



Ser2ftp (ver. 0.1alpha)

A Program for Getting BASIC Stamp and Propeller Output onto the Internet

Disclaimer

This document describes a program that writes data not only to local files but also to those hosted on a remote server. Consequently, the potential for abuse, as well as system disruption and compromise is extremely high. It is incumbent upon the user to verify that this program meets his/her requirements before deploying it on a system that might suffer from inadvertent file writes.

*This program is distributed free of charge in the hope that it will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**.*

Introduction

There are numerous ways to interface Parallax's BASIC Stamp and Propeller microcontroller products to the Internet. Some involve attaching server devices, such as Parallax's PINK module, which allows the micro to serve content directly from an Ethernet port, through the user's NAT router and firewall, to the Internet at large. Though this is an appealing solution in many situations, not everyone can (or wants to) run his own server. Some ISPs prohibit it. But even where it's allowed, not all users are equipped to set up such a system securely or have the necessary bandwidth to serve a wide audience. For this reason, many choose to leave web-serving to the pros and pay to have their sites hosted externally. But this creates a gap between any dynamic content one might wish to serve and the source of that content, residing at the user's location. The program described here, ser2ftp, is designed to bridge that gap.

Ser2ftp.exe runs in the background on a user's PC, to which is attached the user's micro via a serial port (or USB virtual serial port) connection. It can accept commands from the micro to save data either to local files or to a remote computer via the FTP protocol. The kinds of files it can save or transfer include text files (.txt, .htm, .html, etc.), binary files (.bin, .dat), and graphic image files (.gif, .jpg, .png). For creating graphics files, ser2ftp includes a full set of drawing commands, which can be used to produce graphs, charts, and raster graphic images.

Getting Started

To run ser2ftp, it is necessary to create a directory for it to reside in and to copy the files ser2ftp.exe and ser2ftp.ini into that directory. Before running ser2ftp.exe, you will have to edit ser2ftp.ini to conform to your particular needs. Here's an overview of the settings that need to be made:

maxlevel (default 999)

This parameter determines how much information is logged onto the console from things that transpire while ser2ftp is running. All logged items with numbers up to and including **maxlevel** will appear on the console screen. The log levels are as follows:

1. Errors and critical messages.
5. Running commentary of ser2ftp actions.
8. Interpretation of commands received from the attached micro.
10. Echo of every line received from the attached micro.

A typical setting, which logs errors and the running commentary might look like this:

```
maxlevel = 5  #Set to log errors and commentary.
```

Note that the pound sign can be used for inserting comments anywhere in the ini file. The pound sign and everything after it will be ignored by ser2ftp.exe. It can also be used to "comment out" lines that are not needed. Also, any setting commented out or not included in the ini file will either be assigned a default value, if appropriate, or left undefined. Default values are shown in the parameter subheadings in this section.

eol (default LF)

This establishes the end-of-line character used in text files. Lines sent from the micro to ser2ftp must always end with a carriage return (CR). However, this may not be how you want lines in your generated files to end. Files sent to Unix/Linux-based servers, for example, should end with a linefeed (LF); those residing on or sent to Windows-based machines, with a carriage-return/linefeed sequence (CRLF). Valid values for **eol** are: **LF**, **CR**, **CRLF**, and **LFCR**. An example assignment follows:

```
eol = CRLF
```

serialport (default COM1)

This parameter determines which serial communications port the micro is connected to. An example:

```
serialport = COM3
```

baudrate (default 9600)

This parameter defines the communication baudrate with the attached micro. Valid values are 300, 600, 1200, 2500, 4800, 9600, 19200, 38400, 57600, and 115200. Example:

```
baudrate = 19200
```

ftpserver (default 127.0.0.1)

The **ftpserver** parameter establishes the URL of the server to which files are uploaded by ser2ftp. This should be the server's server name or IP address. The default value, 127.0.0.1 is the same as "localhost", the machine upon which ser2ftp is running. Here's an example:

```
ftpserver = ftp.myserver.com
```

ftpuserid (default guest)

This parameter is the username which is used to log into the ftp server. Example:

```
ftpuserid = myuserid
```

ftppasswd (default guest)

This is the password required for ftp login. Bear in mind that **userid** and **passwd** are saved in the ini file in plaintext and sent over the internet unencrypted. *For this reason, it is recommended that you establish a separate ftp account for ser2ftp that allows access to the barest minimum of your server's directory tree necessary for the job at hand.* Here's an example:

```
ftppasswd = mypassword
```

ftpdirdir (default ./)

