As you are now the owner of this document which should have come to you for free, please consider making a donation of £1 or more for the upkeep of the (Radar) website which holds this document. I give my time for free, but it costs me money to bring this document to you. You can donate here https://blunham.com/Misc/Texas

Many thanks.

Please do not upload this copyright pdf document to any other website. Breach of copyright may result in a criminal conviction.

This Acrobat document was generated by me, Colin Hinson, from a document held by me. I requested permission to publish this from Texas Instruments (twice) but received no reply. It is presented here (for free) and this pdf version of the document is my copyright in much the same way as a photograph would be. If you believe the document to be under other copyright, please contact me.

The document should have been downloaded from my website https://blunham.com/, or any mirror site named on that site. If you downloaded it from elsewhere, please let me know (particularly if you were charged for it). You can contact me via my Genuki email page: https://www.genuki.org.uk/big/eng/YKS/various?recipient=colin

You may not copy the file for onward transmission of the data nor attempt to make monetary gain by the use of these files. If you want someone else to have a copy of the file, point them at the website. (https://blunham.com/Misc/Texas). **Please do not point them at the file itself as it may move or the site may be updated.**

It should be noted that most of the pages are identifiable as having been processed by me.

I put a lot of time into producing these files which is why you are met with this page when you open the file.

If you find missing pages, pages in the wrong order, anything else wrong with the file or simply want to make a comment, please drop me a line (see above).

It is my hope that you find the file of use to you.

Colin Hinson
In the village of Blunham, Bedfordshire.

TEXAS INSTRUMENTS

TM990

E155 Euroboard POWER BASIC

MICROPROCESSOR SERIES™

Manual Addendum

TM990/E155 EUROBOARD POWER BASIC

MANUAL ADDENDUM

This addendum in conjunction with the 'TM990 Power BASIC Reference Manual' (MP308) constitute the Reference Manual for TM990/E155 Euroboard Power BASIC.

Texas Instruments Ltd March 1982

IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time in order to improve design and supply the best product possible.

Information contained in these publications is believed to be accurate and reliable. However, responsibility is assumed neither for its use nor for any infringement of patents or rights of others that may result from its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

INDEX

INDEX

1 INTRODUCTION

Euroboard Power BASIC has been specially developed from TI's Development Power BASIC to exploit the advanced features of the high-speed TMS9995 microprocessor (such as the internal RAM and the on-chip decrementer) on the TM990/E155 microcomputer board, and to allow access to the capabilities of the TM990/E board range of microcomputer modules.

While supporting all the facilities contained in Development Power BASIC with the Software Enhancement Package (see section 1 of the Power BASIC Reference Manual for a description of these products), plus a number of new features, this version of the interpreter runs approximately 75% faster than Development Power BASIC (already one of the fastest BASICs commercially available).

This addendum gives an overview and operation guide for Euroboard Power BASIC, and a detailed description of those features which are different from Development Power BASIC. Reference is made to the Power BASIC Reference Manual (MP308) for a detailed explanation of common features. Section 1.7 below is an introduction to the Power BASIC language, for those unfamiliar with BASIC.

1.1 List of Features

o indicates a new feature.

* indicates a modified feature.

The numbers indicate the section that deals with the particular command/statement/function/operation. If the section number is preceded by either 'o' or '*' then those sections appear in this addendum, otherwise they refer to sections in the Power BASIC Reference Manual.

To clear screen — ; " <0C> " OR ; " <03> "
Rand. numbs. — 4 = INP(RNDX 20)+1
Command; say

Commands

CLEar	o	3.5	CONtinue		4.2	LISt	4.3
LOAd	*	3.1	NEW	*	3.4	PROgram	* 3.3
RENumber	o	3.2	RUN		4.7	SAVe	* 3.1
SIZe		4.9					

Statements

BAUD		5.8.4	BASE		5.10	CALL	* 3.7
DATA		5.7	DEF		5.4	DIM	5.3
ELSE		5.6.2	ENABLE	o	3.8.1	END	5.6.8
ERROR		5.6.6	ESCAPE		5.13	FOR	5.6.5
GOSUB		5.6.3	GOTO		5.6.1(G1)	IF	5.6.2

IMASK	* 3.8.2	INPUT	5.8.1	IRTN	5.9.3
LET	5.6.1	NEXT	5.6.5	NOESC	5.13
ON	5.6.4	POP	5.6.3	PRINT	5.8.2
RANDOM	5.12	READ	5.7	REM	5.2
RESTOR	5.7	RETURN	5.6.3	STOP	5.6.7
TIME	5.11	TRAP	5.9.2	UNIT	5.8.3

Functions

ABS	7.2.1	ADR	3.6	ATN	7.2.2
BIT	7.4.7	COS	7.2.3	CRB	7.4.1
CRF	7.4.2	EXP	7.2.4	INP	7.2.5
LOG	7.2.6	MEM	7.4.6	MWD	7.4.9
NKY	7.4.3	RND	7.4.8	SIN	7.2.3
SQR	7.2.7	SYS	7.4.4	TIC	7.4.5
TAB	5.8.2.2				

String Operations

ASC	7.3.1	Assign	6.2	Compare	5.6.2
Concat	6.3	Convert	6.9/10	Delete	6.7
Insert	6.6	LEN	7.3.2	MCH	7.3.3
Pick	6.4	Replace	6.5/8	SRH	7.3.4

The following sections of the Power BASIC Reference Manual refer to other members of the Power BASIC family. You may wish to delete these sections to avoid confusion.

All of section 2, except for Figure 2-12
3.12
4.4 to 4.6
4.8
5.9
5.14

1.2 Summary of Features

Apart from the standard features of BASIC (LET, GOTO, GOSUB, etc), Euroboard Power BASIC allows the user to access control equipment in real time (timing is provided by the TIC function) by either memory-mapped I/O (MEM and MWD functions) or via TI's standard bitwise Communications Register Unit (BASE statement, CRB and CRF functions). Further, it also allows the user to write interrupt service routines in the Power BASIC language and to associate each of these routines to a particular interrupt level (TRAP, IMASK, ENABLE and IRTN statements). Previously written assembly language routines (either burnt into EPROM or built in RAM using the MWD function) can be invoked from an application program and have full access to the Power BASIC variables (CALL statement).

Euroboard Power BASIC provides full character handling

facilities (character assignment, replacement, insertion, deletion, pick, comparison and concatenation operators, plus character search, match and conversion functions), better control structures (ELSE, ON and ERROR statements), varied print formats (with hexadecimal and full decimal formatting), complete error message reporting, and floating point arithmetic (with 11 significant digits).

Applications programs can be loaded from (LOAD command) or saved to (SAVE command) the following: ASR733 digital cassettes; 763/765 bubble memory terminal files; or low cost audio cassettes.

The RENumber command causes the complete stored program to be renumbered (to allow a number of additional statement lines to be inserted into the program) and automatically updates the program to take account of the new line numbers.

A fully tested and debugged application program can be burnt into TMS25xx EPROMs (PROgram command). With these EPROMs installed in a target system, the application program can either be automatically executed on system power-up (by locating the EPROMs at a start address of hex 4000) or as and when required using the "address" option of the LOAD command.

Audio cassette operation and EPROM programming require additional special purpose hardware.

1.3 Hardware Requirements

The minimum hardware configuration to run E155 Power BASIC is as follows:

E-bus chassis and backplane
Power supply
TM990/E155 cpu module
TM990/E358 EIA interface module
RS-232-C compatible terminal and cable

1.4 Installation

The TM990/E155 cpu board must be configured as follows:

EPROM Select (jumpers J2 and J5):

J2	E6 - E7	EPROM type = TMS2564
J5	E15 - E16	EPROM type = TMS2564

ROM Speed Select (jumper J3):

J3	E8 - E9	One wait state

Automatic Wait State (jumper J4)

J4	E12 - E13	One wait state

The Euroboard Power BASIC interpreter consists of two TMS2564 EPROMs that are installed on the E155 cpu module. The EPROM marked "U7" must be installed in socket U7, and the EPROM marked "U6" must be installed in socket U6. With the EPROM held so that the small nick in the casing at one end is at the top, pin 1 of the EPROM is in the top left hand corner. The EPROM should be installed so that pin 1 corresponds to the small "1" marked on the circuit board. As a check, the small nick in the EPROM will then be at the end closest to the E-Bus connector.

The EIA card should be configured for a CRU base address (R12 contents) of hex 80; if a second EIA card is to be included in the system, it must be configured for a CRU base address (R12 contents) of hex 180.

The E358 EIA card is configured as follows:

Interrupt Request (J1):

J1	E3 - E4	Asserted to INTEN-

Current Loop/RS-232-C (J2, J7 and J8):

J2	E7 - E8	RS-232-C
J7	E22 - E21	Current loop circuit disabled
J8	E25 - E24	RS-232 protective ground

Bus Time-out Processor (J3 and J6):

J3	E10 - E9	Processor ON
J6	E19 - E18	Generates READY-

Reset Switch (J4):

J4	E13 - E12	Connected to PRES-

BUSCLK (J5):

J5	E16 - E15	BUSCLK- asserted to TMS9902

The CRU address switch should be set to hex 80 as follows:

CRU Address Switch:	S1	S2	S3	S4	S5	S6
	Off	On	On	On	On	On

A second EIA card can be installed in the system, configured exactly as above, except that the CRU address switch must be set for hex 180:

CRU Address Switch:	S1	S2	S3	S4	S5	S6
	Off	Off	On	On	On	On

1.5 Memory Expansion

A fully populated /E155 board has on-board RAM from hex E000 to hex EFFF. The top 850 bytes of this, and all the TMS9995 on-chip RAM, is used by the interpreter for its workspaces, buffers, system variables, etc. The remaining 3.2K bytes of on-board RAM is available for storing a Power BASIC application program.

Additional RAM, in the form of memory expansion boards (eg /E250 and /E251 memory boards), may be included in the target system. However, for the Power BASIC interpreter to recognise that the additional RAM is available for its use, the memory boards must be configured so that the RAM is contiguous from hex DFFF down towards low memory. See the User's Guide for the appropriate board module(s) for details of how to configure memory at the required address.

Note: The /E155 cpu module requires all memory expansion boards in the system to generate the READY- signal. This is done by setting the READY-/AREADY- jumper (jumper J4 on the /E250 board, for example) to the READY- position.

Note: Addresses above the TMS9995's internal RAM (hex F0FC to hex FFF9) are reserved for memory mapped devices.

1.6 Operation

When the Power BASIC EPROMs have been installed and all jumpers correctly set, insert all the circuit boards into the chassis (with the power off). All boards should be inserted with the components on the right hand side. Press the boards firmly home, to ensure that all connections are properly made.

Connect an RS-232-C compatible terminal, set for FULL DUPLEX operation at any of the following baud rates (Power BASIC will automatically measure the correct baud rate when it is switched on):

110 300 1200※ 2400 4800 9600 19200

※ Note that at 1200 baud, printer output is padded with a time delay to reduce the effective transfer rate to 300 baud. Transfer to and from digital cassette tape occurs at the full rate of 1200 baud. This is compatible with the operation of the TI 733 ASR terminal.

Ensure that the terminal is switched on. Then switch on the power to the Euroboard chassis, press the reset switch on the E358 EIA card, and press the "RETURN" key (sometimes marked "carriage return" or "CR") on the terminal. Power BASIC will respond with the message:

```
EUROBOARD BASIC REV x.x.x
*READY
```

[If there is no response, and the procedure described above has been followed exactly, check that the terminal used provides a "Data Terminal Ready" (DTR) signal (HIGH on pin 20 of the EIA connector). If the terminal does not provide this signal, the cable should be wired to connect pin 6 of the EIA connector to pin 20.]

Euroboard Power BASIC is now ready to accept commands and program statements.

1.7 Introduction to the Power BASIC Language

This section gives a brief introduction to the Power BASIC language and how to use it. If you are already familiar with BASIC, you may wish to skip to section 2.

The following is best read with a Euroboard Power BASIC system available, set up and initialised as described above.

Type in SIZE followed by the "RETURN" key. Power BASIC will respond with details of the memory space allocated to program and variable storage, and the space free for new input:

```
SIZE
PRGM:012H BYTES
VARS:0H BYTES
FREE:0C94H BYTES
```

The figures are given in hexadecimal notation※; the actual numbers will depend on how much memory there is in the system.

