERRORS

BAD ARGUMENT	When *time* > 327.6 or negative
BAD SYNTAX	When any parameters left out
INVALID LINE	When *line number* not found

EXAMPLE

The following example will interrupt 5 times before it is canceled at line 220.

```
10 A = .15
20 ONTICK A,200
30 IF C<4 THEN A=A+1 : GOTO 30
40 END
200 PRINT A
210 C = C + 1
220 IF C = 5 THEN ONTICK 0,200
230 RETI

>run
 145.15
 286.15
 431.15
 575.15
```

The IDLE command may be used to "wait" for an ONTICK interval interrupt.

PEEKB

Syntax: PEEKB(*segment,address*)

              Where: *segment* = 0 to 7, specifies a 64K segment

                    address = 0 to 65535, byte address in a segment

Function: Reads a byte from RAM

Mode: Command, Run

Use: A = PEEKB(1,AD)

Cards: All

DESCRIPTION

This function is used in conjunction with POKEB. Data is retrieved from any memory location. PEEKB inputs 1 byte of data. This function operates in much the same way as XBY does except PEEKB can access 512K of RAM.

See POKEB command for addressing and segment info.

RELATED

POKEB

ERRORS

BAD SYNTAX   If B, *segment*, or *address* is missing.

BAD DATA   If *segment* is > 7, or *address* > 65535

EXAMPLE

The following example reads digital I/O port A and saves it to RAM. The values are then retrieved and printed back.

```
10    FOR N=0 TO 500
20    POKE B1,N*2,LINEB(3,0)
30    NEXT
40    FOR N=0 TO 500
50   A=PEEKB(1,N*2)
60    PRINT A,
70    NEXT
```

PEEKF

Syntax:	PEEKF(*segment,address*)
	Where: *segment* = 0 to 7, specifies a 64K segment
	address = 0 to 65535, byte address in a segment
Function:	Reads a floating point number from RAM. Floating point range is +/- 1E-127 to +/-0.99999999E+127
Mode:	Command, Run
Use:	A = PEEK F(1,AD)
Cards:	All

DESCRIPTION

This function is used in conjunction with POKE F. Data is retrieved from any memory location. PEEK F retrieves a floating point number saved by POKE F.

PEEKF can access up to 512K of ram by selecting a segment and an address. A segment selects a 64K block while the address selects a location within this block.

Each floating point number requires 6 bytes. *address* must be incremented indexed 6 bytes for each value. See POKEB and POKEF commands for addressing and segment info.

RELATED

POKEF

ERRORS

BAD SYNTAX	If B, *segment*, or *address* is missing.
BAD DATA	If *segment* is > 7, or *address* > 65535

EXAMPLE

The following example reads the A-D port, multiplies it by a constant, and saves it to RAM. The values are then retrieved and printed back.

```
10    FOR N=0 TO 500
20    A = AIN(1) * 0.2344
20    POKE F1,N*6,A
30    NEXT
40    FOR N=0 TO 500
50   A=PEEK F(1,N*6)
60    PRINT A,
70    NEXT
```

PEEKW

Syntax:	PEEKW(*segment,address*)
	Where: *segment* = 0 to 7, specifies a 64K segment.
	address = 0 to 65535, word address in a segment.
Function:	Reads an unsigned 16 bit number from RAM
Mode:	Command, Run
Use:	A = PEEKW(0,AD)
Cards:	All

DESCRIPTION

Use this function in conjunction with POKEW. Data is retrieved from any memory location as a single 16 bit (2 byte) number. Numbers in the range of 0 to 65535 are retrieved. Two bytes of data are required for data retrieval.

PEEKW can access up to 512K of ram by selecting a segment and an address. A segment selects a 64K block while the address selects a location within this block.

See POKEB for addressing and segment information.

RELATED

POKEW

ERRORS

BAD SYNTAX	If W, *segment*, or *address* is missing.
BAD DATA	If *segment* is > 7, or *address* > 65535

EXAMPLE

This example takes 500 readings from analog input 0, saves it to segment 1 of a 128K RAM, and then prints out all of the values

```
10     FOR N=0 TO 500
20     POKE W1,N*2,AIN(0)
30     NEXT
40     FOR N=0 TO 500
50    A=PEEKW(1,N*2)
60     PRINT A,
70     NEXT
```

PEEK$

Syntax:	$(n) = PEEK$(*segment,address*)
	Where: *segment* = 0 to 7, specifies a 64K segment
	address = 0 to 65535, starting string address in a segment
Function:	Retrieves a string from RAM.
Mode:	Command, Run
Use:	$(0) = PEEK$(1,210)
Cards:	All

DESCRIPTION

Use this command to retrieve strings stored in RAM memory using the POKE$ command. *segment* specifies the 64K segment to save to. 0 is the base segment where RPBASIC-52 runs its programs. Setting MTOP to a number less than the top of memory will provide a 'protected' area from the Basic program.

Refer to the POKEB statement for addressing and segment information.

NOTE: This command works only when it is assigning another string variable. A BAD SYNTAX error is returned when it is part of a PRINT, IF-THEN, ASC, or other command or function. Use this function only as shown in SYNTAX above.

RELATED

POKE$

ERRORS

BAD SYNTAX If $, *segment*, or *address* is missing. Also when this function is part of another function or command.

BAD DATA If *segment* is > 7

EXAMPLE

The following example assumes MTOP = 30000. It will assign and recover a string from RAM.

```
10 AD = 30000
20 STRING 100,20
30 $(0) = "Test string"
40 POKE$ 0,AD,$(0)
50 $(1) = PEEK$(0,AD)
60 PRINT $(1)
```

PI

Syntax: PI
Function: Stored constant 3.1415926
Mode: Command, run
Use: PRINT PI
Cards: All

DESCRIPTION

PI is closer to 3.141592653, so proper rounding should be 3.1415927. However, trig errors were greater when 7 was used than 6 for the last digit.

POKEB

Syntax: POKEB *segment,address,data*
            Where: *segment* = 0 to 7, specifies a 64K segment
                   address = 0 to 65535, specifies address in a *segment*
                   data = 0 to 255, number to save to RAM
Function: Writes one data byte to RAM.
Mode: Command, Run
Use: POKE B1,2100,D
Cards: All

DESCRIPTION

Use POKEB to write to any one of 512K memory locations. The maximum number of
locations is limited by the amount of RAM installed.

WARNING: RPBASIC-52 does not check the *address*. It is possible to poke into the
             program, stack, or variable areas. Results are unpredictable. Poke into
             memory above MTOP for safest operation.

PEEK and POKE statements and functions access memory by specifying a segment and an address. A
segment is a 65,535 byte block. The largest segment number allowed depends upon the amount of RAM
installed. A system with 32K of RAM can only access 1 segment, numbered segment 0. When 128K is
installed, two segments, 0 and 1, are accessible. A 512K system has 8 segments, numbered 0 through 7.
Another way of looking at a segment is its address equivalent. The general addressing form is: S,AAAA. S
is the segment and AAAA is the address.

RPBASIC-52 always uses segment 0 for variable and program storage. Setting MTOP to a number below the
top of RAM ensures that RPBASIC-52 will not use the memory above that address. In a 32K RAM system,
the top of memory is address 32767. In a 128K or larger system, it is 65535. In 128K or 512K systems, all of
the memory in segment 1 and higher is available for data storage.

Maximum *segment* and *address* for a given system RAM size are:

RAM Size	Maximum Segment	Maximum Address
32K	0	32767
128K	1	65535
512K	7	65535

RELATED

PEEKB, XBY

ERRORS

BAD SYNTAX   If *B, segment, address*, or *data* is missing.
BAD DATA   If *segment* is > 7, *address* > 65535 or negative, *data* > 255 or negative.

EXAMPLES

```
10 POKE B0,64000,D      Pokes to segment 0, address 64000
20 POKE W1,0,A          Pokes a word (2 bytes) to segment 1, address 0
30 POKE $2,30,$(1)      Pokes a string to segment 2, address 30.
```

POKEF

Syntax: POKEF *segment,address,data*
             Where: *segment* = 0 to 7, specifies the 64K segment
                     address = 0 to 65535, location in *segment* to save to
                     data = +/-1E-127 to +/-0.99999999E+127, number to save to RAM
Function: Writes a floating point number to RAM.
Mode: Command, Run
Use: POKE F1,AD,DA
Cards: All

DESCRIPTION

Use POKEF to write floating point numbers into RAM. Program "constants" such as calibration tables are saved to battery backed RAM.

> *WARNING:* RPBASIC-52 does not check the address. It is possible to poke into the program, stack, or variable areas. Results are unpredictable. Poke into memory above MTOP for safest operation.

Each floating point number requires 6 bytes of RAM. When storing to RAM, separate addresses by at least 6 bytes. *address* is the starting address in RAM. Data is written to from *address* to *address* + 6. For example, if the first address was 0, the next is 6, third 12, and so on. An easy way to calculate an address is to use an index number and multiply it by 6. By adding a constant, different sections of RAM may be used. See the POKEB command for segment and address information.

RELATED

PEEKF

ERRORS

BAD SYNTAX   If *W, segment, address*, or *data* is missing.
BAD DATA   If *segment* is > 7, *address* or data > 65535 or negative

EXAMPLES

The following example takes data from an analog input, multiplies it by a constant, and saves it to segment 1 of the 128K RAM.

```
1000  FOR N = 0 TO 7
1010  POKE F1,N*6+100,AIN(N)*1.2383
1020  NEXT
```

The equation "N*6" is an index multiplier.

The next example prints out the data from RAM.

```
500 FOR N = 0 TO 7
510 PRINT PEEKF(1,120+N*6)
520 NEXT
```

The expression "120+N*6" performs two functions. First, 120 is a fixed offset into RAM. This offset is necessary when allocating sections of RAM for storage parameters (strings, byte data, and other floating point numbers). "N*6" indexes the floating point number into RAM so it does not overwrite other valid numbers.

POKEW

Syntax: POKEW *segment,address,data*

        Where: *segment* = 0 to 7, specifies the 64K segment

              address = 0 to 65535, location in *segment* to save to

              data = 0 to 65535, number to save to RAM

Function: Writes an unsigned 16 bit number to RAM.

Mode: Command, Run

Use: POKE W1,AD,DA

Cards: All

DESCRIPTION

Use POKEW to write 16 bit numbers into RAM. The results of an A-D conversion, for example, can be saved.

WARNING: RPBASIC-52 does not check the address. It is possible to poke into the program, stack, or variable areas. Results are unpredictable. Poke into memory above MTOP for safest operation.

See the POKEB command for segment and address information.

RELATED

PEEKW

ERRORS

BAD SYNTAX   If *W, segment, address*, or *data* is missing.