This is the directory that ser2ftp will switch to when it first logs in. The default value (./) is just the login directory established when setting up the user's ftp account. This value should represent the highest directory in the directory tree to which access will be permitted by ser2ftp. When the connected micro

uploads data, it is not permitted to prepend an absolute path to the filename – only a relative path. Also excluded are relative paths like “../” which traverse up the directory tree. This is done to provide a measure of security against inadvertent writing to an undesired directory. However, absolute paths are allowed for **ftpd**dir, so be careful what you specify! Here’s an example:

```
ftpdire = ./mydirectory
```

In this example, the root directory, to which all ftp saves are relative, will be “mydirectory”, which resides one level under the default login directory.

filedir (default *current working directory*)

This is like **ftpd**ir, except that it pertains to files saved to the local system. The default value is whatever the current working directory is when ser2ftp is started (i.e. the “Start In” directory listed under “Properties” for any Windows shortcut to ser2ftp, or the directory from which ser2ftp is launched in a DOS window). Again, for security reasons, the attached micro cannot specify a file which involves traversing above the **filedir** directory. Here’s an example:

```
filedir = c:\jobs\mysavedfiles
```

watchdog (default 120)

This parameter helps to keep your attached micro alive in the event of power-downs, intermittent disconnects, runaway programs, and other faults. It determines the maximum time, in seconds, which may pass without receiving a line of data from the micro. If this time limit is exceeded, the micro will be reset, via a pulse on DTR, and sent an “Event 0” notification. (See “EV1 to EV9” later in this document.) The **watchdog** setting may be changed by a command from the attached micro. A setting of 0 disables the watchdog timeout. Example:

```
watchdog = 1200 #Time out after 20 minutes.
```

Once the watchdog interval has passed and a connection is determined to be defunct, ser2ftp will attempt to restore it at one-minute intervals until communication can be reestablished. This is particularly important when connection is through a USB adapter since, once disconnected, the port has to be closed and reopened to restore communication.

Once ser2ftp.ini has been edited to your satisfaction, you can run ser2ftp.exe. When started from Windows, it will open a DOS window in which logged events are recorded.

resetdelay (default 500)

This parameter determines how long, *in milliseconds*, to delay after resetting the attached micro before sending the reset event string. For the BASIC Stamp, 500 is a reasonable value. The Propeller will require a longer delay, which will depend on how soon the serial UART cog gets loaded and operational. You will have to experiment to find the best value. Example:

```
resetdelay = 1000 #Delay after reset is one second.
```

Basic Operations

Ser2ftp responds to commands sent to it over the serial port from the attached micro. Commands are of the form,

!CMD[:<param>:<param>:...:<param>]^{C_R}, where

CMD is a three-letter command designator. The parameters, if any, are separated by colons (:), and ^{C_R} is the carriage return character (decimal 13). (Brackets [] are used throughout this document to indicate optional data. Angle brackets <> are used around parameter names. *Neither kind of bracket is sent by the micro to ser2ftp.*) Parameters treated as numbers can be decimal (e.g. 123), binary (e.g. %10110),

or hexadecimal (e.g. \$09FC). Numerical parameters preceded by a minus sign (hyphen) are treated as negative numbers. For 9600-baud communication under PBASIC, issuing these commands can be as simple as sending them with a **DEBUG** statement, *viz*:

```
DEBUG "!ECH:\Y", CR
```

For baud rates other than 9600 baud, **SEROUT** to "pin 16" can be used:

```
SEROUT 16, SerialModeValue, ["!ECH:\Y", CR]
```

Every command must end with a carriage return. Ser2ftp waits for complete lines, beginning with **!** and ending with ^c_R before beginning processing. Throughout this document, **DEBUG** will be used in the examples for brevity.

All lines sent to ser2ftp may contain "escape sequences", i.e. characters which have special meaning or are otherwise cumbersome to transmit from a BASIC or Spin program due to syntax constraints. After the line is read, and before it is processed, the following embedded escape sequences are substituted with their corresponding replacements:

\r	Carriage return.
\n	Newline (linefeed).
\q	Double quote mark (").
\c	Colon (:), processed after colon-separated arguments are split out.
\t	Tab character.
\\	Backslash (\).
\m	Current month number (1 - 12).
\M	Current month abbreviation (Jan – Dec).
\d	Current day number (1 – 31).
\D	Current day-of-year number (1 – 366).
\y	Current year number (00 – 99).
\Y	Current year number (1900 - 2038).
\w	Current weekday abbreviation (Sun – Sat).
\W	Current weekday name (Sunday – Saturday).
\h	Hour number (00 – 23) in 24-hour time.
\H	Hour number (1 – 12) in 12-hour time.
\u	Minutes number (00 – 59). <i>Be careful here! \m is the month!</i>
\s	Seconds number (00 – 59).
\S	Seconds number since the beginning of the epoch (Unix time).
\p	a.m. or p.m., as appropriate.
\P	AM or PM, as appropriate.
\Z	GMT, GDT, LST, or LDT, as appropriate.

