Using the Parallax, Inc. PBASIC Tokenizer Library

Contents

INTRODUCTION....2
BACKGROUND....2
SUPPORTED PLATFORMS....3
  WINDOWS SUPPORT....3
  LINUX SUPPORT....3
  MACINTOSH SUPPORT....3
WHAT'S NEW....3
  FOR DEVELOPERS WHO HAVE USED THE PBASIC TOKENIZER LIBRARY V1.16....3
  FOR DEVELOPERS WHO HAVE USED THE PBASIC TOKENIZER LIBRARY V1.14....3
TOKENIZER OVERVIEW....4
TMODULEREC STRUCTURE....5
SOURCE BUFFER....7
SOURCE-TO-TOKEN-REFERENCE STRUCTURE....8
TESTRECALIGNMENT FUNCTION....9
VERSION FUNCTION....11
COMPILE FUNCTION....11
GETRESERVEDWORDS FUNCTION....12
FIELD FORMAT (ERROR)....14
FIELD FORMAT (TARGETSTART, PROJECTFILESTART, PORTSTART, LANGUAGESTART AND ERRORSTART)....14
FIELD FORMAT (PROJECTFILES)....15
  PROCESSING PROJECT FILES....16
FIELD FORMAT (PORT)....17
FIELD FORMAT (EEPROM, EEPROMFLAGS AND VARCOUNTS)....17
  EEPROM AND EEPROMFLAGS....17
  VARCOUNTS....18
FIELD FORMAT (PACKETCOUNT AND PACKETBUFFER)....18
AN EXAMPLE OF THE COMPILING PROCESS....18
ERROR MESSAGES....20
LINUX SERIAL PORT TIPS....21
ACKNOWLEDGEMENTS....21
COPYRIGHTS AND TRADEMARKS....22
DISCLAIMER OF LIABILITY....22

Introduction

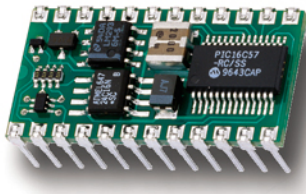
This document discusses the use of the PBASIC Tokenizer Library to generate object code for BASIC Stamp microcontrollers; specifically, the BS2, BS2e, BS2sx, BS2p and BS2pe microcontrollers.

By reading this document, using the information contained herein and using the PBASIC Tokenizer Library software that it describes, you are acknowledging that you have read and agree to the terms and conditions in the Parallax PBASIC Tokenizer Software License.

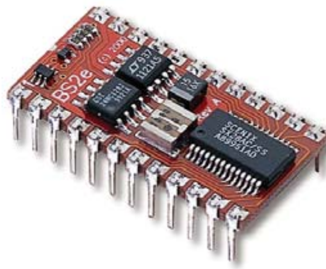
IMPORTANT: If you have used a previous version of the PBASIC Tokenizer, make sure you read the "What's New" section!

Background

The BASIC Stamps (some are pictured below) are small programmable controllers intended for general-purpose use. They are programmed in a variation of the BASIC programming language, called PBASIC, and are frequently used by industrial, commercial and R&D engineers, educators (in classroom electrical/software engineering sessions) and electronic hobbyists.



BASIC Stamp 2



BASIC Stamp 2e

BASIC Stamp 2p24

Parallax has created and maintained integrated development environments (IDEs) for the BASIC Stamps that run on DOS® or Windows® PCs. Software may be downloaded, free of charge, from:
http://www.parallax.com/html_pages/downloads/software/software_basic_stamp.asp

This software allows a user to perform standard programming operations such as 1) edit PBASIC source code, 2) load and save to disk, 3) tokenize (compile) source code, 4) download to BASIC Stamps, 5) Debug code, etc.

Since many Parallax customers are software developers (on various platforms), they would appreciate the ability to write their own BASIC Stamp development system.

To satisfy this need, Parallax has taken the key component of the BASIC Stamp Editor, the PBASIC Tokenizer, and made it available to the public. The PBASIC Tokenizer is a carefully written set of program routines that takes PBASIC source code and converts it to the proper tokens that are suitable for downloading to BASIC Stamps. The PBASIC Tokenizer is made available in a pre-compiled shared library for various platforms.

A properly skilled developer can create a BASIC Stamp development environment using these steps:

1. Read this entire document carefully,
2. Write editor/development software (command-line or GUI-based) on the supported platform,
3. Link in the PBASIC Tokenizer Library,
4. Create a source buffer in memory,
5. Create the TModuleRec structure in memory and test it by using the TestRecAlignment function,
6. Remove the call to the TestRecAlignment function and use the Compile function thereafter, and
7. Add the BASIC Stamp Programming Protocol routines (detailed in an accompanying document).

Supported Platforms

The PBASIC Tokenizer Library is available for the Windows, Linux and Macintosh operating systems. Pay close attention to the function prototypes (formats) of the four published functions, TestRecAlignment, Version, Compile and GetReservedWords, described in their associated sections.

Windows Support

Support for the Windows operating system is provided through a DLL file called "Tokenizer.dll". It uses a standard API interface and can be linked statically or dynamically using the appropriate methods for the programming language of your choice. Refer to your programming environment's documentation on linking to DLL files.

The Windows version was developed and tested on an Intel Pentium® 4 processor running Window 2K and should run properly on any Intel x86-based processor running Windows 95 or above.

Linux Support

The Linux support is provided through a shared library file called "tokenizer.so". It can be linked using the appropriate methods for the programming language of your choice. Refer to your programming environment's documentation on linking to shared library files.

The Linux version was developed and tested on an Intel Pentium® 4 processor running Linux kernel versions 2.2.14-5.0 RedHat 7.3 and Linux kernel version 2.4.18-14 RedHat 8.0 (gcc versions 2.96 and 3.2).

Macintosh Support

The Macintosh support is provided through a shared library file called "tokenizer.shlb". It can be linked using the appropriate methods for the programming language of your choice. Refer to your programming environment's documentation on linking to shared library files.

The Macintosh version was developed and tested on a PowerPC G3 and G4 processor running OS 9 and OS X (10.2).

What's New

For Developers Who Have Used the PBASIC Tokenizer Library v1.16

The following items have been changed in release 1.23.

1. The PBASIC Tokenizer Library v1.23 supports both PBASIC 2.0 and PBASIC 2.5 source code.
2. The TModuleRec structure now includes two new integer fields inserted after the PortStart field (see TModuleRec Structure for more information):
    1. LanguageVersion – Indicates version of the PBASIC language used by the code. 200 = PBASIC 2.00, 250 = PBASIC 2.50.
    2. LanguageStart – Indicates the beginning of the language version in the source.
3. A new function, GetReservedWords, was added to retrieve a list of the reserved words and their associated types using the actual source code to be tokenized. This function enables editors to actively update their syntax highlighting display (if used).
4. A new, optional parameter, called TSrcTokReference, was added to the Compile function to retrieve EEPROM Token-to-Source Code references for PBASIC simulators. See the "Source-to-Token-Reference Structure," page 8, for more information.