BAD DATA   If *segment* is > 7, *address* or data > 65535 or negative

EXAMPLES

The following example takes data from the AIN function and saves it to segment 1 of the 128K RAM.

```
1000   FOR N = 0 TO 7
1010   POKE W1,N*2+100,AIN(N)
1020   NEXT
```

The next example prints out the data from RAM.

```
500 FOR N = 0 TO 7
510 PRINT PEEKW(1,100+N*2)
520 NEXT
```

POKE$

Syntax:	POKE$ *segment,address,string*
	Where: *segment* = 0 to 7, specifies the 64K segment
	address = 0 to 65535, location in *segment* to save to
	string = string variable to save
Function:	Save string variable to RAM memory.
Mode:	Command,Run
Use:	POKE$ 1,30000,$(1)
Cards:	All

DESCRIPTION

POKE$ is used to save literal strings in RAM memory. Strings of any length can be saved.

When poking several strings, memory should be divided into "blocks" equal to the length specified in the STRING statement plus 1. POKE$ does not check to see if it is writing over other variable information.

WARNING: RPBASIC-52 does not check the address. It is possible to poke into the program, stack, or variable areas. Results are unpredictable. Poke into memory above MTOP for safest operation.

Refer to the POKEB statement for segment and address information.

POKE$ requires a string variable in order to work. If *string* is in quotes, a data error is returned.

RELATED

PEEK$

ERRORS

BAD SYNTAX	If *$, segment, address*, or *data* is missing.
BAD DATA	If *segment* is > 7, *address* > 65535 or negative, *string* not valid.

POP

Syntax: POP *variable [,variable,...]*
Function: Takes a value PUSHed to a stack and assigns it to the variable.
Mode: Command, run
Use: POP X,Y,Z
Cards: All

DESCRIPTION

Multiple variables can be POPped off the stack by separating the variables with commas. The first value POPped is the last value PUSHed.

POP and PUSH are useful for transferring data values between subroutines. They allow you to write a subroutine with arbitrary variable names. Data transfers to and from the subroutine can be performed by PUSH and POP, rather than by equating variable names.

RELATED
PUSH, LD@, ST@

ERROR
A-STACK   No *variable* on the stack when the POP instruction executed.

EXAMPLE

```
100     FOR N=0 TO 7
110     PUSH AIN(N)
120     NEXT
130     FOR N=0 TO 7
140     POP A
150     PRINT A*.00214
160     NEXT

>run

 0
 0
 0
 0
 0
 .536
 3.445
 2.334
```

PH0.
PH1.

Syntax: PH0. *expr*

             PH1. *expr*

             Where: *expr* = any mathematical expression

Function: Print in hexadecimal format following the number with an "H".

Mode: Command, run

Use: PH0. PEEKB (1,3000)

Cards: All

DESCRIPTION

The PH0. and PH1. instructions work like PRINT instruction except that it print values in HEX. The value printed is always a truncated integer and is followed with an "H" to indicate hexadecimal format. If *expr* evaluates to a fractional number within integer range, *expr* is truncated and displayed in hex format. If *expr* is not within integer range (0 through 0FFFFH/65535), the normal decimal PRINT mode is used. PH0. suppresses two leading zeros if *expr* evaluates to less than 0FFH. PH1. always prints four hexadecimal digits.

If there is no *expr*, a carriage return - line feed combination (a blank line) will be output. An *expr* may be any combination of instructions/operators and variables, strings, or literal values. More than one *expr* may be output by separating them with commas. Values are printed with a leading space; a list of values separated by commas will thus print with one intervening. This is different from the decimal PRINT instruction which prints values with a trailing blank. Strings and literals are output with no added blanks. If a comma is the last character in the list then a carriage return/linefeed is suppressed.

EXAMPLE

```
100    PH0. A
```

PRINT
PRINT #,
P.
?

Syntax:	PRINT *expr*
	P. *expr*
	? *expr*
	PRINT#*port,expr*
	P.#*port,expr*
	?#*port,expr*
	Where: *expr* = any string, mathematical number, or calculation
	port = serial output port 0 or 1. Your card may have m ore ports.
Function:	Prints value of *expr* to a serial port
Mode:	Comm and, run
Use:	PRINT "String",$(0),AIN(0)*.00214
Cards:	PRINT#, P.#, and ?# only on RPC-320, RPC-330.

DESCRIPTION

PRINT is used to send serial data to any port. Default is COM 0. Use *port* or UO to re-direct output to COM 1 or others.

If there is no *expr*, a carriage return - line fe ed com bination is sent. *expr* is any combination of instructions/operators and variables, strings, or literal values. More than one *expr* may be output by separating them with commas. Values are printed with a leading and trailing space; a list of positive values separated by commas will thus print with two intervening blanks. A "+" is implied. The "-" symbol precedes negative values and takes the place of the normal preceding space. Strings and literals are output with no added blanks. If a comma is the last character in the list then the normal <CR><LF> is suppressed.

The shorthan d versions P. and ? are converted to PRINT after each program line is entered, so a P. or ? is never listed.

The PRINT#*port,* instruction functions exactly like the PRINT instruction, but it directs output to the designated serial port. When using this syntax, any output directed by the UO comm and is bypassed.

P.# and ?# are shorthand for PRINT#.

RELATED UO, CONFIG BAUD

EXAMPLE

```
100     STRING 200,20 : $(0)="String" : B=PI*5
110     PRINT $(0),B,AIN(0)*.00215

>run

String 15.707963  0
```

PUSH

Syntax: PUSH *expr1* [,*expr2*,...]

             Where: *expr* is a numeric value

Function: Puts the value of *expr* to the argument stack. The first value PUSHed and is the last POPped.

Mode: Command, run

Use: PUSH X,Y

Cards: All

DESCRIPTION

PUSH and POP instructions pass values to Basic subroutines. The last value pushed is the last expression in the PUSH instruction, and is also the first popped off the stack. Multiple expressions can be pushed onto the argument stack by separating the expressions with commas.

The PUSH and POP instructions alleviate some of the problems of global variables in Basic. They eliminate the need to equate subroutine variables to global variables used by the program which called the subroutine.

The stack is cleared when a new program is loaded using EXECUTE.

RELATED

POP, LD@, ST@

ERROR

A-STACK   Attempt to push too many values on the argument stack. Typically no more than 32 values may be PUSHed onto the stack before it is full.

EXAMPLE

Please refer to the POP example.

PWM

Syntax: PWM *line, ltime,htime[,cycles]*

Where: *line* = 0 to 8 or 100 to 123. This is card depe ndent. Refer to yo ur hardware m anual.

ltime = 0 to 255, number of 5 ms periods line is low

htime = 0 to 255, number of 5 ms periods line is high

cycles = 0 to 65535, optional number of pulse cycles

Function: Produces pu lse width m odulated outpu t.

Mode: Comm and, Run

Use: PWM 8,3,B,5000

Cards: All

DESCRIPTION

Any digital I/O lines may o utput a Pulse W idth Modu lated signal. Pulses c an run indefinitely or for a specific number of times. PWM m ay be used to control the brightness of a display or send a number of pulses to a motion co ntroller.

WARNING: PWM continues to run in the command mode.

Low and high tim es are referenced from unbuffered outputs. Outputs from high current lines are inverted, so high and low times are reversed.

cycles refer to the num ber of low to high transitions from an unbuffered outpu t. When a P WM output is finished counting, that line goes and remains high.

A PW M output is sh utoff the quickest by specifying 1 for *htime, ltime,* and *cycles.* This can be done as part of a program or in the immediate mode.

RELATED

CONFIG LINE

ERRORS

BAD SYNTAX   If any param eters left out.

BAD ARGUMENT   When any parameters are out of range.

EXAMPLE

The following example sets outputs a PWM signal to line 7. Line 7 is configured for an output on power-up.

```
PWM 7,2,1
```

READ

Syntax: READ *variable* [,*variable*, ...]
Function: Sequentially assigns the values of data provided in the DATA statement to the variables in a list.
Mode: Run
Use: READ X,Y,Z
Cards: All

DESCRIPTION

Multiple variables following one READ instruction must be separated by commas. READ must always be followed by at least one variable.

See RESTORE for examples and more information.

REM

Syntax: REM *any ASCII text*
Function: Allows remarks in a program or on command line
Mode: Command, run
Use: 100 REM You can put any thing you want here
             REM This remark has no line number so will be discarded by RPBASIC-52
Cards: All

DESCRIPTION

The REM instruction lets you add comments to your program. Any text after a REM is ignored. REM instructions cannot be terminated with a colon, but they can follow colons. RPBASIC-52 lets you use REM in command mode and while downloading programs. A REM without a preceding line number is ignored by RPBASIC-52. This allows you to place comments in an off-line source code text file and have them stripped out when you download the text file to the card.

Appropriate comments make your programs easier to understand and maintain, but do slow program execution and consume program memory.

Multiple statements per line following a REM are ignored since they are considered part of the remark. Refer to the example.

EXAMPLE

```
100     REM A comment
120     PRINT A :REM PRINT A+2

>run
 0
```

RESTORE

Syntax: RESTORE

Function: Resets the READ instruction pointer to the beginning of the DATA list.

Mode: Run

Use: RESTORE

Cards: All

DESCRIPTION

After a RESTORE statement is executed, the next READ statement accesses the first item in the first data statement in the program.

ERROR

NO DATA - no DATA list provided.

EXAMPLE

```
100    READ A,B,C
110    PRINT A,B,C
120    RESTORE
130    READ X,Y,Z
140    PRINT X,Y,Z
150    READ A,B,C
160    PRINT A,B,C
150    DATA 1,2,3*2
150    DATA 6,9,12

>run

 1   2   6
 1   2   6
 6   9   12
```

RETI

Syntax: RETI

Function: Return from ONITR or ONTICK interrupt. RETI must be the last instruction of the interrupt subroutine.

Mode: Run

Use: RETI

Cards: All

DESCRIPTION

The RETI instruction causes you to exit from ONTICK, ONTIME (RPC-52 card only) and ONITR interrupts. RETI functions like RETURN, but it clears software interrupt flags so that RPBASIC-52 can acknowledge subsequent interrupts. If you don't execute the RETI instruction in the interrupt procedure, all future interrupts, hardware and software, are ignored.

RELATED

ONITR, ONTICK

EXAMPLE

Refer to ONTICK and ONITR examples.

RETURN

Syntax:	RETURN
Function:	Returns program to next instruction following a GOSUB command or software interrupt (ON LINE, ON KEY PAD, etc.)
Mode:	Run
Use:	RETURN
Cards:	All

DESCRIPTION

RETURN is used as a return from a GOSUB. Program execution continues at the statement following the GOSUB.

RND

Syntax: RND

Function: Returns a pseudo-random fractional number between zero and one inclusive.

Mode: Command, run

Use: A=RND

Cards: All

DESCRIPTION

The RND operator uses a 16-bit binary seed and repeats after 65535 pseudo-random numbers. The initial seed is the value of MTOP. The seed can be changed by writing to address 10CH and 10DH using the XBY command.

EXAMPLE

```
100    A=RND
110    PRINT A
```

SAVE

Syntax: SAVE *[segment]*
               Where: *segment* = 0 up to 7
Function: Save program to flash EPROM.
Mode: Command
Use: SAVE 1
Cards: All

DESCRIPTION

Use SAVE to store programs in flash EPROM. The current program in RAM is saved to the *segment* specified. If no *segment* is specified, 0 is assumed. Up to 8 programs (totaling over 500K bytes) can be saved, depending upon the flash EPROM type. Using EXECUTE, any of these 8 programs can be loaded and run during run-time. Use LOAD to retrieve a program.