The available commands and their meanings are given in the sections that follow.

UTC (Universal Time Capture)

This command takes no arguments and captures the current UTC, or GMT (Greenwich Mean Time), time. The captured time value can be used later to include time information in saved or uploaded files. If no **UTC** or **LTC** commands are received, the data returned by the above escape sequences will be the *local time* at which ser2ftp.exe was started. Example:

```
DEBUG "!UTC", CR
```

LTC (Local Time Capture)

This command takes no arguments and captures the current local standard (or daylight) time. The captured time value can be used later to include time information in saved or uploaded files. If no **UTC** or **LTC** commands are received, the data returned by the above escape sequences will be the *local time* at which ser2ftp.exe was started. Example:

```
DEBUG "!LTC", CR
```

ECH (Echo the arguments back to the micro)

This command will echo everything following it back to the connected micro. This can be used to resynchronize communication, as well as to obtain time and date info from the PC. Here's an example that requests the month, day, and year from the latest requested time capture:

```
DEBUG "!ECH:??\m,\d,\y\r", CR
DEBUGIN WAIT("??"), DEC Month, DEC Day, DEC Year
```

Ser2ftp inserts a 50-millisecond wait between receiving the request and sending the echo to give a slow micro time to get ready for it. Also note that a carriage return is not automatically sent after the echoed data. It has to be inserted into the echoed string, if wanted, as in the above example (\r).

EOL (Set a new end-of-line sequence.)

The end-of-line sequence established by the **eol** parameter in the ini file can be overridden with this command. In the following example, the end-of-line sequence is replaced with *two* carriage returns and a linefeed, effectively double-spacing any text files that follow:

```
DEBUG "!EOL:\r\r\n", CR
```

This capability makes it possible, for example, to write DOS-style text files on the local machine, and Unix-style text files on a remote server. You can also use it to eliminate the **eol** sequence entirely by providing an empty parameter.

WDT (Set watchdog timeout)

This command is used to override any watchdog setting established in the ini file. It takes one argument: the number of idle seconds before resetting the micro. Again, a zero argument disables the watchdog timer. Example:

```
DEBUG "!WDT:0", CR
```

A watchdog timeout triggers an "EV0" event. Such an event resets the micro via a pulse on DTR. Then, after a 0.5-second delay, transmits the string "EV0" to the micro.

EV1 to EV9 (Schedule an event)

In addition to watchdog timeouts, ser2ftp is also capable of managing a single event schedule, by which operations in the micro can be synchronized. When an event numbered between EV0 and EV4 occurs (EV0 being the watchdog timeout, described above), the micro is first reset, then, after a 0.5-second delay, the event string "EV0", "EV1", ..., or "EV4" is sent. Higher-numbered events do not reset the micro: they just send the event string "EV5", "EV6", ..., or "EV9". When an event is scheduled, it completely replaces any other event schedule currently in place (except for the watchdog timeout).

The event schedule is specified in this command's single argument and follows a very strict syntax. (For Linux users, this is the same syntax used by **crontab**.) The argument consists of five or six "fields", separated by spaces. These correspond to minute, hour, day of month, month, day of week, and (optionally) seconds. The following specification for this argument string is quoted from the documentation for Perl's **Schedule::Cron** module, which is used as the basis of ser2ftp's event system:

Field	Values
minute	0-59
hour	0-23
day of month	1-31
month	1-12 (or as names)
day of week	0-7 (0 or 7 is Sunday, or as names)
seconds	0-59 (optional)

A field may be an asterisk (*), which always stands for "first-last". Ranges of numbers are allowed. Ranges are two numbers separated with a hyphen. The specified range is inclusive. For example, **8-11** for an "hour" entry specifies execution at hours 8, 9, 10 and 11.

Lists are allowed. A list is a set of numbers (or ranges) separated by commas. Examples: **1,2,5,9** and **0-4,8-12**.

Step values can be used in conjunction with ranges. Following a range with **"/<number>"** specifies skips of the number's value through the range. For example, **0-23/2** can be used in the "hour" field to specify command execution every other hour [and is equivalent to] **0,2,4,6,8,10,12,14,16,18,20,22**. Steps are also permitted after an asterisk, so if you want to say "every two hours", just use ***/2**.

Names can also be used for the "month" and "day of week" fields. Use the first three letters of the particular day or month (case doesn't matter).

Note: The day of a command's execution can be specified by two fields: day of month, and day of week. If both fields are restricted (i.e., aren't *), the command will be run when *either* field matches the current time. For example, **30 4 1,15 * Fri** would cause a command to be run at 4:30 a.m. on the 1st and 15th of each month, plus every Friday.