SIZE is an example of a Power BASIC command. It is carried out immediately it is keyed in. Commands provide such functions as listing, storing, loading and executing Power BASIC programs. A full list of commands is given in section 4.2 of this Addendum.

Apart from commands, the other type of input that can be

※ See the Software Development Handbook (MPA29, 2nd edition), section 8.13.2.1, for a description of the hexadecimal number system.

entered at the keyboard is statements. A statement is an instruction to do something, but it is not carried out immediately. Type the following statement, which is an instruction to add two numbers together and print the result:

```
30 PRINT A+B
```

followed by "RETURN".

If you make a mistake typing, there are two things you can do: (1) press the "RUBOUT" key to backspace, one by one, over the characters that are wrong, and then type them again; or (2) press the escape key (marked "ESC"), and then type the whole line again.

Power BASIC will perform some checks on the line after you have entered it (ie, after you have pressed "RETURN"), and it may discover an error that you have overlooked. If so, it will print an appropriate error message, type the line again, and position the terminal cursor or printhead where it thinks the error is - so that you can type over it. Here is an example:

```
30 PRINT A.B
***ILLEGAL CHARACTER***

30 PRINT A.B
           ^
           |
   cursor or printhead positioned here
```

Overtype the mistake with the correct characters, and then press "RETURN". Again, you can use the RUBOUT key to backspace over the line, or press ESC and type the whole line again.

A statement is normally preceded by a line number ("30"), to distinguish it from a command. Statements with a line number are not carried out immediately; they are stored in memory for future use. Several statements can be entered and stored, and then executed together as a program. The line numbers determine the order in which the statements will be carried out (it does not matter in which order the statements were actually entered).

In the statement just entered, "A" and "B" are variables: they stand for values which can be set up, referred to, and changed by program statements. In fact, A and B represent storage locations in the computer memory.

The types of values that can be used in Power BASIC programs

are

integers (whole numbers 1, 2, 3, etc) up to 32,767 and down to -32,768.

floating point numbers (ie decimals such as 12.34 and 0.001234)

strings (ie sequences of characters enclosed in quotes - "THIS IS A STRING")

Variables are described more fully in section 2.7 below.

In fact, Power BASIC could not execute statement 30 as it is, because A and B do not yet have any values to add together or print. Variables must be given a value before they can be referred to. If we tried to execute statement 30 as it is, Power BASIC would print an error message:

UNDEFINED VARIABLE AT 30

Until a variable has been assigned a value, Power BASIC does not even know of the variable's existence, and is certainly not prepared to do any calculation with it.

To solve this problem, two more statements can be added to the program, with appropriate line numbers to ensure they are executed before statement 30:

```
10 A=5
20 INPUT B
```

Statement 10 is an assignment statement that introduces the variable "A" to Power BASIC, and assigns the value of 5 to it. Once this statement has been executed, A can be referred to in any statement and will have the value 5 (unless it has been changed in the meantime). Further assignment statements can be used to change the value of A.

Statement 20, when executed, introduces the variable B, and causes Power BASIC to ask for a value for it, and to wait while a number is entered at the keyboard.

Note that neither of these things has actually happened yet. What we have done is to input some statements, to form a program that will be executed at some future time, when we give Power BASIC the command to do so. Despite having entered the new statements, "A" and "B" are still undefined - and will continue to be undefined until we run the program.

To look at the program you have just entered, type

```
LIST
```

(followed by "RETURN"). LIST is another command, like SIZE. Power BASIC will list the program stored in its memory, in the order in which it will be executed:

```
10 A=5
20 INPUT B
30 PRINT A+B
```

If your listing does not look like this, simply re-type the line that is wrong. If you have entered a wrong line number, first type the wrong line number followed immediately by "RETURN" to erase the line, and then enter the correct line. LIST again to check.

So far, we have simply constructed a program - a list of statements - in Power BASIC's memory. To carry out (execute) this program, type the command

```
RUN
```

(followed by "RETURN").

Power BASIC will perform statement 10, assigning the value of 5 to the new variable A. (Of course, you will not observe anything while this is happening).

Halfway through executing statement 20, Power BASIC will output a prompt (?) to the terminal, and will wait for you to enter a value for the new variable B. Type a number, followed again by "RETURN". Power BASIC will add the values of A and B together and print the result, then return to wait for the next command. The whole sequence looks something like this:

```
RUN
? 256
 261

STOP AT 30
```

"256" is the value entered for "B"; Power BASIC has printed "261", which is 256 plus the value assigned to A - correct! Try RUN again, this time entering a different number - say one with a decimal point.

You can also change the program and re-run it, for example to set A to a different value, or to replace "A+B" with a different mathematical formula. (As with most computer languages, "*" is used for "multiply" and "/" for divide; "+" and "-" have the usual meaning. See section 3.8 of the Power BASIC Reference Manual for a full description of operators and expressions).

You can change, or edit, the program simply by retyping the appropriate line, or adding a new one with a suitable line number. Power BASIC also provides a simple line editor, that allows single characters to be changed, inserted or deleted without retyping the whole line. Section 2.4 below gives the commands available for editing single lines. Section 3.5 of the Power BASIC Reference Manual gives some examples of use of the line editor.

Note that the assignment statement "A=5" is not simply a statement of fact, as it would be in mathematics ("A equals 5"). In Power BASIC it stands for an action: "make A equal to 5". Thus a statement such as "A=A+5" is quite legal and meaningful in Power BASIC (provided A has already been assigned a value); try it, and see what it does.

The assignment statement is sometimes also called the LET statement in BASIC, because it can also be written (eg)

```
15 LET A=A+5
```

One useful thing to note is that a statement entered without a line number is not stored, but is carried out immediately, like a command. For example, if you want to know the internally stored value of a variable at any time, simply type (eg)

```
PRINT A
```

In fact, PRINT is used so frequently that it has a one-character abbreviation (namely, a semicolon). The above statement can also be written

```
;A
```

The following is a more complex program, which can be used to explore some of the additional features of Power BASIC. To prepare for entering this program, type the command

```
NEW
```

which will erase all statements and variables from Power BASIC's memory, ready for a new program. Then type the following:

```
10 DIM A(4)
20 $A(0)="THE NUMBER IS"
30 INPUT "INPUT NUMBER", N
40 IF N-INP(N)<>0 THEN PRINT $A(0);N;:: GOTO 60
50 GOSUB 100  ! EVEN OR INTEGER
60 PRINT ", ITS SQUARE IS";N*N;", AND ITS SQUARE ROOT IS";
70 IF N<0 THEN PRINT " UNDEFINED":: GOTO 30
80 PRINT SQR(N)
90 GOTO 30
100 IF INP(N/2)*2=N THEN PRINT $A(0);" EVEN";::RETURN
110 PRINT $A(0);" ODD";
120 RETURN
```

Correct any typing errors as described above. Ensure that you get all the punctuation exactly right. Power BASIC will find some errors - for example, if you leave out one quote mark (") it will be detected, because quote marks always come in pairs surrounding a text string - "THE NUMBER IS". Other errors may not be detected until you run the program.

LIST the program as a final check. You may notice that line 10, for example, is listed as

```
10 DIM A[4]
```

This is perfectly correct; Power BASIC makes no distinction between parentheses () and square brackets []. Your program is stored internally by Power BASIC in a condensed and encoded form, to take up as little space as possible. (For this reason, Power BASIC programs are very compact). When you enter a statement, a part of Power BASIC called the editor takes your input and converts it into this coded form. The editor is also responsible for translating the coded program back into an understandable form and printing it, when you type LIST. While you are entering, changing or listing program statements you are using the editor.

When you type RUN, another part of Power BASIC, called the interpreter, takes over from the editor. The interpreter fetches the stored program statements, one by one, and carries out the coded instructions. (Later on in this manual and in parts of the Power BASIC Reference Manual, "interpreter" is used to refer to the whole of the Power BASIC system including the editor.)

RUN the new program, and enter a number (followed by "RETURN") in response to the question mark prompt. Press the escape key ("ESC") to get out of the program (this will work at any time, even if Power BASIC is not waiting for input). The result should be something like this:

```
UN
INPUT NUMBER? 17
THE NUMBER IS ODD, ITS SQUARE IS 289, ITS SQUARE ROOT IS 4.1231056256
INPUT NUMBER? -6
THE NUMBER IS EVEN, ITS SQUARE IS 36, ITS SQUARE ROOT IS UNDEFINED
INPUT NUMBER? 2.35
THE NUMBER IS 2.35, ITS SQUARE IS 5.5225, ITS SQUARE ROOT IS 1.532970971
INPUT NUMBER? (escape key)
TOP AT 30
```

There are several things to note about this program:

- "A" is a string variable, which holds text rather than numeric values. Where a variable is used to store strings, its name is preceded by a "$" (lines 20, 40, 100, 110). String variables are described in section 2.7.5 below.

- A is also an array (sometimes known as a dimensioned variable), which means that it stands not for a single storage location, but for several storage locations grouped together in memory. Arrays are often used to store text strings, which usually require more space than is available in a single storage location; but they can also be used to store groups of numeric values. Statement 10 (the DIMension statement) declares to Power BASIC that A is going to be an array rather than an ordinary variable, and also declares how big it is. Arrays are described in section 2.7.6 below, and some additional information on string arrays is given in the Power BASIC Reference Manual, section 3.7.4. (When looking at the Power BASIC Reference Manual, note that the storage allocation of Euroboard Power BASIC corresponds to Development Power BASIC and not Evaluation Power BASIC - ie 48 bits and not 32 bits).

- The INPUT statement can specify a text prompt to be printed before the question mark (line 30). See the Power BASIC Reference Manual, section 5.8.1, for a description of all the available options of the INPUT statement.

- The IF .. THEN statement (line 40) allows conditional execution of an action. When interpreted, this statement means: "IF the number just entered is not a whole number THEN print the string stored in the array $A, followed by the number N, and go to line 60 for the next statement" (otherwise do nothing, and continue with line 50). "<>" means "is not equal to"; INP is described below. See the Power BASIC Reference Manual, section 5.6.2.

- It is possible to place more than one Power BASIC

statement on a single line, separated by "::" (lines 40, 70, 100). Each statement is executed, in order. This feature is particularly useful with IF .. THEN statements: all statements after the THEN, on the same line, will be executed if the condition is true, and not executed if it is false.

- Comments can be placed at the end of a statement line, to make the program clearer, by preceding them with "!". Everything after the exclamation mark is ignored by Power BASIC when it executes the statement (line 50). However, comments do take up storage space in program memory.

- Statements such as GOTO and GOSUB can be used to alter the normal flow of program execution (which is in order of increasing line numbers). The GOTO statement in line 40 will cause the Power BASIC interpreter to go directly to line 60 to find the next statement to be executed. The GOSUB statement in line 50 causes a similar transfer to line 100, but in this case Power BASIC "remembers" where it was; when the interpreter finds the RETURN statement in line 120, it goes back to the end of the GOSUB statement (line 50), and executes line 60 next. The GOTO is a permanent diversion of program flow; the GOSUB is a temporary diversion, which is expected to return to the main program. The statements 100 to 120, which are executed out of the main line of the program under control of the GOSUB statement, are called a subroutine. See sections 5.6.1 and 5.6.3 of the Power BASIC Reference Manual.

- The PRINT statement (lines 40, 60, 70, 100, 110) has a number of options, to allow output of text and layout of printed numbers on the page. (Try substituting "," for ";" in one of the print statements, and see what happens). See section 5.8.2 of the Power BASIC Reference Manual for a full description of PRINT.

- SQR (square root) and INP (integer part) are two examples of the standard functions available in Power BASIC. See section 7 of the Power BASIC Reference Manual. It is also possible to define new functions (Power BASIC Reference Manual, section 5.4).

- This program, as it is written, is an endless loop: there is no way to exit from the program except by pressing ESC.

By understanding and experimenting with this program, you will learn some of the most important features of Power BASIC. Section 2 below gives key information about BASIC, in a rather more formal and systematic way than is presented

above. Following that, there are descriptions of the individual commands, statements, and operations that make up the Power BASIC language. Features that are specific to Euroboard Power BASIC are described in this addendum; those that are common to all members of the Power BASIC family are described in the Power BASIC Reference Manual. Section 1.1 above has a complete list, with section numbers to refer to. In addition, section 4 below is a complete Quick Reference Guide to Euroboard Power BASIC.