For Developers Who Have Used the PBASIC Tokenizer Library v1.14

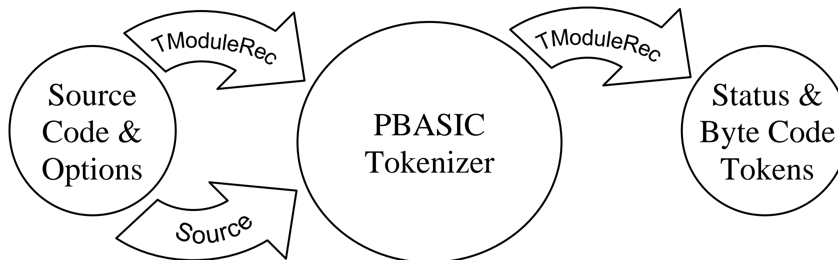
Based on feedback since the first release, v1.14, the following items have been changed in release 1.16.

1. The TModuleRec element "Source" was removed from the structure to provide compatibility with platforms and development languages that impose limitations on fixed structure size. The source code buffer should now be defined separately (still within the calling program, but outside of the TModuleRec structure) and passed as an additional pointer to the Compile function. **It is critical that you modify your code to reflect this since the structure size and layout, and the Compile function's parameter format are not**

compatible with version 1.14 of the tokenizer. See the “TModuleRec Structure” and “Source Buffer” sections for more information.
2. The Compile function has an additional paramenter: a pointer to Source buffer. See the above item.
3. The TModuleRec elements “SourceStart” and “SourceLength” were renamed to “ErrorStart” and “ErrorLength”, respectively, to clear up confusion and more clearly reflect their function. It is not critical that your code be changed to reflect this fact, however it is suggested to do so in order to maintain consistency with the documentation.
4. An additional function, Version, was added to provide the calling program with the ability to retrieve the PBASIC Tokenizer Library’s version number. See “Version Function” for more information.
5. The TestRecAlignment function returns different values due to the changes indicated in item 1. See “TestRecAlignment Function” for more information.

Tokenizer Overview

The tokenizer is designed to be a relatively independent “black-box” device, see the functional overview diagram below. The tokenizer simply accepts text-based source code plus option flags as its input, processes the data and outputs results in the form of status flags and tokenized byte codes.



The form of the input is a shared memory structure, called TModuleRec, and a shared source code text buffer. Note: Optionally, another shared memory structure, TSrcTokReference, may also be used (see “Source-to-Token-Reference Structure,” page 8, for more information). Both the TModuleRec structure and the source code buffer must be created and initialized outside of the tokenizer; this is the responsibility of 3rd-party developer using their desired programming language. A pointer to the structure and buffer is passed to the tokenizer as a parameter of the Compile function. The TestRecAlignment function requires only a pointer to the TModuleRec structure. Both functions modify the data within the TModuleRec structure and exit with a true or false status as necessary. The calling application simply views the modified data in the TModuleRec structure and uses the results to either notify the user or download the compiled tokens to the BASIC Stamp.

The interface to the tokenizer library consists of the following components:
1) TestRecAlignment function (used only during development),
2) Version function (used only when verifying the version of the tokenizer library),
3) Compile function,
4) GetReservedWords function (used to assist in updating an editor’s syntax highlight display),
5) TModuleRec structure (created by the calling program and passed, by pointer, to the tokenizer),
6) Source code text buffer (created by the calling program and passed, by pointer, to the tokenizer),
7) Optional TSrcTokReference buffer (used to assist PBASIC simulators).

These components are described in more detail below.

See “An Example of the Compiling Process” (pg 18) for more information on the compiling process.

TModuleRec Structure

The TModuleRec structure contains 19 fields organized in the format show below. This is a memory structure (sometimes called a record) that holds status flags and data generated mostly by the tokenization process, however, some flags may be set by the calling program as well.

Field Name	Type	Size (in bytes)	Description
Succeeded	bool	1	Indicates whether the tokenizer passed or failed on compile. `True` = Compile passed. All fields (except Error, ErrorStart and ErrorLength) contain valid information about the results of the operation. `False` = Compile failed. Error, ErrorStart and ErrorLength fields contain valid information about the error. All other fields can be ignored.
Error	char *	4	Pointer to null-terminated error message. This field is only valid when Succeeded field is false. See "Field Format (Error)" (pg 14) for more information.
DebugFlag	bool	1	Indicates whether there is debug data (ie: a `DEBUG` command in source code). This is used to determine whether or not to open a Debug Terminal after downloading the code to the BASIC Stamp. `True` = Debug data exists. `False` = No Debug data exists.
TargetModule	byte	1	Indicates which BASIC Stamp Module to compile for. A target module is required to compile the code. The target module is normally determined by the Stamp directive in the source code, but optionally can be indicated by the development environment, as indicated below. This is a dual-purpose field. PURPOSE 1: If `Compile` function is called with `ParseStampDirective` set to `true` (normal use), the tokenizer looks for a Stamp directive in the source code. If a valid directive is found, the compiler stores the value of the target module in this field and compiles. If no Stamp directive is found, the tokenizer generates an error. PURPOSE 2: If `Compile` function is called with `ParseStampDirective` set to `false` (optional use), the tokenizer ignores any Stamp directives in the source and instead uses the value currently in this field to compile the code. This means, the TargetModule field MUST be set to the proper value BEFORE calling Compile. 0=None, 1=<reserved>, 2=BS2, 3=BS2e, 4=BS2sx, 5=BS2p, 6=BS2pe
TargetStart	int	4	Indicates the source position of the target module string (within $STAMP directive). This field is ignored when `Compile` function is called with `ParseStampDirective` set to `false`.
ProjectFiles[7]	char *	4 * 7	Array of seven pointers to the null-terminated paths and names of related project files, if any are found in the Stamp directive. Set to `NULL (0)` if file corresponding to index is not indicated. The calling program can use this to determine which additional files, if any, to load and compile. Note: the tokenizer does not perform any verification of file name format or of the existence of the file; the calling program MUST perform those procedures. See "Field Format (ProjectFiles)" (pg 15) for more information.
ProjectFilesStart[7]	int	4 * 7	Array of seven values each indicating the source position of project file names.
Port	char *	4	Pointer to null-terminated COM port string in Port directive, if found. This can be used to determine the exact serial port to use when downloading the code. See "Field Format (Port)" (pg 17) for more information.
PortStart	Int	4	Indicates the source position of the port name.