Examples:

8 0 * * *	8 minutes after midnight, every day.
5 11 * * Sat,Sun	at 11:05 on each Saturday and Sunday.
* /5 * * * *	every five minutes.
42 12 3 Feb Sat	at 12:42 on 3rd of February and on each Saturday in February.
32 11 * * * 0-30/2	11:32:00, 11:32:02, ... 11:32:30 every day.

An optional sixth column can be used to specify the seconds within the minute. If not present, it is implicitly set to **0**.

The following example illustrates how an event might be scheduled in PBASIC.

```
' {$STAMP BS2pe}
' {$PBASIC 2.5}

Event VAR Byte

LOW 4                                'Used by MoBoStamp to disable serial echo.

DEBUGIN WAIT("EV"), DECL Event
IF (Event = 0) THEN
    DEBUG "!WDT:60", CR              'Set the watchdog timer for 1 minute.
    DEBUG "!EV1:*/15 * * * **", CR  'Schedule event 1 for every 15-minutes.
ELSE
    'Do the 15-minute interval task.
```

```
ENDIF
DO
    DEBUG "!NOP", CR                'Send a NOP for keep-alive.
    SLEEP 30                        'Sleep for 30 seconds.
LOOP
```

This will be a typical pattern for implementing events that reset the BASIC Stamp. The "LOW 4" will disable the serial echo if you're using the MoBoStamp Board. This can be helpful if **EV5** through **EV9** are being employed, because they might inadvertently occur while the BASIC Stamp is sending data. With the serial echo enabled, this data could get garbled. With BASIC Stamp boards other than the MoBoStamp, you just have to be careful not to transmit data when data from ser2ftp is being anticipated.

NOP (No Operation)

Just what it says: it doesn't do anything – except keep the watchdog timer happy, and that's the only reason to use it.

FIL and FTP (Save data to a local file or transfer to a server.)

The syntax for these two commands is identical:

!FIL:<filename>[:<option>:...:<option>]:<end-sequence> ^{c_R}

!FTP:<filename>[:<option>:...:<option>]:<end-sequence> ^{c_R}

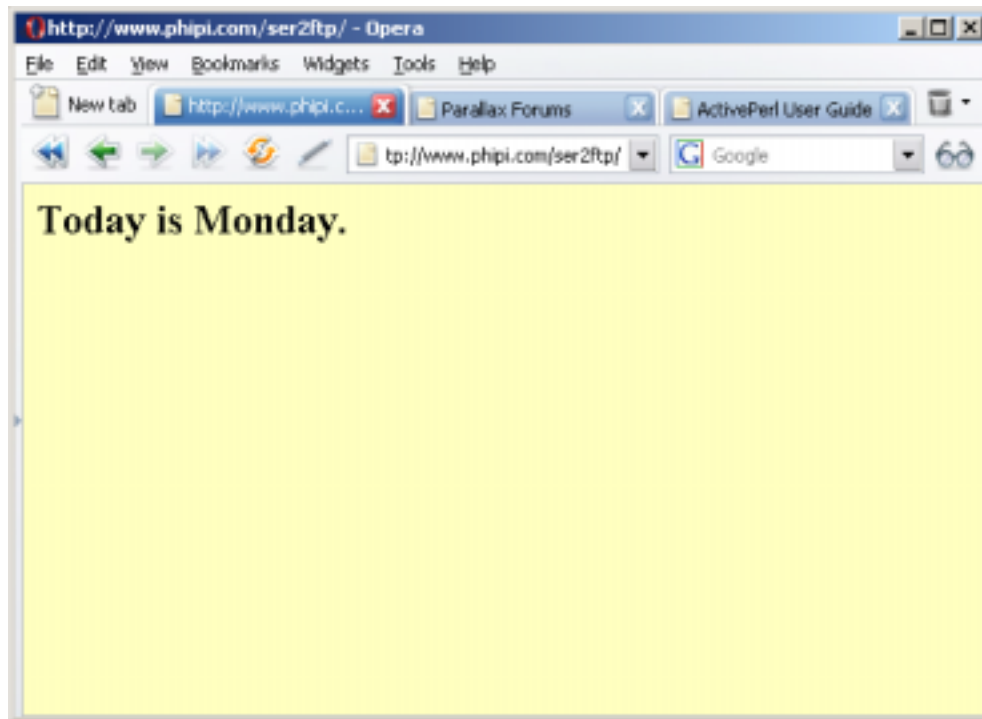
The <filename> contains the relative path and filename info for the file being saved. The type of the file is determined by the three-letter extension in the file name. Special treatment is given to binary files (extensions **.bin** and **.dat**) and graphic image files (extensions **.gif**, **.jpg**, **.png**). All other extensions are assumed to refer to text files. The <option> parameters are used by image files and are described below. The <end-sequence> can be any sequence of printable characters and determines what you will use as an end-of-file. Anything sent to ser2ftp after one of these two commands will get added to the file being saved until the <end-sequence> is received *by itself on a single line*, terminated with a carriage return.