Power BASIC is not limited to mathematical operations. In conjunction with input and output board modules such as the TM990/E350 and TM990/E351, Power BASIC's real time facilities can be used for control and monitoring in experimental, industrial and commercial applications. For bit-oriented input and output, the CRB and CRF functions and the BASE statement provide direct input and output of single bits or fields of information of any size from 1-16 bits, through the TMS9995 processor's Communications Register Unit (CRU)※. Memory mapped input and output is also provided through the MEM and MWD functions. In addition to the standard board modules in the TM990/E range, custom input and output modules can be constructed for special purposes. The E-Bus System Design Handbook (MP402) describes how to do this.

A Power BASIC program can be designed to respond to external inputs in two ways: through polling, and via interrupts. With polled input, the program is written to check (or poll) relevant input signals (using the CRB, CRF, MEM or MWD functions) at regular intervals, and to take appropriate action. With interrupts, certain input signals are wired up so that the appearance of a signal causes an immediate transfer (after the interpreter has finished executing the current statement) to an interrupt subroutine, which performs any necessary action to service the interrupt. When the interrupt subroutine has completed, Power BASIC returns to whatever it was doing before the interrupt occurred. Euroboard Power BASIC provides 15 prioritised interrupt lines that can be used in this way. Section 3.8 below describes Power BASIC interrupts.

Having written a Power BASIC program, you may wish to save it on tape cassette, or program it into EPROM for more permanent storage. Sections 3.1 and 3.3 below describe how to do this. Programs in EPROM can be arranged so that, on power up, Power BASIC will automatically begin executing the program, rather than printing the banner message and waiting for input. In this way, stand-alone systems can be constructed that will work unattended - perhaps controlling a piece of machinery - without even a terminal being

※ See the Software Development Handbook (MPA29, 2nd edition), section 8.9, or the TMS9995 Data Manual, for a description of the CRU.

connected if no terminal is required.

For special purposes, subroutines can be written in 9995 assembly language and called up from a Power BASIC program. Interrupt subroutines can also be written in assembly language, to make use of the 9995 hardware interrupts. The TMS9995 Data Manual or the Software Development Handbook (MPA29) describe 9995 assembly language; section 3.7 below describes how to call an assembly language program from Power BASIC, and section 3.8.3 describes assembly language interrupt handlers.

The Software Development Handbook is a source of further information on software development, including the design of software systems. It includes a description of assembly language for all 9900/99000 family processors, as well as high level languages and the software development process.

2 GENERAL PROGRAMMING INFORMATION

This section provides an overview of the characteristics of Euroboard Power Basic. Some additional information is contained in section 3 of the Power BASIC Reference Manual.

2.1 Power BASIC Interpreter Operation Overview

The interpreter allows the user to enter statements direct from the terminal keyboard and to either immediately execute the statement (if no line number is given) or to store the statements as part of a program. As soon as a statement is entered, the interpreter converts it into a compressed and encoded form (to save user RAM) and in the process performs a certain amount of syntax checking (ie does the statement conform to the language specification). During program entry, the interpreter is able to detect when certain errors have been encountered; such as mismatched parentheses/quotes, or missing "THEN" with an IF or ON statement. (However, as the interpreter does not perform a full syntax check at this time, it is possible to enter and store a syntactically incorrect statement line. Any such statement lines will be discovered, and reported, when execution mode is invoked.)

Detected errors are immediately reported, and until the statement is corrected the line can not be stored nor executed. The necessary modifications to the statement line can be made using the "edit keys" described below (section 2.4).

What happens next depends on which of the following two modes the interpreter is in:

- KEYBOARD MODE is automatically entered when Power BASIC is initialised. In this mode, entering a numbered line causes that line to be stored in the appropriate place in the program space. Entering an unnumbered line causes the statement(s) to be immediately executed and keyboard mode to be re-entered as soon as the necessary processing has been performed.

- EXECUTION MODE is entered by issuing either a RUN, a CONT or a GOTO statement. This causes the Power BASIC interpreter to execute the previously stored program. RUN starts at the lowest line number in the program; CONT continues from the last line that was previously interpreted; GOTO proceeds from the line specified. This mode is terminated by any one of the following conditions:

- Error condition arising.
- STOP or END statement executed.
- Pressing the terminal's ESCape key.

Note: There are a number of statements which can only be issued in keyboard mode (these are referred to as commands).

2.2 Power BASIC Program

A Power BASIC program consists of a number of statement lines, with each line preceded by a line number (an integer number in the range 1 to 32767). This line number indicates the sequence that the statement lines are to be ordered into before the program can be executed. The lowest line number present represents the first line and the highest line number the last line. These statements may either perform some action, such as adding two variables together and assigning the sum to a third variable (eg 'A=B+C'), or may be control statements, that change the execution flow of the system (such as GOSUB 3000).

To save user RAM, a number of statements can be written on one line, using the statement concatenation operator (::), with each statement being executed in turn. The general syntax for a line is:

```
{ line number } statement [ :: statement ]  { ! comment }
```

where { } indicate optional items
[ ] indicate item is repeated as many times as required - 0,1,....

Exceptions:

- DATA must be the only statement on a line.
- DEF must be the only statement on a line.
- ON must be the only statement on a line.
- NEXT should be the first statement on a line, otherwise it may not be located to terminate its corresponding FOR loop.
- REM takes the remainder of a line as comment.
- STOP must be the last statement on a line.

A full list of the Power BASIC commands/statements is provided in sections 4.2 and 4.3 of this addendum.

2.3 Character Set

1) Upper and lower case alphabet.
2) Digits 0 to 9.
3) Special characters
   ! " # $ % ^ ´ ([ ]) * : = - + ; , . ? / < >

Note: The THEN keyword can be abbreviated to ´::´ and PRINT can be abbreviated to ´;´.

2.4 Editing Source Statements

One method of modifying (or editing) a line is to simply retype the line. However, Power BASIC also supports a "sophisticated" line editor that allows the user to easily change previously entered source statements. The available edit commands are:

	ESC	Cancel input line
	RUBOUT	Backspace and remove character
	CR or LF	Enter the edited line
	ctrl H	Backspace the cursor one character
	ctrl F	Forward space the cursor one character
	ctrl I n	Insert N blanks
	ctrl D n	Delete N characters
ln	ctrl E	Display the line LN for editing

An attempt to forward space past the last character entered, or to backspace beyond the first character in the line will only cause the terminal´s bell to be rung.

´ctrl E´ means hold down the CTRL (CONTROL) key while striking the E key.

´ctrl I n´ means hold down the CTRL key while striking the I key, then strike the numeric key corresponding to the number N.

´ln ctrl E´ means type the desired line number (LN) followed by ctrl E.

When the carriage return (CR) or linefeed (LF) key is pressed, all characters displayed are entered, regardless of the position of the cursor.

Entering just a line number followed by a carriage return causes the specified line to be deleted from the stored program. However, if the specified line does not exist then an error message is output. Entering a statement with a line number that already exists causes the original statement to be replaced by the new one. Changing just the line number causes a copy of the original statement to be included in the program with the new line number (the

original statement line remains unchanged).

The editor is automatically invoked when the interpreter encounters a syntax error in a line being entered via the terminal.

If an edit operation fails due to lack of memory (ie you get the error message "STORAGE OVERFLOW"), issue the CLEar command (see section 3.5 below) and then re-issue the edit operation. If this doesn't work, more memory needs to be added to the system (if possible).

2.5 Automatic Line Numbering

The automatic line numbering facility is invoked by terminating an input line with a linefeed instead of a carriage return. This causes the interpreter to output the incremented line number and keyboard mode to be re-entered. The incremented line number is 10 greater than the last line number entered. Entering a line containing just a linefeed initializes the line number to 10. Terminating a line with a carriage return disables this facility.

2.6 Hexadecimal Constants

A hexadecimal constant is one to four hex digits followed by the letter H. A hex constant beginning with one of the letters A - F must be preceded by a zero.

2.7 Variables

A Power BASIC variable can be used to store either an integer number, a real (floating point) number, or a character string depending on the context in which the variable is used. Thus, although a variable may contain a number (integer or real) it can be used as though it contained a character string, and vice versa.

Variable storage starts in high memory and builds down towards low memory as new variables are declared, with each variable being allocated six consecutive bytes of memory. A variable's address is that of the word in lowest memory, ie the word nearest to address zero (in the diagrams below this word is referred to as the 'first word').

2.7.1 Variable Names

A variable name is either an alphabetic character followed by a number in the range 0 to 127 (eg Z100) or an alphabetic string up to three characters long (eg A, ST, and LST). The

variable name can not be identical to a Power BASIC keyword, nor can it form the beginning of a keyword. The following variable names are not valid:

LIS	Begining of LISt (a Power BASIC statement)
MEM	A Power BASIC function
TOT	First 2 letters are the Power BASIC keyword TO
12B	First character is not alphabetic
ABCD	More than 3 characters
I130	Number greater than 127
A.B	'.' not allowed in variable names

Note: There is a maximum of 140 different variable names in any one Power BASIC program.

2.7.2 Variable Declarations

Variables are not explicitly declared in BASIC. Instead a variable is implicitly declared by assigning a value to a valid variable name. For example, to declare the variable TST and assign it the value 100 the following statement can be used:

```
TST=100
```

A value can be assigned to a variable by either a READ (read a value from a DATA statement), an INPUT (accept input from the terminal) or a LET statement. The statement 'TST=100' is an implied LET, as are all statements of the form:

```
<variable>=<expression>
```

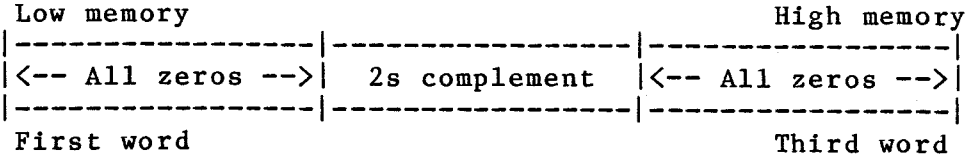
where <expression> may contain function calls:

```
FRD=SIN(PI*NUM)
```

The above statement assumes that the variables PI and NUM have already been declared (assigned a value).

2.7.3 Integer Variable

If a number can be represented in a 16-bit twos complement form, it is stored in integer format. Integer numbers in the range -32768 to +32767 will be stored in this way.

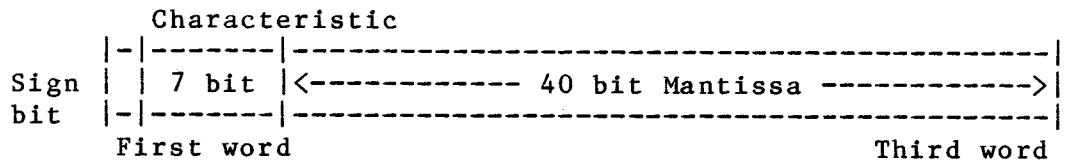


Integer numbers outside this range (up to approximately 11 decimal digits) will be recognised as integers and printed out in integer format, even though they will be stored internally in floating point format (see below). Integer numbers greater than this will be treated as real numbers and will be printed out in floating point format (eg 1.234E11).

2.7.4 Floating Point Variables

Floating point format allows a real number in the range 10E-75 to 10E+74 to be stored. ('E' represents the multiplier 10, the integer number following is the power to which 10 is raised. 2.5E24 means 2.5 times 10 to the power 24. Real constants can be entered in this format if desired.) This representation provides approximately 11 decimal digits of accuracy (a real constant should consist of no more than 11 digits).

A floating point number is represented internally as a fraction multiplied by a power of 16 (this power is known as the characteristic), and is stored as:



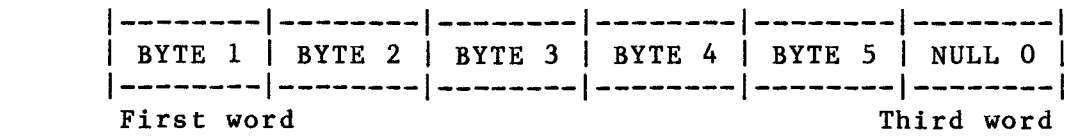
Bit 0 (the most significant bit) is the sign bit and represents the sign of the floating point number: 0 for positive, 1 for negative. Bits 1 to 7 hold the characteristic coded in Excess 64 notation (the characteristic is incremented by 64; this gives the characteristic a range of 0 to 127 representing a true exponent range of -64 to +63). The remaining 40 bits contain the normalised mantissa (the mantissa is normalised if its first hex digit is non-zero). Negative fractions are stored in true form with the sign bit set to one and not in twos complement notation.

