PL65

1.1 WHAT PL65 IS FOR

PL65 stands for Programming Language for 6500 series microprocessors, a derivative of which, the 6502 chip, is used in the ATARI 8-bit family of home computers.

The language has been designed for use as an alternative to assembler and as such it has been created with speed and versatility as the primary objectives. It is implemented as a single pass compiler which generates directly executable machine code output with no intermediate linking process.

Program development is normally broken down into 6 distinct stages: analysis, design, coding, testing, documentation and maintenance. In assembler the first four of these stages must be tackled before even the feasibility of a project can be checked. This can obviously lead to a waste of time and money if a concept proves impractical.

PL65 allows you to get your program concept into working code in the minimum time scale possible by reducing the tedious initial coding stage. Programs can be written and tested using natural high level algorithms and modular design. A fast edit-compile-run cycle allows quick testing of programs and then, when feasibility has been established, the code can be 'tarted up' with the extra frills added and the time critical portions converted to assembler.

PL65 allows you to be as structured or unstructured as you wish, a versatile language should not put any constraints on the programmer. It allows you to write code that needs no operating system to run (as does assembler) and can be originated to run in whatever part of memory you desire. Alternatively you can use the supplied library routines and built in compiler facilities to create programs that are directly runnable with all the built in facilities of the operating system.

Programs can perform a wide range of tasks, from simple utilities to powerful word processors, even arcade games are possible with PL65's fast execution speed and built in assembler.

All programs created with PL65 are independent, that is, they do not need any resident language interpreter to run them. Once you have compiled your program it becomes a completely independent machine code program.

You do not have to pay royalties if you were to sell any program written using PL65 because the program would have been created by you and you alone would hold the copyright. However compiled programs would contain our runtime routines, and we do require you to include a notice in the software or accompanying literature, acknowledging our copyright of these routines.

1.2 DISK ORGANISATION

The PL65 master disk contains the following files:

```
DOS. SYS
PADDLE.LIB
DUP. SYS
SETUP. SYS
RAMD.COM
PL65. COM
READ.ME
GRAPHICS.LIB
UDG.LIB
PEEKPOKE.LIB
SOUND.LIB
DEBUG.LIB
AUTORUN.SYS
KED.COM
TERMINAL.LIB
STRING.LIB
PMG.LIB
SAMPLE.PRG
```

DOS.SYS and DUP.SYS are the standard file management system, the version of DOS is 2.5 which enables both single and enhanced density disk drives to be used.

The standard DOS utility programs SETUP.SYS and RAMD.COM are included. If you have a 130XE computer then you can rename RAMD.COM to RAMDISK.COM - this will then be automatically loaded at power up and lets you use the extra memory as a virtual disk drive (accessed using D8: as the device specifier).

SETUP.SYS allows you to tailor DOS to your own system. When running this program make sure that there are 7 active open files else PL65 may not be able to operate correctly. The supplied DOS is configured to allow up to 2 disk drives, so you should not need to use this program unless you have more than 2 drives (or to enable fast writing).

NOTE: SETUP.SYS can only be run from DUP, it cannot be run from KED or PL65.

PL65.COM is the compiler and KED.COM is the keyboard text editor. Files with the extender .LIB are library source files and those with the extender .PRG are program source files.

All files with the extender .COM can be loaded using option 'L' from the DUP main menu.

READ.ME is a text file that contains up to the minute information on errors or omissions in the manual, changes in the software and other general information. This can be loaded and read using either of the editors.

When you first receive your copy of PL65, format some spare data disks and then copy across KED and the library files to one of these disks. KED is more powerful than PL65's built in editor and is more suitable for creating and editing larger source files.

Since source code entering and editing is a separate process to compiling, it makes sense to use a backed up version of KED to do most of the donkey work rather than put the system disk at risk with constant use.

You can back up KED as many times as you wish, but PL65 is copy protected and will only work when loaded from the master disk.

Once loaded both PL65 and KED can work independently of any system disk so this can be replaced with a data disk.

Booting the disk

Boot the master disk in drive 1 with no cartridges installed. If you have an XL or XE type computer don't forget to hold down the OPTION key to disable BASIC, failure to do this will result in the loss of 8 valuable K of memory.

The control program AUTORUN.SYS will automatically load and run. This program lists the runnable files on the disk (with the extender .COM) and allows them to be run without going into DUP.

Part II   KED the keyboard text editor

The keyboard editor is used to enter source code into the computer. Source code is saved to the disk and can then be accessed by the compiler. Source code is stored in normal ASCII format and thus can be modified by any available text editor or word processor.

KED is loaded using the binary load option "L" from DUP. Remember on XL and XE systems to hold down the OPTION key to disable B6SIC when first booting the system.

2.1 FILENAMES

Whenever KED prompts for or expects a filename, the filename must be entered using the filename rules described in the ATARI DOS literature. If no device specifier is supplied then KED will assume device 'D:'.

It should be noted that the built in operating system has device independence, which means files can be transferred to and from any device on the system (provided the device has the facilities). This can be handy in some cases, ie a printout of a file can be obtained by selecting P: as the save filename. However some devices can cause confusion ie selecting the screen handler as the output device (S:) whilst editing a file will write rubbish to the screen corrupting the file display.

2.2 THE MAIN MENU

Upon loading KED will automatically run and produce the greeting message along with the main menu.

```
KED keyboard editor V1.2
Copyright (C) 1987 Noahsoft

1 Load file
2 Save file
3 Edit file
4 Disk index
5 Create new file
6 DOS
7 Run binary file
8 Utilities

Which ?
```

The file currently in memory is shown below the menu along with the file size and the amount of free memory.

2.3 LOADING FILES FROM DISK

To edit a previously saved file select option 1 from the main menu. KED will ask you for the filename, enter this and then press return.

Once loaded KED will return to the main menu. You may then select option 3 to edit the file.

2.4 SAVING FILES TO DISK

To save a file to disk select option 2 from the main menu. KED will ask you for the filename. If you wish to use the same filename as the input filename then simply press return, else enter the desired filename.

2.5 CREATING NEW FILES

If you wish to create a new file then select option 5, KED will then ask you for a filename.

The filename entered here may be changed later when the editing session is finished and the program ready to save to disk.

Once entered the screen will clear and the main editing screen will appear. With no text within the file the screen will show the minimum file contents (one return character).

2.6 DISK INDEX

Selecting option 4 from the main menu enables the disk directory contents of any drive to be displayed on the screen. KED will ask for a drive (1 to 8), select this using the appropriate numeric key. Drive 1 will be selected if any other than the numeric keys are pressed.

2.7 RETURNING TO DOS

Selecting option 6 from the main menu will exit KED and return control back to the DUP program. Before selecting this option make sure a disk which contains the DOS files is in drive 1.

2.8 RUNNING BINARY FILES

Selecting option 7 from the main menu enables a machine code binary file to be run directly from KED. The file must be a load and go file as described in the DOS handbook. You can use this option to run any program compiled with PL65 or you can run PL65 itself. Upon entering KED will ask for the filename of the binary file, enter this and then press return. The program will load and run automatically.

2.9 UTILITIES MENU

Selecting option 8 from the main menu will display another menu with the utilities options. These utilities enable routine file handling functions such as deleting and renaming to be carried out directly from KED without having to return to DOS. Once entered the menu will be displayed:

```
Utilities

1 Delete file
2 Rename file
3 Format disk
4 Disk index
5 Lock file
6 Unlock file

X Return to main menu
```

The utilities in the menu are selected as with the main menu by typing the option number. All of the utilities work in a similar fashion to the DUP program. For the delete, lock and unlock utilities KED will prompt for a filename to act on - once entered the action will be carried out immediately with no 'are you sure' type message.

The disk index option is duplicated from the main menu for ease of use.

The format disk option will ask for a drive number. It then asks you to type "Y" to carry on, any other key will abort the format.

2.10 EDITING TEXT

To edit a file that has previously been loaded, select option 3 from the main menu. The screen will clear and the first 23 lines of the file will be displayed.

The first thing to notice is the change in the screen characteristics. An extra line is displayed at the top of the screen, this is used to convey file information to the user along with error messages and instructions when using certain functions.

Normally the message line will show the status of various editing options, eg tab setting, caps or lower case selected, typeover or insert mode and the indentation level. The state of the inverse/normal video mode is indicated by the 'caps/lc' annotation, which will be displayed in inverse video when in inverse mode.

2.11 USING THE KEYBOARD

Most of the special editing features are accessed with control key commands - you hold down CTRL while pressing another key. Some functions are selected with the SELECT or OPTION keys, again these are selected by holding down these keys while pressing another key.

To return to the main menu type OPT ESC.

To insert text into the file simply type the characters you wish to appear. If the character you wish to insert into the text requires a key combination that would normally actuate an editing function, then first press the escape key, this will force the character to be inserted.

Owners of older 400 and 800 machines will notice that the action of the caps/lower key has been altered to act like XL and XE machines, that is, it now toggles between caps and lower case (although this is only in the edit mode).

2.12 CURSOR MOVEMENT

The cursor can be moved about the screen using the normal cursor keys in conjunction with the control key. If an attempt is made to move the cursor off the screen vertically then the screen will scroll in the appropriate direction. If an attempt is made to move the cursor beyond the extremes of the file then an error message will be displayed in the message window and the attempt ignored. There are a number of other key combinations to move the cursor around in the file, eg CTRL T, M and B moves the cursor to the top, middle and bottom of the screen. SEL T and B moves the cursor to top and bottom of the file. CTRL W moves the cursor onto the first character of the next word. CTRL A and S moves the cursor to the start and end of the current screen line. CTRL O and P moves the cursor up and down by one complete screen page.

2.13 MODES OF OPERATION

Two modes of operation are supported, they are insert and typeover mode. KED defaults to insert mode, typeover mode can be toggled in and out using OPT M.

Insert mode will cause all non edit function type key presses to insert the key character into the text at the current cursor position, expanding the file.

Typeover mode replaces the character under the cursor with the character entered, which does not expand the file. You can still open up space for an insertion into the file using CTRL INSERT.

The insert mode has an auto indentation feature. Indentation is used in high level languages to make programs more readable.

The indentation level is shown in the message window, if this is greater than zero then every time a RETURN is entered a number of spaces corresponding to the indentation level is automatically inserted into the file at the start of the newly created line.

The indentation level is incremented using CTRL I (or SEL I for incrementing without creating an EOL) and decremented using CTRL U (or SEL U to decrement without the EOL).

2.14 SEARCH AND REPLACE FACILITIES

You can search for the occurrence of a particular text string by using the SEL S function. KED prompts for the search string in the message window. You may enter a string containing any printable characters (including EOL's), the string must be less than 16 characters. Once you have entered the string, press START and the search will begin.

KED searches for an exact match with the string which includes leading and trailing blanks.

If a match is found then the screen will repaint with the line that the matching text is on at the top of the screen. The cursor will be positioned on the first character of the text.

If no match is found then a 'No matching text' message will be displayed in the message window.

Once a search string has been entered, you can search for more than one occurrence at any time simply by pressing the START key.

Searches always start at the current cursor position+1 and proceed towards the end of the file. If you wish to search the whole of the file, then you will have to first position the cursor at the start of the file (using SEL T).

The string replace facility enables one or more occurrences of a particular text string to be replaced with another string. To select this function press SEL R. First KED will prompt for a search string, this should be entered as described above. KED then prompts for a

replace string, this should be entered in exactly the same way as the search string - pressing START when ready. The replace string does not have to be the same length as the search string.

After both strings have been entered KED will ask 'All or Verify'. Enter the appropriate option using the A or V keys. If 'ALL' is selected then all occurrences (forward of the current cursor position) of the search string will be replaced. If 'verify' is selected then KED will stop at each match of the search string (repainting the screen at the string match point) and ask 'Skip or Replace'. Select either option using the S or R keys. If 'Skip' is selected then KED does not replace that occurrence but carries on searching for the next occurrence. If 'Replace' is selected then the string will be replaced.

Some intensive replace actions may take a bit of time, KED will display a 'working' message during this time.

2.15 CUT AND PASTE FACILITIES

KED provides comprehensive facilities for moving blocks of text around within the file. A text block which needs manipulating is selected by using the SEL M function to mark one end of the block and then the cursor is moved to the other end of the block.

You can then choose to either copy or delete the block into the cut buffer with the SEL C and SEL D functions.

Once in the buffer the text can be pasted back into the file at any point and as many times as required. To do this move the cursor to the point in the file that you want the text to appear and then press SEL P.

The contents of the buffer are maintained until overwritten with another SEL D or SEL C action, even if another file is loaded into memory.

The buffer size is limited to 2048 characters. If the block you are intending to delete is larger than this then an 'Exceeds buffer size' message is displayed along with 'Cont Y/N'. If 'N' is selected then the action is aborted. If 'Y' is selected then the whole block will be deleted, but the buffer will only hold that part of the text that its size would allow.

If you need to cut and paste blocks larger than the buffer will allow then use the input and output facilities.

2.16 INPUT AND OUTPUT FACILITIES

Input and output facilities are provided to allow blocks of text to be transferred through the operating system to and from any device on the system.

The text block to be output can be selected by using the SEL M function to mark one end of the block and then moving the cursor to the other end of the block.

You can then choose either SEL K to append the text to the end of a file on disk, or SEL O to output the text to any device on the system. Output devices may include a printer(P:), cassette(C:) or a disk file.

The append function is useful for creating programs using selected routines from various other source files. Instead of re-writing the routines in your new file, you simply load in your old file and then append the area of the program containing the routines you require to the end of your new file.

The output function is useful for creating listings of selected areas of a file on a printer, or it may be used in conjunction with the input text function as a cut and paste facility with the whole disk as the buffer.

The input text function is used to merge a file stored on disk with the file in memory. To do this simply move the cursor to the point in the file that the text is to be inserted, then press SEL N.

All of the input and output functions will require a device/filename, KED will prompt for this in the message window. Enter your filename and/or device and then press START to activate the function.

2.17 EDIT FUNCTION SUMMARY

Cursor movement

```
CTRL - Moves the cursor up one line.

CTRL = Moves the cursor down one line.

CTRL + Moves the cursor left one character.

CTRL * Moves the cursor right one character.

CTRL A Moves the cursor to start of the current screen line.

CTRL S Moves the cursor to the end of the current screen line.

CTRL O Scrolls the screen by one complete screen page towards the start of
the file.
```

```
CTRL P Scrolls the screen by one complete screen page towards the end of
the file.

CTRL W Moves the cursor onto the first character of the next word.

CTRL E Moves the cursor onto the first character of the previous word.

CTRL T Moves the cursor to the top left hand corner of the screen.

CTRL M Moves the cursor to the left hand center of the screen.

CTRL B Moves the cursor to the bottom left hand corner of the screen.

SEL  T Moves cursor to the top of the file.

SEL  B Moves cursor to the bottom of the file.
```

Text deletion

CTRL BS (backspace) Deletes the character under the cursor.

BS Backspace will delete the character to the left of the current cursor position.

SHFT BS Deletes the whole of the current screen line.

SEL BS Deletes all the text from the start of the file up to the current cursor position. A confirmation message is first displayed in the message window, follow the directions given.

SEL CTRL BS Deletes from the current cursor position to the end of the file. A confirmation message is displayed in the message window, follow the directions given. NOTE: this is a three key function.

CTRL Q Deletes the word to the right of the cursor. For this function a word is defined as a sequence of alpha-numeric (upper and lower case letters and the digits 0-9) characters. If the character under the cursor is not alpha-numeric then only the single character is deleted.

Insertion

SHFT INSERT Inserts an EOL character into the file, effectively creating a new line. The EOL char is not inserted at the current cursor position but rather at the start of the current line.

CTRL INSERT Inserts a space character into the file at the current cursor position. The space will be inserted even in typeover mode.

TAB Inserts X spaces into the file at the current cursor position, where X may be set by the user using the OPT T function. Default for tab is 6.

ESC Escape forces the next keystroke to be inserted as an ASCII character directly into the file at the current cursor position, regardless of any option or select or control key being pressed. This enables the ATARI control characters to be inserted into the text. When escape is enabled the message "esc" will appear in the message window.