SAVE automatically determines the type of flash EPROM installed. When an attempt is made to save a program to a segment larger than allowed by the EPROM type, an error message is returned.

The largest *segment* size depends upon the type of flash EPROM installed. The following table shows the largest segment for a particular EPROM.

EPROM type	Size Bytes	Sector size	Segment range
29C256	32K	64	0
29C040	512K	512	0-7

SAVE completely overwrites previous data in memory, up to the program size plus enough bytes to complete a sector. A sector is the number of bytes programmed in a flash at a time. For example, if a program was only 100 bytes long and a 29C040 is installed, 412 bytes of "filler" are also programmed. If a program is 1000 bytes long, 24 bytes of filler are programmed (2 sectors = 512 bytes). Sector sizes are not a concern except to users of BSAVE command.

Maximum program size also depends upon the amount of RAM installed. A 32K RAM can run a 29K program. A 128K or 512K RAM can execute up to a 60K byte program.

To find out the length of the program currently in RAM, type PRINT LEN in the immediate mode. Frequently, the length of a program in RAM is 10% to 30% less than that in a disk file. This is because the code is tokenized and commands are replaced with a single character.

When program requirements are small and data is large, some data can be saved to the flash EPROM using the BSAVE command.

RELATED

BSAVE, EXECUTE, LOAD

ERROR

BAD DATA   If *segment* is > 7 or larger than flash EPROM type.

SGN

Syntax: SGN(*expr*)

Function: Returns +1 if *expr* is greater than zero, zero if the *expr* equals zero, and -1 if *expr* is less than zero.

Mode: Command, run

Use: PRINT SGN(SIN(X)))

Cards: All

DESCRIPTION

Use SGN in level control applications. If a level is high or low, it can direct control to the appropriate program.

EXAMPLE

```
100    ON SGN(A)+1 GOSUB 2000,3000,4000
```

SIN

Syntax: SIN(*expr*)

Function: Returns the trigonometric SINE of *expr* which is assumed to be in radians. The value of *expr* must be in the range of +/- 200,000.

Mode: Command, run

Use: PRINT SIN(PI/2)

Cards: All

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and PI/2. the algorithm used to reduce the value will reduce accuracy when *value* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERRORS

ARITH. UNDERFLOW   *value* or result is less than RPBASIC-52's smallest floating-point value of ± 1E-127

ARITH. OVERFLOW   *value* or result is greater than RPBASIC-52's largest floating-point value of ±.9999999E+127

DIVIDE BY ZERO   Attempt to take TAN(X) when COS(PI/2) = 0

EXAMPLES

```
10   PRINT SIN(PI/2),COS(10*PI),TAN(8*PI/4)
20   PRINT ATN(PI)

>run

 1  1  0
 1.2626272
```

SPC

Syntax: PRINT SPC(*expr*)
             Where: *expr* = number of spaces to print
Function: Sends *expr* number of space characters (20H) from the serial port.
Mode: Command, run
Use: PRINT SPC(A*4),
Cards: All

DESCRIPTION

SPC must be used in conjunction with a print statement.

EXAMPLE

```
100    PRINT SPC(80-A*4),
```

STOP

Syntax: STOP
Function: Breaks program execution; resume with the CONT comm and.
Mode: Run
Use: STOP
Cards: All

DESCRIPTION

The STOP instruction lets you break program execution at specific locations in a program. You can display and modify variables after STOPping a program. STOP is useful for program debugging. The CONT command lets you resume program execution.

The line number printed after execution of a STOP instruction is the line number following the instruction and not the line number containing the STOP instruction.

If you modify a STOPped program, CONT will be unable to continue execution.

RELATED

CONT, GOTO

ERROR

CAN'T CONTINUE Attempt to continue after editing a stopped program, or attempt to execute CONT without a prior STOP or <Ctrl-C>.

EXAMPLE

```
100     PRINT "Tick=",TICK(0)
110     STOP
110     GOTO 100

>run

A= 0
STOP - IN LINE 120
```

STR

Syntax: A = STR(*function,$(n)[,$(n)]*))

               Where: *function* = 0 to 14, specifies string function to perform as described below.

Function: Performs string manipulation, described below, per function number.

Mode: Command,Run

Use: A = STR(0,$(0))

Cards: RPC-320, RPC-330

DESCRIPTION

There are 11 string manipulation functions using STR. Each function is described below.

NOTE: Most of these functions require a string variable (such as $(0)) rather than a quoted string. Functions which will allow quoted strings offer an alternate syntax immediately below the first one.

Syntax: A = STR(0,$(n))

Description:

Returns number of characters in a string. When string is not set equal to something, or the string number is out of bounds, erroneous data is returned. Length limit is 254 characters.

Example:
```
10  STRING 100,20
20 $(0)=" 123456789"
30  PRINT STR(0,$(0))
run
 10
```

Syntax: A = STR(1,$(n))

Description:

Convert letters *A - Z* to lower case. Variable A returns length of the string.

Example:
```
10  STRING 100,20
20 $(0)="Some UPPER case"
30 A = STR(1,$(0))
40  PRINT $(0)
run
some upper case
```

Syntax: A = STR(2,$(n))

Description:

Convert letters *a - z* to upper case. Variable A returns length of the string.

Example:
```
10  STRING 100,20
20 $(1) = "Some lower case."
30 A = STR(2,$(1))
40  PRINT $(1)
run
SOME LOWER CASE.
```

Syntax: A = STR(3,$(n))

Description:

Returns numbers in a string as a real number. Similar to VAL in other Basics. Leading spaces are ignored. First non-number terminates conversion at last valid number. No valid numbers return 0. Number length is limited to the first 12 valid numbers and decimal in a string. This means a number no greater than 99999 99999 999 is converted to a number.

Example:
```
10  STRING 100,20
20 $(2) = "-23.452volts"
30 A= STR(3,$(2))
40  PRINT A
run
-23.452
```

Syntax: A = STR(4,$(n))

Description:

Trims spaces to left of first non-space character. Variable A returns length of trimmed string.

Example:
```
10  STRING 100,20
20 $(0) = "  1234"
30 A = STR(4,$(0))
40  PRINT $(0)
50  PRINT A
run
1234
4
```

Syntax: A = STR(5,$(n))

Description:

Trims spaces from right side of string. Variable A returns length of trimmed string.

Example:
```
10  STRING 100,20
20 $(0) = "ABCDE    "
30 A = STR(5,$(0))
40  PRINT $(0)
50  PRINT A
run
ABCDE
 5
```

Syntax: A = STR(6,$(x),$(y))

               A = STR(6,$(x),"*string*")

Description:

Appends one string into another. This function concatenates two strings in the form of $(x) = $(x) + $(y). Length of new string is returned in variable A. The variable $(y) could be a quoted *string*.

Example:
```
10  STRING 120,40
20 $(0)="First part"
30 $(1)=" Second part"
40 A = STR(6,$(0),$(1))
50  PRINT $(0)
60  PRINT "Length:",A
70 A = STR(6,$(0)," last part")
80  PRINT $(0)
90  PRINT "Length:",A
run
First part Second part
Length: 22
First part Second part last part
Length: 32
```

Lines 50 and 80 print the concatenated string $(0).

Syntax: A = STR(7,$(*put*),$(*get*),*position,length*)
Description:

Extracts a portion of a string from $(*get*) and transfers it over to $(*put*). The actual number of characters moved is returned. *position* starts at 1. When *position* is 0, no characters are placed into $(*put*) regardless of *length*. When *length* is 0, all characters are copied from $(*get*) to $(*put*) starting at *position*.

Example:
```
10  STRING 200,20
20  $(0) = "123456.789"
30  A = STR(7,$(1),$(0),3,5)
40  PRINT $(1)
50  PRINT "Length:",A
run
3456.
Length: 5
```

Syntax: A = STR(8,$(*search*),$(*substring*))
Description:

Scans $(*search*) for occurrence of *substring*. Returns position where entire *substring* first matches *search* string. Returns 0 when *substring* is not in *search* string.

Example:
```
10  STRING 200,20
20  $(0) = ">05M34C3"
30  $(1) = "05M"
40  A = STR(8,$(0),$(1))
50  PRINT "Position match at:",a
run
Position match at: 2
```

The number '0' in $(1) matches $(0) at position 2.

Syntax: A = STR(9,$(*string1*),$(*string2*))
Description:

Compares *string1* to *string2*. Returns position of first mismatch. If both strings exactly match, then 0 is returned.

Example:
```
10  STRING 200,20
20  $(0) = ">05M34C3"
30  $(1) = ">05"
40  A = STR(9,$(0),$(1))
50  PRINT "Mismatch starting at:",a
run
Mismatch starting at: 4
```

Since the first three character positions matched, position 4 is returned as the longer string did not match the shorter one.

String functions 8 and 9 are useful in RS-485 network communication. In the above example, ">05" could be the RPC-320's address. Knowing the mismatch starts at position 4, the address can be assumed correct. If the mismatch started sooner, a smaller number is returned. Hence, the message was not intended for this particular card and the entire message can be flushed.

Syntax: A = STR(10,$(n),*format,variable*)

Description:

   Converts and formats *variable* into a string and puts it into $(n). Variable A returns irrelevant data. Formatting is controlled by the *format* variable. Strings are formatted into one of 3 basic patterns, described below.

   format = 0. Default free format. When number is between ±99999999 and ±0.1, RPBASIC will save integers and fractions. When numbers are outside this range, the F0 format, described next, is used.

   format = Fx. Floating point format. 'x' determines how many digits after the decimal point are saved. When x = 0, the number of trailing digits will vary and trailing 0's are not saved. 'x' is represented as a hex number. When *format* = 0F3H, 3 decimal numbers are printed. An alternate way of setting floating point output is to make *format*= the number of decimal numbers plus 240.

   format = xyH. Force integer and/or fraction output. Command is same as USING(##.##), where 'x' is the number digits left of the decimal point and y is to the right. Maximum value for x and y is 7. Use the hex format to set the number.

Example:
```
10    String 200,20
20   C = 23.45
30   F = 0
40   A = STR(10,$(0),F,C)
50    PRINT "Variable value, before formatting:",C
60    PRINT "String in free format:",$(0)
70   F = 0F2H
80   A = STR(10,$(0),F,C)
90    PRINT "Using floating point format:",$(0)
100 F=52H
110 A=STR(10,$(0),F,C)
120 PRINT "Using #####.## format:",$(0)
run
Variable value, before formatting: 23.45
String in free format: 23.45
Using floating point format: 2.34 E+1
Using #####.## format:    23.45
```

ERROR

   BAD ARGUMENT When *function* is out of range or string data is incorrect.

STRING

Syntax: STRING *total bytes,string length*
             Where: *total bytes* = total number of bytes in memory to allocate
                    string length = maximum number of bytes in a string
Function: Allocate memory for strings
Mode: Command, run
Use: STRING 56,10 : REM Allocate memory for 5 10-byte strings
Cards: All

DESCRIPTION

Prior to using strings, you must use STRING to allocate memory for them. The STRING argument values are computed by this equation:

total bytes = ((*string length* + 1) * number_of_strings) +1

The only way to recover string memory is with a "STRING 0,0" instruction. String memory is reclaimed and then reallocated each time you use the STRING operator. Strings are terminated with a carriage return (0DH or 13) which is the additional byte added to your *bytes per string expr*.