2.7.5 Character String Variables

A character string is a string of characters enclosed within single or double quotes. Paired double quotes can be used to enclose single quotes and vice versa.

A variable is specified as containing a character string by preceeding the variable name with a dollar sign ($). In this form, a variable should be used to store up to 5 characters, as the last byte is used to terminate the string and contains the null character (zero). This is necessary

to ensure that the variable defined immediately before the string variable does not get corrupted.



Non-printable characters may be included in a character string by writing their hexadecimal ASCII representation enclosed in angle brackets (<>). The angle brackets are stored along with the character string and are only interpreted when the string is being input from a terminal, read from a DATA statement, or when the string is being printed. Note: Attempting to use the character sequence '<>' in a string via an INPUT, READ or PRINT statement will cause problems. If these characters are required then the sequence '<3C><3E>' should be used.

2.7.6 Array Variables

An array is a number of variables (stored consecutively in memory) that is referenced by a single variable name. Individual variables (or array elements) are accessed by following the variable name with a number that identifies the position of the variable within the array. The number (this is known as an array subscript) is enclosed in parentheses or square brackets (internally the parentheses are converted into and stored as square brackets).

To allocate the array STR with 10 elements the following statement is required:

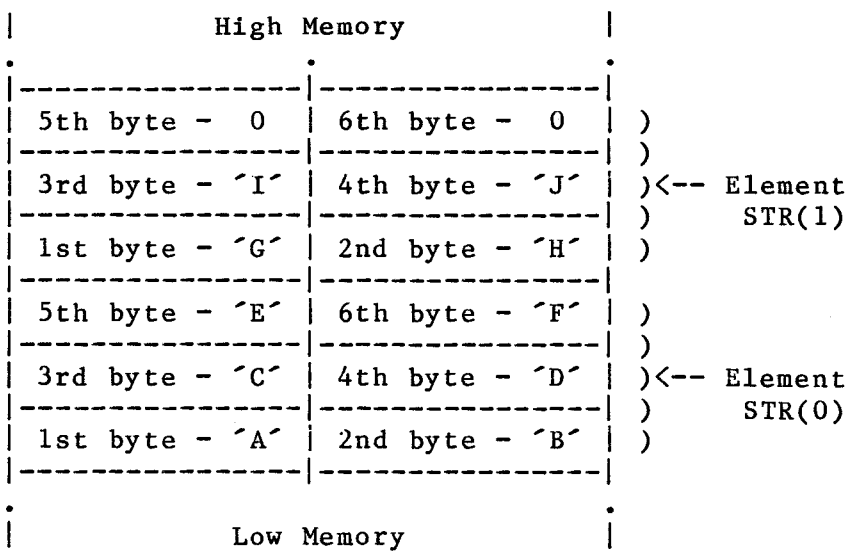
```
DIM STR(9)
```

The elements are referenced by

```
STR(0), STR(1), ...., STR(9).
```

The size parameter supplied to the DIMension statement is one less than might be expected, as Power BASIC automatically allocates space starting from element zero.

Although an array may be used to hold character strings, it is declared (in the DIMension statement) without the dollar sign. Individual bytes of a character string array can be accessed by following the array subscript with a semicolon (;) and the number of the required byte, starting from 1. For example, $STR(1;3) references the third byte of array element STR(1); this corresponds to the letter 'I' in the diagram below, which shows storage of the string "ABCDEFGHIJ".



Power BASIC allows an array to be declared with any number of dimensions. However, for most practical applications, a two dimensional array is usually sufficient.

Note: The variable A and the array variable A(0) refer to two completely different variables.

2.8 System Initialisation

On power-up, the Euroboard Power BASIC interpreter "autosizes" memory to determine how much RAM is available for system use. This autosizing stops as soon as a read/write mismatch is encountered, or when address hex 4000 is reached.

The interpreter allows two EIA cards to be inserted in the system; the first, port A, at CRU base address (R12 contents) hex 80 and the second, port B, at hex 180. The baud rate of the first port is determined automatically as described below. If a second EIA card is present, this is set to operate at 300 baud (when the initialisation has completed, this can be reset using the BAUD command).

If a Power BASIC application program has been "burnt" into EPROM(s) with the 'run flag' set (see PROgram command below), and installed at address hex 4000 then this program is immediately executed; with the first EIA card, if it is present, initialised to 300 baud. (Note that the auto-baud sequence is not performed in this case).

When no application program is present, an auto-baud sequence operation is performed on the first EIA card. This operation consists of waiting until the user presses the A key and then measuring the length of time taken for the

"start bit" to be received. From this, the actual baud rate of the terminal is evaluated and the EIA card is initialised to this baud rate. (Note: A carriage return delay of approximately 250 milliseconds is provided for devices operating at 1200 baud or less. Further, when operating at 1200 baud, output to a terminal is padded with a time delay to reduce the effective transfer rate to 300 baud.)

The system variables/pointers are then set according to the values determined by the auto-sizing operation. The following banner message is then output to port A:

```
EUROBOARD BASIC REV x.n.m
*READY
```

Where x = Language level
n = Release number
m = revision number

At this stage, Euroboard Power BASIC is in keyboard mode waiting for user commands.

Note: The input scanning routine allows data to be entered through either EIA port. If data is available at both ports then port A has precedence.

3 DETAILED DESCRIPTIONS OF MODIFICATIONS

This section describes features which are unique to Euroboard Power BASIC. Other features are described in sections 4-7 of the Power BASIC Reference Manual. A complete list of features and where to find them is given in section 1.1 of this Addendum. Section 4 below gives a complete "quick reference guide" to Euroboard Power BASIC.

3.1 LOAd/SAVe Commands

Power BASIC programs may be stored on "tape" using the SAVE command. Previously stored programs can be copied into memory using the LOAD command; programs that have been programmed into TMS25xx EPROMs can also be "loaded" into memory. Euroboard Power BASIC supports the following devices:

- o Texas Instruments ASR733 digital cassettes
- o Texas Instruments 763 and 765 Bubble memory terminals
- o Audio cassettes through an EIA port (this requires an EIA-to-CUTS standard conversion module)

The general format for these two commands is:

LOAD <exp> or SAVE <exp>

Where <exp> may be any valid Power BASIC expression. If no expression is present then the device is assumed to be an ASR733 digital cassette. The value of <exp> has the following meaning:

Value	Device
0	ASR733 digital cassette
1	763/765 bubble memory file
2	Audio cassette through port B
other	LOAD - TMS25xx EPROM(s)
	SAVE - error

3.1.1 733 ASR Terminal

To save a Power BASIC program on 733 digital cassettes perform the sequence shown below:

- o Ensure that the switches on the bottom row of the ASR switch panel are set as:

  KEYBOARD to LINE
  PLAYBACK to LINE
  RECORD to LINE

PRINTER to LINE

- Insert the cassette tape on which the program is to be SAVEd into either cassette transport. Make sure that the tabs on the bottom of the cassette tape are not in the write protect position.
- Verify that the terminal is ON-LINE.
- Set the RECORD CONTROL switch to OFF.
- Set the selected cassette transport to RECORD.
- Set the TAPE FORMAT switch to LINE.
- Momentarily press the REWIND switch on the selected cassette transport.
- When the END indicator lamp lights up, press the LOAD/FF switch on the selected cassette transport. The READY indicator lamp should light up after a few seconds.
- If the 733 ASR does have the Automatic Device Control (ADC) option, type in "SAVE" followed by a carriage return. Euroboard Power BASIC will issue the necessary device control characters to start (DC2 - RECORD ON) and stop (DC4 - RECORD OFF) the cassette transport.
- If the 733 ASR does not have the ADC option, the user must manually start and stop the cassette. To do this type in "SAVE" (don't enter the carriage return yet), set the RECORD CONTROL switch to ON and then enter the carriage return. When the cassette finally stops, the RECORD CONTROL switch must be set to OFF.

When the save operation has completed, Euroboard Power BASIC automatically returns to keyboard mode and causes the printer to advance the paper to a new line.

A detailed description of each of the switches on the cassette transport is given in section 4 of the Model 733ASR/KSR Data Terminal Installation And Operation Manual (part number 945259-9701). Figure 4-2 of this manual (or Figure 2-12 of the Power BASIC Reference Manual) gives the actual layout of these switches.

To load a Power BASIC program from 733 digital cassettes perform the following sequence:

- Ensure that the switches on the bottom row of the ASR switch panel are set as:

  KEYBOARD to LINE
  PLAYBACK to LINE
  RECORD to LINE
  PRINTER to LINE

- Insert the cassette tape from which the program is to be LOADed into either cassette transport.

- Verify that the terminal is ON-LINE.

- Set the RECORD CONTROL switch to OFF.

- Set the selected cassette transport to PLAYBACK.

- Set the TAPE FORMAT switch to LINE.

- Momentarily press the REWIND switch on the selected cassette transport.

- When the END indicator lamp lights up, press the LOAD/FF switch on the selected cassette transport. The READY indicator lamp should light up after a few seconds.

- If the 733 ASR does have the Automatic Device Control (ADC) and the Remote Device Control (RDC) options, type in "LOAD" followed by a carriage return. Euroboard Power BASIC will issue the necessary device control characters to start (DC1 - PLAYBACK ON) and stop (DC3 - PLAYBACK OFF) the cassette transport. With the RDC option, the cassette transport will accept the BLOCKFOWARD (DLE 7) control characters to start the cassette and read the next record.

- If the 733 ASR does not have the ADC option, the user must manually start and stop the cassette. To do this type in "LOAD" followed by a carriage return and momentarily press the PLAYBACK CONTROL CONT switch to START.

If the 733 ASR does not have the RDC option, it may not be possible to load a Power BASIC program from cassette at 1200 baud. If this is the case, then the terminal must be set to operate at 300 baud. In either case it will be necessary to perform a manual start/stop operation as described above.

When the load operation has completed, Euroboard Power BASIC

automatically returns to keyboard mode and causes the printer to advance the paper to a new line. At this point, the program has been loaded into memory. If any errors occurred, the appropriate error message will be output followed by the statement line in error. As the statement line can not be correctly stored it will be ignored, and the loading operation will continue from the next statement line stored on the cassette tape.

3.1.2 763/765 Bubble Memory Terminal

The bubble memory terminal must be configured as follows:

```
LINE MODE:       EIA/ 1200 BAUD/ EVEN PARITY/ FULL DUPLEX/
OPTIONS ON:      EDC/ DC3/ DC1.3/ DC2.4/
OPTIONS OFF:     PCHECK/ ABMPRT/ AUTOABM/ EOTDIS/ BUFFER/
ABM:
RECORD FILE:     <record file name>
PLAYBACK FILE:   <playback file name>
TRANSMIT EOL:    <carriage return symbol>
RECEIVE EOL:     CRLF
KEY:
```

<carriage return symbol> refers to the symbol produced by the bubble memory terminal whenever the carriage return key is struck. (Note: When setting the TRANSMIT EOL parameter, the <carriage return symbol> must be enclosed within double quotes.)

The STATUS (or ST) command should be used to check that the terminal is configured correctly. If any of the configuration parameters is incorrect then use the CHANGE (or CG) command to modify the appropriate parameter (eg CHANGE SPEED TO 1200, or CG PORT TO EIA, etc). When the terminal has been correctly configured, it needs to be on-line before Power BASIC can communicate with it; this is performed by the ONLINE (or ON) command. By this time the ON LINE and COMM indicator lights (lower right hand side of the keyboard) should be on.

Note: Although the baud rate of the bubble memory terminal is user selectable up to 9600 baud, the actual throughput of the terminal is limited to 240 characters per second (2400 baud). However, at speeds above 1200 baud the printer will lose characters and it may even "lock up" (if this happens it is necessary to use the ESCape key to return to keyboard mode).