CTRL C Changes the case of the character under the cursor (if it's an alpha char).

Indentation

CTRL I Increments the indentation level and starts a new line. This function has the same effect as pressing the RETURN key, however the indentation level is incremented. This means that a number of spaces is inserted at the start of the next line. All future presses of the RETURN key will force these spaces to be inserted. The indentation level is indicated in the message window.

CTRL U Decrements the indentation level and inserts an EOL character into the text. This function has the same effect as the increment indent function except that the indentation level is decremented. The spaces are still inserted but one less than the last line.

SEL I Increments the indentation level. This function is used when the indentation level needs to be changed without affecting the file. The new indent level will become effective the next time the RETURN key is pressed.

SEL U Decrements the indentation level. This function is used to decrement the indentation level without affecting the file. The new indent level becomes effective the next time the RETURN key is pressed.

Cut and paste

SEL M Mark the current file position. This function marks one end of a text block and is used in conjunction with the copy and delete block functions.

SEL D Delete the block of text between the current cursor position and the previously marked position into the cut buffer. A maximum of 2048 characters can be deleted in one go.

SEL C Copies a block of text between the current cursor position and the previously marked position into the cut buffer. A maximum of 2048 characters can be copied in one go.

SEL P Paste the contents of the cut buffer into the file at the current cursor position.

Search and replace

SEL S Search file for match with text string. The text string will be prompted for in the message window. A maximum of 16 characters can be entered and is terminated by pressing the START key which activates the search. If no match is found within the file then a "no matching string" message will be displayed else the screen will display the file contents at the point where a match is found. Future searches for the same string can be activated by simply pressing the START key. KED always searches for an exact match with the search string which includes trailing and leading blanks.

SEL R Replace string. The editor will prompt for a search string which is entered as described above. The editor will then prompt for a replace string, which is entered in the same way as the search string. The editor then asks "All or Verify", select the desired option with the A or V keys. If "All" is selected then all matching strings throughout the file (forward of the current cursor position) will be replaced with the replace string. If the verify option is selected then the editor will stop at each occurrence of the string and ask "Skip or Replace". The desired action can then be selected with the S or R keys.

START Searches forward of the current cursor position for the next occurrence of the last entered search string. If no search string has been entered (using SEL Sl, then an error is reported.

Input and output

SEL O Outputs the block of text between the last marked file position and the current cursor position to device. The output filename will be prompted for in the message window. The filespec may be any legal ATARI device (to output to a printer enter P:) and is terminated with the START key.

SEL K Appends the block of text between the last marked file position and the current cursor position to a file stored on disk. The filename will be prompted for in the message window and is entered in the same way as for the output text function.

SEL L Loads text from a file stored on disk and inserts into the file in memory at the current cursor position. The input filename will be prompted for in the message window and is entered in the same way as in the output text function.

Editing options

OPT ESC Exits the editing mode and returns to the main menu.

OPT L Set left margin. The margin can be moved across the screen using the keys shown in the message window. Press the return key to return to edit mode.

OPT R Set right margin. The margin can be moved across the screen using the keys shown in the message window. Press the return key to return to the edit mode.

OPT T Set tab. The number of spaces inserted by pressing the tab key can be altered using this function. Follow the directions in the message window to increment or decrement the tab. Pressing return exits back to the edit mode.

OPT I Set invisible EOLs. This function turns off the screen display of EOL characters.

OPT V Set visible EOLs. This function turns on the screen display of EOL characters.

OPT M Toggles the editing mode between typeover and insert.

OPT F Displays the amount of free memory left that the file may expand into.

OPT C Changes the background color. Succesive presses will cycle through all sixteen of the available colors.

OPT B Changes the background brightness. Succesive presses will cycle through the eight available brightness levels and then onto the next color.

OPT N Changes the character brightness. Succesive presses will cycle through the eight available brightness levels.

Part III COMPILER OPERATION

The system disk is booted in drive 1 and will power up into the AUTORUN.SYS program. This program allows you to select KED, PL65 or DUP run directly. If you have an XL or XE type computer, don't forget to hold down the option key to disable BASIC.

If you are in the DUP program you can run PL65 or KED by using option 'L' from the DUP menu. The two programs have the filenames

D:KED.COM and D:PL65.COM

If you are already in KED you can run PL65 directly using the 'Run binary file' option from the main menu.

3.1 COMPILER MAIN MENU

Once loaded the compiler will display the greeting message and the main menu.

```
PL65 compiler Ver 1.2
Copyright (C) 1987 Noahsoft

1 Compiler
2 Editor
3 Options
4 Disk index
5 Xref
6 DOS
7 Run binary file

Which ?
```

PL65 is a menu driven system. To select a particular option from a menu simply press the option number. See the section on KED for a description of options 4, 6 and 7.

3.2 COMPILER MODE

Selecting option 1 enters the main compiler mode, the user will then be promted to enter four filenames. These are the source filename, output filename, list filename and the error log filename.

Of these four files only the first - the source filename - is needed by the compiler, the other three are optional.

The source filename is the filename of the program that you want to compile. This file may be compiled from a disk drive or it may be in memory (previously loaded from the editor menu). To compile a file from memory enter 'M:' as the source filename.

The output filename is the file that the compiled object code is written to. This is optional to allow dummy runs for error checking purposes.

The list file is the file to which a listing of the program which includes a line map of memory location addresses for the start of each program line is written. If you have a monitor program you can use the information in this file to inspect the compiled object code.

NOTE: Line maps addresses are only accurate if each logical line ends in a semi-colon.

The error log file is the file to which a copy of the error reports displayed on the screen during compilation, are written.

The compiler will prompt for each of these filenames in turn. If an optional file is not required just press the RETURN key.

It should be noted that error log files and list files are not limited to the disk drives, but can be any legal ATARI operating system output device. To direct a listing to the printer simply select 'P:' as the list filename.

Once the filenames have been entered PL65 will begin to compile the source file.

The compiler can be halted at any time (either during compilation or when entering the filenames) by pressing the break key.

3.3 ERRORS AND THE ERROR LOG

During compilation a running record of compiler actions is automatically displayed on the screen, this is the error log. If an error log filename has been entered then a copy of these reports is sent to the selected file.

In its simplest form the error log will show the filename of the file currently being compiled. It also shows files that are LINKed or INCLUDed with the main file.

The main reason for the error log is to report errors in the source text. PL65 can detect syntax and program inconsistency type errors, it can't find errors in the logic of your programs.

The ability of the compiler to recover and carry on compiling after an error depends largely on how badly flawed the source file is.

If an error occurs during compilation then an error message and number will be shown on the screen along with the current line of input source text. The point in the line where the compiler thought the error occurred will be highlighted in reverse video.

If an error occurs inside a procedure then the procedure name will also be displayed.

eg:

```
Error 31 in routine-PLOT
POS(X,Y)
```

This simplifies exact location of errors within the source text.

Because of the single pass nature of the compiler, some errors can't be checked until a procedure END statement is reached. All of these errors will indicate that the END statement is in error.

eg:

```
Unresolved reference to SKIP in routine-INIT
END
```

If you have a number of references to the same label and then forget to declare the actual label then there will be one error report for each reference.

If a FORWARD procedure is declared without an corresponding BODY then this will not be reported until the end of the compilation.

The compiler will report that the END part of the MAIN routine is in error.

Operating system errors (those with numbers >127) will cause the compilation to abort this is also true of errors marked FATAL in the error message section of this manual.

Error cascades occur most frequently when using nested program flow statements. These types of errors are when one badly placed error results in a program flow breakdown and thus a whole string of errors are reported.

eg:

```
WHILE A<10 DO
   IF A=3 THEN
      GOTO LOOP
   ELSE
     REPEAT
        B=A*3+4
   ENDIF
ENDWHILE
```

In the above example the compiler will not allow the IF and the WHILE to terminate because the innermost level (the REPEAT) has a missing terminator (UNTIL). When the procedure END is reached the compiler will report that all of the statements are unterminated.

Some errors may mask other errors - this happens because the compiler error recovery routine works by skipping ahead in the text to find a reference point to continue compiling. These errors can be found by recompiling after correcting the original errors.

When the compilation is finished the total number of errors will be shown.

A summary of error messages are contained in appendix B of this manual. Most of the errors are self explanatory, a more detailed description of those errors which may need further explanation follows.

Error 1 Symbol already in use.

There is already a symbol of this name within the same scope. See the section on identifier scope.

Error 6 No such symbol.

You have tried to access an identifier that does not exist.

Error 8 Variable declaration or BEGIN expected.

This error occurs in the local variable declaration part of a procedure header. You have attempted to do something other than declare a variable or constant.

Error 9 Syntax error.

A valid statement, procedure call or assignment statement is expected here.

Error 11 Memory full.

See the section on memory management.

Error 13 Identifier expected.

This error occurs when an identifier is expected. It means that the first character of the identifier is not a letter.

Error 14 Incorrect expression argument.

The expression operand is incorrect. See the section on runtime expressions for valid operands.

Error 26 Identifier character invalid

The second or subsequent characters of an identifier is not alpha-numeric or an underline character.

Error 27 Illegal expression argument type.

This error occurs in compile time expressions. The expression operand is incorrect. See the section on compile time expressions for valid operands.

Error 28 Unconvertable digit in number.

The compiler has attempted to interpret an expression operand as a number. This error means that one of the characters in the number is not a valid digit.

Error 29 Command expected.

The compiler is expecting a command and this isn't one. See the language description for valid commands.

Error 30 Can't use statement name as identifier.

You have attempted to use a reserved word or symbol as an identifier.

Error 36 Compilation completed with no MAIN routine.

You have either forgot to include the MAIN routine or failed to LINK to the file in which it is contained.

Error 37 Unexpected EOF.

The file finishes at an illegal point in the program. This error may occur if you have INCLUDed a file which has no ENDFILE at the end.

Error 38 To many parameters in procedure definition.

Definition exceeds the maximum number of parameters allowed (128).

Error 39 Expression too complicated.

This error may occur in runtime expressions. It usually means that the level of parenthesis is too deep for the compiler to handle. The compiler can handle around twenty levels of parenthesis so an expression has to be very complicated indeed for this to happen.

Error 40 Compiler stack overflow.

The level of statement nesting in a procedure is too deep. The compiler will allow around 64 levels of statement nesting. If this error happens split the nesting by putting halve of the nest into a separate procedure.

Error 41 Compiler stack underflow.

This is an internal compiler error check, it should not occur under normal circumstances. If it does then it could mean that a succession of errors has confused the compiler in such a way that it has had to

abort the compilation. If the error occurs with no other preceeding error then contact us.

Error 43 BASED type must be a POINTER.

See the section on variables for BASED types.

Error 44 Branch out of range.

This is an assembler error. The 6502 branch instructions have a limited range. You will have to restructure your program to make the label closer to the branch.

Error 46 Has illegal addressing mode.

This is an assembler error. The addressing mode for this particular instruction is invalid.

Error 47 Value >255.

This error means that a compile time expression has evaluated to a 16 bit number when a byte value was expected, in assembler some operands must be byte values.

Error 48 Statement expected - illegal use of symbol.

This error occurs within procedures. It may mean that you have tried an assignment to an identifier which is not a variable, or you may have tried to call a procedure which is actually a function or some other type.

Error 50 Too much DATA for size of variable.

The supplied data is greater that the dimensioned size of the variable.

Error 53 String comparison error.

This happens in runtime expressions. You have tried to compare a string to something which is not a string.

Error 63 Not a FORWARD procedure.

You have attempted to declare the BODY of a procedure when the identifier is not that of a FORWARD procedure.

Error 64 Undefined parameter error in BEGIN.

This error may occur if the parameter list of a procedure has an error. The corresponding BEGIN statement has found an inconsistency with which it cannot cope.

Error 65 Parameters not allowed in MAIN and INTERRUPTS.

See the section on procedure types for more information.

Error 70 No matching structure initiator.

This error occurs when a statement terminator is encountered which has no matching initiator,
eg: ENDIF without IF.

3.4 MEMORY MANAGEMENT

Once loaded into memory PL65 has to provide memory resources for three different applications.

1 - The text area, for source files loaded by the editor.
2 - The symbol table, generated during compilations.
3 - The compiling workspace, the temporary area that code is compiled into before being written to disk.

All three of these areas are constantly variable in size.

There is around 20k of free memory when PL65 is first run, this is soon eroded when large text files are loaded into memory.

At the start of a compile sequence the symbol table is cleared. Each new identifier which is declared is added to the symbol table which grows in size. PL65 has to remember all global identifiers and it has to have these in memory so that they are immediately available.

If an error 11 is generated during compilation this means that no more memory is available for the symbol table to grow into. This error should really be impossible with the full amount of free memory, but it is possible if large text files are held in memory.

PL65 allows complete disk to disk compilations so there is no real need for a text file to be in memory. If you start getting troubles with error 11 then save your text file to disk and erase the file in memory (by creating a new file), and then compile the file directly from disk.

3.5 EDITOR MODE

PL65 contains a built in editor which has a subset of the facilities available in KED. The object of the micro editor is to allow correction of minor errors in source text without having to keep swapping between editor and compiler programs.

PL65 also has the ability to directly compile a source file which has been loaded by the editor. This is achieved by selecting M: as the source filename.

When the micro editor is selected from the main menu, the editor menu is displayed.

```
Micro edit

1 Load file
2 Save file
```

```
3 Edit file
4 Disk index
5 Create file

X Return to main menu

Which ?
```

Loading, saving and creating files is similar to KED, instead of ESC use the break key to abort an action.

The disk index option is duplicated from the main menu for ease of use.

Typing X returns control back to the main menu. The file in memory is not altered or disturbed by any compiler action.

3.5.1 EDITING FILES

Upon entering the edit mode the screen will clear and display the first 24 lines of the file. Micro edit differs from KED in that no message line is displayed at the top of the screen. Messages will be displayed in reverse video at the bottom of the screen only when required.

Cursor control

The cursor can be moved up down left and right using the normal cursor keys in conjunction with the CTRL key. Further cursor control functions are tabulated below.

CTRL O Moves the cursor by one complete screen page towards the start of the file.

CTRL P Moves the cursor by one complete screen page towards the end of file.

CTRL A Moves the cursor to the start of the current screen line.

CRTL S Moves the cursor to the end of the current screen line.

CTRL W Moves the cursor to the start of the next word.

CTRL E Moves the cursor to the start of the previous word.

CTRL T Moves the cursor to the top left hand of the screen.

CTRL H Moves the cursor to the middle left hand of the screen.

CTRL B Moves the cursor to the bottom left hand of the screen.

SEL T Moves the cursor to the top of the file.

SEL B Moves the cursor to the bottom of the file.

Other facilities

CTRL C Changes the case of the character under the cursor.

CTRL BS (backspace) Deletes the character under the cursor.

SHFT BS Deletes the whole of the current screen line.

BS Deletes the character to the left of the cursor.

SHFT INS Inserts a new line at the start of the current screen line.

TAB Inserts 6 spaces at the current cursor position.

OPTIONs

The following functions are enabled by holding down the OPTION key in conjunction with the described keys.

OPT I Turns off screen display of EOL characters.

OPT V Turns on screen display of EOL characters.

String Search

SEL S Sets up search model. The editor will prompt for a search string at the bottom of the screen. The string is entered (up to 16 characters) and then START is pressed to initiate the search. The search will start from the current cursor position and proceed towards the end of file. If no match is found then an appropriate message will be displayed. If a match is found then the screen will repaint with the line that the matching string is on at the top of the screen, the cursor is positioned on the first character of the matching text. If the whole file is to be searched then SEL T should be typed to position the cursor at the top of the file.

START Searches forward of the current cursor position for the last entered search string.

3.6 COMPILER OPTIONS

Selecting option 3 from the main menu will display another menu containing the compiler options along with the current state of each option. The options are:

```
1 ORG address.......$2100
2 Stack address.....$2000
3 Zero page address.$80
4 Zero page length..S80
5 Append MEMLO......YES
6 Make autorun......YES
7 Append Runtimes...YES

X Return to main menu
```

This menu allows you to control the locating of your programs in memory. At power up PL65 sets default values for each of the shown parameters. The values are chosen such that a program may be compiled in the correct format to be runnable using option "L" from the DUP program menu.