WARNING:

STRING causes RPBASIC-52 to execute the equivalent of a CLEAR instruction since string and numeric variables occupy the same memory space. In other words, the STRING instruction clears all variables, interrupts and stacks. Allocate string memory early in your program and don't reallocate it unless you can accept the loss of all variables.

RELATED

ASC, CHR, STR

ERRORS

MEMORY ALLOCATION   Memory not allocated for strings
C-STACK   STRING used in a subroutine, clearing the stack.

EXAMPLES

```
10      STRING 1000,40
20      $(0) = "Up to 40 characters in this string"
.
.
100    $(2) = COM$(1)
```

SQR

Syntax: SQR(*expr*)

               Where: *expr* is any valid mathematical expression, number, or variable greater than 0

Function: Returns a positive square root.

Mode: Command, run

Use: PRINT SQR(A)

Cards: All

DESCRIPTION

expr must be positive. Any calculation is accurate to ±5 least significant digits.

ERRORS

ARITH. UNDERFLOW   *expr* or result is less than RPBASIC-52's smallest floating point value of ± 1E-127

ARITH. OVERFLOW   *expr* or result is greater than RPBASIC-52's largest floating point value of ±.99999999E+127

BAD ARGUMENT   Attempt to take SQR() of a negative number

EXAMPLE

```
100    FOR N = 1 to 10
110    A=SQR(N)**2
120     IF (A-N)<>0 THEN PRINT A,N
130     NEXT

>run
 2.0000001  2
 2.9999999  3
 5.0000001  5
 6.0000002  6
 6.9999999  7
 7.9999998  8
```

ST@

Syntax: ST@ *expr*
           Where: *expr* = 0 to 65535
Function: Takes a floating-point number from the argument stack and stores it to data memory at the address.
Mode: Command, run
Use: PUSH B : ST@7E00
Cards: All

DESCRIPTION

ST@ is used in conjunction with PUSH, POP, and LD@. Use these commands to save and retrieve floating point numbers to program RAM.

NOTE: LD@ and ST@ cannot use extended RAM. Only segment 0 RAM (used for running Basic programs) is used. Use PEEK and POKE commands to access this memory.

WARNING: When 128K and 512K RAM are installed, all of memory is cleared on power up and reset. Do not use LD@ or ST@ to save floating point numbers in segment 0. Use POKE and PEEK type commands instead.

32K RAM systems have address 7E00H set as MTOP. This location up to 7FFFH may be used to store variables.

expr is the address in RAM of where a number is stored.

Each floating-point number requires six bytes of memory. *expr* in the ST@ and LD@ instructions specify the high address. A number is stored at locations *expr* through *expr-6*.

RELATED

LD@, PUSH, POP

ERROR

expr location should be above MTOP. Otherwise the data may be overwritten.

EXAMPLE

```
100     A=AIN(0)*.234
110     PUSH A
120     ST@7F00H
   .
   .
300     LD@7F00H
310     POP B
320     PRINT "Analog value retrieved=",B

>run

Analog value retrieved=",B
```

TAB

Syntax: PRINT TAB(*position*)
             Where: *position* = 1 to 255
Function: Specifies a column number at to begin printing.
Mode: Command, run
Use: PRINT TAB(5), "Pressure", TAB (20),"Temperature"
Cards: All

DESCRIPTION

TAB is used with PRINT. It is used to print data in table form. If the cursor is past the requested column, the instruction is ignored.

ERROR

BAD ARGUMENT   When *position* is negative or out of range.

EXAMPLE

```
100    PRINT TAB(5),"Pressure",TAB(20),"Temperature"
110    FOR N=0 TO 6
120    PRINT TAB(7),AIN(0)*.237,
130    PRINT TAB(23),AIN(1)*1.324
140    NEXT

>run
     Pressure        Temperature
      116.13            237.3
      116.14            237.3
       116.13            237.4
       116.14            237.4
       116.11            237.0
       116.16            237.6
       116.13            237.5
```

TAN

Syntax: TAN(*expr*)

Function: Returns the trigonometric tangent (sin/cos) of *expr* which is assumed to be in radians. *expr* must be in the range of +/- 200,000.

Mode: Command, run

Use: PRINT TAN(PI/4)

Cards: All

DESCRIPTION

SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and PI/2. the algorithm used to reduce the value will reduce accuracy when *value* is large. To maintain accuracy, keep the arguments for trig functions as small as possible.

ERRORS

ARITH. UNDERFLOW   *value* or result is less than RPBASIC-52's smallest floating-point value of ± 1E-127

ARITH. OVERFLOW   *value* or result is greater than RPBASIC-52's largest floating-point value of ±.9999999E+127

DIVIDE BY ZERO   Attempt to take TAN(X) when COS(PI/2) = 0

EXAMPLES

```
100    PRINT SIN(PI/2),COS(10001*PI),TAN(5*PI/4)
110    PRINT ATN(TAN(PI/4))/PI

>run

1  -1 1
.24999996
```

TICK

Syntax: TICK(*timer*)

Where: *timer* = 0 to 3. It specifies the timer number.

Function: Returns a time from one of 4 process clocks in 5 ms increments.

Mode: Command,Run

Use: A = TICK(2)

Cards: All

DESCRIPTION

There are four tick timers updated 200 times per second. Each timer is independent of each other in that they may be read and cleared separately (see CLEAR TICK). All timers are updated at the same time.

This function is separate from the real time clock and is not battery backed. All timers reset to 0 on power up or reset.   Timers continue to run in command mode and cannot be turned off.

TICK(n) returns time in thousandths of a second (in 5 ms intervals) up to 65535.995 seconds, or approximately 18.2 hours. The timer then starts at 0 again.

Tick timers are not affected by to the ONTICK instruction.

RELATED

CLEAR TICK, ON TICK

ERRORS

BAD SYNTAX   If any parameters left out

BAD ARGUMENT   When *timer* > 3 or negative or left out

EXAMPLE

The following example clears tick timer number 3, delays for a time, then prints tick timers 0 and 3.

```
10 CLEAR TICK(3)
20 FOR X = 0 TO 1000
30 NEXT
40 PRINT TICK(0),TICK(3)

 124.6  .425
```

TIME (function)

Syntax: A = TIME(*n*)

Where: *n* = 0 to 4

        0 = hours
        1 = minutes
        2 = seconds
        3 = hundredths of a second
        4 = seconds since midnight

Function: Returns the hour, minute, second, or hundredths of a second from the real time clock

Mode: Command, Run

Use: A=TIME(1)   Returns minutes

Cards: All

DESCRIPTION

A DS1216DM must be installed in the data RAM socket, usually U5. Refer to your hardware manual for exact location. The numerical value of the hour, minute, or second is returned.

TIME(4) returns the seconds plus hundredths of a second since midnight. This is useful when time stamping an event.

A HARDWARE error is returned when the RTC module is missing or bad. Use ONERR to trap for this kind of error. Error code is 50 at address 101H

RELATED

TIME (command)

ERRORS

BAD ARGUMENT   When *n* out of range, negative
HARDWARE   RTC module missing

EXAMPLE

The following program converts a TIME number into a string.

```
100 STRING 200,20
110 $(0) = "  :  :  " :REM SETUP FOR HH:MM:SS
120 HR = TIME(0)
130 MN = TIME(1)
140 SC = TIME(2)
150 T = HR/10 :REM use T for temporaries
160 ASC($(0),1) = INT(T) .OR. 48 :REM 10's of hours
170 ASC($(0),2) = (T-INT(T))*10) .OR. 48 :REM 1's
180 T = MN/10  :REM minutes conversion
190 ASC($(0),4) = INT(T) .OR. 48
200 ASC($(0),5) = ((T-INT(T)) * 10) .OR. 48
210 T = SC/10  :REM temp for seconds
220 ASC($(0),7) = INT(T) .OR. 48
230 ASC($(0),8) = ((T-INT(T)) * 10) .OR. 48
240 PRINT "Time:",$(0)
```

TIME (command)

Syntax:	TIME *hours,minutes,seconds*
	Where: *hours* = 0 to 23
	minutes = 0 to 59
	seconds = 0 to 59
Function:	Sets the time to the real time clock
Mode:	Command, Run
Use:	TIME 18,17,00 Sets time to 6:17:00 PM
Cards:	All. Note consideration for RPC-320, RPC-330

DESCRIPTION

Time uses a 24 hour format. Hundredths of a second are set to 0 when TIME is executed.

The RPC-320 and RPC-330 use an optional DS1216DM clock module. This module is shipped with the clock off to conserve battery power. To turn the clock on, execute the DATE command first. Failure to do so causes a HARDWARE error. DATE need only be done once to turn on the clock. Subsequent changes to TIME can be performed without using DATE. Refer to your hardware manual for installation location.

RELATED

TIME (function), DATE (command)

ERRORS

BAD ARGUMENT   When *hours, minutes*, or *seconds* is out of range

HARDWARE   Clock module is missing or not turned on. Error code 50 at address 101H

UI0
UI1

Syntax: UI*n*

               Where: *n* = 0 or 1, is the serial port number

Function: Directs serial input to COM 0 or COM 1 when using GET and INPUT statements.

Mode: Command, run

Use: UI1

Cards: All

DESCRIPTION

UI0 and UI1 can be placed anywhere in a program, allowing RPBASIC-52 to accept input from either COM0 or COM1.

The original BASIC-52 required an assembly language routine to use another serial port. RPBASIC-52 has done this already.

RELATED

GET, UO0, UO1, INPUT

EXAMPLE

The following example prints out and inputs data from COM 1.

```
100 UI 1
110 UO 1
120 INPUT "Enter name:",$(0)
130 UO 0
140 UI 0
```

UO0
UO1

Syntax: UO*n*
             Where: *n* = 0 or 1, is the serial port number
Function: Directs PRINT output to COM 0 or COM 1 serial port.
Mode: Command, run
Use: UO1
Cards: All

DESCRIPTION

UO0 and UO1 can be placed anywhere in a program, allowing RPBASIC-52 to accept print to either COM0 or COM1.

The original BASIC-52 required an assembly language routine to use another serial port. RPBASIC-52 has done this already.

RELATED

UI0, UI1, INPUT

EXAMPLE

The following example prints out and inputs data from COM 1.

```
100 UI 1
110 UO 1
120 INPUT "Enter name:",$(0)
130 UO 0
140 UI 0
```

USING
U.

Syntax: PRINT USING (*format*)
            PRINT U.(*format*)
            Where: *format*

    USING(Fn) n is the number of significant digits. A minimum of 3 significant digits are always printed. Maximum value of n is 8.

    USING(#.#) The number of # symbols determines how many significant figures of the output value will be displayed before and after the decimal point. The maximum total number of "#" symbols is 8. Integers (decimals truncated) are printed when there are no "#" symbols after the decimal point or if no decimal point is given. If a value cannot be printed in the requested format, RPBASIC-52 outputs a "?" and prints the value in USING(0) format.