LanguageVersion	int	4	Indicates the version of the target PBASIC language. This can be used to retrieve or set the version of PBASIC used by the source code. A value of 200 means PBASIC 2.00. A value of 250 means PBASIC 2.50.
LanguageStart	int	4	Indicates the source position of the language version number.
SourceSize	Int	4	Indicates the size (in bytes) of the source code that was place in the Source buffer. This value should be set just before calling the `Compile` function.
ErrorStart	int	4	Indicates the source position of the start of text containing an error, called the "error selection". The tokenizer fills this field when a compile error occurs and it is only valid when the Succeeded field is `false`. The calling program can use this value, along with ErrorLength, to select and highlight the offending text in the source code.
ErrorLength	int	4	Indicates the number of characters in the error selection. The tokenizer fills this field when a compile error occurs and it is only valid when the Succeeded field is `false`. The calling program can use this value, along with ErrorStart, to select and highlight the offending text in the source code.
EEPROM[2048]	byte	2048	Contains the tokenized data if compile is successful. This indicates the actual "compiled" EEPROM contents of the BASIC Stamp after the PacketBuffer data is properly downloaded. The calling program can use this information, along with EEPROMFlags, to generate a "memory map" of the results of the compilation. See "EEPROM and EEPROMFlags" (pg 17) for more information.
EEPROMFlags[2048]	byte	2048	Contains status flags for each location in the EEPROM field. The calling program can use this information, along with EEPROM, to generate a "memory map" of the results of the compilation. BIT 7 : 0=unused EEPROM location, 1=used EEPROM location BITS 0..6 : 0=empty, 1=undefined data, 2=defined data, 3=program data
VarCounts[4]	byte	4	Array of 4 values each indicating the number of bits, nibbles, bytes and words, respectively, used by user RAM variable declarations. ELEMENT 0=bits, ELEMENT 1=nibbles, ELEMENT 2=bytes, ELEMENT 3=words See "VarCounts" (pg 18) for more information.
PacketCount	byte	1	Indicates the number of tokenized packets to download to BASIC Stamp.
PacketBuffer	byte	2304	Actual packet data to download to the BASIC Stamp. This data begins at element 0 and consists of packets of 18 bytes each. ELEMENT 0..17 : Packet number 1 ELEMENT 18..35 : Packet number 2 … ELEMENT 2286..2303 : Packet number 128 See "Field Format (PacketCount and PacketBuffer)" (pg 18) for more information.

Notes: * Indicates pointer to the type that precedes it.
Bool means Boolean value (True/False).
Byte means 8-bit value.
Char means byte-sized character value (8-bits).
Int means integer value (32-bits).

IMPORTANT:

- It is the calling program's responsibility to create this structure and allocate memory for it.
- It is critical that the structure be organized in the exact order as listed above.
- The tokenizer is designed for 32-bit processors. To aid in alignment compatibility between processors and programming languages, each field of the TModuleRec structure is aligned at the start of a 32-bit boundary (the most common method used). This means, for example, even though the Succeeded field is a Boolean type (1 byte), since the next field, Error, is a character pointer (4 bytes) there are 3 "spacer" bytes that are inserted after the Succeeded field and before the Error field. This alignment continues throughout the

structure wherever the next element would overlap a 32-bit boundary if placed directly after the current field. The table below describes the alignment of the first few fields when viewed at a byte (8-bit) resolution. The double-lines indicate 32-bit boundaries.

Byte Number	Field	Portion
0	Succeeded	Entire field
1	*spacer*	*n/a*
2	*spacer*	*n/a*
3	*spacer*	*n/a*
4	Error	Byte 0 (low byte)
5		Byte 1
6		Byte 2
7		Byte 3 (high byte)
8	DebugFlag	Entire field
9	TargetModule	Entire field
10	*spacer*	*n/a*
11	*spacer*	*n/a*
12	TargetStart	Byte 0 (low byte)
13		Byte 1
14		Byte 2
15		Byte 3 (high byte)
16	ProjectFiles[0]	Byte 0 (low byte)
17		Byte 1
18		Byte 2
19		Byte 3 (high byte)
20	ProjectFiles[1]	Byte 0 (low byte)
21		Byte 1
22		Byte 2
23		Byte 3 (high byte)

- Since field alignment is so critical and with the potential for misalignment so high, a special function has been provided, called "TestRecAlignment" solely to test for proper alignment.

Source Buffer

The Source buffer is a byte array of up to 65536 bytes, where the PBASIC source code should be stored before calling the Compile routine. It is the calling program's responsibility to create this buffer and allocate memory for it. A pointer to the Source buffer must be given when calling the Compile function.

This buffer is modified during compile-time; DO NOT use this area as the only source code buffer for your entire application, but rather as a temporary storage place only for the tokenizer. Because the buffer is modified at compile-time, filling the buffer once and calling the Compile function two times will result in a success the first time and a failure the second. The calling program MUST fill the buffer before each call to Compile.

Source code characters must be in byte-size ASCII format, not Unicode "wide-char" (two-byte per character) format. The characters can be in the ASCII range 0 – 255 and any characters from ASCII 0 to 8 and 10 to 31 will be considered to be end-of-line markers.

The calling program can allocate the Source buffer to be any size, up to a maximum of 65536 bytes. At the moment of compilation, it only needs to be large enough to hold the entire source code plus one extra character.

Do not free (release) the Source buffer memory in-between compilation; some pointers in the TModuleRec structure point to locations within the Source buffer after compilation. A calling program should allocate the Source buffer once, at startup, and free (release) it only once, upon program termination.

Source-to-Token-Reference Structure

The Source-to-Token-Reference structure is an optional array of two words (defined as indicated below) used to retrieve information about which Source characters relate to which EEPROM token bits. This structure is optional since it is intended for use by PBASIC simulators. It is the calling program's responsibility to create this structure and allocate memory for it. If this information is desired, a pointer to the Source-to-Token-Reference buffer must be given when calling the Compile function, otherwise set the `*Ref` parameter to 0 or NULL.

```
struct TSrcTokReference
    {
    /*2 bytes*/   word      SrcStart;
    /*2 bytes*/   word      TokStart;
    };.
```

The array of this structure should be 2338 elements in size.

Upon a successful compile, the TSrcTokReference array will contain a list of SrcStart/TokStart pairs. The SrcStart field contains the index of the first character in the source buffer that relates to the token bit index indicated by the TokStart field. The last character of a token is always the character that just precedes the next SrcStart value (or the end of the source buffer, if no more SrcStart/TokStart values exist). The last token bit of a particular token is always the bit that just precedes the next TokStart value (or the end of the EEPROM data if no more SrcStart/TokStart values exist).