Files are allocated within the bubble memory terminal using the CREATE (or CF) command. When a file is created, the user specifies:

- File name (a maximum of six alphanumeric characters,

the first character must be alphabetic) which can not be KEY or TO.

o Format to be used, either C (continuous) or L (line). In line format one statement line will be stored in one physical record (if the record size is 70 characters and the statement line only needs 20 characters, the remaining 50 characters worth of storage will not be used). In continuous format a number of statement lines can be stored in one physical record.

o Maximum number of physical records that the file can hold (files can not be extended or reduced once created).

o Record size (ie the maximum number of characters that can be stored in one record) up to 80 characters. If this parameter is not specified a record size of 80 characters is assumed.

For example, to create the file BAS001, using line format, to hold 150 records the following is required:

```
CREATE   BAS001 L 150 <skip key>
```

The CATALOG (or CL) command can be used to list the characteristics of all the existing files (it also shows how much space is left).

After a SAVE operation, the Power BASIC program will be contained in <record file name>. A LOAD operation will cause the contents of <playback file name> to be stored in program memory (assuming that the file contains a valid Power BASIC program). <record file name> and <playback file name> can be the same file.

Power BASIC programs can be created when the bubble memory terminal is not connected to the /E155 microcomputer module (ie when it is off-line) using the EDIT (or ED) command. Please note that if this is done then the file must be terminated with a line that only contains the escape character.

Any errors encountered while loading a Power BASIC program will cause the error line and the appropriate error message to be output to the terminal (the whole line will be thrown away as it can't be stored correctly) and processing will continue from the next input line.

While accessing a bubble memory file, the keyboard is disabled until the operation has completed.

NOTE: The STATUS, CHANGE, CREATE, EDIT, ONLINE, and CATALOG

commands refer to the commands available under the bubble memory terminal's internal monitor and not to Euroboard Power BASIC. The terminal's monitor can be entered at any time by striking the CMD key (top right hand side of the keyboard); when this is done the COMMAND indicator lamp should light up and the monitor should reply with a "filled in arrow" symbol. Also note that the terminator key for a monitor command is the SKIP key and not the CR key. For further information on the operation of the bubble memory terminal refer to 'Model 763/765 Memory Terminals Systems Manual' (part number 2203665-9701) and/or 'Model 763/765 Memory Terminals Operating Instructions' (part number 2203664-9701).

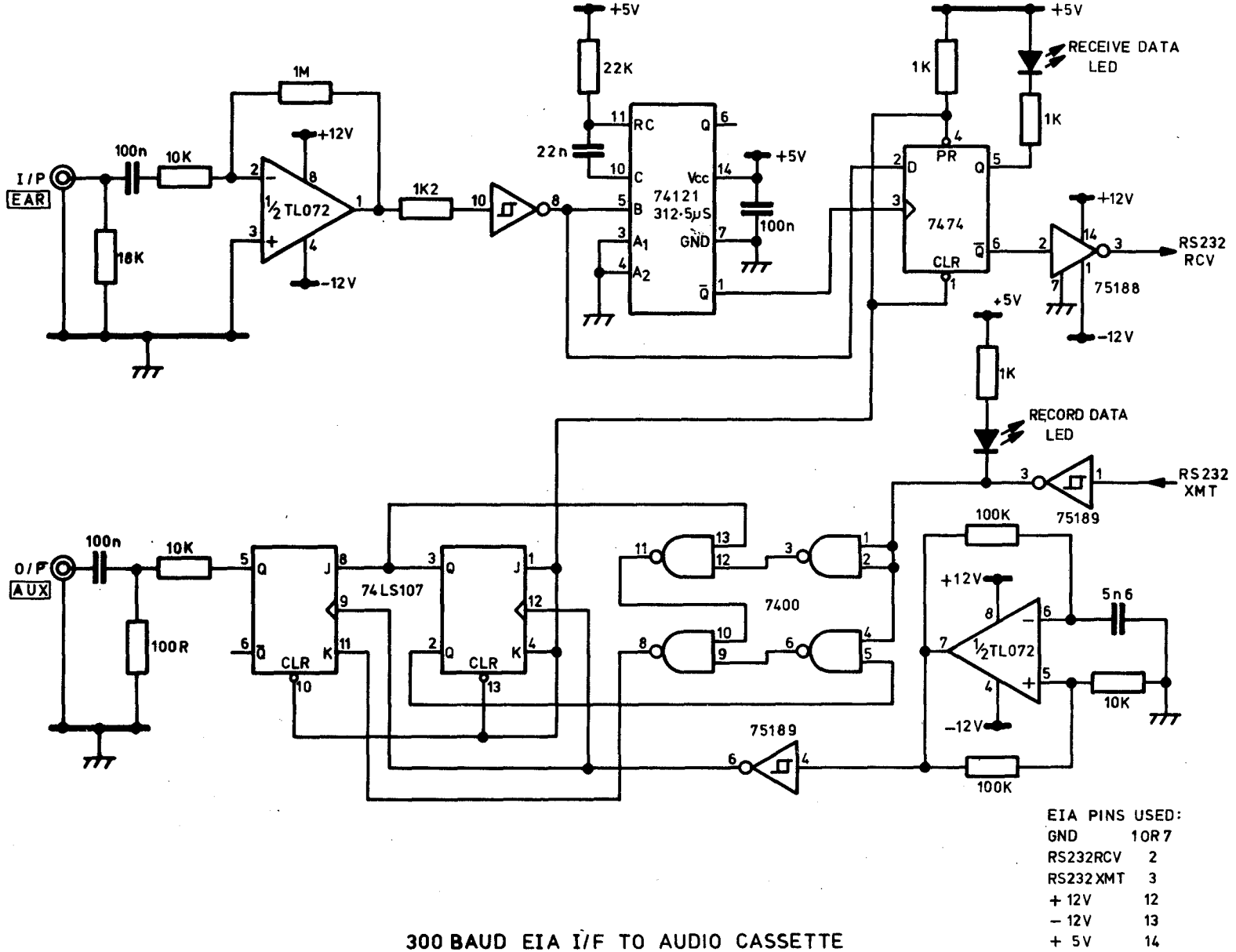
Bubble Memory Connector P1	EIA Card
1	1
2	5
3	3
4	6
8	20
9	6
11	4
14	2
15	7

The necessary cable configuration to connect a bubble memory terminal with a Euroboard EIA card is shown above. The terminal's connector P1 is used. Note that connector P1 is protected by a flap that is located immediately below connector P2.

3.1.3 Audio Cassette (CUTS Standard)

Computer User's Tape Standard (CUTS) is a universal standard for audio cassette interfaces that is designed to operate at 300 baud. Logic level '1' is defined by a 2400Hz signal and logic level '0' is defined by a 1200Hz signal.

The circuit in Figure 1 implements the CUTS audio cassette interface and connects directly to the second EIA port (configured at CRU address hex 180); power is conveniently taken from the EIA connector itself. This circuit can be built up and connected to a second TM990/E358 card (see section 1.4 above for the appropriate jumper settings). Alternatively, a custom card can be built which contains both the EIA serial terminal interface and the audio cassette interface. A circuit for the serial terminal interface is given in the E-Bus System Design Handbook (MP402), section 7.2.



300 BAUD EIA I/F TO AUDIO CASSETTE

Figure 1 EIA to Audio Cassette (CUTS Standard) Interface

The connections marked AUX and EAR plug into the input and output sockets of the cassette tape recorder.

Note: Although the circuit above does not show it, it is necessary to wire pin 6 of the EIA connector to pin 20 to provide the DTR signal.

The procedure for recording information on an audio cassette is: allow the tape to run for several seconds (say 10 seconds) to give a good leader; record the data; leave a trailer (say 5 seconds).

Having built the audio tape interface it is necessary to determine the correct volume control setting for each recorder to be used. This is performed (from Euroboard Power BASIC) by entering and executing the following sections of code:

a) Write the sequence "U* ..... U*" to the cassette

```
100 BAUD 1,5      ! 2nd port to 300 Baud
110 UNIT 2        ! Output to 2nd port only
120 PRINT "U*";   ! Alternate '1's and '0's
130 K=NKY(0)      ! Key pressed?
140 IF K=0 THEN GOTO 120  ! N - loop
150 UNIT 1        ! Back to main port
160 STOP
```

When RUN, this program will continually write "U* ..... U*" to the cassette until a key is pressed on the terminal.

b) Read the tape back

```
210 BAUD 1,5      ! 2nd port to 300 baud
220 BASE 0180H    ! Point to 2nd port
230 IF CRB(21)=0 THEN GOTO 230 ! Wait for character
240 A=CRF(7)      ! Get character
250 $B=%A%0::PRINT $B;  ! Print character
260 CRB(18)=0::GOTO 230 ! Reset RBRL and loop
270 STOP
```

The recorder may take a little while to settle down after changing the volume setting; the correct setting has been found when the read routine above is consistently receiving 'U*' characters (say several correct lines). When satisfied that the volume control has been correctly set, press the ESC key to escape from the read routine and return to keyboard mode.

Note: When the read routine receives a character from the audio cassette drive it immediately prints it at the

terminal connected to port A. To ensure that characters are not lost from port B when a carriage return is output to port A, the baud rate for port A must not be less than 300 baud.

To save a Power BASIC program on audio cassette perform the following sequence:

- Insert the AUX plug into the cassette recorder's AUX (or MIC) socket.
- Ensure that the EAR plug is not installed.
- Start the cassette recorder running in record mode.
- Wait a few seconds to ensure a good leader.
- Type in the command 'SAVE 2' followed by carriage return.
- When the save operation has completed, Euroboard Power BASIC automatically returns to keyboard mode and causes the printer to advance the paper to a new line.
- Leave the cassette recorder running for a few seconds to generate a suitable trailer.
- Remove the AUX plug.

To load a Power BASIC program on audio cassette perform the following sequence:

- Insert the EAR plug into the cassette recorder's EAR socket.
- Ensure that the AUX plug is not installed.
- Position the cassette recorder's read head directly over part of the leader.
- Start the cassette recorder running in playback mode.
- Type in the command 'LOAD 2' followed by carriage return.
- When the load operation has completed, Euroboard Power BASIC automatically returns to keyboard mode and causes the printer to advance the paper to a new line.
- Turn off the cassette recorder and remove the EAR

plug.

If an error occurs during the load operation then the whole input line is thrown away, as it can't be stored correctly (processing continues with the next input line), and when the operation has completed the message 'TAPE READ ERROR' will be output. LIST the program to find the missing line(s).

While the interpreter is accessing an audio cassette, the UNIT flag is temporarily reset to disable all keyboard output. All input to port A is ignored except for the escape key (ESC) which can be used to abort the operation.

When using an audio cassette the baud rate of port B should be set to 300. This is automatically done on power-up. However, if you have been using this port with a faster (or slower) device it will be necessary to execute the 'BAUD 1,5' Power BASIC statement.

3.1.4 EPROM

When the value of <exp> supplied to a LOAD command is greater than 2, the interpreter assumes that the "stored" Power BASIC program has actually been programmed into EPROMs and that the value of <exp> specifies the start address of that program.

The program's start address is calculated from:

Program's Start Address = EPROM Address + Offset

where "EPROM Address" is the memory address of the first EPROM containing the program within the system memory map. "Offset" is the offset within this EPROM to the actual program and should be the same value as that supplied to the PRO command's "EPROM START ADDRESS:" prompt when the program was being burnt into EPROM (see section 3.3 for details about the PRO command).

For example, a "LOAD 05862H" command would indicate that "EPROM Address" is hex 5000 (ie the program's EPROM is located at system memory address hex 5000) and that "Offset" is hex 862 (ie the program that we are interested in is located at an offset of hex 862 bytes within the EPROM).

When a Power BASIC program is burnt into EPROM a number of system pointers are also burnt into the EPROM. These pointers allow the program (when loaded) to be listed and executed while still resident in EPROM without having to copy it into RAM. However, this does mean that it is not possible to change a program that has been "loaded" from

EPROM; attempting to do this will cause unpredictable results to occur.

The interpreter does not attempt to verify that the address supplied actually contains an EPROM-based Power BASIC program. If this is not the case, attempting to access the "loaded" program (in any way) may cause upredictable results to occur.

If the "auto-run" option was selected when the program was burnt into EPROM (ie the answer to the "RUN?" prompt was "Y") then the interpreter will automatically execute the Power BASIC program when it has been "loaded". Otherwise the interpreter will return to keyboard mode when the operation has completed.