Text Files

Text files are the simplest to generate and are always line-oriented. Each line sent should end with a carriage return. Each line written to the file, however, is terminated with the character specified by **eol** in the ini file, or as overridden by the **EOL** command. Here's an example:

```
DEBUG "!LTC", CR                'Capture local time and date.
DEBUG "!FTP:index.html:end", CR  'FTP file to index.html.
DEBUG "<html><body bgcolor=\q#FFFFFF\q><h2>"
DEBUG "Today is \W.", CR        'Print full name of weekday.
DEBUG "</h2></body></html>", CR  'End of html stuff.
DEBUG "end", CR                 'End of file, as defined in FTP command.
```

In a browser window, the resulting file will look like this:



Binary Files

Binary files are sent as sequences of hexadecimal digit pairs, two digits for each byte. Only lines consisting of nothing but an even number of hex digits are saved. All other lines are ignored. Neither the required carriage returns nor the **eol** characters are saved to the file – only the data. Here's an example:

```
DEBUG "!FIL:mydata.bin:EOF", CR      'Save mydata.bin on local PC.
FOR I = 1 to 16                      'Write 16 bytes to file.
  DEBUG HEX2 I                       'Write one byte of data.
NEXT
DEBUG CR, "EOF", CR                  'Terminate the line, then the file.
```

This will send one "line" of data to ser2ftp: **0102030405060708090A0B0C0D0E0F^C**. Ser2ftp interprets each pair of hex digits as one byte, so it creates a binary file 16 bytes long.

Image Files

And image file can be either a GIF, JPG, or PNG bitmap. The type is specified by the three-letter extension in the filename. Also specified must be the size of the image and the color space being used. The parameters are the "options" referred to above. The image size is given by two options, **X** followed by a number, and **Y** followed by a number: e.g. **X640:Y480**. The color space is given by one of the following:

- BW** Black and white. Each pixel consists of one bit: 0 = black, 1 = white.
- G16** 16 gray levels. Each pixel consists of four bits, ranging from 0 (black) to 15 (white).
- G256** 256 gray levels. Each pixel consists of eight bits, ranging from 0 (black) to 255 (white).
- C16** The 16 DOS VGA colors:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----
- RGB** Any of the 16 million 24-bit colors.

All color spaces but **RGB** are paletted and can be used with any graphics file. The **RGB** color space can be used only with JPG and PNG files.

A typical beginning to a graphics file might look like this:

```
DEBUG "!FTP:images/wind.gif:X200:Y200:C16:end", CR
```

This file will be a GIF image, 200 x 200 pixels, with 16 colors, and will be written to the "images" subdirectory on the server.

Images are created by executing drawing commands, similar to the HPGL commands used by pen plotters. Each command consists of two letters, possibly followed by a list of arguments. The syntax is fairly loose and is designed to be as compact or as readable as the user wishes. (For BASIC Stamp programs, compact is probably best, since long strings chew up program memory pretty fast.) One line can contain any number of drawing command, terminated with a carriage return:

$\langle \text{cmd} \rangle [\langle \text{data} \rangle, \dots, \langle \text{data} \rangle] [;] \langle \text{cmd} \rangle [\langle \text{data} \rangle, \dots, \langle \text{data} \rangle] \dots^C_R$

*Note that drawing commands do **not** begin with an exclamation point (!), but that lines must still end with a carriage return to be processed. A typical line might look like this:*

```
DEBUG "SP2MV100,100LI200,200", CR
```

This selects color 2, then moves to position (100,100) and draws a line from there to (200,200).

Effects produced by drawing commands are not remembered between image files. When a new image file is begun, all settings revert to their default values:

Current X:	0
Current Y:	0
Pen:	0 for paletted color spaces, black for RGB.
Background:	0 for paletted color spaces, black for RGB.
Palette:	Default for the color space chosen.

Image coordinates are always given as an (X,Y) pair of numbers and originate with (0,0) in the *upper* lefthand corner, increasing to the right and *down*. Coordinates may either be absolute or relative to the last point plotted or moved to. An absolute coordinate is always a positive number, so is indicated without a leading sign. A relative coordinate is preceded with either a plus (+) or minus (-) sign, as in the following example:

```
DEBUG "MV100,100LI-100,+0", CR
```

This moves to point (100,100), then draws a horizontal line from there to (0, 100).