ORG address

This is the address in memory to which the next compilation will be originated to. To change this address enter the option number the compiler will prompt for a new ORG address, enter the new address (in hex) and press return. The menu will then be updated with the new address.

PL65 actually relocates the runtime code to start at this address and the code generated from a compilation is added to the first address free following this code.

STACK address

This is the address in memory of the software stack. The stack needs one complete 256 byte page of memory. The user has the option of relocating the stack to any page in memory. To change the stack base enter the appropriate option number, the compiler will then prompt for the new stack address. Enter the new stack address (in hex) then press return. An error will be issued if the address you enter does not fall on a page boundary, (the least significant byte must be zero).

Zero page address

The zero page address is the start of the zero page memory area that is allocated for use by a compiled program. A number of bytes is pre-allocated for use by the runtime code the rest of the area is for use as POINTER variables. The compiler allows you to change this address to enable programs to be developed for use on other computer systems. Typically half of the zero page area will be reserved for use by the computers resident operating system, however PL65 programs can be operating system independent. If no operating system is required then the whole of this area can be claimed. To enter a new start address select the appropriate option from the menu. The compiler will prompt for the new address, enter this then press return, the menu will be updated.

Zero page length

The zero page length can be set in conjunction with the zero page start address to reserve just a portion for use by the compiled code. To change this enter the appropriate option number. Enter the new length (in hex) and then press return. The compiler will issue an error if the entered length is insufficient to handle the runtime code plus one pointer variable, or if the length plus the zero page start address results in a value greater than 255.

Append MEMLO

MEMLO is an operating system database variable which can be set by a loaded machine code program to indicate the first free memory location past the newly loaded code. This is very handy variable to have ready for the programmer since it can be used for a variety of programming needs (eg a word processor can use this to determine the start area to load in text). This data can be automatically appended to the end of your program or not. The menu shows the current state either a "YES" to include it or a "NO". To change the current state of this option simply select the option.

Make autorun

This option determines whether a compiled program is to become an auto-run file or not. This is achieved by appending the entry address of the MAIN routine to the end of the file. The options menu will show the current state of this option, either "YES" or "NO". A program which has this option will autorun when loaded using the binary load option "L" from the DUP main menu.

Append Runtimes

This option determines whether a compiled program has the runtime code included. This option is intended for future expansion and is currently unsupported, its use is in specialist applications which generate very large programs. It allows an executive file to load parts of a larger program that reside on disk, thus the runtime code needs only to be included with the executive.

3.7 CROSS REFERENCES

A cross reference of a compiled program can be obtained after a compilation provided no errors are produced in the error log. This is done by selecting option 5 from the main menu.

The compiler will ask for a filename. Enter here the file to which the xref will be sent, this may be a printer (P:) or a disk filename (D:filespec) or any other legal device.

If no copy is desired then press return - the xref will then be displayed on the screen only.

The first thing the xref will show will be some auxillary information about the final symbol table size and the amount of free memory left after the compilation.

Next the contents of the symbol table are listed in reverse order, that is the MAIN routine first. The symbol table will only contain the global identifiers of the program.

The format of the listing is:

```
Type Address Params Identifier
```

The 'type' part is a decimal number corresponding to the type of identifier.

The 'address' part is a hexadecimal number which is the address of the symbol. In the case of variables this is the address in memory of the first element of the variable. For procedures the address is the entry address of the procedure. For constants the address is the constant value. Some symbol types may have no address eg: INCLUDE filenames.

The 'params' part is a hexadecimal number corresponding to the number of elements assigned to the symbol. For variables this number will show 1 element for scalars or in the case of arrays the number of array elements. For procedures this number correspond to the number of parameters declared in the procedure parameter list. Lastly but not least the 'identifier' itself is displayed.

Symbol table types

The following is a list of the global symbol table types

```
 1 - BYTE
 2 - INT
 3 - BYTE ARRAY
 4 - INT ARRAY
 5 - CONST
 6 - PROC
 7 - FUNC
 8 - MAIN
 9 - INTERRUPT
14 - STRING
17 - INCLUDE FILE
18 - STRING FUNCTION
19 - BASED BYTE
20 - BASED INT
```

Part IV   LANGUAGE DESCRIPTION

4.1 PL65 Source files

Source files are created using either the editor program KED or any other word processor or editor. The compiler will accept any normal ASCII file as input, however certain rules have to be adhered to.

PL65 will only read one complete logical line of text at a time, these lines must be less than 128 characters long (the length of the input buffer), lines longer than this will cause a fatal error.

A logical line is a line of text terminated by an EOL character. KED will allow you to create lines of any length so make sure there is an EOL character at least every three screen lines.

If you need to include a text string which is longer than a logical line then break it up using the appropriate separator.

Eg: a string constant can be split using the & concatenator

```
"A STRING"&
" AND SOME MORE TEXT"
```

Data strings are split using a comma

```
"A STRING",
" AND SOME MORE TEXT"
```

Strings split in these ways are identical to unsplit strings.

Source code can only include reverse video characters as part of a string constant, DATA string or comment.

Spaces are used as delimiters but are in all other respects are ignored by the input parser, that is to say, you must separate identifiers by something and in the absence of any suitable character then a space must be included.

```
FOR A=l TO 100 DO
```

In the above example a space is necessary to separate the FOR statement from the variable identifier A, but there is no need to (although you could if you wanted) place a space between the A and the equal since non alpha-numeric characters will automatically delimit the elements.

You may sprinkle spaces liberally thoughout your source program, any number of spaces can be included where a single space may appear.

EOL (end of line) characters may also be used liberally in the same way as spaces, they also are used as delimiters and they can be placed anywhere where a space may appear. The only other action caused by an EOL is to force a new line to be read in from the current input file.

Spaces and EOL's can be used to good effect to create a more readable program with identation.

eg:

```
WHILE count=0 DO
  IF height<maxheight THEN
    height=height+l
  ELSE
    height=maxheight
  ENDIF
ENDWHILE
```

Notice how this has been broken into many lines, with indentation to make it more readable. This shows that spaces and EOL's can be freely inserted between the elements of statements without changing its meaning.

By aligning each statement initiator to the same indentation as the corresponding terminator, the start and finish of a statement can be seen immediately.

4.2 PROGRAM STRUCTURE

PL65 is a procedural structured language, programs consist of a list of commands and statements.

Commands control the compiler and make actions occur at compile. time. A command may do such things as creating variables, including or linking source files or initiating the start of a code segment (procedure).

Statements are things that make the compiler create the actual code and are used within procedures to make an action occur at run time.

The following is a list of the commands and high level statements available in PL65:

COMMANDS

```
! comment command
LINK command
INCLUDE command
CONST command
BYTE command
INT command
POINTER command
STRING command
PROC command
FUNC command
MAIN command
INTERRUPT command
BODY command
```

STATEMENTS

```
Assignment statement
IF statement
WHILE statement
REPEAT statement
FOR statement
CASE statement
GOTO statement
CALL statement
ON statement
RETURN statement
TRAP statement
NOTRAP statement
Call procedure statement
```

The procedure is the main programming block, it contains a list of statements, it is akin to BASICS subroutine.

Each procedure has a level - the level being determined by the position in the program that the procedure is defined. Procedures at

the beginning of the program have the lowest level, they contain the finest detail of the program. As each procedure is defined they may call any procedure at a lower level than themselves, thus program detail lessens with higher level procedures. Ultimately the MAIN routine is defined, at runtime program execution begins with the first statement in this routine. MAIN controls just the outer level of the program ie: for a game program MAIN might look something like this

```
MAIN()
BEGIN
  initialise()
  REPEAT
    titlescreen()
    playgame()
  FOREVER
END
```

The simplicity and readability of such a routine is obvious and self documenting. Compare this to BASIC's

```
10 GOSUB 100
20 GOSUB 200
30 GOSUB 300
40 GOTO 20
```

BASIC programmers will notice the lack of line numbers in PL65. Once the language syntax is understood it becomes obvious that they are not needed.

PL65 programs should be created by first defining on paper what you want to do. The example MAIN routine is a good starting point for a game. Next you may want to design a title screen, this could be created in a procedure called 'titlescreen'. You may then want to plan out the game. The game could be controlled with a procedure called 'playgame'

eg:

```
PROC playgame()
BEGIN
  drawscreen()
  init_scores_and_things()
  WHILE lives>0 DO
    startgame()
  ENDWHILE
END
```

As you work down through the procedures to the finest detail, you eventually know exactly what variables and library routines the program requires. The program can then be layed out as follows:

1 - The INCLUDE filenames of the library routines needed.
2 - The global variable declarations.
3 - The program procedures.
4 - The MAIN routine.

Some programs may be short enough to be contained wholly in the MAIN routine, eg a program to calculate the number of prime numbers over a given range could be written:

```
--! Sieve of Eratosthenes
--! (mixworx modified syntax of PL65 comments to match LUA/SciTE lexer)
--!
INCLUDE D:TERMINAL.LIB

CONST size=8190, false=0, true=l
BYTE flags[size+1]
INT prime, count, k, i

MAIN()
BEGIN
  WRTLN("1 iteration only")
  count=0
  FOR i=O TO size DO
    flags[i]=true
  NEXT
  FOR i=O TO size DO
    IF flags[i] THEN
      prime=i+i+3
      k=i+prime
      WHILE k<=size DO
       flags[k]=false
       k=k+prime
      ENDWHILE
      count=count+1
    ENDIF
  NEXT
  WRITE(count) WRTLN(" primes")
END
```

The standard procedures for output to the screen are contained within the library file TERMINAL.LIB.

The declarations at the start of the program show how constants and variables may be declared - the array 'flags' has an example of a compile time expression used to declare the length of the array in terms of a pre-declared constant, the resultant length of the array will actually be 8191. This extra one is added because the program references elements 1 to 8191 of the array, but array indices actually start at zero and go up to the declared length-1.

The constant 'size' is referenced not only in the array declaration but also within the MAIN routine in three different program loops. Using a constant this way allows us to change the range of values that the program calculates simply by changing the declared value of 'size'.

4.3 IDENTIFIERS

Any user defined item declared within a program be it a variable, constant, procedure or any such related item must be given a unique name. This is so it can later be identified when referenced within your program. These names (identifiers) must be declared within the following rules:

1 - The first character of an identifier must be an alpha letter.
2 - The second and subsequent characters must be alpha-numeric or an underline character
3 - The total length of an identifier must not exceed 32 characters.

PL65 distinguishes between upper and lower case hence the following example shows three completely different identifiers.

Somename SOMENAME somename

Reserved words may not be used as identifiers (PL65 statements and commands).

Although the programmer has an otherwise unlimited choice of possible identifier names at his disposal, it is accepted practice in all languages to choose identifiers whose meaning in the program is suggested by their names. The length of an identifier does not alter the size of the compiled program since the identifiers are not included in the finished code.

Identifier scope

Identifier scope is defined as the area of the program within which a declared identifier is recognized. PL65 requires that the association of each identifier with the quantity it denotes must be unique throughout its scope.

All identifiers must be declared before the first part of the program that they are referenced.

An identifier declared outside of a procedure is defined as being GLOBAL, that is, it will be recognized anywhere after its declaration.

An identifier declared within a procedure is defined as LOCAL. The scope of a local identifier is limited to the current procedure, that is, it will only be recognized inside the current procedure. It is legally possible for LOCAL identifiers to have the same name as GLOBAL identifiers in which case the LOCAL identifier will take precedence. After the procedure is terminated the GLOBAL identifier will again become active.

4.4 COMPILE TIME EXPRESSIONS

PL65 allows two types of expressions, these are runtime expressions and compile time expressions. Runtime expressions are described in the statement section of this manual.

Compile time expressions allow static program constants to be calculated at compile time in terms of the symbols that already exist. They are evaluated using 16-bit positive integer math.

These expressions are used where the syntax graphs show a CEXPR symbol.

Compile time expressions may not form part of a statement except in the case of an assembler address operand.

CEXPR OPERATORS

The legal operators of compile time expressions are:

```
*    /    AND
+    -    OR    XOR
```

the precedence of these operators are as shown.

Parenthesis may be used to force the expression to be evaluated in any desired sequence.

CEXPR operands

Operands may be:

Decimal numbers eg 1, 2, 36, 9999
Hexidecimal numbers (preceded by $) eg: $FF, $A, $2FAE
Binary numbers (preceded by %) eg: %10, %10101,
%1111100111001111
ASCII numbers (in single quotes) eg: 'a', 'b', '*'

NOTE: ASCII numbers are created by the compiler by substituting the character code of the character in quotes.

Any previously declared identifier can be used as a CEXPR operand. Identifiers will actually be referenced by their address.

NOTE: Runtime expressions use the dot operator to access an identifiers address but this is automatic in compile time expressions.

The ? symbol, which leaves the declared parameters of an identifier. (See the section on runtime expressions for more detail)

The @ symbol, which references the compiler location counter.

Examples of compile time expressions are:

```
CONST SIZE=8190
BYTE FLAGS[SIZE+1]
BYTE INFLAG=FLAGS+50
CONST DUMMY=@
```

In these examples the constant SIZE is declared to have the value 8190, future references to SIZE will substitute this value. The array FLAGS will be dimensioned to 8190+1 or 8191. INFLAG has its absolute address set to the 50th element of FLAGS. DUMMY holds the current value of the location counter.

Assembly language operands use compile time expressions,

eg:

```
LDA SIZE AND $FF
LDY SIZE / 256
```

will load the accumulator with the low byte of SIZE and the Y register will be loaded with high byte of SIZE.

4.5 COMMENTS   (--! mixworx modified syntax … supports scintilla-lua-lexer)

Comments are written notes that are included within source program text to improve readability and provide program documentation. A PL65 comment is a sequence of characters delimited on the left by the character ! and on the right by either an EOL or another !.

```
!THIS IS A COMMENT !
--!THIS IS A COMMENT !--
```

These delimiters instruct the compiler to ignore any text between them, and not to consider such text as part of the program proper.

A comment may contain any printable ASCII character including spaces and reverse video characters.

A comment may not be embedded inside an identifier or within a character string constant. Apart from this, it may appear anywhere that a blank character may appear. Thus comments may be freely distributed throughout the program.

4.6 THE LINK COMMAND

The LINK command is used when a program becomes too large to manage in a single text file. It enables the program to be split into smaller more manageable files.

LINK is used in the form:

```
    LINK filespec
```

where filespec is a valid ATARI operating system device specifier

```
eg:
```

```
LINK D:PART2.PRG
```

LINK should be the last command in a program sub file, it ends the compilation of the current file and transfers compilation to the new file specified by the filename following the LINK command. Any source code contained in the current file following the LINK command is ignored.

4.7 THE INCLUDE COMMAND

The INCLUDE command enables commonly used routines to be put in their own text file. These routines can then be accessed by any program.

INCLUDE is used in the form:

```
INCLUDE filespec
```

INCLUDE causes the compilation of the current file to be suspended and transfers control to the file specified in the INCLUDE command.

eg:

```
INCLUDE D:TERMINAL.LIB
```

When the INCLUDE file is terminated, control returns to the file from which the included file was called.

INCLUDE files can be nested up to four deep, that is to say a program file can INCLUDE a file which can INCLUDE another file which can INCLUDE another file.

INCLUDE files will only be loaded once. If a file includes a file that has previously been included then the message "already resident" will be shown in the error log and the INCLUDE will be ignored. Thus for example if two library source files need D:TERMINAL.LIB to communicate with the operating system then the file will only be compiled once.

The INCLUDE command must be the last thing on the current line. When control returns from the included file, compilation will continue with the next line.

The last command in an INCLUDE file must be an ENDFILE.

NOTE: Both LINK and INCLUDE can only be used outside of a procedure.

4.8 THE CONST COMMAND

Constants are used to assign a fixed value to an identifier. Once declared a constant can be used throughout its scope - at each occurrence the compiler will substitute the assigned value. This enables an often used value to be declared at the start of a code segment, later if this value is to be changed by the programmer, instead of reading through the whole program and changing each occurrence, only the constant declaration need be changed.