    USING(0) The default output format for RPBASIC-52 floating-point values. Displayed as a decimal integer and fraction if the value is between +/- 99999999 and +/- 0.1.

Function: Used with PRINT to format subsequent expressions.
Mode: Command, run
Use: PRINT USING(F3),A
Cards: All

DESCRIPTION

The same formatting capability is available using the STR(10,...) function.

Formatting is "remembered" until it is reset or changed.

RELATED STR

ERRORS BAD SYNTAX - Missing # to the left of the decimal point or a space between USING and the left parentheses.

EXAMPLE

```
110     PRINT USING(F3),PI*100

>run

 3.14 E+2
```

WDOG

Syntax: WDOG *[time]*
               Where: *time* = 0, 1, or 2
                       no parameter = toggle watchdog timer
                       0 = turn off watch dog timer
                       1 = set timeout interval to 0.38 seconds
                       2 = set timeout interval to 2.8 seconds
Function: Resets or sets watchdog timer.
Mode: Run
Use: WDOG
Cards: All. Cards use this command in different ways. Refer to your hardware manual to verify operation.

DESCRIPTION

The watchdog timer is a supervisory function for applications that cannot afford to "crash".

WDOG 1 or WDOG 2 enables the watchdog and sets the interval. WDOG is executed periodically by your program to prevent the card from resetting. WDOG 0 turns off the watchdog timer.

Different cards may use different time out periods. Refer to your hardware manual.

RELATED

ERROR

BAD ARGUMENT when *time* is out of range

EXAMPLE

The following example shows how ONTICK can be used to reset the timer. The watchdog timer is set for 2.8 seconds.

```
10    WDOG 2
20    CLEAR TICK(0)
30    ONTICK 2,200
40    GOTO 40
200   PRINT TICK(0)
210   WDOG
220   RETI
```

When a watchdog timeout occurs, only the CPU is reset. The effect is the same as performing a hardware reset, except a hardware reset pulse is not issued. Digital I/O at J3 does not change. Digital lines L0 - L8 are reset to power up conditions as is the display port.

XBY

Syntax: XBY(*addr*)
            XBY(*addr*)=*expr*
            Where: *addr* = 0 to 65535 (0FFFFH) is a memory address
                   expr = 0 to 255 is data to save
Function: Read/write external data memory, segment 0 only.
Mode: Command, run
Use: XBY(99)=35
Cards: All

DESCRIPTION

   XBY retrieves or assigns a value to external data memory. This command is equivalent to PEEKB and POKEB.

RELATED

   CBY, DBY, PEEKB, POKEB

ERROR

   BAD ARGUMENT Invalid *addr* or attempt to assign an out of range value to a XBY(*expr*).

EXAMPLE

```
100    XBY(47536) = XBY(47536) .OR. 3
```

CONFIG COMMANDS

CONFIG commands configure various I/O to user defined parameters.

All CONFIG commands are unique to RPBASIC-52. There is no equivalent in the original version. Some commands are not available for all cards.

CONFIG AIN

Syntax: CONFIG AIN *channel, mode, range*
                Where: *channel* = 0 to 7, analog input channel
                        mode = 0 or 1, differential or single - ended
                        range = 0 or 1, ±2.5 or 0 to 5 volt input
Function: Determines type of analog input for measurement
Mode: Command, Run
Use: CONFIG AIN 3,1,0
Cards: RPC-320, RPC-330 Refer to your hardware manual for applicability if your card is not listed here.

DESCRIPTION
All inputs are configured for single - ended, 0 to +5V inputs on power up. Inputs, or pairs of inputs, may be changed to differential and/or ±2.5 volt input.

Differential inputs use adjacent channels, as described in *Chapter 10, Analog Input, Initialization.* Inputs are pseudo-differential, meaning the input signal is measured with respect to ground. See *Chapter 10* for more information.

mode of 1 specifies single ended while 0 means differential.

range = 0, a ±2.5V input is chosen while a 1 sets 0 to +5 volt input.

Refer to *Chapter 10, ANALOG INPUTS* in your hardware manual for examples and configuration information.

ERROR
BAD ARGUMENT When any parameter is out of range.

CONFIG BAUD

Syntax: CONFIG BAUD 0,*baud*
            CONFIG BAUD 1,*baud,rs-485*
            Where: *baud* = Baud rate number. See tables below.
                       rs-485 = Parameters for RS-485 port. See table below.
            Specify 0 or 1 for serial port.
Function: Set baud rates for COM0 and COM1.
Mode: Command, Run
Use: CONFIG BAUD 1,3,OFF
Cards: All. *baud* code will vary from card to card.

DESCRIPTION

Power up baud rate is 9600 for both ports. Serial parameters change immediately after this command is executed. Communication parameters are set at 8 data bits, 1 stop, no parity.

Use the table below for COM0 and COM1 *baud* code on the RPC-320 and RPC-330:

baud code	Baud rate		*baud* code	Baud rate
0	38400 (COM0), 57600 (COM1)	4		2400
1	19200	5		1200
2	9600	6		600
3	4800	7		300

Notice *baud* code 0 gives different rates for COM0 and COM1.

rs-485 configures COM1 for RS-232, RS-422, and 4 wire RS-485. Set jumper W4 as needed. Power up default is 0, or RS-232 configuration.

rs-485	Configuration
0	RS-232
1	RS-422 (transmitter and receiver always on)
2	RS-485, 4 wire (Tx on during transmit, receiver always on)

ERROR

BAD ARGUMENT   When any parameters are out of range.
BAD SYNTAX   When any required parameters are missing.

CONFIG DISPLAY

Syntax: CONFIG DISPLAY *type*

Where: *type* = 0 to 3, defines the display type

0 = LCD 4 x 40 character

1 = LCD 4 x 20 character

2 = Vacuum florescent 4 x 20 character

3 = LCD - 5003 graphics display

4 = Vacuum florescent 4 x 20, IEEE Centry series

Function: Defines the display type used with DISPLAY and related commands.

Mode: Command, RUN

Use: CONFIG DISPLAY 1   Configures display port and operation for LCD 4 x 20.

Cards: All

DESCRIPTION

The display type must be set in order for the DISPLAY drivers to work properly.

Other kinds of LCD and Vacuum florescent displays may also be used. However, certain options such as character positioning may not work properly or at all.

RELATED

DISPLAY, CLEAR DISPLAY

ERROR

BAD ARGUMENT   When *type* out of range.

CONFIG FREQ

Syntax: CONFIG FREQ *channel,interval*
            Where: *channel* = counter num ber, 0 or 1
                   interval = number of 5 milli-second periods between readings. Range is 1 to 255. An
                   interval of 0 turns off this multitasking routine.
Function: Sets up multitasking to read a counter every *interval*. The counter is read using the FREQ com mand.
Mode: Comm and, Run
Use: CONFIG FREQ 0,100
Cards: RPC-210, RPC-320, RPC-330 (cards with LSI 7166 counter chip)

DESCRIPTION

Comm and sets up RPB ASIC operating system so FREQ function can operate. This comm and defines a
counter and time interval between counter reads.

Longer *interval* smooths out readings. Short intervals (*interval* between 1 and 10) are not recommended.

RELATED

FREQ

ERROR

BAD ARGUMENT   *channel* > 0 or 1 (depending upon the card)
                          interval > 255

EXAMPLE

See the FREQ command for an example.

CONFIG LINE

Syntax: CONFIG LINE 0,*configuration 0,port C*

                CONFIG LINE 100,*configuration 1,port A,port B,port C*

                Where: *configuration n* = port configuration per tables below.

                      port A = Digital I/O port A output data

                      port B = Digital I/O port B output data

                      port C = Digital I/O port C output data

Function: Configures digital I/O ports for inputs and outputs.

Mode: Command, Run

Use: CONFIG LINE 0,1,128

                CONFIG LINE 100,3,255,0,240

Cards: All. Check line ranges for your card.

DESCRIPTION

Upon power up or reset, digital I/O port J3 (lines 100-123) are configured for inputs. Lines at P6 are configured for inputs (L0-L3) and outputs (L4-7). Outputs L4 and L5 are low and L6 and L7 are high. The status of these lines is changed using this command.

There are two digital I/O line number groups on the RPC-320. One group, 0-8, access lines at the terminal strip on the card. Line number 0 is used to specify these lines. *port C* simply specifies which lines are high and low.

The second digital group is specified as line 100 and determines the configuration for digital I/O port J3. An 82C55 is used to interface the 24 digital I/O lines. The 82C55 consists of 3 ports organized as follows:

Port A   Eight lines that can be programmed as all inputs or all outputs.

Port B   Eight lines that can be programmed as all inputs or all outputs.

Port C   Eight lines which can be programmed in one group of eight lines or two groups of four lines as all inputs or all outputs.

The following table is used for the *configuration 0 or 1* parameter. It determines which port, or part of a port, is an input and output.

configuration 0	Lines 4-7	Lines 0-3 (Upper and lower Port C)
0	Output	Output
1	Output	Input
2	Input	Output
3	Input	Input

configuration 1	Port A	Port B	Upper C	Lower C
0	Output	Output	Output	Output
1	Output	Output	Output	Input
2	Output	Input	Output	Output
3	Output	Input	Output	Input
4	Output	Output	Input	Output
5	Output	Output	Input	Input
6	Output	Input	Input	Output
7	Output	Input	Input	Input
8	Input	Output	Output	Output
9	Input	Output	Output	Input
10	Input	Input	Output	Output
11	Input	Input	Output	Input
12	Input	Output	Input	Output
13	Input	Output	Input	Input
14	Input	Input	Input	Output
15	Input	Input	Input	Input

port A, B, and *C* parameters set the output status. When a port is configured as an input, any value can be used. When a *port* is configured as an output, the value may be determined by corresponding a bit output with a value.

```
Bit      7 6 5 4 3 2 1 0
Status   0 0 1 0 0 0 1 1  = 23H = 35 decimal
```

Lines 0, 1, and 5 will go high while the others will go low. In this example, *port* would equal 35 or 23H (either one will work).

When J3 is connected to an opto rack, a '0' at a bit position turns ON a module while a '1' turns it off. (***NOTE:*** The LINE command reverses the meaning of '0' and '1' while LINE # does not).

The value for an output at *port C* is computed in the same manner even if one half is an input.

The following example configures lines at J3 so port A and B are all outputs and port C is all inputs. With the high current output installed at U12, lines 7 and 8 are 'ON' or low while the other high current outputs are 'OFF'. Line 19 will also be low. Lines at port C are pulled high or low according to jumper W7.

```
CONFIG LINE 100,5,254,130,0
```

WARNING: When configuring lines for outputs using CONFIG LINE, lines will go low momentarily (less than 10 micro-seconds) until they are set high again as per the data in the command line.

Some other lines are affected when CONFIG LINE 0 is executed. These lines are card dependent. Refer to the cards hardware manual under *DIGITAL I/O* for more information.