3.2 RENumber Command

The general format for this command is:

REN

This command renumbers the stored Power BASIC program so that on completion the first line number is 10 with an increment of 10 between successive lines. The line numbers contained within the program code are automatically updated to reference the new line numbers.

The command is implemented using a two-pass approach. On the first pass, the command determines how much "free RAM" exists in the system and where it is. This area of memory is then used to dynamically build a table of line number references. The whole program is scanned and every time a line number is encountered within the program an entry is added to this table. Each table entry consists of the line number and the address in the program memory where it was found. If there is insufficient free RAM to complete this table, then the table area is reset to its original state, the error message 'STORAGE OVERFLOW' is output to the terminal and an immediate return is made to keyboard mode.

However, if the first pass does complete successfully (ie there was sufficient memory available) then, and only then, is the second pass executed. This approach ensures the integrity of the stored program (ie it can not get corrupted due to starting to update the program and not being able to complete the operation).

If there are any line numbers in the form of expressions (eg TRAP 5 TO 50*A4), this is detected by the first pass, and the following warning message is output (it is not possible to update these references automatically):

** UNCONVERTED STMT REF AT NEW LINE xx **

where xx is the line containing the unconverted reference.

When the renumbering is complete, check the line(s) indicated and convert the statement references by hand. All other references will be converted correctly.

In the second pass, each line number entry in the table is checked to determine whether or not the line it references actually exists. If it does, that line's new value is used to update the memory locations that reference the line. When a line is referenced that does not exist then the following warning message is output:

** NON-EXISTENT STMT REF xx AT NEW LINE yy **

where xx is the line that does not exist.
yy is the line with the reference to xxx.

When all table entries have been done, the free RAM area is put back to its original state and a return is made to keyboard mode.

If the SIZe command is executed immediately before and then again straight after executing the RENumber command, there may be a slight discrepancy between the two program sizes shown. This is nothing to worry about; it is the result of an internal reorganisation and does not affect the operation of the stored program.

If the program to be renumbered is very large, or requires a vast amount of data storage (for instance, very large arrays) then it is possible that the command will fail due to the limited amount of temporary storage space available. This can be overcome by either of the following sequences:

- Issue the CLEar command.
- Re-issue the REN command.

or

- SAVE the program.
- Execute the NEW command.
- LOAD the program.
- Re-issue the REN command.

If these do not work then the only thing to do is to add another memory expansion board to the system (if possible).

3.3 PROgram Command

The general form for this is:

PRO

This command allows the user to "burn" the stored Power BASIC program into TMS25xx series EPROMs using the EPROM programming board. These EPROMs can then be inserted into the target system and will either be automatically executed on system power-up (if the EPROMs are installed starting at address hex 4000 with the 'auto-run' flag set), or can be executed when required using the LOAD (see section 3.1) and RUN commands.

Before the program is burnt into EPROM it is a good idea to perform the following sequence:

- o SAVE the program.
- o Execute the NEW command.
- o LOAD the program.

This ensures that all temporary Power BASIC variables (ie variables that are currently defined but not actually required by the program) are deleted from the interpreter's "variable name table".

The PROgram command is prompt driven (ie it outputs prompt messages to the terminal and waits for correct user response). For all of these prompts (except the 'EPROM TYPE' and 'EPROM START ADDRESS' prompts) a 'Y' (yes) or 'N' (no) response is required, and only these two characters are echoed back to the terminal. However, the ESCape key can be used to terminate the programming session. This outputs the following message.

PROGRAMMING TERMINATED

Any other response causes the prompt to be re-displayed.

The first prompt is:

RUN?

This is asking whether the 'auto-run' flag, contained in the first word of the first EPROM, is to be set (see above).

Power BASIC then asks for the EPROM-type with the following prompt:

EPROM TYPE (2516, 2532, 2564, 2528)?

If the response is an invalid EPROM type, the above prompt is re-displayed. When a valid response is given, the EPROM programming board is configured to the specified type. (If the response is incorrect then the user must strike the ESC key and then re-issue the command.)

The Power BASIC interpreter then checks to ensure that the

on-board dc/dc converter is working. If not, the message

EPROM PROGRAMMER NOT FOUND

is output and the command terminates.

When the programming board is configured, the following prompt is output

EPROM START ADDRESS:

A response of a carriage return will cause programming to start from the first location within the EPROM. Usually, this is what is required.

However, it is possible to burn two or more (small) programs into a single EPROM. To burn a second or subsequent program into an EPROM, specify the address within the EPROM where programming is to start, followed by a carriage return. This address must be greater than the address of the last byte of the last program to be burnt into that EPROM. It is important that this value is entered correctly. If an 'illegal' value is entered (eg A8 instead of 0A8H, or *, or MMMM) then a start address of 0 is assumed. See section 2.6 above for the correct format of hexadecimal constants. If the value is a floating point number, or if it is outside the EPROM's range (eg 02000H for a TMS2516) then the prompt is re-issued.

Before the carriage return key is struck, the value can be modified using the RUBOUT key. All other control characters are ignored.

Note: The value entered is forced to a word boundary so that, if (eg) the value 07FH is entered, programming will start from EPROM location 07EH - the preceding word boundary.

The next prompt is

SPLIT INTO MS AND LS BYTES?

If the EPROM is to be used with a memory board that has an 8 bit wide data path (eg TM990/E250) then the answer to this prompt should be 'N'. However, if the memory board has a 16 bit wide data path (eg TM990/E255) then the answer should be 'Y'.

This is followed by the prompt

MOUNT EPROM
EPROM READY?

When the first/next EPROM has been mounted in the EPROM

socket the programming cycle starts. This is indicated by the message

PROGRAMMING

If the response to the 'SPLIT INTO MS AND LS BYTES?" prompt was yes then the message output will be either

PROGRAMMING MS BYTE
or PROGRAMMING LS BYTE

The most significant byte (MS - bits 0 to 7) is programmed first.

During this cycle the keyboard is continuously scanned. The only character that is recognised during this stage is the ESCape key (all others are ignored), which causes the programming cycle to shutdown gracefully and the termination message to be displayed.

After the programming cycle has completed (either the EPROM is full or all the Power BASIC program has been burnt into EPROM) the message

VERIFYING

is output and the programmed EPROM is verified. If the EPROM's contents do not match the memory contents, the message

VERIFY ERROR!

is output and the 'MOUNT EPROM' phase is re-entered.

If any of the Power BASIC program remains to be burnt into EPROM, the 'MOUNT EPROM' phase is re-entered. Otherwise the message

PROGRAMMING COMPLETE

LAST BYTE PROGRAMMED WAS xxxx

is output at the terminal and the command terminates. "xxxx" is the address of the last location within the EPROM that was programmed. This value should be carefully recorded if any further programs are to be burnt into the same EPROM: add 2 to obtain the correct reply to the next "EPROM START ADDRESS" prompt (see above).

3.4 NEW Command

The general form for this command is:

NEW <exp> or NEW

Either form will erase any stored Power BASIC program, and reinitialise the memory. After a NEW command, any previously entered statements and all variables stored will be lost.

This command allows the user to specify the lower limit of RAM that is to be used by the Power BASIC interpreter. The Power BASIC environment will consist of all memory from the TMS9995's internal RAM down to this limit (<exp>). Any memory that resides below this limit will be temporarily deallocated from the Power BASIC environment; however, it can still be accessed via the MWD and MEM functions.

The interpreter requires a certain amount of memory to operate: the top 850 bytes of the /E155's on board RAM and all of the TMS9995's internal RAM. If <exp> lies within this region (or even above it) or if it specifies an address that was below the limit found by the interpreter during the autosizing phase on power-up then the error message 'INVALID MEMORY LIMIT' is output and the command is ignored.

When the NEW command is issued without an <exp> parameter, the lower RAM limit will revert to that found on power-up.

3.5 CLEar Command

The general form for this command is:

CLE

This command deallocates all the memory occupied by Power BASIC variables (ie it clears and returns the memory to the interpreter) without affecting any stored program.

However, after issuing this command, an attempt to access the contents of a Power BASIC variable or its address (eg in a PRINT statement, CALL statement or on the right hand side of a LET statement) will result in the error "UNDEFINED VARIABLE".

3.6 ADR Function

The ADR function returns the address of the specified variable, string variable, array element or byte offset in an array element. This is used in conjunction with the LET (assignment), PRINT and CALL statements.

The general form for this is:

ADR(<variable>)

Valid uses are:-

```
B = ADR( <var> )
PRINT #, ADR( <var> )
CALL .....,ADR( <var> )
```

where <var> can be one of the following:
A, $A, A(index), $A(index), or $A(index ; offset)

Although the following yields a valid address, the data that it references is of no real value as it will be somewhere within the internal evaluation stack and is thus subject to change.

```
B = ADR( <expression> )
```

Sections 2.7.3 to 2.7.6 above describe Euroboard Power BASIC's variable storage mechanism. This function (unless explicitly overridden) returns the address of the first byte (ie lowest memory address) for the specified variable.

The ADR function has been included to allow an area of RAM that belongs within the Euroboard Power BASIC environment (eg elements of a dimensioned array) to be accessed as though it were temporarily outside the environment. The following program illustrates this.

```
1000 DIM TMP(50)          ! 300 byte area
1010 B = ADR( TMP(0) )    ! B refs that area
1020 C = B                ! Save address
1030 MWD( B ) = 020CH     ! 'LI  R12,>600' instruction
1040 B = B + 2            ! Next word
1050 MWD( B ) = 0600H     ! 2nd word of instruction
1060 .
     .
2500 CALL "ANY",C,.....   ! Execute stored program
```

Operations of this kind should be executed with care, to prevent corruption of stored Power BASIC programs.

3.7 CALL Statement

The general format for this statement is:

```
CALL  <name>,<address>,<parms>
```

Where <name> is a string (this is not used by the interpreter but it must be present). <address> is the address in memory of the assembly language routine's entry point. <parms> is the list of parameters to be passed over to the called routine, and each parameter can be any valid Power BASIC expression.

The CALL statement handler has been modified from that in Development Power BASIC to invoke the specified assembly language routine via a BLWP instruction. The BLWP vector is built in RAM using a special workspace and the routine's entry point (<address> above). This workspace is allocated from the Euroboard Power BASIC environment but is only ever used by the CALL statement handler.

This allows up to 12 parameters (as opposed to only 4 in Development Power BASIC) to be passed to the called routine. R0 to R11 of the workspace contain the 16 bit parameter values, the actual number of parameters passed over is stored in R12, and R13 to R15 contain the Power BASIC context.

Before storing a parameter in the appropriate register, the parameter is evaluated and the result is fixed into a 16 bit value. If the called routine requires a floating point number, or a string (from either a simple variable or a dimensioned variable) then the ADR function should be used to give the address of the particular variable or array element. (Note: In this implementation of the call handler, the parenthesis mechanism for passing a variable's address has been removed.) Sections 2.7.3 to 2.7.6 above describe Euroboard Power BASIC's variable storage mechanism.

A return to Euroboard Power BASIC is made by executing an RTWP instruction in the called routine. Care must be taken to ensure that R13, R14 and R15 of the special workspace are not modified, otherwise execution of the interpreter could become erratic.

3.8 Interrupts

Euroboard Power BASIC supports a "pseudo interrupt" mechanism that polls 16 dedicated CRU input lines at the end of each statement. This means 15 Power BASIC interrupts ('level 1' to 'level 15') are available, whereas the TMS9995 microprocessor itself only implements 6 interrupts. (The TMS9995 hardware interrupts are also available, for assembly language subroutines outside the Power BASIC environment).

As the interpreter can not respond to an interrupt until the currently executing statement is completed, this method of handling "interrupts" introduces no overhead. Software control also allows very flexible interrupt handling for application programs.

If any of the interrupt signals (which are prioritized) is present, Power BASIC will perform the interrupt subroutine associated with that signal, and then return to the main program. The TRAP statement (below) associates an interrupt

signal with its appropriate Power BASIC interrupt subroutine.

Two separate methods are available for masking/enabling individual interrupt signals. First, the ENABLE statement allows each individual interrupt signal to be "switched" on or off. Power BASIC will not recognise any interrupt signal that is not enabled.