Constants can be declared using the CONST command as follows:

```
CONST identifier=CEXPR
```

where CEXPR is a valid compile time expression. eg:

```
CONST five=5, six=b, MAXVOL=20
```

The scope of constants is the same as variable scope. Constants are local when declared in the local variable part of a procedure header.

4.9 VARIABLE TYPES AND ATTRIBUTES

Four types of variable are supported, these are:

```
BYTE INT POINTER and STRING
```

All variables must be declared before the first point in the program at which they are referenced.

Depending on where variables are declared, they may be GLOBAL, LOCAL or PARAMETER (see the rules on scope). A variable is GLOBAL when declared outside of a procedure. A variable is LOCAL when declared as part of the local variable list in a procedure header.

A variable is defined as being a PARAMETER when defined as part of the parameter list in the procedure header. PARAMETER variables have the same scope as LOCAL variables - that is, they can only be accessed by the current procedure.

All variables be they GLOBAL, LOCAL or PARAMETER are held statically in memory and (unless they have been initialized with a DATA command) no initialization is performed at runtime. Thus variables will hold random data and each should be initialized with an assignment statement.

4.9.1 BYTE variables

BYTEs hold numbers in the range 0 to 255, they are declared as follows:

```
BYTE identifier
```

BYTE arrays can be declared by subscripting identifiers, as follows:

```
BYTE identifier[CEXPR]
```

where CEXPR is a valid compile time expression.

Arrays can have any length, the compiler will allocate 1 byte of memory for each element of the array.

Program references to arrays should be subscripted to one less than the declared length, eg if an array is declared with a length of 4

```
BYTE name[4]
```

then to reference this array you will actually need to reference name[0] to name[3], which cover the four elements.

More than one variable can be declared with the same BYTE command by including a comma after each identifier ie:

```
BYTE start, middle[20], end
```

Absolute variables

A facility exists to enable the absolute address of the declaration to be set, this is achieved as follows:

```
BYTE identifier=CEXPR
```

```
BYTE NMIEN=$D40E
```

In this case the compiler allocates no memory within the program data space for the variable, the address of the variable in memory is that set by the expression. Both BYTE and BYTE arrays may be declared in this way. Program references to variables declared in this way are identical to ordinary variables.

DATA initialization

BYTE and BYTE arrays can be initialized with data provided the declaration is not BASED and provided the address of the declaration has not been set to an absolute memory location. e g:

```
BYTE ident[6] DATA 0,1,2,3,4;
```

In the example the array variable will be initialised with the data following the DATA command such that at runtime identC01 will hold 0 and identC11 will hold 1 etc. The data list must be terminated with a semicolon.

DATA for BYTE variables can also be entered as a character text string, eg:

```
BYTE ident[5] DATA "HELLO";
```

Will store the ASCII character code values of the characters in quotes in each successive element of the declared array.

Character data can be mixed with numeric data by separating the elements with commas ie:

```
"HELLO",$9B;
```

NOTE: An error will be reported if there is more DATA in the data list than the declared size of the variable. However it is possible to supply less data than the declared size, in which case the remaining uninitialized portion will contain random data at runtime.

4.9.2 INT variables

INT variables are positive only 16 bit integers. They hold numbers in the range 0-65535.

Integers are held in two successive bytes of memory in the normal LO HI format. INT's are declared as follows:

```
INT identifier
```

To declare an INT array the identifier may be subscripted ie:

```
INT identifier[CEXPR]
```

Where CEXPR is a valid compile time expression.

Arrays of both INT and BYTE may be any length.

The compiler will allocate 2 bytes of storage for each element of INT arrays.

Multiple declarations may be made using the same INT command as in the BYTE command.

Absolute INT's

The compiler will allocate in-line storage space for the variable but the programmer has the option to assign the variable to an absolute address eg:

```
INT name=$200
```

This instructs the compiler that the address of the variable is the location 200 (hex) in this case no storage is allocated by the compiler in-line. This way it is possible for many variables to share the same address or for the program to share a variable with another program (such as the operating system).

NOTE: If an integer variable is assigned to a page zero location then it will be treated as a POINTER variable.

DATA initialization

INT's can be assigned DATA in the same way as BYTE's with the exception that character data may not be used.

```
INT table[5] DATA 5, 10, $2E4F, 1655, 78;
```

4.9.3 POINTER variables

POINTER variables are functionally equivalent to INT's. In actuality they are assigned to page zero locations and thus not only are they accessed quicker, they also enable BASED variables to be declared.

Pointers are declared as follows:

```
POINTER identifier
```

More than one POINTER can be declared with the same command by separating each identifier with a comma, eg:

```
POINTER point1, point2, point3
```

POINTERS may not be subscripted nor may they be assigned DATA.

The number of POINTER variables that can be declared in a program is limited by the availability of zero page RAM - this is set prior to a compilation from the compiler options menu. The default setting allows around 58 POINTER's. Remember, the library files will use some of these.

4.9.4 BASED variables

Sometimes a direct reference to a data element is inconvenient. This happens when the location of the element remains unknown until it is computed at runtime. In this case it is necessary to write code to manipulate addresses of data elements rather than the data elements themselves.

To permit this type of manipulation, PL65 uses "BASED" variables. A BASED variable is a variable which is pointed to by another variable. A BASED variable is not allocated storage by the compiler. At different times during runtime the location of the variable may be in different places in memory, since its base may be changed by the program. A BASED variable is declared by first declaring its base, which must be a POINTER variable, and then declaring the BASED variable itself.

eg:

```
POINTER ITEM
BYTE itemof BASED ITEM
```

Given these declarations, a reference to 'itemof' is, in effect, a reference to whatever byte value is pointed to by the current value of ITEM.

This means that the sequence

```
ITEM=$200
itemof=$C0
```

will load the byte value $CO (hex) into the memory location $200 (hex).

A variable is made BASED by inserting the word BASED after the declaration identifier and then following with the identifier of the POINTER (which has already been declared) to be the base.

The following restrictions apply to BASED variables:

The BASE must be a POINTER.

The BASED variable may not be subscripted-that is, it may not be an array.

The BASED variable must be a scalar INT or BYTE (not another POINTER).

4.9.5 STRING variables

String variables hold character data, they are declared as follows:

STRING identifier$[CEXPR] where CEXPR is a valid compile time expression.

The dollar sign incorporated after the identifier does not form part of the identifier name, but it is expected by the compiler after every string declaration and reference. This is simply to improve program readability by distinguishing strings from numeric variables.

It should be noted that because the dollar sign does not form part of the identifier, it should not be included when referencing the identifier as part of an compile time expression or when referencing using the dot operator in runtime expressions.

String arrays can be declared to any length, the actual memory allocated to the string will be one byte per character plus four bytes of size information. Two of these bytes hold the length of the string, this may vary at runtime. The other two bytes hold the maximum length of the string, this is used by the runtime code for error checking.

The dot operator discussed elsewhere in this manual will return the address of the first character of the string variable. The address of the variable which holds the runtime length of the string can be calculated as:

.ident-2 ((.ident – 4 ???))

STRING variables may be initialized with character data, eg:

```
STRING A$[50] DATA "A STRING", $9B, " MORE STRING";
```

If strings are not initialized with a DATA command then, at runtime, the string will be initialized to have a length of zero.

4.10 PROCEDURES

Procedures are the main elements of a program since they contain the actual code that is to be executed. There are 4 types of procedure each having a different role:

`PROC`

This is the standard procedure containing a subroutine.

`FUNC`

In essence this is identical to PROC except it returns a value and thus can only be invoked as part of a runtime expression.

`INTERRUPT`

This is a special procedure that allows interrupt processing to be achieved using high level code. INTERRUPT procedures can't be assigned parameters since they can't be directly called from an executing procedure, however an executing interrupt procedure is free to call any standard procedure or function.

`MAIN`

This is the main program loop from which all the sub-procedures are ultimately called. MAIN can have no parameters and can not be assigned any other identifier.

All four types described above have the same attributes in terms of coding.

All of the following notes referring to procedures apply equally to FUNC's, INTERRUPT'S and MAIN with the exception of those points described under the relevant section.

Procedures are declared as follows:

```
PROC identifier(parameter list)
local variable declarations
BEGIN
    statement 1
    statement 2
    . . . . .
    statement n
END
```

The procedure header contains an identifier and a parameter list followed by a list of the local variable declarations. The actual code for the procedure is initiated with a BEGIN followed by the statement list and terminated by an END.

The parameter list is a list of variable declarations that accompanies the procedure declaration - the object of this list is to define the data to be communicated from calling procedures. The compiler keeps track of the amount of data that is to be communicated and at each point in the program that a call is made, the supplied parameters is compared to the declared parameters. If there is a mismatch then an error will be reported.

The parameter list may consist of any variable type ie BYTE, INT (both scalar and arrays), STRINGs and POINTERs. In addition a special type of parameter can be defined - this is the PROC parameter.

Parameter variables are treated within the procedure exactly the same as local variables, that is, they can be used in expressions and can be assigned to, eg:

An equivalent procedure to BASIC's SETCOLOR command could be written as follows:

```
PROC SETCOLOR(BYTE REG,EOL,LUM)
BYTE RCOL[5]=$2C4
BEGIN
    RCOL[REG]=COL*16+LUM
END
```

In this example the parameter list has declared 3 BYTE variables REG, COL and LUM. A call to this procedure can now supply three items of data, eg:

```
SETCOLOR(0,12,A*2+B)
```

When the procedure is entered REG will be set to the value 0, COL will be set to the value 12 and LUM will be set to the result value of the expression in the call. SETCOLOR can now use these values to compute a value in the color assignment expression.

From this example it can be seen that the parameter list is really just a list of deferred assignments. The calling procedure decides the values and the defined procedure decides where to store those values.

The total number of parameters that can be passed is limited by the size of the software stack. All parameters are passed as 16 bit integers and with a software stack of 256 bytes it follows that 128 parameters can be passed. This may be limited further if functions are

nested in procedure call expressions, since each function will require its own parameters to be passed.

NOTE: There are no run time checks for stack overflow and so it is quite possible for parameters to be corrupted if the programmer is not careful.

If a parameter is declared as an array then each element of the array is counted as one parameter. Thus parameter arrays may not be declared with lengths that exceed 128 elements, eg:

A procedure declared with a parameter array as follows:

```
PROC aroutine(BYTE data[8])
```

would mean that a calling procedure needs to supply 8 items of data, eg:

```
aroutine(7,50,23,64,95,77,127,127)
```

When the procedure is entered data[0] would hold the value 7, data[1] would hold the value 50, data[2] would hold the value 23 etc.

String arrays when declared in the parameter list will only count as 2 parameters regardless of the dimensioned size since only the address and length of the string will be required in order for the string to be copied.

So using the information above

```
PROC NAME (BYTE pram1, pram2[20], INT pram3[8], STRING A$[20])
```

will actually require 31 parameters to be passed.

The compiler does not distinguish parameter types in a procedure call thus a procedure defined as follows:

```
PROC NAME(STRING A$[21])
```

will look identical to:

```
PROC NAME(INT addr, length)
```

to any calling procedure.

It is important to note that the 2 examples are not the same since in the first case the whole string will be copied into the parameter string array and in case 2, just the address and the length of the string will be accepted into the 2 integer variables.

A call to the above procedure can be written as:

```
NAME(addr1,len1)  or  NAME("HELLO")  or  NAME(A$)
```

In the first of the above examples the string whose address and length are the result of 2 expressions will be passed to the procedure whereas case 2 generates a string constant and passes the address and length automatically. In the last example the contents of a string variable will be passed as an address and length. Any part of a string variable can be passed using the rules described in the string assignment section of this manual.

A special case exists that allows another procedure to be declared as a parameter. The reason for procedure parameters can best be explained with the following example:

```
PROC POS(INT X BYTE Y)
BEGIN
    POKE($54,Y)
    DOKE($55,X)
END

PROC PLOT(INT X BYTE Y)
BEGIN
    POS(X,Y)
    PUT(6,COLOR)
END
```

In the above example the runtime action of PLOT is to pull both X and Y parameters from the stack and store them into the parameter variables, the next action is for the variables X and Y to be loaded back onto the stack and the procedure POS is called. PLOT has no other use for these parameter variables. Obviously this is a waste of time and memory thus PLOT can be re-written:

```
PROC PLOT(PROC POS)
BEGIN
    PUT (6,COLOR)
END
```

Any call to PLOT will still require the same parameters to be passed but instead of the parameters being pulled from the stack by PLOT, they are passed directly to the procedure POS.

This special type of indirect parameter passing can only be used when the PROC parameter would normally be the first statement in the statement list.

PROC parameters can be intermixed with normal parameters, eg:

```
PROC dosomething(INT A,B PROC POS BYTE C,D)
BEGIN
    .   .   .
END
```

In the above example 6 parameters would be supplied in the call. The runtime action would be:

First - The parameter variables D and C are pulled from the stack
Second - The procedure POS is called using up 2 parameters.

Third - The parameter variables B and A are pulled from the stack.
Fourth - The statement list is executed.

4.11.2 ORDER OF PARAMETER PULLING

Parameters are pulled from the stack in the reverse order to which they were pushed. It is important to understand this "Last In First Out" rule in order to avoid confusion when trying to assign to variables directly from the parameter list. For instance, it would seem easy to create an equivalent procedure to BASIC's POKE statement with the following parameter list trick:

```
PROC POKE (POINTER address BYTE byte BASED address)
BEGIN
END
```

In this example what we might expect to happen is for the POINTER to be set to the address value and then the BYTE to be set to the supplied byte value, which being BASED to the POINTER should store directly into the memory location.

In fact what will happen is that the BYTE will be pulled first using the "Last In First Out" rule and will store into the memory location that the POINTER was set to the last time the routine was called, which may be undefined (if it is the first time called) and crash the program. The POINTER will be pulled after the BYTE.

4.12 THE LOCAL VARIABLE LIST

The code for a procedure starts with the BEGIN command but it is first necessary to declare any local variables, eg:

```
PROC NAME(INT addr1,len1)
BYTE A,B
INT TEMP1,TEMP2
STRING C$[20]
BEGIN
    . . .
```

Local variables are for use solely by the procedure within which they are declared. They can't be accessed outside of the procedure.

The memory allocated to them is static, that is to say, it can't be reclaimed after the procedure is terminated. It would seem that there is no advantage in having local variables over global variables - in terms of memory usage and actual code, this is true. However the reason local variables are useful is that it removes the necessity to check whether a variable of the same name has already been defined

- the compiler will always use the correct variable even if another procedure has one of the same name, or even if a GLOBAL identifier has the same name.

The local variable list may also include constants which are local to the procedure.

4.13 FORWARD PROCEDURES

Situations sometimes arise when two procedures need to call each other. In normal single pass compilers this would be impossible. PL65 solves this problem with the FORWARD procedure attribute.

By including the word FORWARD at the end of a procedure header you can declare that a procedure exists without having to include the body of the procedure at that point in the program.

```
PROC NAME(INT addr1,addr2) FORWARD
```

The parameter list is not repeated when the main body of the procedure is compiled but rather is written thus:

```
BODY NAME
BEGIN
```

Local variables for FORWARD procedures are declared between the BODY command and the BEGIN command as per normal.

```
BODY NAME
INT temp1, temp2
BEGIN
```

An example of the use of FORWARD procedures can be found in the library routine TERMINAL.LIB. Here CIO needs to write an error to the screen if no trap is set, and yet WRITE needs CIO in order to achieve this.

It should be noted that PL65 procedures are not naturally recursive, that is, the parameter and local variables are static and not dynamically allocated on procedure entry. This means that if a procedure were to call itself through use of the FORWARD attribute then the variables will be changed. This should be taken into account when designing procedures.

((BUG !!! parameters are on stack while locals are fixed ???))

4.14 FUNCTIONS

Functions are procedures that return a value, they can only be invoked by their inclusion as operands in expressions. They are declared as follows:

```
FUNC identifier(parameter list)
```

after which the body of the function is identical to a procedure definition.

All exits from the function, ie END and RETURN must be followed by a valid expression.