RELATED
LINE (both statement and function)

ERROR
BAD ARGUMENT   *configuration* > 15 or negative
                port > 255 or negative

APPENDIX A - Network example program

File: NET3XX.BAS

```
rem RPC-3xx networking
rem Uses COM1 as network port
rem To use com port 0 and get going faster, REM out the following lines:
rem 130, 150, 1510, 1530
rem Line 1510 must still exist, so rem AFTER the line number

rem Change the following lines
rem 160 for COM 0 instead of 1
rem 1000 change COM$(1) to COM$(0)

rem If your card does not have analog output, comment out line 2560

rem command D assumes a display.  Adjust the CONFIG DISPLAY command
rem at line 140


rem Demo program is limited to 5 commands.  If adding more, change
rem limit check in line 1210

rem Data packet to the card is:

rem  <CR>>ncd...ds

rem Where
rem <CR> = carriage return character 0DH
rem > =  command signature
rem n = card number.  May be number 0-9 or letter
rem c = command.  May be number, letter, or combination
rem d...d = data as required for command
rem s = optional checksum of string
rem        a ?? means ignore checksum

rem Command types for this demo are:

rem    A = set line 8.  Data following A is 0 or 1
rem        Example:  >00A0??
rem    B = set analog output channel, data
rem        Example:  >00B043??
rem                        ||- 1 to 4 digits of data
rem                        | - channel no 0 or 1
rem    Shows how to convert a string number into one usable by
rem     BASIC
rem C = return position from counter 0 or 1
rem    Example:  >00C0??
rem                        ||-counter #
rem                        | - command
rem    Shows how to take a "real" number and convert it to a
rem     string.
rem D = Send message to display port
rem    Example:  >00DCheck station 5??
rem     E = Power up acknowledge.  Used to inform host of reset condition
rem    Example:  >00E??
rem    F = Is everything OK or is there a problem
rem    Example:  >00F??
rem    Command F returns an An.  If n = 0, everything OK
rem    Error codes in STATUS are set somewhere else
rem    Routine clears STATUS when polled

100 STRING 2000,40       :REM allocate memory
120 $(0) = ">00"         :REM assign card ID.  It is modified at line 150
125 $(3) = ">99" :REM All units go into safety mode

REM set up RS485 port on board for 19200
rem NOTE: this is board dependent.  Check your cards manual to make sure
```

```
rem 130 CONFIG BAUD 1,1,2

rem set the display type for command D

140 config display 1

rem Read lines 0-3 to determine card address.
rem Card number starts from ASCII '0' and goes up from there.

rem 150 ASC($(0),3) = (lineb(5,2) .AND. 15)+48

REM declare tasking and define conditions
REM To 1000 when either 40 characters are in or a <cr> received

rem 160 ON COM$1,40,13,1000
160 on com$0,40,13,1000

300 GOTO 300            :REM hang out here

REM Handle interrupt here
REM Since all variables are global, local variables used here start
REM with the letter 'o'.  This helps prevent inadvertent value changes
REM to other parts of the program

rem 1000 $(1) = COM$(1)     :REM get data
1000 $(1) = com$(0)

rem Check for emergency safety mode code

1005 if str(8,$(1),$(3)) = 1 then 5000

REM see if card ID is in this packet
REM If 0 returned, is not this card

1010 IF STR(8,$(1),$(0)) <> 1 THEN RETURN

REM Parse out command.  For this demo, assume
REM it is only 1 letter long and starts with
REM capital letter A.  If command is negative
REM can return a NAK (negative acknowledge)
REM to sender or ignore it.

1020 OA = ASC($(1),4)-65
1030 IF OA < 0 then 1500

REM Make sure checksum is OK
REM Add up values in string for length - 2

1040 ocksum = 0
1050 ole = str(0,$(1))
1060 for oc = 1 to ole-2
1070  ocksum = asc($(1),oc) + cksum
1080 next

rem strip off excess

1090 ocksum = ocksum .and. 0ffH

REM Get checksum values
REM IF second to last character is a ?, then don't check checksum
REM convert last two characters into decimal

1100 oc = asc($(1),ole-1):REM get first digit
1110 if oc = 63 then 1200:rem if ?, skip rest of checksum test
1120 gosub 1600      :rem convert ASCII to number
1130 och = oc*16     :REM assign high byte first
1140 oc = asc($(1),ole)  :rem get last hex digit
1150 gosub 1600
1160 oc = oc+och     :rem make checksum value
```

```
rem if last two digits don't sum to message, then return a negative
rem acknowledge error and bail out

1170 if oc <> ocksum then $(2) = "N2" : goto 1510

rem Checksum is good

REM If status command, go process it

1200 IF oa = 4 THEN 4000
1210 if oa > 5 then 15REM if not in command, is error

REM Check for valid power up acknowledge
REM if not acknowledged, then state so

1220 if oflag = 0 then $(2) = "N3" : goto 1510

rem process command
rem GOSUB's could also be used here.  However, goto's are faster as
rem exiting the routine makes a direct branch to the condition

rem  Cmdn letter  A    B    C     D     E    F

1240 on oa goto 2000,2500,3000,3500,4000,4500

rem If more commands, check for limit.  If over, then subtract command
rem and make another ON GOTO

REM Common return point for successful completion of a command

REM Return acknowledge to sender.
REM Used for commands

1400 $(2)="A"
1410 GOTO 1510            :REM to common output & exit

REM Return negative acknowledge to sender.
REM N1 = unrecognized command
REM N2 = checksum bad
REM N3 = power up not acknowledged.  Needs command 5.
REM N4 = bad data
REM N5 = something is wrong.  Can add error conditions as needed

1500 $(2)="N1"
1510  rem UO1
1520 PRINT $(2)
rem 1530 UO0        :REM back to main port
1540 RETURN

REM convert ASCII HEX number into a number 0 - 15
REM Enter with oc = ASCII value of number (0-9 or A-F which is 48-
REM 58 or 65 to 70)
REM If problem, oc returns -1.  If OK, returns number 0 to 15

1600 if (oc < 48) .or. (oc > 70) then oc = -1 : return
1610 if oc > 58 then 1640

rem value between 0 and 9.  Simply subtract 48 and exit

1620 oc = oc-48
1630 return

rem Value should be between A-F

1640 if oc < 65 then oc = -1 : return
1650 oc = oc - 55
1660 return

REM Send back acknowledge
```

```
1700 $(2) = "A"
1710 GOTO 1510


rem Bad data

1750 $(2) = "N4"
1760 goto 1510

REM set a line according to data
rem For this example, line 8 is assumed to be controlled

rem Get desired status.  Position 5 in string is 1 or 0

2000 oc = asc($(1),5) - 48

rem make sure data is 0 or 1

2010 if oc < 0 then 1750
2020 if oc > 1 then 1750

rem Set line according to input and send back acknowledge

2030 line 8,oc
2040 goto 1700

rem Command B
rem Set analog output
rem Command format:  >XXBcdddd??
rem                   ||||||-dddd = data 1 to 4 numbers
rem                   ||     - channel no. 0 or 1
rem                   |      - this command no

rem get the channel no.

2500 oc = asc($(1),5) - 48

rem Data starts at position 6 and could be 1-4 numbers long
rem Extract the last part of the string into $(4)

2510 od =str(0,$(1))     :rem get length of string
2520 od =str(7,$(4),$(1),6,od-7) :rem get only data

rem convert string number into usable number then output it
rem Check limits.  If out of range, then return error

2530 od = str(3,$(4))
2540 if (od < 0) .or. (od > 4095) then 1750
2550 if (oc < 0) .or. (oc > 1) then 1750

2560 aot oc,od

2590 goto 1700

rem Command C
rem Return counter value
rem Command format:  >xxCc??
rem                   |-counter number 0 or 1 (RPC-330)
rem                     (could be 4-11 also)
rem change limit check in 3010 for your card

rem get the channel no.

3000 oc = asc($(1),5) - 48

3010 if (oc < 0) .or. (oc > 1 ) then 1750

3020 oc = count(oc)

rem convert number to a string and output
```

```
3030 oc = str(10,$(2),0,oc)


rem Force letter A to first spot.  This is a space as set by format above

3040 asc($(2),1) = 65

rem output string as it is

3050 goto 1510

rem Command D
rem Send string to display
rem Command format:  >xxDCheck station 2??

rem NOTE:  Position is set by another command (exercise left to
rem       the student)

rem Extract the string to display

3500 oc = str(0,$(1)) : rem get length
3510 oc = str(7,$(4),$(1),5,oc-6)

3520 display $(4)

3530 goto 1700

rem Set power up acknowledge flag (OFLAG)

4000 oflag = 1
4010 goto 1700


rem Command F
rem General status of card
rem Syntax: >xxF??
rem Returns An
rem  Where n = code or codes of system.  0 = all ok
rem variable STATUS is global and indicates system status

4500 oc = str(10,$(2),0,status)

rem Force letter A to first spot.  This is a space as set by format above

4510 asc($(2),1) = 65

rem optionally clear STATUS flag

4520 status = 0

rem output string as it is

4530 goto 1510

rem Emergency or safety shut down
rem Set lines as appropriate here
rem Do not return an acknowledge as message applies to all
rem cards on network

5000 rem shut down code here

5200 return
```