Second, the interrupts are prioritized (level 1 highest priority, level 15 lowest priority). A high priority interrupt can interrupt a lower priority interrupt routine, but not vice versa. The Power BASIC interpreter maintains an interrupt mask that determines what priority of interrupts will be serviced. The mask is automatically set on receiving an interrupt, to disable all interrupts of the same or lower priority until the interrupt subroutine has completed. By setting the mask level appropriately in the main program (using the IMASK statement) the user can enable certain interrupts and disable others.

The new mechanism is implemented as a 16 bit STCR operation at a CRU base address (R12 contents) of hex 120. Bit 0 of this read (input pin 0) is ignored. The remaining 15 bits are interpreted as "interrupt lines". These CRU "interrupt lines" can be implemented using one of the EBUS I/O boards (eg the /E350 or /E352) that is configured for a CRU base address (R12 contents) of hex 100.

As an interrupt is active low, a value of zero ('0') on an input pin indicates the presence of an interrupt, while a value of one ('1') on that input pin indicates the absence of an interrupt.

To use this new interrupt mechanism the user must:

1) Allocate a Power BASIC subroutine to a particular interrupt level using the TRAP statement (as Development Power BASIC)

2) "Enable" all necessary interrupts, using the ENABLE statement (not in Development Power BASIC)

3) Set the interpreter's internal interrupt mask to an appropriate level, using the IMASK statement (modified from Development Power BASIC)

4) The last statement in the interrupt subroutine should be an IRTN statement (no change).

5) Before the IRTN statement is executed the user must reset the device that generated the interrupt, otherwise the interrupt subroutine is liable to be re-entered immediately.

When building systems from E-Bus modules, note that Power BASIC "interrupts" are implemented as CRU read operations, and are not transmitted over E-Bus as interrupt codes.

3.8.1 ENABLE Statement

This statement allows the user to specify whether an interrupt is to be "enabled" or "disabled". (Although an interrupt is "enabled" the interpreter will not recognise it until the internal interrupt mask has been set to allow it through.)

(The use of this statement is analogous to enabling/disabling interrupt mask bits in the TMS9901.)

The general format for this statement is:-

ENABLE <interrupt level 1>,....,<interrupt level n>

Where <interrupt level i> can be an expression. A positive value indicates that the level is to be "enabled" while a negative value means that the level is to be disabled. Note: Only the least significant hex digit is used to determine the interrupt level. For example

ENABLE 6,3,15,-10,-11,0105H

Is translated to mean that interrupt levels 3, 5, 6 and 15 are to be "enabled" and that levels 10 and 11 are to be "disabled".

3.8.2 IMASK Statement

The IMASK statement does not actually affect the processor's status register (ST) which is permanently set to 4 (this allows the user to "hook in" assembly language interrupt handlers for interrupt levels 1, 2 and 4 - see section 3.8.3 below).

This statement determines whether or not an "enabled" interrupt is recognised by the interpreter. When the IMASK value is "0" the interpreter does not interrogate the "interrupt lines". All other values cause the "interrupt lines" to be read; any "enabled" interrupts are compared against the stored IMASK value and if one of them is of a higher priority then that interrupt is serviced.

Consider the following statements:

```
10 TRAP 3 TO 3000
20 TRAP 6 TO 6000
30 TRAP 10 TO 10000
40 ENABLE 6,3,10
50 IMASK 10
```

If after executing the ENABLE statement "interrupt line" 10 is low, then the Power BASIC subroutine starting at line 10000 will be executed.

Recognising an interrupt causes the existing IMASK value to be saved along with the current ELSE flag and CRU base address on a 16 level deep stack; a new IMASK value is generated that is one less than the interrupt level being serviced (this effectively inhibits further interrupts at that level until the present one has been completed) and the specified interrupt subroutine is executed.

Once the interrupt subroutine has been completed the original IMASK value, ELSE flag and CRU base address are restored from the stack and program execution continues from where it was interrupted.

Issuing an IMASK statement in an interrupt subroutine causes the interpreter's status to be reset to the IMASK value but only while the subroutine is being executed. In interrupt subroutines the IMASK statement should not be used to permit the same or lower priority interrupts to be recognised as this can cause the interrupt stack to overflow.

Note: The "interrupt lines" are not scanned when the user is operating in keyboard mode. Interrupts are now scanned AFTER a statement has been executed. This allows the user to ESCape from the program, to inspect and/or modify Power BASIC variables and to continue execution without problems. Further, when the interpreter has recognised an interrupt, the "interrupt lines" are ignored until the first statement of the interrupt subroutine has been executed. This ensures that the user is able to inhibit all other interrupts (using IMASK 0) if he so wishes.

3.8.3 Assembly Language Hardware Interrupt Handlers

When the system is powered up the following instructions are written into RAM (starting at location hex EF6C)

```
0EF6C     BLWP  @>0000    (for interrupt level 1)
0EF70     RTWP
0EF72     BLWP  @>0000    (for interrupt level 2)
0EF76     RTWP
0EF78     BLWP  @>0000    (for interrupt level 4)
0EF7C     RTWP
```

If the BLWP vector address is not modified by the user then when the processor recognises a level 1, 2 or level 4 interrupt a RESTART (interrupt level 0) operation will be performed.

By changing the BLWP vector address (possibly using the MWD function) the user can cause assembly language routines to trap the hardware interrupts of the TMS9995.

It is not intended that actual hardware interrupts are handled by a Power BASIC interrupt routine. However, it is possible for an assembly language routine to cause an interrupt routine to be executed. This can be done in either of two ways:

- If a TMS9901 Programmable Systems Interface is located at a base address of hex 100 (R12 contents), then writing a zero (SBZ instruction) to the appropriate pin will cause the interpreter to "see" an interrupt on that line.

- By setting the appropriate bit in the memory word at hex address EED2 to a '1'. (Bit 15 corresponds to interrupt level 1, and bit 1 corresponds to interrupt level 15.) This word is masked onto the 16-bit value read from the CRU "interrupt lines" so that it appears that a "polled" interrupt has been generated.

Note: Before an assembly language routine does initiate a Power BASIC interrupt routine it should first ensure that the real hardware interrupt is cleared. Further, both of the above mechanisms are subject to the normal interrupt sequence (ie the IMASK value must be set to allow that interrupt level through and the interrupt must be enabled).

4 QUICK REFERENCE

4.1 GENERAL

Character Set

1) Upper and lower case alphabet.
2) Digits 0 to 9.
3) Special characters
   ! " # $ % ^ ´ ([ ]) * : = - + ; , . ? / < >

Abbreviations

::	for	THEN
;	for	PRINT

Hexadecimal Constants

A hexadecimal integer constant is one to four hex digits followed by the letter H. A hex constant begining with one of the letters A - F must be preceded by a zero.

Variable Names

A variable name starts with an alphabetic character optionally followed by up to two additional alphabetic characters or a number in the range 0 to 127. The variable name may not be the same as a Power BASIC keyword; nor can it form the begining of a keyword.

Edit Commands

	CR	Enter line into program source
	LF	Enter line into program source and enable the auto-numbering facility
	ESC	Cancel input line, return to keyboard mode
	DEL/RUBOUT	Backspace and delete character
	Ctrl D n	Delete N characters
	Ctrl I n	Insert N blanks
	Ctrl H	Backspace 1 chaaracter
	Ctrl F	Forwardspace 1 character
ln	Ctrl E	Display line LN for editing

4.2 POWER BASIC COMMANDS

Power BASIC commands may not appear within a program.

Command	Function
CLEar	Return and clear all memory occupied by Power BASIC variables to the interpreter
CONtinue	Continue execution from last break
ln LISt	List current program from specified line LN=Null, Line=First line number LN≠Null, Line=LN
LOAd exp	Load BASIC program from specified device EXP=Null, Device=733 digital cassette EXP=0, Device=733 digital cassette EXP=1, Device=Bubble memory terminal EXP=2, Device=Audio cassette EXP=Address, Device=2532 EPROM
NEW exp	Clear system for new program EXP=Null, RAM limit set by autosizing EXP≠Null, RAM limit=EXP
PROgram	Burn current program into 25xx EPROM(s)
RUN	Clears all variable space, pointers, and stacks and executes current program from first line number
SAVe exp	Save current program on specified device EXP=Null, Device=733 digital cassette EXP=0, Device=733 digital cassette EXP=1, Device=Bubble memory terminal EXP=2, Device=Audio cassette
SIZe	Display size of current program
RENumber	Renumber current program.

4.3 POWER BASIC STATEMENTS

Power BASIC program lines are of the form:

```
{ line number } statement [ :: statement ] { ! comment }
```

Where { } Indicate optional items
[ ] Indicate item is repeated as many times as required - 0,1,....

Exceptions:

DATA must be the only statement on a line
DEF must be the only statement on a line
ON must be the only statement on a line
NEXT should not be preceded by ´:: statement(s) ´
REM must not be followed by ´:: statements(s) ´
STOP must be the last statement on a line

BAUD exp1 , exp2

Sets the baud rate of the serial I/O port(s) of the TMS9902 Asynchronous Communications Controller.

EXP1=0, port=A (CRU address hex 080)
EXP1≠0, port=B (CRU address hex 0180)
EXP2=0, baud rate=19200
EXP2=1, baud rate=9600
EXP2=2, baud rate=4800
EXP2=3, baud rate=2400
EXP2=4, baud rate=1200
EXP2=5, baud rate=300
EXP2=6, baud rate=110

BASE exp

Sets CRU base address to EXP for subsequent CRU operations.

CALL ´name´ , add { , parm }

Transfers control to the assembly language subroutine NAME located at ADD. Up to 12 parameters, PARM, are allowed in the statement (each separated by commas); these are passed to the subroutine in R0 to R11 of the call handler´s workspace. R12 contains the count of the number of parameters supplied. The /E155 Power BASIC interpreter´s environment is saved in R13, R14 and R15.

DATA item [, item]

Defines an internal data block for access by READ. ITEM is either an expression or a string.

DEF FNi { (arg) }= statement

Defines a single line arithmetic statement containing a maximum of 3, single letter, dummy variables ARG (each separated by commas). i is the function identifier. When calling FNi the dummy arguments are replaced by the actual

parameters, which may be any Power BASIC variable, array element or expression.

DIM var (num [, num])
Allocates user space for the dimensioned array VAR. NUM is the number of elements in a dimension; each dimension starts at element 0.

ELSE statement [:: statement]
When the most recently executed IF THEN statement is false, all subsequent ELSE statements up to the next IF THEN statement are executed; otherwise they are ignored.

ENABLE exp [, exp]
Enables/disables the specified 'polled interrupt' level. If EXP is negative then the specified level is disable. Used in conjunction with the IMASK statement.

END
Terminates program execution and return to keyboard mode.

ERROR ln
Specifies a Power BASIC subroutine, starting at line LN, that is to be executed via a GOSUB statement when an error occurs.

ESCAPE
Enables the ESCape key to interrupt program execution.

FOR var = exp1 TO exp2 { STEP exp3 }
The FOR statement is used with the NEXT statement to open and close a program loop. Both identify the same FOR variable VAR. EXP1 is the start value, EXP2 is the end value and EXP3 is the stepsize. If STEP is omitted, a stepsize of 1 is assumed.

GOSUB ln
Transfers control to a Power BASIC subroutine starting at line LN. The address of the statement following is stored on the GOSUB stack.

GOTO ln
Transfers control to line LN.

IF cond THEN statement [:: statement]
The statement(s) following the THEN keyword are executed if the condition COND is true.

IMASK exp
Sets the internal interrupt mask to EXP (in the range 0 to 15). Must be used in conjunction with the ENABLE statement.

IRTN

Is used to return from an interrupt routine; it restores the program environment existing prior to taking the interrupt.

(;)

INPUT item [, item]

Take input (numeric or string) from the terminal and store it into next variable ITEM in the INPUT list. Input is prompted with a question mark (?) for numeric data and a colon (:) for character data. A double question mark (??) signifies an illegal number.

{ LET } var = exp

Evaluate EXP and store the result in the variable, string variable or array element VAR.

NEXT var

Delimits a FOR loop. The variable VAR must match the FOR variable.

NOESC

Disables ESCape key on the terminal.