They may simply calculate a value for the expression (which needs no statement list) or they may carry out complicated actions before returning a value, eg a simple function to multiply two numbers:

```
FUNC mult(INT a,b)
BEGIN
END a*b
```

The following function reads the console switches. It has no parameters but it needs to store a number into a hardware register before returning with the switch value.

```
FUNC console()
POINTER temp
BYTE cswitch BASED temp
BEGIN
  temp=$D01F
  cswitch=8
END cswitch
```

4.15 STRING FUNCTIONS

String functions are declared by including a dollar symbol after the identifier as follows:

```
FUNC identifier$(parameter-list)
```

The difference between a string function and a normal function is that each exit from the function must be followed by a valid string factor ie:

```
RETURN A$
```

String functions can only be called where a string-factor is required and ordinary functions can only be called as part of a numeric expression.

Both types of functions can also be declared FORWARD in the same way as procedures.

The library procedure STR$ is an example of a string function - it returns the ASCII string of a number and can be used in a string assignment, eg:

```
A$=STR$(B*4+5)
```

String functions allow you to create your own string handling routines, these can be a very powerful tool in text processing programs.

4.16 INTERRUPT PROCEDURES

INTERRUPT procedures enable the handling of interrupts using high level code. These procedures can have no parameter list since they cannot be called directly from an executing procedure. Interrupts are declared as follows:

```
INTERRUPT identifier()
```

The body of the interrupt procedure is identical to a normal procedure.

Be careful when calling ordinary procedures from interrupt procedures as this will currupt both local and parameter variables of the called procedure.

Interrupts can be declared FORW4RD.

The actual mechanism for enabling an interrupt is system dependent thus no high level facility is built in to handle this. But in a typical system it will be necessary to store the address of the interrupt routine into a specified memory location and then enable a flag.

For example, a display list interrupt called DLINT could be used to change the background color halfway down the screen,

```
INTERRUPT DLINT()
BYTE colreg=$D018
BEGIN
  colreg=0
END
```

We can then write a procedure to enable the interrupt and to control when the interrupt happens:

```
PROC int_enable( )
POINTER point
INT B BASED point
BYTE C BASED point
```

```
BYTE NMIEN=$D40E    --! interrupt register
INT SDL=$230        --!Display list address
BEGIN
  point=$200 B=.DLINT --!Store address of interrupt into vector
  point=SDL+15 C=$82  --!Interrupt at line fifteen
  NMIEN=$C0           --!Enable interrupt
END
```

Interrupt procedures may not use the TRAP command and it is inadvisable to try to call the ERROR procedure from an executing interrupt since this may stop the interrupt from running to completion.

4.17 STATEMENT LIST

The statement list is initiated by a BEGIN command which is followed by the actual statements. The list may comprise as many statements as the user desires and is limited only by the available memory at compile time. The list may have no statements at all, this can happen in functions where only a value needs to be returned to the calling expression.

The statement list is terminated with END.

Each statement may optionally be separated by a semi-colon, this can improve program readability but more importantly it will help the compiler to recover more easily from errors in source text.

Compound program flow statements such as IF, REPEAT etc may have other statements included in their body, this is known as nesting. PL65 syntax dictates that each nesting of statement must be terminated from the innermost level of nesting outwards, eg:

```
IF height>maxheight THEN
  WHILE count<10 DO
    count=count+1
  ENDIF
ENDWHILE
```

is incorrect - the compiler will produce an error. The WHILE loop must be terminated first.

Statement nesting can be as many levels deep as required and is limited only by the compiler stack, this will typically allow around 64 levels, which should be adequate for even the most complicated of tasks.

There is no stacking involved at runtime for any statements, this allows complete freedom to use GOTO in and out of statement nests (that includes FOR/NEXT loops).

The following section describes each of the statements, but it is first necessary to describe runtime expressions.

4.18 RUNTIME EXPRESSIONS

Most of the high level statements work in conjunction with expressions. A PL65 expression consists of operands combined by means of the various arithmetic logical and relational operators, eg:

```
A+B
A+B-C
A*B+C/D
A*(B+C)-(D-E)/F
```

where +, -, * and / are operators for addition, subtraction, multiplication and division, and A, B, C, D, E and F represent operands. Parenthesis serve to group operands and operators, as in ordinary algebra.

All mathematics in PL65 is done in 16-bit unsigned integer format with underflow and overflow indication unspecified. This reflects the nature to which PL65 is intended to be used (as a replacement for assembler).

Although the compiler uses unsigned mathematics, numbers can be supplied using a unary -, eg: -1. These numbers are computed as 0-1 and thus still produce an unsigned internal format.

It should be noted that for most operations unsigned and signed mathematics produce the same result and 0-1 is the same in both formats, only the way it is output in ASCII is different. However relational operations may produce different results, eg -1<1 will produce a false result since -1 in unsigned format is actually 65535 and thus is greater than 1.

4.18.1 OPERANDS

Operands are the building blocks of expressions. An operand must be something which has a specific value at runtime. Thus in the above example A, B, C etc might be the identifiers of variables which have values at runtime.

Literals

Numeric literals may be used as operands in expressions.

Literals can be used in any of 4 different formats.

```
1 - DECIMAL        eg: 1, 2, 500, 6000

2 - HEX              preceeded by $  … eg:  $FF,  $3E2E

3 - BINARY        preceeded by % … eg:  %1,  %10 %1011001110111001

4 – ASCII            in single quotes … eg:  'A', 'B', '*', '='
```

ASCII literals are created by the compiler by substituting the character code of the character in quotes.

((WHERE?? These numbers are computed as 0-1 and))

Constants

Constants can be declared and assigned to an identifier, ie:

```
CONST fifty=50, FALSE=0, TRUE=1
```

references to these identifiers will result in the value assigned to them being substituted.

Variable references

Any fully qualified variable reference may be used as an operand in an expression. When the expression is evaluated, the reference is replaced by the value in the variable.

Because all expression evaluations are performed using 16-bit math, BYTE variable values are converted by adding a zeroed high byte.

Function references

A function reference includes the identifier of the predeclared function along with any parameters required by the function declaration. The value of a function refence is the value returned by the function.

A function to peek a memory location could be written:

```
FUNC PEEK(POINTER address)
BYTE byte BASED address
BEGIN
END byte
```

This could then be invoked as part of an expression as follows:

```
A=PEEK($D01F)*5+3
```

Sub-expressions

A sub-expression is simply an expression enclosed in parenthesis. A sub-expression may be used as an operand and is used to force an expression to be evaluated in a sequence that would not normally occur due to the rules of precedence.

4.18.2 OPERATORS

Location references

A location reference is formed by using the dot operator. The form of the dot operator is,

```
.identifier
```

where the identifier is that of any symbol previously declared.

The object of location references is to enable the address of identifiers to be referenced rather than the value they contain.

If the identifier is of an array then the value of the reference is the address of the first element of the array. If the identifier is that of an procedure then the value of the reference is the entry address of the procedure.

A location reference to a CONST will result in the same value as a normal CONST reference.

Identifier element referencing

A facility exists for referencing the declared element part of a symbol which complements the location part reference described above. This is accessed with the ? operator, eg:

```
?identifier
```

What this does is to return the declared parameters or elements of a previously declared identifier, eg if INT name[20] has been declared then ?name would return 20 since this is the declared elements.

In the case of the identifier being a procedure the ? operator would return the number of declared parameters.

This facility enables the storage of variables to be calculated. BYTE arrays have a 1 to 1 ratio with this number and INT's have a 2 to 1 ratio, eg if an array has been declared

```
INT A[20]
```

then we can calculate the memory this has used with the expression:

```
?A*2
```


((husty!!, to jsou metadata!! reflection!!))

4.18.3 ARITHMETIC OPERATORS

There are five principle arithmetic operators which are:

```
+  -  *  /  MOD
```

These perform addition, subtraction multiplication and division.

MOD returns the remainder of a division as opposed to / which returns the quotient.

All of these operators perform unsigned binary integer arithmetic on two operands. In the case of an overflow or underflow due to integer math or a divide by zero, the result of any of the operations is unspecified.

4.18.4 UNARY OPERATORS

There are 4 types of unary operators. They take single operands, to which they are prefixed. The unary operators are:

```
-  NOT  SHL  SHR
```

The unary - has the effect of computing the two's complement of the operand thus:

```
-1 = 65535
```

NOT produces a result in which each bit of the operand is complemented.

SHL and SHR are unary shift operators they effectively multiply and divide by two, eg:

SHL 4 shifts the operand left resulting in 8. The most significant bit of a left shift is lost so SHL 32768 results in 0.

SHR 4 shifts the operand right resulting in 2. The least significant bit of a right shift is lost so SHR 3 results in 1.

All unary operators can be nested, ie SHL SHL 2 results in 8 and NOT NOT 2 results in 2.

4-18.5 LOGICAL OPERATORS

There are 4 logical operators in PL65, these are:

```
NOT  AND  OR  XOR
```

These operators perform logical operations on 16 bits in parallel.

Although NOT is a unary operator it is shown here because it is also logical.

The remaining operators each take two operands, and perform bitwise operations

AND performs a 16 bit AND,
eg 7 AND 3 results in 3

OR performs a 16 bit OR,
eg 4 OR 3 results in 7

XOR computes the exclusive OR of two operands,
eg 1 XOR 0 results in 1
    1 XOR 1 results in 0

4.18.6 RELATIONAL OPERATORS

Relational operators are used to compare operands. They are:

```
<     less than
>     greater than
<=    less than or equal
>=    greater than or equal
<>    not equal
=     equal
```

Relational operators are always binary operations, taking two operands. The comparisons are always performed assuming the operands are unsigned integers. If the specified relation between two operands is true then a value of -1 ($FFFF in hex) is returned. A false results in a value of 0 being returned, thus

```
2>1       results in $FFFF
NOT 2>1   results in 0
2<1       results in 0
4>=4      results in $FFFF
```

Expressions producing true and false results can be combined meaningfully using logical operators, eg:

```
IF A>1 OR A<5 THEN
    dosomething()
ENDIF
```

Numeric data items can only be compared (or assigned) to other numeric items. The same is true when comparing strings.

Strings are compared using the same relational operators (= <> <= >= < >) that are used for comparing numbers. String comparisons are made by taking one character at a time (left to right) from each string and evaluating each ASCII character code. If the character codes are the same, the characters are equal. If the character codes are different then the character with the lowest code is the lower. The comparison stops when the end of either string is reached. All things being equal then the string with the shortest length is considered less than the longer string. Leading and trailing blanks are significant.

String comparisons produce a numeric true/false result and as such can be used within a numeric expression, eg:

```
IF A$="YES" THEN
    dosomething()
ENDIF
```

This shows how true and false values resulting from relational operations are useful in conjunction with program flow statements, ie IF, WHILE etc.

All of the program flow statements will treat a zero result as false and any non zero value as true.

4.18.7 OPERATOR PRECEDENCE

Operators in PL65 have implied precedence, which is used to determine the manner in which operators and operands are grouped together.

In general operands are bound to the adjacent operator of highest priority, or to the left of one in the case of a tie.

The PL65 operators are listed below in order of highest precedence:

```
unary  -  NOT  SHL  SHR

*  /  MOD
+  -
<  <=  <>  =  >=  >
AND  OR  XOR
```

Parenthesis can be used to override the assumed precedence in the same way as used in ordinary algebra. Thus the expression (A+B)*C will cause the sum of A and B to be multiplied by C, instead of adding A to the product of B and C.

4.19 ASSIGNMENTS

Results of computations can be stored into variables. At any given moment a variable has only one value, but this may change with program execution.

The form of the assignment statement is:

variable = expression

The expression may be any PL65 expression as described above. The expression is evaluated and the result is stored in the variable whose identifier is to the left of the '='. The variable may be a scalar BYTE, INT or POINTER - if an array is specified then the identifier must be followed by [expression], eg:

```
A=1
B=A+5
LIST[50]=50*B
```

NOTE: Although expressions are always evaluated in 16-bit integer format, if the result is to be stored into a BYTE variable then this will automatically be adjusted by dropping the high byte.

4.20 STRING ASSIGNMENTS

There are three ways in which strings can be represented, these are:

1 - String constants, which are held between double quotes, eg:

```
"THIS IS A STRING CONSTANT"
```

Control characters can be embedded into the string constant by using the compiler pseudo function CHAR, eg:

```
"THIS IS A STRING CONSTANT "&CHAR($9B)
```

will include the return character into the constant. This is necessary because the editor and PL65 reserve some of the control characters for internal use. Particularly the EOL character is used to split logical lines and thus can't be directly embedded into text strings.

Constants and control characters can be intermixed using the & concatenator. The CHAR function is not a function call but simply instructs the compiler to include the ASCII character into the string.

((viz. Action! ' apostrophe char literal, tady je to CHAR($xx) !!!))
((pro Action! potřebujeme změnu tady ➔ ' ➔ ~ … nebo prostě nějaký jiný znak, možná # !!!))

2 - String variables, which hold variable data.

3 - String functions, functions which return a string.

A string variable can be assigned to hold data transferred from any of the above types. The form of the string assignment is:

```
identifier$ = <<string-factor>>
```

The string assignment causes data on the right of the '=' to be transferred to the string variable whose identifier is on the left side of the '=', which erases the data already in the variable and the variable length becomes the length of the string expression on the right side.

The $ sign following the variable identifier does not form part of the identifier name, but it is expected by the compiler after all string

identifiers. The reason for this is simply to improve program readability by distinguishing between normal numeric and string identifiers.

The string assignments may be expanded by including subscripts to the variable identifier, ie:

```
Identifier$[expr] = <<string-factor>>
```

In this case the string to the right of the '=' is transferred to the variable whose identifier is on the left side of the '=', starting at the position specified by the expression inside the [] brackets. The length of the variable then becomes the length of the string expression on the right plus the value of the expression inside the brackets. Remember array subscripts start at 0, eg:

if we have declared a variable:

```
STRING A$[20]
```

and then assign to the variable

```
A$="HELLO THERE"
```

then executing the procedure:

```
WRTSTR(A$)
```

```
will print on the screen HELLO THERE
```

and then execute the statements:

```
A$[6]="AGAIN THERE"   WRTSTR(A$)
```

will print on the screen HELLO AGAIN THERE. Thus we have only changed the part of the variable after the point defined by the expression.

The string assignment can be further expanded to include just part of a string, eg:

```
A$[6,10] = "string"
```

the effect of this is that the string to the right of the '=' is copied to the string variable whose identifier is to the left of the '='. The string will only occupy the part of the variable specified by the two expressions enclosed in the string brackets. If the space allocated is less than the supplied string length, then the string will be truncated to fit into the available space.

The resultant length of the string variable will remain unchanged unless the value in the second expression in the variable assignment is greater than the current string length.

NOTE: There are 2 runtime error checks associated with string assignments - see APPENDIX B for more information.

4.20.1 STRING CONCATENATION

A string can be appended to the end of a string variables as follows:

```
identifier$ + <<string-factor>>
```

The resultant length of the string variable becomes the original length + the length of the string-factor, eg:

The variable assignment:

```
A$="HELLO"
```

and then

```
A$+" AGAIN THERE"
```

would result in the string HELLO AGAIN THERE in the variable.

NOTE: The variable in this type of assignment can't be subscripted.

((aha, něco jako =+ operator pro stringy, no teda NIC MOC SYNTAX TOHLE !!!))

4.21 IF statement

The IF statement enables one of two sets of statements to be selectively executed on a condition. The form of the IF statement is as follows:

```
IF expression THEN
    statement 1
    .....
    statement n
ELSE
    statement 1
    .....
    statement n
ENDIF
```

The IF statement has four parts:

1 - An IF part with condition.

2 - A THEN part which consists of another statement or collection of statements.

3 - An optional ELSE part which consists of another list of statements.

4 - An ENDIF part to terminate the structure.