Numerical arguments immediately preceding a subsequent command don't need any punctuation after them *unless they're given in hexadecimal*. In this case, a semicolon (;) should be inserted to disambiguate in case the next command starts with one of the letters "A" through "F". Here's an example:

```
DEBUG "SP$FF0088;CI100", CR
```

All the drawing commands are described in the sections to follow. To test the example code snippets yourself, embed them in the following PBASIC code template (changing the **\$STAMP** directive as needed):

```

' {$STAMP BS2pe}
' {$PBASIC 2.5}

Pen    VAR Byte
Type   VAR Byte
Size   VAR Byte
Dir    VAR Byte
x      VAR Byte
y      VAR Byte

LOW 4

DEBUGIN WAIT("EV0")
DEBUG "!WDT:0", CR
DEBUG "!FIL:example.gif:X250:Y150:C16:end", CR
DEBUG "BG15ER249,149", CR

'=====Example code goes between these lines.=====
'=====

DEBUG "end", CR
END

```

In the example images, unless specified otherwise, the image size is 250x150, the palette is **C16**, and the background color has been changed to white. Also a black border has been drawn around the image to delineate its boundaries.

Pen Color

PC<pen number>,<rgb color>

Just because you've selected a color palette doesn't mean you can't change the colors. The Pen Color command allows you to do just that and should be used before any actual drawing takes place. It takes two arguments: the pen number and the color assigned to that pen. The color is a 24-bit RGB value and is almost always given in hexadecimal: two digits for Red, followed by two digits for Green, then two digits for Blue. For example, \$FF00FF would yield magenta. The pen number has to lie within the limits for the palette you've chosen. For example, here's a command that changes pen 5 to yellow:

```
DEBUG "PC5,$FFFF00", CR
```

The Pen Color command is for paletted color spaces only, so should not be used with the **RGB** color space.

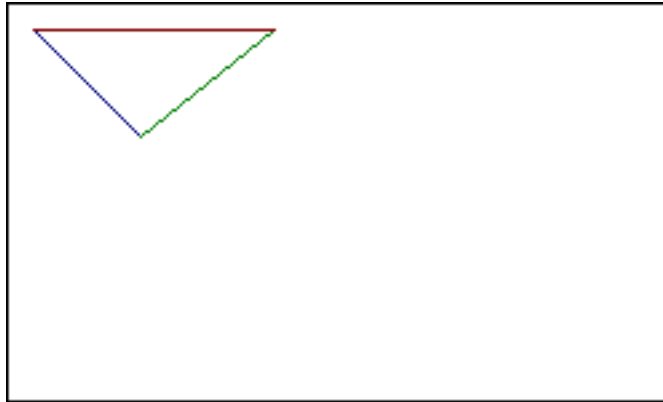
Select Pen

SP<pen number> *or* **SP**<rgb color>

This command selects the pen to be used for all subsequent drawing commands, until another Select Pen is encountered. It takes one argument: the pen number for paletted color spaces, or an RGB value (e.g. \$808080) for the **RGB** color space. (See the explanation of RGB colors above.) Here's an example that draws a triangle in three colors:

```
DEBUG "SP1MV10,10LI50,50SP2LI100,10SP4LI10,10", CR
```

Here's what the image looks like:



Background Color

BG<pen number> or **BG**<rgb color>

When a graphics file is first created, it is filled with a background corresponding to color zero – usually black. You can change this easily with the Image Background command, which takes one argument: either a pen number or an RGB color value, depending on the color space. Here's an example that changes the background color to white when using the **RGB** color space:

```
DEBUG "BG$FFFFFF", CR
```

The image background should be applied only once before doing any other drawing. Otherwise you'll erase anything else you've already drawn.

Line Thickness

LT<thickness>

This command sets the thickness of any lines drawn with the current pen. It takes one argument: the thickness, in pixels, of any subsequent lines drawn. Extremely heavy lines drawn with a large value for Line Thickness will likely not look the way you expect them to, due to the primitive way they're capped at the ends. But for thicknesses of 2 or 3, it's a convenient way to add emphasis to outlines.

Move To

MV<x>,<y>

This command moves the "pen" to another location without drawing anything. It takes two arguments: the new X and Y location. Examples of the MoVe command can be seen above.

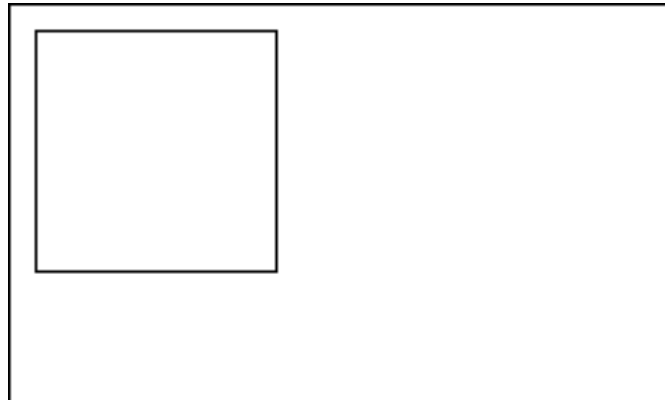
Line To

LI<x1>,<y1>,<x2>,<y2>,...,<xn>,<yn>

This command draws a straight line from the current position to the position given by its X and Y arguments, using the current pen. You can also chain arguments using the Line command, as shown in the example below:

```
DEBUG "MV10,10LI10,100,100,100,100,10,10,10", CR
```

This will draw a square, 90 units on a side, with diagonals at (10,10) and (100,100). Here's what it looks like:



If the Line command is given without any arguments, it will draw a single pixel at the current position.