Every row of the TSrcTokReference array contains valid data unless the TokStart value is 0, indicating the end of the list.

For example, assume we're using the following source code on a Windows-based computer:

```
'{$STAMP BS2}
'{$PBASIC 2.5}
DEBUG "Hi"
IF IN0=1 THEN LOW 1 ELSE HIGH 1
```

When copied to the Source byte array in its pure text form, as is required, it would appear as:

Element #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Character	'	{	$	S	T	A	M	P		B	S	2	}	<cr>	<lf>

Element #	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Character	'	{	$	P	B	A	S	I	C		2	.	5	}	<cr>	<lf>

Element #	31	32	33	34	35	36	37	38	39	40	41	42
Character	D	E	B	U	G		“	H	I	“	<cr>	<lf>

Element #	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73
Character	I	F		I	N	0	=	1		T	H	E	N		L	O	W		2		E	L	S	E		H	I	G	H		2

And after compilation, the TSrcTokReference array would contain the following:

Element #	SrcStart	TokStart
0	31	14
1	43	73
2	57	145
3	63	160
4	68	181
5	0	0

Notice that row (element) 0 of the TSrcTokReference indicates that the first token starts at source character 31. The ending character of that token corresponds to the next source start minus 1, ie: 43 - 1 = 42. Characters 31 through 42 correspond to: `DEBUG "Hi"` (followed by a carriage return and line feed).

Row 1 indicates a starting character of 43 (with the ending character being the next start, 57, minus 1, or 56). This corresponds to: `IF IN0=1 THEN` (followed by a space).

The next three rows indicate the tokens for: `LOW 2`, `ELSE` and `HIGH 2`, respectively.

The last row, having a TokStart value of 0 indicates the end of the list.

TestRecAlignment Function

One of the three available function calls, TestRecAlignment, is provided only to aid development with the tokenizer. It will never need to be used after the calling program's TModuleRec structure definition is verified to be correct.

The format of this function is:

```
bool   TestRecAlignment(TModuleRec *Rec)
```

This function simply takes a pointer to the TModuleRec structure (created by the calling program) and fills each field with a specific value. After creating the TModuleRec structure for the first time, call the TestRecAlignment function and read the resulting values from each field in the TModuleRec structure to determine that their structure is properly sized and aligned.

After calling TestRecAlignment and passing a pointer to the TModuleRec structure, a properly sized and aligned structure will contain the values shown in the table below.

Field	Value
Succeeded	False
Error	Null
DebugFlag	True
TargetModule	2
TargetStart	3
ProjectFiles[0]	Pointer to string: “4”
ProjectFiles[1]	Pointer to string: “5”
ProjectFiles[2]	Pointer to string: “6”
ProjectFiles[3]	Pointer to string: “7”
ProjectFiles[4]	Pointer to string: “8”
ProjectFiles[5]	Pointer to string: “9”
ProjectFiles[6]	Pointer to string: “10”
ProjectFilesStart[0]	11
ProjectFilesStart[1]	12
ProjectFilesStart[2]	13
ProjectFilesStart[3]	14
ProjectFilesStart[4]	15
ProjectFilesStart[5]	16
ProjectFilesStart[6]	17
Port	Pointer to string: “18”
PortStart	19
LanguageVersion	20
LanguageStart	21
SourceSize	22
ErrorStart	23
ErrorLength	24
EEPROM	All are 25
EEPROMFlags	All are 26
VarCounts[0]	27
VarCounts[1]	28
VarCounts[2]	29
VarCounts[3]	30
PacketCount	31
PacketBuffer	All are 32, except that last 18 bytes contain the strings pointed to by ProjectFiles[0..6] and Port

If any values are incorrect:

1. Start with the field just before the first incorrect value and scrutinize it’s size (review your compiler’s documentation).
2. Review the TModuleRec details, above. Pay close attention to the section marked “IMPORTANT” and the example indicating space-padding for 32-bit boundaries and when those issues apply.
3. Try filling the defined structure with a number of temporary byte-sized fields, run the program again and pay close attention to the values seen in consecutive fields. Knowing that multi-byte values are stored low-byte to high-byte order, you may be able to gain insight as to how your code (or compiler) is arranging the fields.

Version Function

The Version function returns an integer value indicating the version of the tokenizer library. Note: This version has no direct relationship with the version of PBASIC language supported by the compiler.

The format of this function is:

```
int     Version()
```

This function takes no parameters.

The value returned is in the format: XYY ;where X is the major version number and Y is the minor. For example, if the Version function returned 116, that indicates the version of the tokenizer is 1.16. If Version returned 123, that would indicate the tokenizer is version 1.23.

Compile Function

The Compile function is the one the calling program will use on a regular basis to compile the PBASIC source code.

The format of this function is:

```
bool Compile(TModuleRec *Rec, char *Src, bool DirectivesOnly,
             bool ParseStampDirective, TSrcTokReference *Ref)
```

This function takes five parameters:

Rec	:	A pointer to an existing TModuleRec structure.
Src	:	A pointer to an existing Source buffer.
DirectivesOnly	:	A Boolean value that provides an option of only tokenizing the “compiler directives” from the source code, rather than the entire source. This option is helpful when the calling program needs to determine only the target module, serial port or project files that may be specified by the PBASIC source code. TRUE : Causes the tokenizer to only parse the directives from the source code. FALSE : (Normal mode) Causes the tokenizer to parse the entire source code, including all directives (depending on the ParseStampDirective parameter).
ParseStampDirective	:	A Boolean value that provides an option of parsing the Stamp directive from the source code, rather than accepting a value in the TargetModule field of the TModuleRec structure. TRUE : (Normal mode) Causes the tokenizer to parse the Stamp directive, if present, from the source code. If a valid directive is found, the compiler stores the value of the target module in the TargetModule field and compiles. If no Stamp directive is found, the tokenizer generates an error. FALSE : Causes the tokenizer to ignore any Stamp directives in the source and instead use the value currently in the TargetModule field to compile the code. This means, the TargetModule field MUST be set to the proper value BEFORE calling Compile.
Src	:	An optional pointer to an existing TSrcTokReference buffer. Set to NULL when not used.

Assuming you're using the C programming language and the defined `TModuleRec` structure is called "`ModuleRec`" and the defined source code buffer is called "`Source`," a normal call to `Compile` will look like:

```
if ( Compile(ModuleRec, &Source[0], False, True, NULL) )
  { <statements to run if Compile is successful> }
else
  { <statements to run if Compile fails> }
```

If there are no errors when compiling, the `Compile` function will return `true` and all the fields within the TModuleRec structure (except Error, ErrorStart and ErrorLength) will contain valid information.