The meaning of the IF statement is that if the condition of the IF part is true then the statements in the THEN part are executed. Otherwise the statements of the ELSE part are executed, eg:

```
IF A>9 THEN
    B=B+C
ELSE
    A=A+1
ENDIF
```

The ELSE part of an IF statement is optional and may be omitted, in this case if the IF part is false program control passes to the first statement following the ENDIF.

```
IF A>9 THEN
    B=B+C
ENDIF
```

4.22 WHILE statement

WHILE is used to create a controlled program loop and has the form:

```
WHILE expression DO
    statement 1
    statement 2
    …..
    statement  n
ENDWHILE
```

The WHILE loop has the following actions:

First the expression following the WHILE is evaluated to form a true or false flag.

If the flag is true then the statements following DO are executed.

The ENDWHILE causes a jump back to the WHILE and the expression is re-evaluated.

The process repeats until the expression returns a false result at which point program execution is transferred to the first statement following the ENDWHILE.

The description assumes that the block contains no statements that cause control to pass out of the block, ie GOTO or RETURN, eg:

```
A=0
WHILE A<10 DO
    WRITE(A)
    WRTSTR(" ")
    A=A+1
ENDWHILE
```

will display on the screen:

```
0 1 2 3 4 5 6 7 8 9
```

4.23 REPEAT statement

The REPEAT statement has the form:

```
REPEAT
    statement 1
    statement 2
    .....
    statement n
UNTIL expression
```

The REPEAT statement causes the following actions to occur:

The statements encompassed by the REPEAT and UNTIL are executed.

The expression following UNTIL is evaluated producing a true or false result.

If the result is false then the program branches back to the first statement following the REPEAT.

The loop continues until the expression produces a true result at which point execution passes to the first statement following the UNTIL part.

REPEAT also has the option of creating an infinite loop using the FOREVER terminator.

```
REPEAT
    statement 1
    statement 2
    .....
    statement n
FOREVER
```

In this case execution always branches back to the REPEAT statement.

The only way to exit from such a loop is to include a GOTO or RETURN statement within the block.

4.24 The FOR statement

The FOR statement begins an iterative loop and enables a group of statements to be executed a number of times.

The simplest form of the FOR/NEXT loop is:

```
FOR index=initial-expression TO limit-expression DO
    statement 1
    statement 2
    .....
    statement n
NEXT
```

The index must be an INT scalar variable (not an array) and "initial-expression" and "limit-expression" are both valid expressions producing a numeric result.

The runtime actions of the FOR/NEXT loop are described below:

1 - The initial expression is evaluated and the result assigned to the index variable. This is done only once, before the first time through the loop.

2 - The limit-expression is evaluated and its value is assigned to a local variable which is not directly accessible to the program. This is done only once, before the first time through the loop.

3 - The statements encompassed in the loop are executed.

4 - The index variable is incremented by one. If this causes the new value to be less than the old value (due to integer math wrap around), the loop is exited - passing control to the first statement following NEXT. If the new value is not less than the old value then it is compared to the limit value. If the index value is greater than the limit value then the loop is exited, otherwise the loop repeats at step 3.

The above description assumes that the loop contains no statements which pass control out of the loop ie GOTO and RETURN.

NOTE: The index may be referenced by any statement within the loop.

The loop characteristics may be changed by an assignment statement which changes the index variable. The loop will always terminate if the index passes through zero.

There is no need to follow the NEXT with the index variable. The compiler automatically keeps track of which NEXT accompanies which FOR.

4.24.1 STEP option

FOR / NEXT loops may contain a STEP option. This has the effect that at each iteration of the loop the index will be incremented by the value of the STEP expression. The STEP expression is calculated only once at the start of the loop along with the initial and limit expressions.

The form of a FOR / NEXT loop with a STEP part is as follows:

```
FOR index=initial-expr TO limit-expr STEP step-expr DO
    statement 1
    . . .
    statement n
NEXT
```

Rules for the termination of a FOR / NEXT loop with a STEP part are the same as for the simple loop.

4.24.2 DOWNTO option

DOWNTO may be used instead of TO to make the loop count down. DOWNTO loops will terminate when the index becomes less than the limit expression or when the integer math causes an underflow (when index-step causes the index value to pass through zero), eg:

```
FOR A=9 DOWNTO 0 DO
    WRITE(A)
    WRTSTR(" ")
NEXT
```

will display on the screen:

```
9 8 7 6 5 4 3 2 1 0
```

Do not get confused into trying to implement a loop which passes through zero. Consider the following example:

```
FOR A=9 DOWNTO -1 DO
    WRITE(A)
    WRTSTR(" ")
NEXT
```

At first sight the loop appears perfectly normal, BUT remember -1 in integer math is computed as 0-1, this results in the positive value 65535. What in fact will happen at run time is that the loop will perform

one iteration, displaying '9' on the screen, the index will then be decremented by one resulting in the value 8. This is then compared to the limit value, and since 8 is less than 65535 then the loop will terminate.

4.25 CASE statement

The CASE statement enables one of a series of statements to be selectively executed. The form of the statement is:

```
CASE expr
    OF expr1 DO statement1 ENDOF
    OF expr2 DO statement2 ENDOF
    OF expr3 DO statement3 ENDOF
ELSE
    statement
ENDCASE
```

The runtime actions of the CASE statement are described below:

First the expression following the CASE is evaluated. The result of this is then compared to the result of the expressions in each of the OF parts in turn until a match is found.

The statements encompassed by the corresponding DO and ENDOF parts are then executed and then control passes to the first statement following the ENDCASE.

If no test expressions are found to match, then the statements following the ELSE part are executed and control then passes to the first statement following the ENDCASE.

NOTE: The ELSE part may be omitted in which case if no test expressions match the CASE expression then control passes directly to the first statement following the ENDCAS, eg:

a routine to input and test for keys from the keyboard may be written:

```
OPEN(1,4,0,"K:")
CASE GET(1)
    OF 'A' DO someproc() ENDOF
    OF 'B' DO anotherproc() ENDOF
ELSE
    WRTSTR("Invalid key")
ENDCASE
CLOSE(1)
```

4.26 LABELS and GOTO's

Situations sometimes occur within a procedure when a particular action becomes either clumsy or inefficient to implement using the standard structured program flow statements. PL65 provides the label and GOTO statement to overcome these problems.

Any statement may be labeled for identification and reference. A label is declared with a colon character followed by the label identifier as follows:

:identifier

The colon must be followed by a valid identifier.

The GOTO statement alters the sequential order that a program is executed by transferring control directly to the labeled statement.

```
IF A>50 THEN
    GOTO outside-range
ELSE
    A=A+1 ENDIF
. . . .
:outside-range
    WRTSTR("overflow")
```

NOTE: The appearance of the label in the GOTO is not a label definition - it is a label reference. Labels are always local to the current procedure. Any number of labels may be assigned to the same statement. Any number of GOTOs may reference the same label.

GOTO may reference labels which are not defined until after the reference (forward reference). In this case the legality of the GOTO cannot be checked until the label is defined or until the procedure is terminated.

4.27 CALL statement

The call statement is used to call a labeled subroutine which exists within the same procedure. The statement is primarily intended to allow forward calls to machine language subroutines (in the same way as GOTO may be used instead of JMP), but may also be used to call a high level subroutine.

NOTE: The general use of CALL is not recommended as it offers little protection to the user.

CALL may be used to call a high level or machine language subroutine as follows:

```
CALL identifier
```

The identifier must be a label.

Subroutines must be terminated with the assembler mnemonic RTS (not RETURN).

The programmer must also ensure that a GOTO is used before the start of the subroutine to skip past the subroutine code, if this is not done then a program crash will result, eg:

```
PROC DEMO()
BYTE A
BEGIN
    GOTO skip
    :mult_A_by_2
    A=A*2
    RTS
    :skip
    A=5; CALL mult_A_by_2
    WRITE(A)
END
```

4.28 ON statement

The ON statement enables program execution to be selectively branched to any one of a list of labeled statements depending on the result of an expression.

The ON statement has the form:

```
ON expr GOTO label0, label1,..,labeln;
```

The runtime action of ON is as follows:

First the ON expression is evaluated.

If the result is 0, then control branches to the statement labeled by label0.

If the result is 1, then control branches to the statement labeled by label1

and so on.

If the ON expression yields a result which is greater than the number of labels in the labels list then the program execution continues with the first statement following the labels list, eg:

```
FOR A=0 to 3 DO
    ON A GOTO L0, L1, L2;
    WRITE(A)  WRTSTR(" does not cause a branch")  GOTO skip
    :L0  WRITE(A) WRTSTR(" branches to L0")  GOTO skip
    :L1  WRITE(A) WRTSTR(" branches to L1")  GOTO skip
    :L2  WRITE(A) WRTSTR(" branches to L2")
    :skip  WRTLN()
NEXT
```

4.29 RETURN statement

The return statement is used to force an exit from a procedure. It can take one of three forms depending on the type of procedure within which it is used.

```
1 - RETURN                 …from a procedure
2 - RETURN expr            …from a function
3 - RETURN string-factor   …from a string function
```

The RETURN statement enables a procedure to be exited without terminating the procedure definition and without having to include structures to pass control to the procedure END statement, eg:

```
FUNC test-value(INT value)
CONST false=0,true=1
BEGIN
    IF value>30 THEN
        RETURN false
    ENDIF
END true
```

NOTE: See the advanced programming section for information on using RETURN in primative procedures.

4.30 TRAP and NOTRAP statements

The TRAP statement interfaces to a special runtime procedure called ERROR which handles non-linear program flow due to special circumstances. It enables a vector to be set up within the program such that whenever the procedure ERROR is called, program execution branches directly to the statement labeled by the identifier following the TRAP statement.

If no TRAP is set then ERROR acts as any other procedure and simply returns from the point at which it was called. In this case only the system variable ERRNUM is altered (with the value supplied to ERROR).

At runtime when the TRAP statement executes it saves the processor stack position and the TRAP vector (the address of the labeled statement) into special system variables, the next time that ERROR is called then the stack will be restored and the program will branch to the labeled statement which will reside within the same procedure in which the TRAP was set.

This arrangement allows ERROR to be called from lower level procedures than that at which the TRAP is set, thus if the TRAP is set in MAIN then global errors anywhere in the program can be handled.

Once a TRAP has been executed, it will automatically be disabled, so if they are to be handled correctly the first statement in the TRAP handler should be another TRAP statement.

If a TRAP is set in a lower level procedure as one already set, then the previous one will be cancelled.

An enabled TRAP can be disabled at any time or level with the NOTRAP statement. The TRAP will automatically be disabled when the procedure, within which it was set, is exited.

The following example shows how a TRAP in MAIN can be used to TRAP global errors. Any procedure that calls the system routine ERROR will cause program execution to branch to the TRAP handler 'handle-error'.

In this example the handler displays the error number and then gives the program user the option to continue at a preset warmstart program vector or to abort the program.

((twl husty !! vyjimky !! exceptions !! … resp jedna vyjimka … try catch … throw !!! ☺)))

```
MAIN()
BEGIN
    TRAP handle-error
    REPEAT
        --! main body of program !
        initialize()
:warm
        doprogram()
    FOREVER
:handle-error
    TRAP handle-error
    WRTSTR("Error ")
    WRITE(ERRNUM)
    WRTLN(" has occurred.")
    WRTSTR("Cont y/n")
    CLOSE(1)
    OPEN(1,4,0,"K:")
    IF GET(1)='Y' THEN
        GOTO warm
    ENDIF

END
```

Now, further down the program, a procedure can utilize the ERROR procedure to range check values on entry, eg:

```
PROC SETCOLOR(BYTE REG,COL,LUM)
BYTE RCOL[5]=$2C4
BEGIN
    IF REG>4 THEN
        ERROR(5)
    ENDIF
    RCOL[REGl]=COL*l6+LUM
END
```

The above example shows how an equivalent procedure to BASIC's SETCOLOR command can be declared. The procedure stores a value into an operating system variable. The problem is that if the procedure is supplied with a value for REG which is out of range, it will corrupt another of the operating system variables. To overcome this REG is range checked. If the value of REG is greater than 4 then the ERROR procedure is called which will cause program execution to branch to the TRAP handler in MAIN. In the example the handler will print on the screen:

Error 5 has occurred.
Cont y/n

and the handler will act on the user's response.

NOTE: See APPENDIX B for reserved error numbers.

4.31 PROCEDURE CALLS

Any procedure previously declared using PROC can be called directly as if it were a high level PL65 statement. Procedures are called simply by including the procedure name identifier followed by the required parameters enclosed in parenthesis.

If the called procedure requires no parameters then the parenthesis will still be expected, but with nothing in them.

Each parameter may be any valid numeric expression, or alternatively a string (which will count as two parameters), eg:

if we declare a procedure:

```
PROC plot(INT x,y)
BEGIN
    ....
END
```

then a call would be written:

```
plot(4,5)      or     plot(2*2, 2*3)
```

if 'plot' had no parameters then the call would be written:

```
plot()
```

NOTE: If more than one parameter is to be supplied then each parameter expression must be separated by a comma.

Part V   ADVANCED PROGRAMMING

The following section contains information for the advanced programmer.

5.1 STRUCTURE OF THE COMPILED PROGRAM

The compiler generates a compound file of object code which is written out to the user specified disk file. Compound files are explained fully in the ATARI DOS user handbook, the following is a brief description:

Compound files consists of two identification bytes of $FF followed by the code blocks which are made up of:

1 - a start address
2 - a finish address
3 - the actual code

The actual number of code bytes in each block is calculated by:

```
finish_address – start_address + 1
```

This arrangement enables the object code to be loaded directly into memory using the binary option "L" from the DUP menu.

The compiler will only write actual code and data into the file - uninitialized variables will not occupy any disk file space.

5.2 PL65's ASSEMBLER

PL65 allows beginners to gently ease themselves into machine language programming by combining an easy to understand high level syntax which easily interfaces to low level code.

It is beyond the scope of this manual to teach programming using assembly language - the object of this section is to describe the rules governing the use and integration of assembly language with PL65. Beginners are advised to purchase one of the many books available on machine code, eg:

PROGRAMMING THE 6502 by RODNEY ZAKS.

The complete set of 6502 mnemonics are incorporated into the language and they can be freely intermixed with high level code, however some rules have to be followed in order not to interfere with the normal running of PL65 these are listed below:

1 - Never cause a premature exit (RTS) from a function or interrupt without complete knowledge of what you are doing.

2 - If you must use the X register then save it first and restore it before executing any high level statement. The X register holds the index of the runtime software stack and thus any corruption of this can have undesirable results. A special page zero location is reserved to hold this value while user machine code is executing (XSAVE).

3 - Be careful with the SED (set decimal) instruction if this is not reset after use it will upset the math routines. Also it may cause problems with interrupts.

4 - High level statements will alter the contents of the registers. If you intermix high level statements with machine code then remember this. The only statements which do not alter the registers are GOTO and CALL.

Mnemonic differences

PL65 mnemonics are identical to standard 6502 mnemonics with the exception of the accumulator rotate and shift instructions:

```
ASL A, LSR A, ROL A, and ROR A
```

these have been changed to:

```
ASLA, LSRA, ROLA, RORA
```

This avoids confusion between the processor accumulator and the variable A (if declared).

The context of some of the mnemonics have changed due to the single pass nature of the compiler. These differences are described below.

Address operands

All addressing modes are supported, opcodes which take an address operand are computed using the rules governing compile time expressions. That is to say you can't reference any symbol before it has been declared, and all symbol references will return addresses.

Branch instructions

The branch instructions:

```
BED, BNE, BCC, BCS, BVS, BVC
```

must be followed by a valid label identifier. This allows branches both forwards and backwards within the same procedure. Branches can't take a compile time expression as a computed branch.

JMP and JSR

JMP and JSR take compile time expressions as operands and thus can't jump forwards within a procedure. However the high level statements CALL and GOTO reference labels directly and since they compile JMP and JSR this is no problem. So the rule is, if you want to jump backwards (or to a known absolute address) then use JMP and if you want to jump forwards use GOTO.

The AND instruction

AND can cause problems because it is both an expression operator and a statement, eg:

the instruction sequence:

```
LDA NUMB AND $FF
AND NUMB2
```

will be interpreted by the compiler as a number ANDed twice using the compile time expression rules - when it is intended that the second AND a new instruction. To avoid this confusion be sure to preceed each AND instruction with a semi-colon separator.