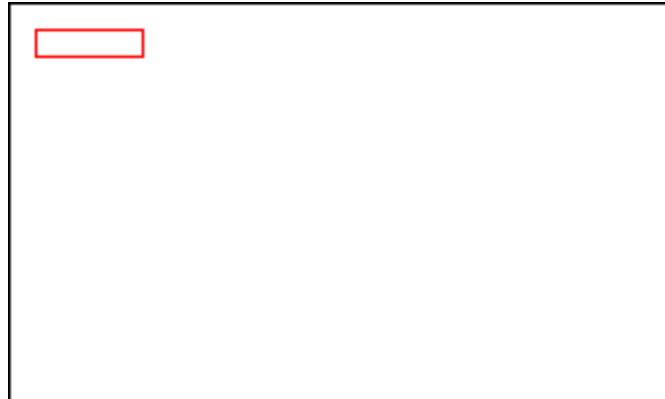
Edge Rectangle

ER<x>,<y>

This is the command to use for drawing a box. It takes two arguments: the X and Y values of the point diagonally opposite the current point. Here's an example:

```
DEBUG "SP12MV10,10ER50,20", CR
```

This will draw a red rectangle, 40 pixels wide and 10 pixels high, with its upper left corner at position (10, 10). After **ER** is executed, the current pen position remains unchanged. Here's what it looks like:



Fill Rectangle

FR<x>,<y>

This is the same as Edge Rectangle except that, instead of drawing the outline, it fills the box with the current pen color. Here's the same example as above, but filling the rectangle:

```
DEBUG "SP12MV10,10FR50,20", CR
```

Edge Circle

EC<radius>

This command will draw the outline of a circle with center at the current pen position and radius in pixels given by its single argument. See the next command for an example.

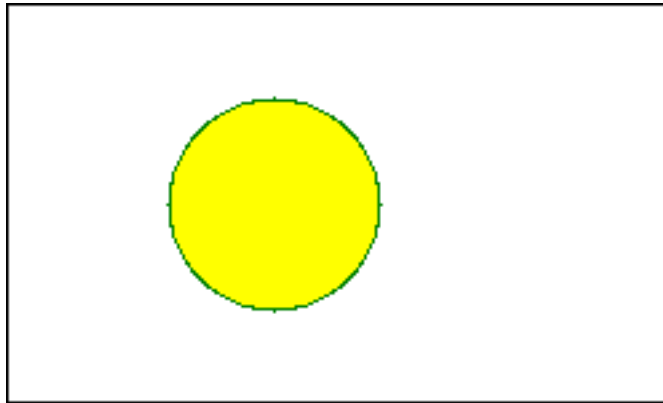
Fill Circle

FC<radius>

This is the same as Edge Circle, except that it fills the circle using the current pen. In the following example a circle with radius 40 is drawn with a center at location (100, 75). It is first filled with pen 14 then outlined with pen 2:

```
DEBUG "MV100,75SP14FC40SP2EC40", CR
```

Here's what it looks like:



Edge Wedge

EW<radius>,<start angle>,<arc length>

A wedge is like a slice of pie, and – not surprisingly – wedges are used to draw pie charts. Wedges begin and end at the pointy end, i.e. the center of the arc. The Edge Wedge command takes three arguments: the radius, the beginning angle, and the angular extent of the wedge. In wedge coordinates, East is angle 0; North, 90; West 180; and South 270. See the next section for an example.