If there is an error when compiling, the `Compile` function will return `false` and the TModuleRec structure will be modified with the Succeeded field set `false`, the Error field will not be `NULL (0)` and will point to an error string and the ErrorStart and ErrorLength fields will indicate the offending text of the source code.

GetReservedWords Function

The `GetReservedWords` function is used to retrieve a list of PBASIC reserved words (keywords) that are valid for the current TargetModule and LanguageVersion. An editor with syntax highlighting can use this function to actively update its syntax display.

The format of this function is:

```
bool GetReservedWords(TModuleRec *Rec, char *Src)
```

This function takes two parameters:

Rec : A pointer to an existing TModuleRec structure.
Src : A pointer to an existing Source buffer.

The TargetModule and LanguageVersion fields of the TModuleRec structure must be set prior to calling `GetReservedWords`. While this can be done manually, most practical implementations will call `GetReservedWords` right after a successful call to `Compile`; a successful compile operation will automatically set the TargetModule and LanguageVersion according to the directives parsed from the source code.

The `GetReservedWords` function returns the reserved words as a list of String/Type pairs in the Src buffer, starting at element 0. Each reserved word string is null-terminated and the byte following the null is the Type ID. The next String/Type pair starts on the byte following the previous reserved word's Type ID. The end of the list of reserved words can be determined either by reading TModuleRec's SourceSize field or by looking for a null following the last string's Type ID (ie: a null string).

For example, if the `GetReservedWords` routine returned only two reserved words, called `COMMAND` and `OPERATOR`, with types of 10 and 11, respectively, upon returning the Src buffer would look like the following:

Element #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Character	C	O	M	M	A	N	D	null	<10>	O	P	E	R	A	T	O	R	null	<11>	null	--
ASCII Value (decimal)	67	79	77	77	65	78	68	0	10	79	80	69	82	65	84	79	82	0	11	0	na

Additionally, the `SourceSize` field of the `TModuleRec` structure would be equal to 19. Note that element 7 is the null-terminator for the first string, `COMMAND`, and element 8 contains its type. This is followed immediately by the start of the next string, `OPERATOR`, at element 9, and that string is null-terminated at element 17 and element 18

contains its type. Element 19 would normally be the start of the third reserved word string, however, since there are only two strings total, element 19 is null (a null string) to indicate the end of the list of reserved words.

Assuming you're using the C programming language and the defined TModuleRec structure is called "ModuleRec" and the defined source code buffer is called "Source," a typical use of `GetReservedWords` would be:

```
if ( Compile(ModuleRec, &Source[0], False, True, NULL) )
  { /* Compile was successful */
  if ( GetReservedWords(ModuleRec, &Source[0]) )
    { <statements to run if GetReservedWords succeeds> }
  else
    { <statements to run if GetReservedWords fails> }
  }
else
  { <statements to run if Compile fails> }
```

If there are no errors, the `GetReservedWords` function will return `true`, the Source buffer will contain the list of reserved word String/Type pairs and the SourceSize field of the TModuleRec structure will be set accordingly.

If there is an error, the `GetReservedWords` function will return `false` and the TModuleRec structure's Error field will not be `NULL (0)` and will point to an error string. The typical errors that can occur with GetReservedWords are related to either failing to set, or incorrectly setting, the TModuleRec's TargetModule and LanguageVersion fields.

The following table lists the possible TypeIDs and their meanings:

TypeID	Type	Examples
0	Editor Directive	$STAMP, $PORT
1	Target Module	BS2, BS2E, BS2SX
2	Conditional Compile Directive	#DEFINE, #IF, #SELECT
3	Instruction	OUTPUT, HIGH, LOW
4	Declaration	CON, VAR
5	Pre-Defined Variable	INS, OUTS, DIRS
6	Variable Type	WORD, BYTE, NIB, BIT
7	Variable Modifier	HIGHBYTE, LOWNIB, BIT15
8	I/O Formatter	DEC, HEX, BIN, REP, SKIP
9	Conditional Operator	<, <=, =>, <>, =, AND, OR, NOT
10	Binary Operator	HYP, ATN, MAX
11	Unary Operator	SQR, ABS, ~
12	Constant	99, $FF, $11
13	Period	.
14	Comma	,
15	Question Mark	?
16	Backslash	\
17	At Sign	@
18	Parenthesis	(,)
19	Square Bracket	[,]
20	Curly Brace	{, }

Field Format (Error)

If an error occurs while compiling, the Error field will point to a null-terminated string indicating the error that occurred. The error string is always formatted as "###-message" where *###* is a three digit number indicating the unique Error ID and *message* is the text message associated with the Error ID. If a visible Error ID is not desired, simply crop the first four characters of every error by adding 4 to the Error pointer before displaying the message. See "Error Messages" (pg 20) for a listing of possible errors.

Field Format (TargetStart, ProjectFileStart, PortStart, LanguageStart and ErrorStart)

There are five fields in the TModuleRec structure that reference a character position within the Source buffer; TargetStart, ProjectFileStart, PortStart, LanguageStart and ErrorStart. The value in these fields is the absolute character index of the beginning of the respective string exactly as it appears in the Source field byte array. The Source field treats the source code as if it were one large line of characters (rather than multiple lines of characters) even though end-of-line markers (typically CR/LF pairs or just an LF character) occur at various places within that string.

For example, on a Windows-based computer the end of lines are marked with a CR (13) character followed by a LF (10) character. This means the following source code:

```
'{$STAMP BS2} 'test
DEBUG Hi
OUTS=65535
```

when copied to the Source byte array in its pure text form, as is required, would appear as:

Element #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Character	'	{	$	S	T	A	M	P		B	S	2	}		'	t	e	s	t	<cr>	<lf>
ASCII Value (decimal)	39	123	36	83	84	65	77	80	32	66	83	50	125	32	39	116	101	115	116	13	10

Element #	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
Character	D	E	B	U	G		H	i	<cr>	<lf>	O	U	T	S	=	6	5	5	3	5	
ASCII Value (decimal)	68	69	66	85	71	32	72	106	13	10	79	85	84	83	61	54	53	53	51	53	

Since there's an error in this code (Hi is not a symbol and is not encased in quotes) the Compile function would return `false` and the Succeeded, Error and ErrorStart and ErrorLength fields would be set to the following:

Succeeded : false
Error : pointer to string: "110-Undefined symbol"
ErrorStart : 27
ErrorLength : 2

The ErrorStart field indicates that the offending text starts at character location 27 and the ErrorLength field indicates it contains 2 characters (characters 27 – 28). The calling program (if it were a GUI editor) would find character 27 though 28 and highlight them (see below) and then display the error message, "110-Undefined symbol."

```
'{$STAMP BS2} 'test
DEBUG Hi
OUTS=65535
```

Depending on how the editor treats multi-line text data, it may have to convert the starting location, 27, to a row/column pair (row 1 / column 6, in this case), paying close attention to the number of characters used to indicate end-of-line.