GOSUB

ON exp THEN GOTO line [, line]

Transfer control, via a GOSUB or a GOTO statement, to the line specified by the value of the expression (when EXP=i use the ith LINE in the list). If EXP is outside the specified range (less than 1 or greater than the number of LINEs in the list) then drop through to the next statement line.

POP

Removes the top item from the GOSUB stack.

PRINT exp [, exp]

Prints (without formatting) the value of EXP.

RANDOM exp

Sets the seed for the random number generator to the value of EXP.

READ var [, var]

Takes input from the internal DATA block and stores it in the next VAR in the READ list.

REM text

Inserts comment lines (REMarks) into a user program. The whole line is taken as a comment.

RESTOR { ln }

Resets the DATA pointer to the specified DATA line LN. If LN is not present, the pointer is set to the first DATA statement.

RETURN

Return from a Power BASIC subroutine, the return address is the last entry in the GOSUB stack.

STOP

Terminates program execution and returns to keyboard mode.

TIME { item }

Interrogate/set the 24 hour time of day clock.

ITEM=Null - Output time in HR:MN:SD format
ITEM=$var - Store time in string variable VAR
ITEM=exp1,exp2,exp3 - Set clock to specified time
(EXP1=hours; EXP2=mins; EXP3=secs)

TRAP exp TO ln

Defines the entry point, LN, of a Power BASIC interrupt subroutine for 'polled interrupt' level EXP. Level 0 can not be serviced by the TRAP statement.

UNIT exp

Designates the device(s) to receive all printed output.

EXP=0, disable all output
EXP=1, I/O port=A
EXP=2, I/O port=B
EXP=3, I/O ports A and B

4.4 OPERATORS

Arithmetic Operators

A=B	Assignment
A-B	Subtraction
A+B	Addition
A*B	Multiplication
A/B	Division
A^B	Exponentiation
-A	Unary minus
+A	Unary plus

Relational Operators

Return values of '1' (TRUE) or '0' (false).

A=B	TRUE if equal, else FALSE
A==B	TRUE if approximately equal (+ or - 9.5E-7), else FALSE
A<B	TRUE if less than, else FALSE
A<=B	TRUE if less than or equal, else FALSE
A>B	TRUE if greater than, else FALSE
A>=B	TRUE if greater than or equal, else FALSE
A<>B	TRUE if not equal, else FALSE

Boolean Operators

Return values of '1' (TRUE) or '0' (FALSE). A non-zero value variable is considered TRUE; a zero-valued variable is considered FALSE.

NOT A	TRUE if FALSE (zero), else FALSE
A AND B	TRUE if both TRUE (non-zero), else FALSE
A OR B	TRUE if either TRUE (non-zero), else FALSE

Logical Operators

Perform bitwise operations on the operand(s). Operand(s) are converted into 16 bit integers before the operation.

LNOT A	1s complement
A LAND B	Bitwise AND
A LOR B	Bitwise OR
A LXOR B	Bitwise exclusive OR

Operator Precedence

1) Expressions in parentheses
2) Exponentiation and negation
3) *,/
4) +,-
5) <=,<>
6) >=,<
7) =,>
8) ==,LXOR
9) NOT,LNOT
10) AND,LAND
11) OR,LOR
12) Assignment (=)

4.5 ARITHMETIC FUNCTIONS

Absolute Value Function - ABS

ABS (exp)
Return the absolute value of EXP.

Arctangent Function - ATN

ATN (exp)
Return the angle (in radians) whose tangent is EXP.

Cosine Function - COS

COS (exp)
Return the cosine of the angle EXP. (EXP in radians.)

Exponential Function - EXP

EXP (exp)
Return the value of the constant 'e' raised to the power of EXP.

Integer Part Function - INP

INP (exp)
Return the integer part of EXP. If EXP is too large for 16 bit integer format then floating point format is used.

Natural Logarithm Function - LOG

LOG (exp)
Return the natural logarithm of EXP.

Random Number Function - RND

RND
Return a pseudo random number between 0 and 1. The random number seed is set using the RANDOM statement.

Sine Function - SIN

SIN (exp)
Return the sine of the angle EXP. (EXP in radians.)

Square Root Function - SQR

SQR (exp)
Return the square root of EXP.

4.6 CRU FUNCTIONS

To use the following CRU functions it is first necessary to set the CRU base address via the BASE statement. (The value supplied to the BASE statement is twice the actual hardware base address.)

CRB Function

CRB (exp)

Read the CRU bit specified by the CRU hardware base address plus EXP (EXP is valid over the range -128 to +127).

CRB (exp1)= exp2

Set/reset the CRU bit specified by the CRU base address plus EXP1. If EXP2=0 then reset ('0') the selected bit, otherwise set ('1') the bit. EXP1 is valid over the range -128 to +127.

CRF Functions

CRF (exp)

Read EXP CRU bits from the CRU hardware base address. EXP is valid over the range 0 to 15. If EXP=0 then 16 bits will be read.

CRF (exp1)= exp2

Output EXP1 bits of the value EXP2 to the CRU lines starting at the CRU hardware base address. EXP1 is valid over the range 0 to 15. If EXP1=0 then 16 bits will be output.

4.7 MEMORY FUNCTIONS

BIT Function

BIT (var , exp)
Read the EXPth bit of the variable VAR. EXP should start from 1 not 0.

BIT (var , exp1)= exp2
Modify the EXP1th bit of the variable VAR. The selected bit is set to '1' if EXP2 is non-zero, otherwise it is set to '0'. EXP1 should start from 1 not 0.

MEM Functions

MEM (exp)
Read the memory byte specified by EXP.

MEM (exp1)= exp2
Set the memory byte specified by EXP1 to the value EXP2.

MWD Functions

MWD (exp)
Read the memory word specified by EXP.

MWD (exp1)= exp2
Set the memory word specified by EXP1 to the value EXP2.

4.8 MISCELLANEOUS FUNCTIONS

ADR Function

ADR (var)

Returns the address of the specified variable, string variable, array element, string array element, or byte offset within a string array element. Should only be used with the PRINT and CALL statements, and on the right hand side of an assignment (LET) statement.

NKY Function

NKY (exp)

Samples the keyboard in run-time mode. If EXP=0 then return the decimal value of the last key struck. (Zero is returned if no key was struck.) If EXP≠0 then compare the last key struck with the decimal value of EXP and return a value of 1 (they are the same) or 0 (they are not the same).

SYS Function

SYS (exp)

Obtain system parameters generated during program execution.

EXP=0, parameter=input control character
EXP=1, parameter=error code number
EXP=2, parameter=error line number

TIC Function

TIC (exp)

Samples the real time clock and returns the current TIC value minus the value of EXP. One TIC equals 40 milliseconds (using a TMS9995 processor at 3MHz, as on the standard TM990/E155 CPU board). TIC (0) obtains the current value.

4.9 STRING OPERATIONS

<$var>	denotes either a literal string, enclosed in quotes, or a string variable
$<var>	denotes a string variable

A varible is specified as being a string variable by preceeding the variable name by a dollar sign ($).

An individual byte within a dimensioned string variable can be accessed by following the last array subscript with a semicolon (;) and the byte position.

Character Assignment. Copy characters into the string variable until a null (zero) byte is found.

```
$<var> = <$var>
```

Character Pick. Copy EXP characters into the string variable and then terminate the string with a null byte.

```
$<var> = <$var> , exp
```

Character Concatenation. Concatenate the strings into the string variable (in the specified order) and terminate the completed string with a null byte.

```
$<var> = <$var> + <$var> [ + <$var> ]
```

Character Replacement. Copy EXP characters into the string variable (do not add the null byte).

```
$<var> = <$var> ; exp
```

Character Insertion. Insert the characters into the string variable.

```
$<var> = / <$var>
```

Character Deletion. Delete EXP characters from the string variable.

```
$<var> = / exp
```

Byte Replacement. Replace the specified byte(s) by the character equivalent of the ASCII code(s) EXPs.

```
$<var> = % exp [ % exp ]
```

String Comparison. Character strings can be compared by:

```
IF <$var> <relop> <$var> { , <exp> } THEN <sequence>
```

where <relop> = relational operator

if the second string is followed by a comma, the expression following indicates the number of characters to be compared.

Convert from ASCII to Binary. A character string can be converted into a number by:

```
var1 = <$var> , var2
```

VAR1 is where the number will be stored. VAR2 is an 'error variable' (the delimiting character is stored in the first byte of this variable).

Convert from Binary to ASCII. A number is converted to a string simply by assigning the number to a string variable. The string is automatically terminated with a null character.

```
$<var> = exp
```

Formatted conversions can be made by preceding EXP with the formatting operator (#) and a string.

```
$<var> = # <$var> , exp
```

String Functions

```
ASC ( $<var> )
```

Returns the ASCII decimal value of the first character in the specified string.

```
LEN ( $<var> )
```

Returns the length of the specified string. Zero is returned if the string is the null string.

```
MCH ( $<var1> , $<var2> )
```

Return the number of characters that are the same in the two strings. A zero is returned if no match is found.

```
SRH ( $<var1> , $<var2> )
```

Return the character position of where the first string is located in the second. A zero is returned if the search is unsuccessful.

4.10 INPUT OPTIONS

INPUT feature item [del feature item]

ITEM	Either a variable, a string variable, or an array element
DEL	One of:
,	Delimit ITEMs in the INPUT list
;	Delimit ITEMs in the INPUT list. Suppress <CR> <LF> if at the end of the statement line
FEATURE	One of:
string	Prompt with STRING then get input
? ln	Upon an invalid input or control charcater, a GOSUB to the line LN is executed. On return, the line following the INPUT is executed.
% exp	Requires entry of exactly EXP characters - for one ITEM only
# exp	A maximum of EXP characters to be entered - for one item only
;	Suppress prompting
null	Prompt (? for numeric, : for character) and and then get input

4.11 PRINT OPTIONS

PRINT feature item [del feature item]

ITEM	Either a variable, an expression, a string variable, a string, or an array element
DEL	One of:
,	Delimit ITEMs in the PRINT list and TAB to the next print field
;	Delimit ITEMs in the PRINT list. Suppress <CR> <LF> if at the end of the statement line

FEATURE	One of:
string	Output STRING
TAB (exp)	TAB to column specified by EXP
# exp	Print EXP in hex free format
# , exp	Print EXP in hex (word)
# ; exp	Print EXP in hex (byte)
# string	Decimal formatting. STRING can be composed of:

9	Digit holder
0	Digit holder or force 0
$	Digit holder and floats $
S	Digit holder and floats sign
<	Digit holder before decimal and floats on negative number
>	Appears after decimal if negative
E	Sign holder after decimal
.	Decimal point specifier
,	Comma in output - suppressed if before significant digit
^	Translated to decimal point on output

4.12 FLOATING POINT XOP PACKAGE

For use with assembly language routines.

FORMAT XOP GA , OP

where GA - General memory address operand
OP - XOP number

FPAC - Floating Point Accumulator

XOP no.	Function
0	LOAD FPAC with 6 byte number addressed by GA
1	STORE FPAC in 6 byte number addressed by GA
2	ADD 6 byte number addressed by GA to FPAC, store result in FPAC
3	SUBTRACT 6 byte number addressed by GA to FPAC, store result in FPAC
4	MULTIPLY FPAC by 6 byte number addressed by GA, store result in FPAC
5	DIVIDE FPAC by 6 byte number addressed by GA, store result in FPAC
6	SCALE adjusts FPAC's exponent to value of byte addressed by GA
7	NORMALISE FPAC - 1st hex digit of mantissa is non-zero. Operand not used
8	CLEAR FPAC. Operand not used
9	NEGATE FPAC - change 1st bit. If FPAC=0 then no change. Operand not used
10	FLOAT FPAC's 2nd word - 16 bit twos complement number to floating point. Operand not used

Converting Integer to Floating Point

1) Set words 1 and 3 of 6-byte reserved area to zero.
2) Store integer number in 2nd word of area.
3) LOAD this 6-byte number into FPAC.
4) FLOAT FPAC.
5) STORE FPAC in 6 byte area.

```
DECNO BSS  6
FLPT  BSS  6
      .
      CLR  @DECNO
      CLR  @DECNO+4
      LI   R0,NUM
      MOV  R0,@DECNO+2
      XOP  @DECNO,0
      XOP  0,10
      XOP  @FLPT,1
      .
```