Splitting 16-bit numbers

The normal method of selecting the high or low byte of a 16-bit number is:

```
LDA #HIGH NUMB
LDY #LOW NUMB
```

or in some assemblers:

```
LDA #> NUMB
LDY #< NUMB
```

PL65 has no facility for this so bytes can be separated as follows:

```
LDA #NUMB/256
LDY #NUMB AND $FF
```

which does the same thing.

Using assembler to the best advantage

Assembly language is best used to do simple things to speed up code. Some small routines are more easily expressed using assembler, ie:

A BYTE variable is incremented in high level as follows:

```
A=A+1
```

in assembler this could be done with one instruction:

```
INC A
```

Mixing large chunks of assembler routines with high level code can get quite confusing and can lead to baffling program bugs, these are best kept in their own procedures.

Try to write your programs entirely in high level code to start with, and then when you are satisfied that the logic of the routines is correct, convert the time critical parts to assembler.

5.3 BUILT IN PROCEDURES

The runtime code which is automatically included with the compiled code contains a number of useful routines and addresses which are callable directly from the source program.

SYSTEM EOUATES

The system equates are useful addresses that may be used by the user. These addresses will always be corrected whenever the runtime code is altered either by relocation caused by changing the options from the options menu or if the compiler version number changes.

The equates described are intended primarily for use with primative procedures to enable the stack data to be manipulated directly with machine code.

r, XSAVE

This constant returns the zero page address where the user may temporarily store the X register when using machine code. STACK

This constant returns the base address of the software stack. This address will always be on a page boundary, eg 82000, 82300 etc.

This constant holds the address of the built in routine which pushes 16 bit values onto the software stack. This routine can be called by the

user by loading the accumulator with the LO byte of the number, the Y register with the HIGH byte and then executing JSR PUSH.

PULL

r\ This constant holds the address of the built in routine which pulls a single 16 bit number from the software stack. The routine is called with a JSR PULL. Upon return the number will be held in the accumulator (LO byte) and in the Y register (HIGH byte).

SYSTEM VARIhBLES ERRNUM

The BYTE variable which holds the error number, stored by the procedure ERROR.

PROCEDURES

This procedure is a very efficient block memory move routine which also includes direction sensing to remove the overwriting problems that normally occur when moving blocks up and down in memory.

This procedure fills an area of memory with a specified value.

This procedure stores the supplied error number into the GLOBAL system variable ERRNUM. If a trap has been set then control is passed to the TRAP vector (after first restoring the processor stack position) otherwise no other action is tiken.

Note:- Errors 1 and 2 are reserved for use by the runtime code. Error 3 is used by the VAL function in STRING.LIB. Also error numbers greater than 128 are operating system errors which will be utilised if TERMINRL.LIB is included. The user may use any other value in the range 4 to 127 in the call.

5.4 RCCESSING THE SOFTWRRE STACK

The software stack can be manipulated by machine code directly, this is necessary in primative procedures because the auto code that normally handles this is not included.

The software stack occupies one complete 256 byte page of memory. The actual location in memory of the stack may be changed by the user, for this reason the system equate STACK is provided. The stack may be accessed using the 6502 abs,X instructions, where the X register holds the index into the stack.

For a completely empty stack the index should be SFF. In actual fact because the stack occupies a complete page of memory the absolute index point is not particularly important.

Numbers are entered onto the stack in 16 bit (2 byte) values only, thus for each number added the index will be decremented by 2.

eg in order to push the value $20 to the stack the code will be

LDA #0 DEX STA STFICK,X LDA #%20 DEX STA STACK,X

Notice that the high byte is pushed first to maintain the LO HI format.

Numbers can be pulled from the stack using the opposite method to above.
 Note:- The runtime code already provides routines to PUSH and PULL

stack values. The above example only illustrates how numbers are stored on the stack. To access other than the first number on the stack add the number offsetd2 to the stack base, eg to access the second number use

STFICK+Z,X for the LO byte and STACK+3,X for the HI byte

. . and for the third number use

STACK+4,X and STACK+5,X etc.

5.5 PRIMRTIVE PROCEDURES

Primative procedures allow complete control of programs by suppressing some of the automatic stack handling activities carried out by the runtime code.

R11 procedure types except INTERRUPTS are simply subroutines which can be called with a 6502 JSR instruction, and terminate with a RTS instruction. INTERRUPT procedures terminate with an RTI instruction.

In standard procedures the entry and each exit point ie the BEGIN, END and RETURN statements cause the automatic compilation of handling code. The actual code compiled is dependant on the type of procedure. In PROC and FUNC types the BEGIN code transferes the parameter data from the stack to the parameter variables. In INTERRUPT procedures it saves all the processor registers and in MAIN it initialises the program.

The END and RETURN statements compile either an RTS or RTI depending

on the type of procedure. In INTERRUPTS they also restore all the processor registers and in FUNC's they have an additional runtime expression which leaves'a value on the stack.

Declaring a procedure as 'primative' stops the compiler generating the automatic high level code, thus BEGIN and END simply mark the begin and end of a code segment allowing pure machine code routines to be generated. In actual fact the END and RETURN statements still compile the RTS or RTI instruction but nothing else.

Primative procedures are declared by including a "t" in the procedure header between the name and the opening bracket for the parameter list e9 9

PROC NAMEt(1NT addrl,addr2)

Primitive procedures inhibit the allocation of memory for the parameter variables. If data is to be communicated then a parameter

list is still necessary but at runtime the data will simply sit on the software stack and will have to be manipulated by user generated machine code.

Primative functions have the added effect that the expression that normally accompanies the END and RETURN statements is not

expected by the compiler. Thus it is important that the programmer creates code which leaves a value on the stack.

/-

eg A function to implement a fast multiply by two can be written,

FUNC fast-mult*(INT numb) BEG IN ASL STACK,X ROL STACK+l,X

END

You can use any high level statement except TRAP in primative procedure these will work as normal. 5.5.1 PRIMATIVE PRRRMETER PASSING

Primative calls to any procedure or function can be made by including an aterisk after the procedure identifier and before the parameter opening bracket ie

This facility allows the parameters for the procedure to be generated by machine code rather than at high level.

eg if the procedure PLOT needs 2 parameters, then these could be supplied as follows

LDA #10;LDY #0;JSR PUSH !push screen X value LDA #12;LDY #0;JSR PUSH !push screen Y value PLOT* ( )

In actual fact the primative call simply suppresses the compilers error report of the parameter mismatch (errors 31 and 32). This enables some parameters to be supplied in the parenthesis as per normal and the rest to be supplied with machine code. eg the above call to PLOT could have been written

LDA #10;LDY #0;JSR PUSH - PLOTt(l2)

5.6 THE COMPILER LOCATION COUNTER

4s mentioned in the language description the @ symbol references the location counter. The location counter is the internal counter used by the compiler to determine where in memory the next section of code or variables is to be located.

The facility exists to set the location counter to any desired address thus allowing code or data to be located into any part of memory. This is equivalent to the ORG command used in assemblers. It is assumed that the user is familiar with assembler location counters, it should be changed only with great care.

The location counter is set as follows, where CEXPR is a valid compile time expression. /Before changing the location counter it is necessary to remember the current location address in order for the compiler to be able to carry on at the same address where it left off.

Setting a constant to the current address is the easiest way eg CONST DUMMY=@ and then set the location counter

and after the code is complete resets the location counter.

5.6 THE COMPILER LOCATION COUNTER

. 4s mentioned in the language description the @ symbol references the location counter. The location counter is the internal counter used by the compiler to determine where in memory the next section of code or variables is to be located.

The facility exists to set the location counter to any desired address thus allowing code or data to be located into any part of memory. This is equivalent to the ORG command used in assemblers. It is assumed that the user is familiar with assembler location counters, it should be changed only with great care.

The location counter is set as follows, where CEXPR is a valid compile time expression. /Before changing the location counter it is necessary to remember the current location address in order for the compiler to be able to carry on at the same address where it left off.

Setting a constant to the current address is the easiest way eg CONST DUMMY=@ and then set the location counter

and after the code is complete resets the location counter.

Part VI LIBRllRY FILES

The PL65 language library source files included on the system disk are intended to offer interfaces to the operating system of the host computer and standard utility routines. These routines are not included in the 'built in' runtime code because not all programs will need them and it is a waste of memory to include a significant portion of code if it is not going to be used. You may decide that you only need selected routines provided in a library file for a particular program, ie you may need the PLOT and DRAW procedures from the graphics routines and not LOCATE and GFILL, in which case you can create another file with these deleted. Be careful though that you do not delete a procedure that is required by other library files.

This section details the usable procedures which are created by these files.

6.1 TERMINAL-LIB operatinq system interface

This file contains standard operating system interface routines. The following is a description of the procedures and functions which are directly callable from your own programs.

This is the main procedure interface to the central 110 subsystem. This procedure expects you to supply the iocb no (0-81, command, and the address and length of the block of memory that is to be transmitted or received through the operating system. If the operating system reports an error then CIO will vector through the runtime routine ERROR, settlng the global variable ERRNUM to the reported error number, thus if any TRQP has been set by the user then this will automatically execute. Q11 further routines described in this section pass through CIO thus will automatically be error checked.

CLOSE(iocb) The.specified block is closed.

n

The block specified by iocb is opened using the auxl and dux2 data, the file-spec is held in the memory locations specified by address and length. Remember the rules on parameter passing, addr and len may be a string,

will open the keyboard for input.

The user should refer to the ATARI-BASIC manual for full information on the auxl and dux2 data values. Briefly aux2 would normally be zero, auxl controls the input/ouput OPEN direction. Q value of 4 will OPEN

for input, a value of 8 will OPEN for output and 12 will OPEN for input and output.

P

Outputs the specified byte to.the iocb. This is equivalent to the PUT# command in ATARI BfiSIC.

This function returns a byte from the specified device.

Note:-in ATARI BASIC GET#l,A would retrieve a byte. In PL65 this would be written A=GET (1)

.

Transmits a block of memory at address and of length bytes to iocb. Receives a block to memory with a specified maximum length.

Transmits a block of memory to device. The number of bytes actually sent will be a maximum of len but may be shorter if terminated with an EOL byte.

GREC(iocb,addr, len) Receives a block from device to memory location start address

n specified by addr. The actual number of bytes will be a maximum of len, but may be less if an EOL char is received.

STATUS(iocb)

Function which returns the status of the last operation of the specified iocb.

The string of the number is written to the screen in free format with no leading or trailing blanks.

Writes the specified string to the screen with no trailing EOL char. eg WRTSTR("he1lo") prints 'hello' on the screen.

n WRTLN(addr, len)

This procedure is the same as WRTSTR but it also sends an EOL at the end of the string.

Sends an EOL to the screen. INPUTS ( )

A string fuction which gets a line of text of up to 128 characters from the terminal. This can be used in a string assignment, eg

6.2 PEEKPOKE-LIB memory access

This file contains the standard PEEK and POKE routines which are familiar to the BASIC programmer. PL65 actually has no need for these routines as BASED variables can be used to achieve the same action. However they are included for those who cannot live without them.

PEEK(addr) Function that returns the value stored in memory location.

Stores byte in memory location specified by addr. DEEK(addr Function that returns the 16 bit integer stored at memory locations

specified by addr (LO byte in first location, HI byte in memory location addr+l).

n

Stores 16 bit integer into memory locations (LO into addr, HI into addr+l)

.

6.3 STRING.LIB number-RSCII conversion

This file contains standard ASCII conversion routines. They enable string data to be converted into numbers and visa-versa.

A11 string conversions are performed with reference to the GLOBAL variable BRSE. This can be altered by the programmer to any value in the range 0-255 to produce any numeric base desired, eg BRSE=16 will make all future uses of STR$ produce hexidecimal and BASE=8 will make all future uses of STRB produce octal. BASE is initialised at compile time to 10 for decimal conversion.

The procedures provided are STRS(number) This function returns the string of a number.

Function that returns the numeric value of a string. If the procedure finds an error in the ASCII text such that it cannot convert to a number then the procedure ERROR is called with an error number of 3 Conversion,is done with reference to the variable BASE.

. This function returns the length of a string. B=LEN(A$) sets the variable B to the current length of A%. String function which returns a one character string of the character , number supplied. would assign an ASCII space to the variable.

This function returns the ASCII character code value of the first character of the supplied string, eg

would set A to the value 65.

6.4 GRAPHICS-LIB ATARI qraphics handler

This file contains routines to interface to the standard graphics routines which are contained within the operating system ROM,s. The procedures are intended to be compatible with the equivalent ATARI-BASIC commands.

The selected graphics mode is enabled. Users should refer to the relevant ATARI literature on the modes available.

COLOR

COLOR is a variable that holds the current color for which future uses of PLOT and DRAW will draw their data. In ATARI-BASIC the color would be selected using the statement COLOR n, in PL65 this would be written COLOR=n

. -1

The SETCOLOR procedure accepts identical data to the equivalent ATARI BASIC command.

The equivalent ATARI BASIC command is PLOT co1,row. The color used in the plot is that last assigned to COLOR. Action of the PLOT procedure is to plot a single point on the screen at the co-ordinates supplied in col (column) and row.

The equivalent ATARI BASIC command is DRAWTO co1,row. The color used for the draw is that last assigned to COLOR. Action of the draw procedure is to draw a straight line from the current screen co-ordinates, determined from the last PLOT or DRAW, to the co-ordinates supplied in the procedure call.

\

This procedure has no ATARI-BASIC equivalent which is strange since it forms part of the operating system as does the PLOT and DRAW commands. PL65 includes it for completeness.

The fill works by filling each horizontal line from top to bottom of a specified area. The fill starts at the left and proceeds across the line to the right until it reaches a pixel with a non zero value, it then proceeds with the next line. Beware, the fill command will go into an infinite loop if a fill with zero data (COLOR=0) is attempted on a line which has no non-zero pixels.

The equivalent ATARI-BASIC command is LOCATE col,row,var where var is a variable. PL65,s version is a function, thus would be written var=LOCATE(col,row).

The procedures in this file enable simple control of the built in sound chip.

This procedure acts in the same way as BASIC's SOUND command.

This procedure acts in the same way as BASIC's SOUND procedure but an -additional

time parameter is included. This automatically cancels the

sound after timet20mS (milli-seconds) and 'time' may be in the range 0-255. The timing is implemented using a vertical blank interrupt routine. This allows normal processing to continue whilst the sound is played. If you intend to include your own vertical blank routine then you will have to take this routine into account.

This procedure changes just the pitch of the selected channel. This can only be used on a channel which has previously had its distortion and volume set by the SOUND-procedure.

6.6 PADDLE-LIB

This file defines four variable arrays, the absolute addresses of which are set to the shadow registers of the joystic and paddle ports. The arrays are,

These variables can be used in the same way as the equivalent controller functions in BASIC.

eg STICK 0 can be read as follows and the trigger button of the stick can be read as follows Refer to the BASIC reference manual for the values returned.

6.7 UDG-LIB user defined character qraphics routines

This file contains procedures which have no equivalent in ATARI BASIC. They enable the easy handling of user defined character sets.

At compile time an array called CHARRAM is created. This holds the data for the new character set. The file provides the following useful procedures.

The main procedure which enables the user defined characters by downloading the old character set into RAM and setting all the relevant hardware registers. This procedure should be called before defining new characters, it may also be called to erase an existing user defined set and set up for a new one.

This procedure will redefine the desired character using the eight user supplied data bytes.

The character set is not built in normal ASCII order due to the way the hardware acts on screen data. This procedure rearranges the character code so that ASCII character constants can be used.

eg The internal representation of the space character is code 0, in ASCII it is code 32. Thus to redefine the space character using DEFCHAR would mean that the supplied value for 'char' is 0. Using DEFASC however allows the actual character constant to be suppied, ie to redifine ASCII 'A' would be written

Sets the hardware to use the internal character set. CUSTOM ( ) Sets the hardware to use the custom defined character set.