Field Format (ProjectFiles)

Some BASIC Stamp modules can accept multiple programs downloaded to different “program slots.” This set of project files (files related to a particular project) can be specified within the Stamp directive for those BASIC Stamps that support it (refer to the BASIC Stamp Manual for more information). The PBASIC Tokenizer parses these optional path and file names from the Stamp directive and provides that information via the ProjectFiles and ProjectFilesStart fields.

After compiling, each of the seven elements of the ProjectFiles array may point to a null-terminated string indicating the path and file name of the respective project file. A maximum of seven files (in addition to the main file) can be specified and they are always numbered contiguously from the first element (0) toward the last element (6) automatically. For example, there can never be a project file in element 2 without there also being projects files in element 0 and 1. Any unused project file position will cause the respective element of ProjectFiles to be set to NULL (0). Therefore, after compiling, the calling program should read the ProjectFiles array from element 0 to element 6 until it finds a NULL pointer to determine both the total number of additional files to load as well as the path and names of those files.

It is important to note that the tokenizer does not perform any verification of platform-specific file name format or of the existence of the file; the calling program MUST perform those procedures. The tokenizer follows simple rules when parsing these names:

1. Project file names can be quoted or unquoted.
2. If a project file name is unquoted (not encased in quotes) it can contain any of the following 84 characters:
   `!#$%&’()+-./0123456789:;=@ABCDEFGHIJKLMNOP`
   `QRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz~`
   That is, all characters from ASCII 32 to 126 except for the space character and the following nine characters:
   `″*,<>?{|}`
3. If a project file name is quoted (encased in quotes) it can contain any of the 84 characters allowed in unquoted names plus the space character ' ', comma character ' , ' and left and right bracket characters ' { ' and ' } '. That is, all characters from ASCII 32 to 126 except for the following six characters:
   `″*<>?|`

The excluded characters are those that are either have a special meaning to the tokenizer or are the most common special-meaning characters in file systems.

The following is an example of Stamp directives containing valid project file names (according to these rules):

```
'{$STAMP BS2e, Test.bse, led, C:\temp\yo.x, "hello world.bs2", "./" }
```

The tokenizer would parse this directive into four separate project file names and would generate the following values in the ProjectFiles and ProjectFilesStart fields (“start” values assume the above line of code is the very first line in source):

ProjectFiles[0] = pointer to string: Test.bse	ProjectFilesStart[0] = 16
ProjectFiles[1] = pointer to string: led	ProjectFilesStart[1] = 26
ProjectFiles[2] = pointer to string: C:\temp\yo.x	ProjectFilesStart[2] = 31
ProjectFiles[3] = pointer to string: hello world.bs2	ProjectFilesStart[3] = 46
ProjectFiles[4] = pointer to string: ./	ProjectFilesStart[4] = 65
ProjectFiles[5] = null pointer	ProjectFilesStart[5] = 0
ProjectFiles[6] = null pointer	ProjectFilesStart[6] = 0

It's important to note that the strings pointed to by the ProjectFiles array are all null-terminated (C-style) and will not include any enclosing quotes, but will include all space characters that appear before and after the name (if it was enclosed in quotes).

Processing Project Files

The calling program is responsible for verifying platform-specific format of file names and the existence of the referenced files. This section discusses the suggested method of processing the files from the example above.

ProjectFile[0], Test.bse, is a file without a path. The intended meaning is "Test.bse exists in the same folder (directory) as the file that specified this Stamp directive." On a Windows platform, Test.bse would be the same as .\Test.bse. On a Linux platform, Test.bse would be the same as ./Test.bse.

ProjectFile[1], led, is a file without a path or extension. The path issue is exactly like that of Test.bse. The intended meaning of the lack of extension is that "led" is really "led.bse" since the Stamp directive specified "BS2e" as the target module and .bse is the default extension for BS2e source code files.

ProjectFile[2], C:\temp\yo.x, is a file with a fully qualified path and extension. This format is not normally used by BASIC Stamp users, but is still valid; most prefer to keep all related files in the same folder and just use the shorter (no path) name. The thing to note here is that this path would only be valid on a Windows platform and not any other platform. A Linux-based environment, for example, should highlight this entire name (using ProjectFilesStart[2] as the starting character and the length of the ProjectFiles[2] string as the number of characters) and display an appropriate message indicating that the file was not found. Another note is the extension, .x; 1) it is a valid part of the name, 2) should be acceptable to the calling program and 3) does not constitute the need for any error message.

ProjectFile[3], hello world.bs2, is a file without a path. The path issue is exactly like that of Test.bse. The extension, .bs2, doesn't match the default extension of the target module, BS2e, but is still a valid part of the name and should be accepted.

ProjectFile[4], ./, is a valid project file name under the tokenizer's rules, but is not a valid file name (under Linux it is just a relative path indicator). The calling program should highlight this name (using ProjectFilesStart[4] and length of string in ProjectFiles[4]) and display an appropriate message indicating that the file was not found.

Additionally, the ' \ ' character is the folder (directory) separator character in Windows-based path names and ' / ' is the folder (directory) separator character in Linux-based path names. To limit the number of issues that may arise from development by users native to a specific platform, it is suggested that the calling program be aware of these issues and do the appropriate translation automatically, in the background. For example, Windows-based BASIC Stamp environments should automatically translate all ' / ' to ' \ ' before processing the path and file name further while Linux-bases BASIC Stamp environments should translate all ' \ ' to ' / '.

For a project (that contains a Stamp directive specifying other files), it is suggested that each file should be loaded and compiled automatically and each associated packet buffer should be downloaded to the BASIC Stamp specifying the proper program slot as detailed in the BASIC Stamp Programming Protocol document. This is how the BASIC Stamp Windows editor from Parallax handles it.

Field Format (Port)

If a valid Port directive is found while compiling, the Port field of the TModuleRec structure will point to a null-terminated string indicating a serial port name.

Some applications or situations require multiple BASIC Stamps connected to a computer at the same time. If two modules of the same BASIC Stamp type are connected to two serial ports, it can become cumbersome to edit and download the file for one BASIC Stamp then edit and download a different file for another BASIC Stamp and make sure to always download the proper code to the proper BASIC Stamp. For this reason, the Port directive was created.

The Port directive allows a PBASIC program to specify, to the IDE, exactly what serial port to use when downloading it. Since the software was initially Windows/PC-based, the target name of the port is always specified as COM# ;where # is the unique numerical ID of the port.