APPENDIX B - Modem example program

File: MODEM.BAS

```
rem Modem communication program
rem Based around BASIC-52 software for RPC-320, 330

rem General operation.
rem This program is designed as a receiver.  Dialing out from
rem a modem is simply a matter of sending a ATDT <phone no>
rem command and responding appropriately to whatever is dialing.
rem Additional cycles (explained later) must be added to handle this

rem Receiving is a matter of going through a series of steps, or
rem cycles.  The first cycle is to detect Ringing message.  Then
rem Connect <baud>.  After that, a password is entered.
rem Line 2300 sets the password.  After 3 fails, it hangs up.

rem Commands are then processed.  Processing is done as part of
rem the main loop rather than in the interrupt.

rem Commands are processed at line 2400.  At this point the card
rem could be treated as a network, processing commands.  A more
rem sophisticated command handler is in the RS-485 demo program. (Appendix A)

Rem ONTICK acts as a communication timer.  Should there be a
rem period of inactivity while the modem is on line, it issues a
rem hang up command to the modem.  Timeout for this example is
rem 10 seconds.  It is controlled by the variable CTIM

rem The program is designed so that on a communication problem,
rem it will hang up and reset the modem.  The OK string from the
rem modem is treated as a "ready to receive" indication from the
rem modem.  If no OK is received, it will go through a hang-up-
rem reset process every 10 seconds until it receives one.
rem The NOKFL variable is set to 1 if no OK message is received.
rem This is read by the main loop since what to do with an inoperative
rem modem is application dependent.

rem If a NO CARRIER message is received from the modem after connecting,
rem the modem will be reset.  If you expect to ever send this string over,
rem modify the program at lines 1500+ else the modem will be reset.

rem Some modem messages such as NO DIALTONE, BUSY, and NO ANSWER are not
rem processed since these are outgoing dependent.  However, they can be
rem processed by adding CYCLEs.

rem To run this program "as is", you should have 2 PC's available.
rem This program has DEBUGging print statements throughout.
rem They may be removed as required.
rem One is connected to the card, the other to a phone line through
rem a modem.  Configure the modem per the RPBASIC software manual.
rem Connect a modem to an RPC-320, 330,
rem or other software compatible card (one that recognizes ON COM$)
rem to COM 1.  Don't forget to put a null modem adapter between the
rem modem and card.  Connect a PC or other such device to COM 0.
rem Download this program.

rem Connect the modem to a phone line.  Run your other PC's modem
rem program.  Run this program on the RPC card.  You will see initialization
rem messages and status displayed.  You should see RD and SD lights blinking
rem on the external modem.  What you are looking for is

rem  cycle = 0  atim = 0

rem on the bottom of the screen.  Dial up the RPC card from the other PC.
rem You should see a progression of messages such as RING, CONNECT
rem and the CYCLE count will increase.  Pay attention to your dial up
rem PC.  You should see a short sign on message and a prompt for a
```

```
rem password.  Enter 'password'.  Use lower case.  The password is
rem set at line 2300.

rem You are then prompted for a command.  Commands for this demo are
rem prefixed with  '>03'. The command is a number following the '3'.
rem To return the current analog reading on channel 0, type '>030'
rem You will probably get 0 if there is no voltage on channel 0.
rem To return the status at line 0, type '>031'.  You will probably
rem get a 1.  A '>032' will disconnect from the line.

rem If you do nothing, the modem will reset by the time  atim = 9
rem as printed on the screen.  When that happens, the modem disconnects
rem and resets.

rem Other things to consider.
rem If you are going to be sending out data for long periods of time,
rem be sure to change the variable ctim or reset atim periodically.
rem This program is designed to hang up if there is inactivity for a period
rem of time.  Default is 10 seconds.

rem CYCLE 4 is the hang up/reset modem cycle.  When something sets this cycle
rem in motion, nothing in this program can get it out.  CYCLE 4 starts by
rem assuming that nothing is being transmitted out.  It does wait a period of
rem time to ensure the time dependent escape sequence gets recognized.
rem A potential problem is in downloading information.  Running at 1200 baud,
rem characters are sent out at about 120 characters/second.  If you are sending
rem out lots of data, chances are the serial buffer in the card will get full.
rem At this baud rate, it will take about 2 seconds to empty.  If you go into
rem CYCLE 4 right after a data dump, the escape sequence will not be recognized
rem immediately.  Since CYCLE 4 keeps trying to reset the modem, it will
rem eventually reset it.  In the mean time, you may get "strange" data on
rem the receiving end.

rem This program was moderately tested.  It recovers from no connects,
rem disconnects, and modem power off/on conditions fairly well.  Keep
rem in mind each modem tends to operate a little differently and some
rem adjustments might have to be made.  The biggest problem we had was
rem in "dead" times.  Manufactures claim they need 1 second of dead time
rem before sending the escape sequence, but we found one needed more.  Also
rem you may need to pause a little longer after getting the CONNECT message
rem before sending out a sign on message.
rem We used USR, Practical Peripherals, and "no name" modems.

rem variable definition

rem cycle = communication cycle counter
rem mcycl = main loop multi tasking cycle
rem flag(n) = main task dispatcher flag
rem atim = actual time since last communication (in seconds)
rem ctim = commanded time for timout
rem htim = hang up/reset timer
rem $(0) = input string from com 1 buffer
rem $(1) = working search string in com 1 interrupt routine
rem $(2) = NO CARRIER string constant
rem cia = Communication Interrupt variable A - working variable
rem passw = pass word tries
rem okflg = flag to indicate OK was received   1 = got it
rem nokfl = flag to indicate OK was NOT received 1 = not got it
rem The main loop looks at nokfl and resets it.  Reason being is
rem modem may be bad, not powered, or not connected.
rem Application requirements dictate what to do in case of a bad
rem modem.  This flag is reset by the main loop.  This program
rem is set up to continuously try to reset the modem.
rem nocfg = no carrier flag  1 = "NO CARRIER" string found

rem Variable root tim was used because time is a key word.

rem initialize strings, arrays, interrupts

10 config baud 1,5,0 :rem 1200 baud for this example
```

```
20 string 1000,50:rem 20 strings, 50 bytes
30 dim flag(15)  :rem flags for main task dispatcher
40 okflg = 0      :rem OK received flag
50 ontick 1,1000 :rem communication watchdog and system timer
60 on com$ 1,49,13,2000 :rem interrupt on 49 chars or <cr>
70 ctim = 10      :rem communication timeout if on line.
80 $(2) = "NO CARRIER"  :rem string constant
90 nocfg = 0      :rem no carrier flag
100 htim = 0
110 clear com(1) :rem get rid of any previous stuff

rem Send reset to modem
rem NOTE:   If you are allowing for downloads to the card,
rem         skip around line 150.  This can be done by
rem         checking for a flag set in expanded memory (segment 1)
rem         If it is set, then don't do card reset.

150 cycle = 4     :rem do modem reset

rem other initialization as needed by the program

rem Main program loop
rem This is a multi-tasker dispatcher.  It performs various tasks
rem as dictated by other interrupts or programs
rem the array FLAG is used to indicate a process should be performed

rem For this example

rem flag(0) = send back analog input channel 0 value
rem flag(1) = send back digital 0 value
rem flag(2) = hang up
rem flag(3) through flag(14) are used for other process functions
rem for this example, only the first 11 flags are processed.

200 for mcycl = 0 to 14
210 if flag(mcycl) = 0 then 300m when a 1, then do a process
220 if mcycl > 5 then 250

rem   mcycl =        0    1    2    3    4    5
230 on mcycl gosub 10000,11000,12000,13000,14000,15000
240 goto 300

rem   mcycl    =    6    7    8    9    10
250 on mcycl-6 gosub 16000,17000,18000,19000,20000
260 goto 300

rem do mcycl 11-14 or more here

300 next

rem DEBUG

400 print "cycle =",cycle,"  atim =",atim,cr,

rem if there are other tasks that have to be done, then do them here

500 goto 200

rem ONTICK processing
rem Communication timeouts checked
rem if on line, some communication must be received in 10 seconds
rem Exception processing is:  Hang up (waits 3 seconds)
rem                           Long data send (ctim set longer)

rem If you need to do other things, then add them as needed.

rem Gosub to routine based on current cycle
rem Cycles are:
rem 0 = waiting for RING  1400
rem 1 = looking for CONNECT   1500
```

```
rem 2 = looking for password.  If ok send log on.  If not, tell user 1500
rem 3 = looking for command.  If ok, set MCYCLE.  If not, tell user 1500
rem 4 = Send esc, look for OK, send hang up, look for OK, send reset
rem      look for OK 1600
rem 5 = Send out sign on message after a few seconds delay

rem        cycle =    0   1   2   3   4   5   6   7
1000   on cycle gosub 1400,1500,1500,1500,1600,1900

rem other ONTICK stuff

1390 reti

rem cycle 0 waiting for ringing
rem This is idle.  No checking is done

1400 return

rem    Cycle 1, 2, or 3

rem Looking for CONNECT, password, or command.
rem Look for NO CARRIER flag.  If set, then reset modem
rem Check 10 second counter atim and compare with ctim

1500 atim = atim+1  :rem the '1' is changed based on current ON TICK time
1510 if nocfg = 1 then 1550 : rem if no carrier, reset all
1520 if atim < ctim then return

rem no communication received  Hang up and reset modem

rem DEBUG

1540 print : print "no CONNECT, password, or command in time"


1550 cycle = 4
1560 htim = 0
1570 nocfg = 0
1590 return

rem    cycle 4

rem Wait 2 seconds, send esc, look for OK, send hang up
rem look for OK

rem ATIM value is used to determine what part of cycle
rem 2 seconds are allowed for each step
rem first wait   htim=0
rem send esc htim = 2
rem look for OK, send hangup  htim = 5
rem look for OK, send reset   htim = 8
rem look for OK.  got into cycle 0  htim = 11

1600 htim = htim + 1
1610 if htim = 2 then 1650
1620 if htim = 5 then 1700
1630 if htim = 8 then 1750
1635 if htim = 11 then 1850
1640 if htim > 12 then htim = 0 : return :rem if really large, then reset
1645 return : rem if none of the above

rem send out escape sequence.  Look for OK

1650 uo 1
1660 print "+++",
1670 uo 0
1680 okflg = 0    : rem reset flag

rem DEBUG
```

```
1685 print : print "Sent +++"

1690 return

rem look for OK
rem If have it, send hang up
rem If not, set flag (nokfl) and continue as modem could have
rem been hung up on and lost carrier
rem Send out hang up command any way

1700 uo 1
1710 print "ATH0"
1720 uo 0
1730 nokfl = not(okflg).and.1 : okflg = 0

rem DEBUG

1735 print : print "Sent ATHO."

1740 return

rem Look for OK (must have it).  If not there, reset htim=0
rem nokfl set to alert system, and redo cycle
rem Send out reset string to modem.  This is a simple one.

1750 if okflg  = 0 then htim = 0:nokfl = 1: return
1760 uo 1
1770 print "ATZ"
1780 uo 0
1790 okflg = 0

rem DEBUG

1795 print : print "Sent ATZ"

1800 return

rem Look for OK (must have this one also).  If not there, reset
rem htim =0 and redo cycle
rem clear COM(1) to flush out any other erroneous data

1850 clear com(1)
1860 if okflg = 1 then cycle = 0  : return
1870 htim = 0
1890 return

rem    Cycle 5  tick processing
rem Send out sign on message after 3 seconds of waiting

1900 htim = htim + 1
1910 if htim < 3 then return

rem print sign on message and request password

1920 uo 1
1930 print "Remote Processing modem demo"
1940 print "Enter password..."
1950 uo 0

rem DEBUG

1955 print : print "Printed RPC sign on message"

1960 clear com(1)

1970 cycle = 2
1980 atim = 0
1990 return
```

```
rem ON COM$ processing


rem get current input

2000 $(0) = com$(1)
2010 atim = 0 : rem if anything came in, reset actual com time

rem ignore any <cr><lf>.  Check for lf

2020 if str(0,$(0)) = 0 then return

rem if first character is lf, then filter it out

2030 if asc($(0),1) <> 10 then 2060
2040 cia = str(7,$(0),$(0),2,str(0,$(0))-1)  :rem get rid of <lf>
2050 goto 2020:rem check for any length of string

rem DEBUG

2060 print : print "Received string:",$(0)," Cycle=",cycle

rem Check for NO CARRIER string.  If there, then set flag and
rem continue.  Other parts of program may use flag

2070 cia = str(8,$(0),$(2))
2075 if cia > 0 then nocfg = 1

rem process according to current cycle
rem CYCLE is defined as follows:
rem 0 = waiting for RING
rem 1 = looking for CONNECT
rem 2 = looking for password.  If ok send log on.  If not, tell user
rem 3 = looking for command.  If ok, set MCYCLE.  If not, tell user
rem 4 = Send esc, look for OK, send hang up, look for OK, send reset,
rem      look for OK  This routine just looks for OK
rem 5 = send out sign on message after 2 second delay for CONNECT

rem     cycle =    0    1    2    3    4    5    6
2080 on cycle gosub 2100,2200,2300,2400,2500,2600

2090 return

rem check if RING message.  If so, then  set cycle for 1

2100 $(1) = "RING"
2120 cia = str(8,$(0),$(1))
2130 if cia = 0 then return  :rem if something else, just ignore it
2140 cycle = 1

rem DEBUG

2150 print "Got RING.  To cycle 1"

2190 return

rem       cycle = 1

rem check for CONNECT message
rem if not, hang up by setting cycle 4
rem if CONNECT, then wait before sending sign on

2200 $(1) = "CONNECT"
2210 htim = 0
2220 cia = str(8,$(0),$(1))
2230 if cia > 0 then 2270
```

```
2240 cycle =4



rem DEBUG

2255  print "No CONNECT received.  Input string=",$(0)

2260 return

rem hold off any xmission for 3 seconds before sending sign on

2270 cycle =5
2280 passw = 0

2290 return


rem       cycle 2

rem Looking for password.
rem If tried 3 times, hang up

2300 $(1) = "password"
2310 cia = str(8,$(0),$(1))
2320 if cia > 0 then 2350
2330 passw = passw + 1
2335 uo 1 : print "Invalid password.  Re-enter" : uo 0
2340 if passw = 3 then cycle = 4 : htim = 0 : passw = 0
2345 return

rem check on length to make sure its all correct

2350 if str(0,$(0)) <> str(0,$(1)) then 2330

rem successful log in.  Tell user to put in valid command

2360 cycle = 3
2370 uo 1
2380 print "Password accepted.  Enter command"
2390 uo 0
2395 return

rem       Cycle 3

rem Process a command.  If valid, set flag(n)

rem To make sure no erroneous data looks like a command, all commands are
rem prefixed with ">03".  Idea is the likely hood of 4 random characters
rem making a valid command is unlikely compared to just 1

2400 $(1) = ">03"
2410 if str(8,$(0),$(1)) <> 0 then 2450
2420 uo 1 : print "Invalid command.  Re-enter"
2430 uo 0
2440 return

rem command is number in 4th position
rem Line 2460 checks for valid command limit

2450 cia = asc($(0),4)-48
2460 if (cia < 0) .or.(cia > 2) then goto 2420
2470 flag(cia) = 1 : rem indicate do this

2490 return
```

```
rem     cycle 4

rem Look for OK
rem If have OK, then reset cycle to 0
rem If message is not OK, simply leave

2500 $(1) = "OK"
2510 if str(8,$(0),$(1)) <> 1 then return

rem got OK
rem Signal system and let tick timer do next

2520 okflg = 1

2590 return


rem     cycle 5
rem send out sign on message after 1 second delay
rem Clear out COM if got here

2600 clear com(1)
2610 return

rem mcycl 0 processing from main loop
rem send back analog channel 0 to modem

10000 uo 1
10010 print ain(0)
10020 uo 0
10030 flag(0) = 0
10090 return


rem mcycl 1 processing
rem Send back digital status from line 0

11000 uo 1
11010 print line(0)
11020 uo 0
11030 flag(1) = 0
11090 return

rem mcycl 2 processing
rem hang up

12000 htim = 0
12010 cycle = 4
12020 flag(2) = 0
12030 uo1
12040 print:print "Hanging up"
12050 uo 0
12090 return
```