Character redefinition example

Binary data can be used to good effect to create a character on screen using the screen editor,

PROC redefine ( ) BEGIN UDG( )

DEFCHAR(0, !redefine space character! %

11 ,

% 111111 , %11 11 11, %11111111, % 111111 , % 1111 , % 11 11 ,
Zlll 111)

END.

The characters are more easily seen by first using spaces in the place of the binary zeros and then filling these in after.

The initialization procedure 'UDG' need only be called once- at the start of a program. If the graphics mode is changed after this procedure is called the the hardware will revert back to the ROM character set, if this happens call 'CUSTOM' to switch the custom set back in.

6.8 PMG-LIB ctlaver missile qraphics routines

This file contains routines that give simple control of the built in player missile graphics capabilities of the hardware.

Each player can be 8 bits wide by 8 bytes deep (missiles 2 bits by 8 bytes) and the shapes for these players can be pre-defined and stored in shape tables. There are 3 shape tables which are,

1- The primary shape table for players. Whenever a shape is moved on the screen it is from this table that the data is taken. There are four shapes held in this table, one for each player (numbered 0 to 3).

2- The secondary shape table. This table holds 16 shapes which can be defined by the user. Any shape held in this table can be transfered to any primary shape.

,\

3- The missile shape table. This table holds the shape data for the four missiles.

By using shape tables animation can be easily achieved on the screen. First by defining up to 16 secondary shapes, then by transferring a secondary shape to one of the primary shapes. This can then be plotted on the screen. Another secondary shape can be transferred to the primary shape, this will not alter the image on the screen until the next time the player is plotted or moved. Thus by storing a sequence of images in the secondary shape table, loading the primary shape and then moving the player- a f.airly good animation can be achieved.

Note:- These routines do not represent the ultimate in speed, they are intended to be general purpose and easy to use. Having said that they are fairly nippy.

At the beginning of this file two constants are declared, these are HEIGHT and SHAPES. These control the height of the players and the number of secondary shapes available and are preset to the values 8

and 16.

These constants can be changed to any value to achieve different heights of players and more shapes if required.

PMG.LIB contains the following useful routines.

This routine enables player missile graphics and sets up both the hardware and software for the TV-line resolution (1 or 2) as supplied to the routine. The screen memory is allocated in a BYTE array 2048 bytes long called PMRAM.

This routine sets the color of the selected player. The supplied color and luminence is the same as for the SETCOLOR command in BASIC.

The size of the selected player is set to the supplied value , 0=

r\ normal, l=Z*width, 2=normal, 3=4fwidth. DEFSHAPE(shape,

... n data bytes determined by HEIGHT)

The secondary shape is Pefined using the supplied data bytes. The first data byte in the list forms the top of the shape. Shapes are created using the same method as for user defined characters.

The primary shape (0 to 3) is loaded with the shape data of the secondary shape (0 to SHhPES-1). This will not alter the screen display of the player until the next plot or move.

The player is plotted on the screen at the X,Y co-ordinates. The Y pos range is 0 to 128 in 2 line resolution mode and 0 to 256 in 1 line resolution. The Y position is automatically masked so that selecting Y , position 129 in 2 line mode will actually plot at position 1.

The selected player is moved by one screen ordinate in the direction specified. The directions are,

Moving players off the screen will cause the player to wrap to the opposite end of the screen.

MLOAD(missile,..n data bytes)

e selected missile shape is defined using the supplied data bytes. ssiles are only 2 bits wide so the most significant 6 bits of each ta byte are ignored.

Plots the selected missile on the screen at the supplied X,Y co-ordinates.

Moves the selected missile in the supplied direction. Directions are the same as for players.

CPMR ( )

Clears the screen of all players and missiles. This does not affect any of the shape tables.

CLRP.(player)

Clears the selected player from the screen. This does not affect the shape tables.

P

Clears the selected missile from the screen. This does not affect the shape tables.

Collision detection

The following routines are collision detection functions which return a bit pattern value of the form,

The four bits determine the type of collision, a 1 bit means that a

collision has been detected since the last HITCLR. The bits refer from right to left to the playfield or player number that the selected object has collided with.

eg To read the collision status of missile 3, /-'

If A has the value 1 then missile three has collided with playfield type 0. If A had the value 2, then this would indicate a collision with playfield type 1. If A had the value 4 then this would indicate a collision with playfield type 2. It is also possible for combinations of collisions to occur.

Clears all the collisions detection bits. MPFCOLL(missi1e) Returns the status of the missile to playfield collision bits. PPFCOLL(player1 Returns the status of the player to playfield collision bits. MPLCOLL(missi1e) Returns the status of the missile to player collision bits. PPLCOLL(p1ayer)

n Returns the status of the player to player collision bits. X,Y position arrays

There are four BYTE arrays which can be read by a program. These arrays hold the screen positions of each object. PXPOSC43 Which holds the X position of each player. PYPOSC41 Which holds the Y position of each player. MXPOSC41 Which holds the X position of each missile. MYPOSC41 Which holds the Y position of each missile. eg The co-ordinates of player 2 can be read as follows, x=PxPosC2l;Y=PYPosC2l

Example routine

The following example shows how player-missile graphics are enabled. It defines two players and two missiles and moves them in circles on the screen.

```
!PMGRAPHICS DEMO! INCLUDE PMG.LIB INT A,B,C

MAIN( ) BEG I N

PMGRAPHICS(2) !2 line resolution DEFSHAPE(0,1,2,3,4,5,6,7,8)
DEFSHAPE(1,8,7,6,5,4,3,2,1) PCOLOR(0,5,8) PCOLOR(1,7,8)
LOADSHAPE(0,O) LOADSHAPE(1.1) MLOAD(0,1,2,1,2,1,2,1,2)
MLOAD(1,1,2,3,1,2,3,1,2) PPLOT(0,60,50) PPLOT(1,.70,60)
MPLOT(0,80,70) MPLOT(1,90,80) REPEAT FOR A=0 TO 7 DO FOR
B=0 TO 10 DO PMOVE(0,A) PMOVE(1,7-A) MMOVE(0,A)
MMOVE(1,7-6) NEXT NEXT

END

FOREVER .
```

6.9 DEBUG-LIB runtime debuqqinq utilities

Most program bugs can be found simply by examining the source listings. Occasionally however bugs appear that seem to defy logic.

DEBUG.LIB can help on these occasions by providing runtime procedure testing facilities.

To use the facilities in DEBUG.LIB the following changes need to be made to your source file. 1- INCLUDE DEBUG.LIB at the start of the file

2- Make the first statement in MRIN a call to the debug initialisation routine eg MAIN( )

BEG IN /-'buginit( ) ! rest of your routine!

3- Include the following as the first statement (after the retrapping statement) in your TRAP handler,

IF ERRNUM=128 THEN DEBUG( )

END IF

This extra bit 1s necessary because although DEBUG sets up a vertical blank break key sensing routine, sometimes the operating system 1/0 sensing routines get there first and take over.

After recompiling your program- arm yourself with a cross reference listing of the program (see COMPILER OPERATION section), you will need this to find out the addresses of the procedures and variables.

Once recompiled your prog'ram should run normally, to enter the debug mode press the break key. Once entered a "debug" message will be displayed followed by a ">" prompt, you can now select one of five commands.

All of the commands are entered using a single letter followed by a hexidecimal number of 1-4 digits, some commands need extra values, these are entered separated by commas.

The commands are detailed below. Dnnnn

This command displays the contents of eight successive memory locations starting at the address nnnn, eg D4EB5 might display

The display command is useful for displaying the contents of variables, a BYTE variable value is the first number displayed after the address, an INT variable value is made up of two successive memory locations thus in the above example the INT value (in LO HI format) is 823F14.

Cnnnn , nn This command is used to change the contents of a memory location, eg -. . !

will store the value $A5 in memory location 34EB5. Gnnnn

The GO command will cause program execution to jump to the given memory location. This is a 'leap in and see what happens command', if the processor meets a RTS instruction then control will pass back to DEBUG mode. If you jump into the middle of a code area don't be surprised if the computer goes to sleep.

The RUN command will re-run the program as if it were just loaded from disk, with the exception of those changes made using the CHANGE command.

Tnnnn,nnnn, ... nnnn

The TEST command allows procedures and functions to be tested with supplied parameters, eg

will supply the values 10 and 20 (in hex) to the procedure whose address is at 350D2. The entry address of a procedure is the number in the second column on the cross reference listing. The third numbers shows how many parameters the procedure requires (note:- this is also shown in hex).

This command will allow up to sixteen parameters to be passed, each

parameter can have a value in the range 0-$FFFF. A procedure with no parameters is called with no trailing commas eg

T50D2

When the called procedure returns DEBUG will display the returning parameters, eg

>T50D2,10 Routine returns 2 values 20 49A3

The above example shows a typical reply from a string function which returns two values corresponding to the length and address of a string. Note that the returned values are displayed in reverse order.

A numeric function should return one value and procedures will not return any values.

Appendix A ERROR MESSAGES SUMMARY

1- Symbol already in use. 2- ";" expected. 3- Already in procedure. 4- "(" expected. 5- ")" expected. 6- No such symbol. 78- Variable declaration or BEGIN expected. 9- Syntax error. (statement expected)

10- THEN expected. 11- Memory full. (FATAL) 1213- Identifier expected. 14- Incorrect expression argument. 15- DO expected. 1617- "[" expected. 18- "I" expected. 19- "%" expected. 20- "=" expected. 21- "," expected. 2223- INT variable expected in FOR/NEXT loop. 24- TO or DOWNTO expected. 25- TRAP not allowed in INTERRUPT or primative procedures. 26- Identifier character must be alpha numeric. 27- Illegal expression argument type. (in compile time expression) 28- Unconvertable digit in number. 29- Command ex pec ted

.

30- Can't use reserved word as identifier. 31- To many parameters in procedure call. 32- Not enough parameters in procedure call. 33- Can't assign DATA to parameters. 34- Can't LINK until procedure terminated. 35- INCLUDE'S nested too deep. 36- Compilation completed with no MAIN routine. 37- Unexpected end of file. 38- Too many parameters in procedure definition. 39- Expression to complicated, processor stack overflow. 40- Compiler stack overflow. (FATAL) 41- Compiler stack underflow.(FATAL) 4243- BASED type must be a POINTER. 44- Branch out of range. 45- Trying to GOTO, CALL, TRAP or branch to a symbol which is not a label. 46- Has illegal addressing mode.(assembler) 47- Value >255. 48- Statement expected- illegal use of symbol. (not a procedure or

assignable variable) 49- Error in address index mode.(assembler) 50- Too much DATA for size of variable. 51- Not an executable statement. 52- Error in string constant(" or CHAR expected) 53- String comparison error. 54- ELSE or ENDCASE expected. 55- GOTO expected. 56- OF expected. 57- Compare string invalid. 58- Variable identifier expected. 59- String factor expected in assignment. 60- String factor expected in return from string function. 61- Expression does not yield a numeric result. 6263- Not a FORWARD procedure. 64- Undefined parameter error in BEGIN. (parameter error in PROC header) 65- Parameters not allowed in MAIN and INTERRUPT. 66- BEGIN not allowed here. 67-.Invalid PROC identifier in parameter list (this identifier is not a procedure) 686970- No matching structure initiator. 71- ENDIF expected. 72- ENDWHILE expected. 73- UNTIL expected. 74- NEXT expected. 75- ENDCASE expected

.

76- ENDIF expected. 77- ENDCQSE expected. 78- '

expected after ASCII constant.

79- Error in INCLUDE filename. 80- Too many POINTER,s (no more page zero RAM available). 81- Unterminated IF. 82- Un termina ted WHILE.

/83- Unterminated REPEAT. 84- Unterminated FOR. 85- Unterminated CASE. 86- Unterminated OF. 87- Unterminated CASE.

The following are system errors. They do not occur during compilations.

90- Stack is not on a page boundary 91- New zero Page ORG is not possible with current allocated length. 92- Zero page length not possible.

RUNT I WE ERRORS

There are only two runtime errors, both of which are used in the string handling routines. The reason for so few error checks is to keep the runtime code as fast as possible. Because string handling is inherently slow and can be very destructive, the inclusion of error checks on array assignment subscripts is warranted.

Because of this it is important to keep a tight rein on your program and include plenty of error checks. A facility is built into the runtime code (see ERROR and TRAP) to enable errors to be handled more easi 1 y.

Be careful with variables, especially arrays . The compiler will allow

you to assign numbers to arrays with subscripts which are out of range of the declared size, and as variables are intermixed with the code then a crash is inevitable.

The two built in errors are Error 1

String to long to fit into allocated variable size. If no TRAP is set then the string will be truncated to fit into the available space. Error 2

Array subscript out of range of declared size. If no TRRP is set then the assignment will be ignored.

In addition the library file STRING.LIB reserves error 3 to indicate an invalid character in the VAL function.

The library file TERMINAL.LIB reserves error numbers greater than 127.

Runtime memory map

This map shows the standard mapping of a compiled program in memory. It assumes that no relocation has taken place.

SYNTAX FLOW CHOHTS

The syntax flow charts give a precise language definition and allow quick look up reference on specific language element syntax.

Elements shown in upper case specify 3 PLb5 reserved word or a language construct. Those shown in lower case specify a pre-defined user construct. Note:- Comments are not shown in these flow charts.

CEXPR (corpile tire expression) -->-- [TERN --)-+------)--------+----:,----
+---->----+--l A l l +-(-l l

t l

l l l l

8 l OH l l CTERH --(-+----<----+----<----+----i----+

l l

XOR l r\ CFRCT

VAR (variable)

t----------------(-----------t-------\------I I

I I

-)--+->I

I I

I I

V

I I

I

I I

I I

I I

I I

BYTE ->-t->- IDENT -+-tI

I I

I I I I y

+ I

I I

[CEXpR] ->-+->-+>,

I I

, +-::+---------)--------I

------------(------------------------- t t I

I I

= -- CEXPR -)--+ ---------) '1

I I

DATfi -)-+-)I

I t-,L

+-------------(--------------I

I

I I

I I @ string "+

+-->-- BASED -- pointer ident ------------------)-------------------------------
<--------------------+ +- ---------)

+- ------- -- +

I I

I I

t->- INT ->-t->- IDENT ->-t-t- [CEXPR] -)-+->-+-:>--------- t I

I

+ I

I = -- CEXpR --1--+-----j---+->-+----+-----)-------- ------+->.-+->---+-->-+ --
-- ;--- ---< ----- + CEXpR ---->---+-tI I

A I I I

' ! -+

I I + I I

I I

I I

I , I A

i I

I I

I I

I I

t I I +->- BASED -- pointer VAR-LIST

STATE (statement)

STATE (continued)

IDENT iidentifier) STRING-CONSTANT

STRING-FACTOR -j-+)I I v I I

I I

+-------------------)----------------------I I

t-j- string ident -- $ ->-t->- [-- EXPR -t- , -- EXPH -t- 1 -->------v

I I +------>-----I 1

t STRING-CONSTANT ------------------------,--------------------t->I

t I I

t I

I I I

EXYR iexpression) HEL (relational; SEXPR (Sirole Exorxrion!

->- FACT ->-+--)-----+------+------+-------+---1

I

I I

I A

I I

I I +-- FACT ----{--+

I I

f I

I I

I

+--->--+--(---+ I

I I I non

I I

ACT ifactor)

+------------>------------I

I I I fun[ident -- (--+->-+-)I t I I

+-j- STRING-FACTOR --+ I

:

I I

I I

EXPR ---j------- +-:>-+ I

I + ---- (---STRING-FACTOR ->-+-)I

t-}I

+'L I I

t-}I

t->I I

t->(--+ I

= --t I

< j --t a I

>= --+ I {= --+ ---- :---+

) --t-j- STRING-FACTOR --->-----t I

) -++

I I

I I I I I

I I

NODE Notes:- 1- Hot all modes are applicable to every instruction. 2-
BEXPR is a compile tire expression which results in a value (256

SINGLE I I I

t->- SED --)----t I I

t-)- CLD -->----+ I

I I I

t-j- SEE --)----t 1 I

I I

t->- CLC --j----t I I

I I

t->- SEI --)----t I

t-)- CLI --)----t I

t-)- CLV -->----+ I I I