BASIC Stamp environments developed for other platforms should accept the COM1, COM2, etc. designators to mean the corresponding serial port. For example: in Linux, "COM1" would translate to "/dev/ttyS0", "COM2" would translate to "/dev/ttyS1", etc. Linux developers: see "" (pg) for additional information.

Field Format (EEPROM, EEPROMFlags and VarCounts)

There are five fields that provide information about a PBASIC program's code space (EEPROM) and variable space (RAM) usage. These are the EEPROM, EEPROMFlags, VarBitCount, VarCounts and VarBases fields. This information can be used to create a memory map feature, similar to that of the BASIC Stamp editor's Memory Map screen.

EEPROM and EEPROMFlags

The EEPROM field is an array of 2048 bytes contains the exact data that will be placed in the BASIC Stamp's EEPROM upon successful download. The EEPROMFlags field is an array of 2048 bytes containing status information about the corresponding bytes in the EEPROM field.

Not all the EEPROM field's values are downloaded each time. Some EEPROM locations may be unused and others may just be reserved (so the tokenized program code can not accidentally overwrite them). To display a memory map of EEPROM usage, the calling program must examine the EEPROMFlags for each EEPROM element.

The EEPROMFlags value is a bit pattern consisting of a used/unused flag and a "usage" value:

EEPROMFlags Format (for each element)								
Bit number	7	6	5	4	3	2	1	0
Description	0 = unused 1 = used	reserved	reserved	reserved	reserved	reserved	00 = empty 01 = undefined data 10 = defined data 11 = program data	

The used/unused flag (EEPROMFlags bit 7) indicates whether the corresponding data in the EEPROM field will be downloaded to the BASIC Stamp. An unused region will be left in its present state (in the actual EEPROM chip) upon download. This is to optimize download speed depending on program size.

If an EEPROM location is marked as empty (EEPROMFlags bit 1 and 0 = 00) the location has not been referenced in any way by the program; EEPROM element's value is ignored.

If an EEPROM location is marked as undefined data (EEPROMFlags bit 1 and 0 = 01) the location has been reserved, by a DATA directive, but no data has been placed there; EEPROM element's value can be ignored.

If an EEPROM location is marked as defined data (EEPROMFlags bit 1 and 0 = 10) the location has been used to store a specified value; EEPROM element's value is the "defined data".

If an EEPROM location is marked as defined data (EEPROMFlags bit 1 and 0 = 11) the location has been used to store a PBASIC program's token; EEPROM element's value is the "token".

VarCounts

The VarCounts field is an array of four bytes each indicating the number of bits, nibbles, bytes and words, respectively, used by user RAM variable declarations. VarCounts[0] = number of bit variables, VarCounts[1] = number of nibble variables, etc.

The tokenizer always arranges variables in RAM in the most efficient manner possible: all words first, followed by all bytes, followed by all nibbles and finally all bits. The calling program can simply use the VarCounts field (starting at element 3 and moving towards element 0) to determine how many words, byte, nibbles and bits to "consume" on the RAM map of a memory map display.

Field Format (PacketCount and PacketBuffer)

After a successful compile, the PacketCount field will contain the number of packets (1 to 128) that need to be downloaded to the BASIC Stamp and the PacketBuffer will contain the actual packet data. No additional process of the packet data is required; the packets should be sent to the BASIC Stamp in the exact order they appear.

Each packet is 18 bytes long. The first packet is stored in PacketBuffer[0] through PacketBuffer[17], the second packet at PacketBuffer[18] through PacketBuffer[35], etc.

Refer to the BASIC Stamp Programming Protocol document for details on downloading these packets.

An Example of the Compiling Process

For this example, we'll use the following as the PBASIC source code:

```
'{$STAMP BS2}

Counter VAR   BYTE          'Counter for LED blinks

FOR Counter = 1 to 20       'Loop 20 times
  PULSOUT 0,50000           'Blink LED on and off for 1/10 second
  PAUSE 250                 'Pause 1/4 second
NEXT
STOP
```

Follow these steps to compile the above source code (assuming the TModuleRec structure is called ModuleRec and the source buffer is called Source):

1. Copy the entire source (as pure text; byte-wide characters) to the `Source` byte array. This source is 203 characters long so it will be copied to `Source[0]` through `Source[202]`. Note: Source size includes spaces, tabs and end-of-line markers; CR+LF. The end-of-line markers may vary depending on platform and editor function.
2. Set `ModuleRec.SourceSize` to the actual size of the source, 203.
3. Call the `Compile` function with a pointer to the ModuleRec structure and the Source buffer, DirectivesOnly set equal to False and ParseStampDirective set equal to True

```
Compile(ModuleRec, &Source[0], false, true, NULL);
```

Note: syntax of this line will vary depending on the programming language used and whether ModuleRec and Source are defined as a static object or pointers to memory within the calling program. Additionally, the last parameter may point to a defined TSrcTokReference structure if Source-to-Token reference is desired).

4. Use the Boolean result returned by the `Compile` function (not shown here) or view the `ModuleRec.Succeeded` flag. If `true`, the compile succeeded, if `false`, the compile failed.

Assuming the compile succeeded, the ModuleRec structure will have the following values:

```
Succeeded                : true
Debug                    : false
TargetModule             : 2
TargetStart              : 9
LanguageVersion          : 200
LanguageStart            : 0
VarCounts[0..3]          : 0, 0, 1 and 0, respectively
EEPROM[0..2015]          : all $00
EEPROMFlags[0..2015]     : all $00
EEPROM[2016..2047]       : $00, $00, $00, $00, $00, $00, $00, $00,
                           $00, $18, $14, $20, $8C, $0E, $D8, $C8,
                           $0E, $60, $4A, $AE, $E8, $9F, $49, $C1,
                           $50, $C3, $6F, $8D, $D1, $03, $07, $C0
EEPROMFlags[2016..2047]  : $80, $80, $80, $80, $80, $80, $80, $80,
                           $83, $83, $83, $83, $83, $83, $83, $83,
                           $83, $83, $83, $83, $83, $83, $83, $83,
                           $83, $83, $83, $83, $83, $83, $83, $83
PacketCount              : 2
PacketBuffer[0..35]      : $FE, $00, $00, $00, $00, $00, $00, $00,
                           $00, $00, $18, $14, $20, $8C, $0E, $D8,
                           $C8, $7C, $FF, $0E, $60, $4A, $AE, $E8,
                           $9F, $49, $C1, $50, $C3, $6F, $8D, $D1,
                           $03, $07, $C0, $60
```