APPENDIX C- ERROR MESSAGES

The RPBASIC-52 error processor helps identify errors.

When running a program, error messages are expressed as:

```
ERROR: XXX - IN LINE NNN

NNN Instruction
_____X
```

where XXX is the type of error and NNN is the program line number where the error occurred. The "_____X" identifies the very approximate location of the error. For example, a BAD ARGUMENT error occurring at line 100 is expressed as:

```
ERROR: BAD ARGUMENT - IN LINE 100
100    DBY(257)=5
_____X
```

In Command mode, only the error type is printed since there are no line numbers in Command mode.

RPBASIC-52 errors include:

```
A-STACK
ARITH. UNDERFLOW
ARITH. OVERFLOW
ARRAY SIZE
BAD ARGUMENT
BAD SYNTAX
C-STACK
CAN'T CONTINUE
DIVIDE BY ZERO
I-STACK
MEMORY ALLOCATION
NO DATA
HARDWARE
```

A-STACK

The argument stack pointer is out of bounds. Too many expressions were pushed or tried to pop non-existent data off the stack.

ARITH. UNDERFLOW

The result of an arithmetic operation is beyond the lower limit of RPBASIC-52 floating-point numbers. RPBASIC-52's smallest floating-point number is $\pm 1E-127$. An operation such as 1E-100/1E28 would cause an ARITH. UNDERFLOW error.

This example produces a correct result:
```
>?1e-100/1e26
 1.0 E-126
```

This exam ple produces a n expected e rror:

```
?le-100/le28
ERROR: ARITH. UNDERFLOW
READY
```

This exam ple produces a n incorrect expo nent:

```
>?le-100/.9e28
 1.1111111 E-0
```

ARITH. OVERFLOW

The result of an arithmetic operation exceeds the upper limit of RPBASIC-52 floating-point numbers. RPBASIC-52's largest floating-point number is ±.99999999E+127. An operation such as 1E100*1E28 causes an ARITH. OVERFLO W error.

ARRAY SIZE

An array was accessed that is outside the dimension boundaries defined by a DIM instruction. For example:

```
DIM A(100)
PRINT A(102)

ERROR:  ARRAY SIZE
READY
```

BAD ARGUMENT

The argument of an operator is out of limits. For example, A=AIN(300) generates a BAD ARGU MENT error since the value assigned by the AIN operator is limited to the range 0 to 7.

BAD SYNTAX

An invalid co mma nd, instruction, or operator or ha ve attemp ted to use a reserve d key word as part of a variable was entere d. This is a generic "I don't know w hat this is" responce by a com puter.

C-STACK

More control stack memory was used than it has available. The control stack has of 158 byte of memory. A FOR-NEXT loop uses 17 bytes, and DO-UNT IL, DO-WHILE, and GOSUB each use three bytes. This means you limited to nine FOR-NEXT loops. Executing a return before a GOSUB, or a WHILE or UNTIL before a DO instruction, or a NE XT befo re a FOR also generates a C-STA CK error.

CAN'T CONTINUE

A program was edited after stopping.

DIVIDE BY ZERO

A num ber was divid ed by zero or a sta tement suc h as TAN (PI/2).

I-STACK

There is not enough internal stack space to evaluate an expression. Usually this is caused by an excessive number of parentheses.

MEMORY ALLOCATION

Accessing a string that is outside the defined string limits or assign MTOP a value that does not contain any RAM.

NO DATA

A READ instruction does not have valid associated DATA instruction. NO DATA - IN LINE XXX error message displays a line number where it expected to find the data.

APPENDIX D - Data storage

STRING STORAGE

BASIC-52 stores string variables between MTOP and top of variable space, call VARTOP. String $(0) would be stored from VARTOP to [VARTOP + (bytes_per_string + 1)]. String $(1) is stored from [VARTOP + (bytes_per_string + 2)] to [VARTOP +2 * (bytes_per_string + 1)], and so on.

All strings are terminated with a carriage return (0DH, 13 decimal).

VARIABLE STORAGE

Scaler variables are numbers not in a dimension. Dimensioned or arrayed variables (commonly referred to as "arrays") are those whose identifier includes a single-dimensioned expression.

Scaler variables: PART, A1, B

Dimensioned variables: TEMP(5), PRESS(A)

Scalars are stored starting at VARTOP-1, with storage growing down at eight bytes per variable.

FLOATING-POINT FORMAT

RPBASIC-52 stores all floating-point numbers in a normalized packed binary-coded decimal (BCD) format. All numbers are normalized, so the most significant digit in a floating-point number is never zero unless its actual value is zero.

To demonstrate the floating-point format, see how RPBASIC-52 stores 12345678.

LOCATION	VALUE	DESCRIPTION
X	88H	exponent: $81H = 10^1$, $80H = 10^0$, $7FH = 10^{-1}$, etc. Zero is represented by a zero exponent
X-1	00H	sign bit: 00H = positive, 01H = negative Other bits are temporary values used only during calculations
X-2	78H	least significant two digits
X-3	56H	next least significant two digits
X-4	34H	next most significant two digits
X-5	12H	most significant two digits

So we have .12345678 X 10^8 which is 12345678.

The POKEF command stores numbers in RAM in this same format. PEEKF expects to read a number in this format.

APPENDIX E - Software revision history

;V1.02   added
;   24 key keypad scanning
;   Took out BELL when backspacing beyond beginning of line
;   Took out extra CRLF when entering in just a CR for a command.

;V1.03   added
;   CARD function
;   CONFIG LINE 100 only now re-initializes port without writing to serial EEPROM.

;V1.04   Changed
;   Release for RPC-320

;V1.05   Fixed
;   BSAVE returned a hardware error when verify was bad. In fact,
;   save was ok. Caused by RAM and EPROM pointers getting swapped

;V1.06   Fixed
;   LCD graphics hardware CS and reset are reversed in RPC-320. Compensated
;   in software.

;V1.07   Fixed
;   MTOP was useless in any system, especially a 32K RAM.
;   In 32K RAM system, MTOP = 7DFF. This will give user 512
;   bytes of free RAM. 128K and 512K RAM versions not affected.
;   STR(6,...) broken. Was not popping stack.

;V1.08   Fix
;   Variables E and F would get dropped if followed by a space
;   Changed token table in MAIN1 and 320_MA20 to add bogus token
;   and command name.
;   Added delays (nop's) between data strobe writes to LCD display to compensate for faster CPU
;   Changed both LCD4x40 and LCD4x20 assembly files

;V1.09   Fix
;   STR(7, ...) did not put in a CR into the put string, causing
;   longer strings to be printed.

;V1.10   Initial release for RPC-330
;   Added (330 only)
;   AOT command
;   COUNT function and command for added counter
;   added ON COM, ON COUNT, ON LINE, ON KEYPAD for RPC-320, RPC-330

V1.11   11/29/95
        Added day of week to DATE command and function

V1.12   12/01/95
        Added code to use Atmel 29C040A type flash

V1.13   01/12/96
    Added code to support IEE centry series display (3602-100-05420)
    Includes CONFIG DISPLAY 4
    Added PRINT #port

V1.14 03/28/96
    Fixed bug in ON COUNT. Returns error for lines > 100

V1.15 06/26/96
    PEEK$ could cause basic to lock up under right conditions.

V1.16 02/18/97
    ON LINE OFF could cause program to lock up if running ON COM.
    Syntax error when DISPLAY used with IF-THEN-ELSE.
    Added PEEKF and POKEF commands.

V1.17 04/16/97
    Fixed keypad debounce. Speed up by about 1%.

V1.18 08/05/97
    PRINT sends a CRLF sequence seemingly at random when printing from both ports and trying to print a variable.

V1.19 12/01/98
    Added FREQ and CONFIG FREQ.

V1.20 08/18/99
    Pointer to baud rate table not getting set properly

V1.21 11/25/00
    Added SPI in and out commands