Error Messages

The following is a list of possible error messages from the tokenizer.

```
101-Expected character(s)
102-Expected terminating "
103-Unrecognized character
104-Expected hex digit
105-Expected binary digit
106-Symbol exceeds 32 characters
107-Too many elements
108-Constant exceeds 16 digits
109-Constant exceeds 16 bits
110-Undefined symbol
111-Undefined label
112-Expected a constant
113-Cannot divide by 0
114-Location is out of range
115-Location already contains data
116-Expected '?'
117-Label is already defined
118-Expected '\'
119-Expected '('
120-Expected ')'
121-Expected '['
122-Expected ']'
123-Symbol is already defined
124-Data occupies same location as program
125-Array size cannot be 0
126-Out of variable space
127-EEPROM full
128-Symbol table full
129-Expected ':' or end-of-line
130-Expected ',', end-of-line, or ':'
131-Expected 'STEP', end-of-line, or ':'
132-'NEXT' must be preceded by 'FOR'
133-Expected ','
134-Expected ',' or ']'
135-Expected a variable
136-Expected a byte variable
137-Expected a variable modifier
138-Variable is already bit-size
139-Expected a smaller-size variable
    modifier
140-Variable modifier is out-of-range
141-Expected a constant, variable, unary
    operator, or '('
142-Expected a binary operator or ')'
143-Expected a comparison operator or '['
144-Expression is too complex
145-Limit of 255 GOSUBs exceeded
146-Limit of 16 nested FOR-NEXT loops
    exceeded
147-Limit of 6 values exceeded
148-Expected a label
149-Expected a label, variable, or
    instruction
150-Expected '='
151-Expected 'THEN'
152-Expected 'TO'
153-Expected a filename
154-Expected a directive
155-Duplicate directive
156-Unknown target module.  $STAMP
    directive not found
157-Nothing to tokenize
158-Limit of 16 nested IF-THEN statements
    exceeded
159-'ELSE' must be preceded by 'IF' or
    'CASE'
160-'ENDIF' must be preceded by 'IF'
161-Expected a label, variable,
    instruction, or 'ENDIF'
162-Expected a label, variable,
    instruction, or end-of-line
163-Limit of 16 nested DO-LOOP statements
    exceeded
164-'LOOP' must be preceded by 'DO'
165-'WHILE' or 'UNTIL' conditions cannot
    appear after both 'DO' and 'LOOP'
166-Expected 'WHILE', 'UNTIL', end-of-
    line, or ':'
167-'EXIT' only allowed within FOR-NEXT
    and DO-LOOP structures
168-'IF' without 'ENDIF'
169-'FOR' without 'NEXT'
170-'DO' without 'LOOP'
171-Limit of 16 EXIT statements within
    loop structure exceeded
172-Expected variable or 'WORD'
173-Expected a word variable
174-Label is missing ':'
175-Pin number must be 0 to 15
176-Expected a label, variable,
    instruction, or 'NEXT'
177-Expected a label, variable,
    instruction, or 'LOOP'
178-Limit of 16 nested SELECT statements
    exceeded
179-Expected 'CASE'
180-'CASE' must be preceded by 'SELECT'
181-Limit of 16 CASE statements within
    SELECT structure exceeded
182-Expected a label, variable,
    instruction, or 'ENDSELECT'
183-'ENDSELECT' must be preceded by
    'SELECT'
184-'SELECT' without 'ENDSELECT'
185-Expected 'GOTO' or 'GOSUB'
186-Constant cannot be less than 1
187-Invalid PBASIC version number.  Must
    be 2.0 or 2.5
188-Expected number, editor directive,
    #DEFINE'd symbol, or '-'
```

```
189-Illegal operator in conditional-
    compile directive
190-Expected #THEN
191-'#IF' without '#ENDIF'
192-'#SELECT' without '#ENDSELECT'
193-'#ELSE' must be preceded by '#IF'
194-'#ENDIF' must be preceded by '#IF'
195-Illegal symbol in conditional-compile
    directive
196-Expected a user-defined symbol
197-Limit of 16 nested #IF-#THEN
    statements exceeded
198-Expected a character or ASCII value
199-<user defined error message>
200-Expected '#ENDIF'
201-Expected '}'
203-Expected '}'.  Can not specify more
    than 7 additional project files.
204-Expected a target module: BS2, BS2E,
    BS2SX, BS2P or BS2PE
208-Expected COM Port name: COM1, COM2,
    etc
218-Limit of 16 nested #SELECT statements
    exceeded
219-Expected '#CASE'
220-'#CASE' must be preceded by '#SELECT'
221-'#ENDSELECT' must be preceded by
    '#SELECT'
222-Expected a declaration, run-time
    statement, or '#ENDIF'
223-Expected a declaration, run-time
    statement, or '#ENDSELECT'
224-Expected '#ELSE'
225-Expected number, editor directive, or
    #DEFINE'd symbol
226-Expected statements to follow previous
    'CASE'
227-Expected a constant, variable or
    'WORD'
228-'ELSEIF' must be preceded by 'IF'
229-Limit of 16 ELSEIF statements within
    IF structure exceeded
230-'ELSEIF' not allowed after 'ELSE'
```

Linux Serial Port Tips

The following notes are for Linux developers only.

Use of serial port hardware can be tricky, the following tips may be of help:

- Type: `setserial /dev/ttyS0`
    - If the response is: `/dev/ttyS0, UART: unknown, Port: 0x03f8, IRQ: 4` you will not be able to access the port unless you know your UART type and set it using setserial.
- You may have to `chmod` on `/dev/ttyS*` to give your non-superuser login access to the serial port.
- Some computers, such as IBM Thinkpads, are shipped with the serial port disabled. The discussion below is specific to Thinkpads, but may apply to other computers as well:
    - You must "power on" and "enable the serial interface.
    - To "power on" the serial interface (a.k.a. "serial A", a.k.a. "serial port 1"):
        - Boot to DOS and type `PS2 SERA ON`
        - If you are using kernal 2.4.x and the 2.x version of tpctl then you can switch the power off again (and on again) using: `tpctl --rs1=off` and `tpctl --rs1=on`
    - To "enable" the serial interface
        - In DOS type: `PS2 SERA ENABLE`
        - Or, in Linux using the tpctl program, type `tpctl --rs1=enable`

Acknowledgements

Parallax, Inc. would like to thank the following people for their efforts in making the PBASIC Tokenizer Library available to the public:

- The entire Parallax, Inc. staff, especially Stephen Swanson, for various support activities,
- Mark Richardson for his efforts in the Linux port.
- The anonymous Macintosh developer for his efforts in the Macintosh port.

Copyrights and Trademarks

Copyright © 2002-2004 by Parallax, Inc. All rights reserved. PBASIC is a trademark and BASIC Stamp is a registered trademark of Parallax, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Disclaimer of Liability

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products.