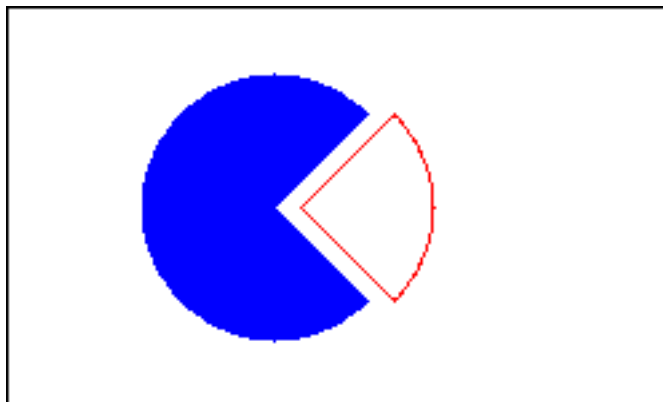
Fill Wedge

FW<radius>,<start angle>,<arc length>

This is like the Edge Wedge command, except that the wedge is filled with the current color. In the following example, two wedges are produced to form a pie chart. One is filled; the other, outlined and pulled out from the pie by 10 pixels:

```
DEBUG "MV100,75SP9FW50,45,270MV+10,+0SP12EW50,-45,90", CR
```

Here's what it looks like:



Tick Type

TT<type>

A tick mark is a small shape placed in an image without actually having to draw it. It can be used to draw short line segments on graph axes to label the coordinates or to indicate the positions and types of points in scatter plots. The Tick Type command takes one numerical parameter, which selects a tick mark from among the following types:

- 0 Horizontal line
- 1 Vertical line
- 2 Outlined square
- 3 Filled square
- 4 Outlined circle
- 5 Filled circle

Tick Size

TS<radius>

This command selects the size of subsequent tick marks placed into the image. It takes a single parameter, the "radius" in pixels of the tick mark. For circles, "radius" means half the diameter (as usual); for lines, half the line's length; for squares, half the length of one side. If the tick size is zero, a single point will be plotted.

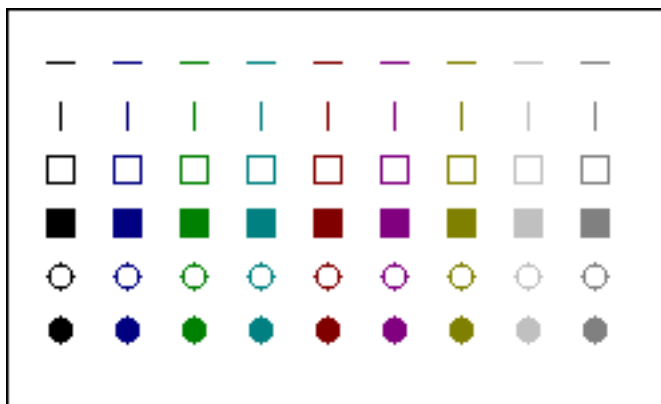
Tick

TI<x1>,<y1>,<x2>,<y2>,...,<xn>,<yn>

This command draws the selected tick (**TT**) at the selected size (**TS**) using the current pen, at locations determined by a list of coordinate pairs following the command. (It works like the **LI** command in this regard.) If no coordinates are given, a tick is plotted at the current point. The tick mark is always centered at each selected location, and each location becomes the current location after the tick mark is drawn. The following example draws several types of tick marks of size 5 in different colors.

```
DEBUG "TS5", CR
FOR Pen = 0 TO 8
  DEBUG "SP", DEC Pen, "MV", DEC Pen * 25 + 20, ",0", CR
  FOR Type = 0 TO 5
    DEBUG "TT", DEC Type, "TI+0,+20", CR
  NEXT
NEXT
```

Here's what the resulting image looks like:



Draw Pixels

PX\$<hex pixel digits> or **PX%**<binary pixel digits>

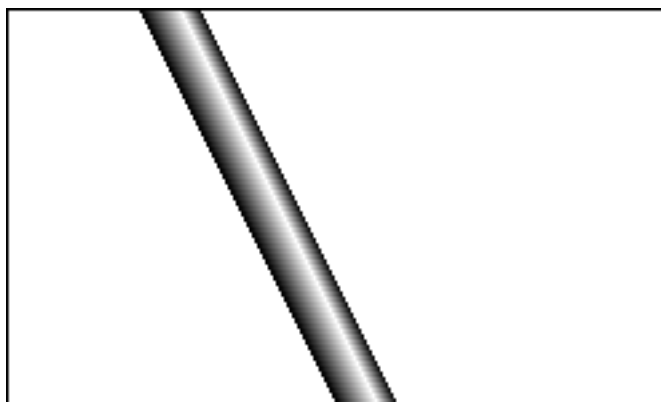
This command allows mapping a row of pixels to the image, starting at the current point, and moving to the right one pixel at a time. The current point also moves right with each pixel plotted. This command takes one long argument, which is a sequence of hex or binary digits (preceded by either **\$** or **%**) representing either the pen number for paletted images, or the RGB color for **RGB** images. The following table shows which sequences are permitted for each color space:

Color Space	Digits Allowed	Digits per Pixel	Example
BW	0 - 1	1 (0 to 1)	%111011011110
G16, C16	0 - F	1 (0 to F)	\$073F5
G256	0 - F	2 (00 to FF)	\$073F54
RGB	0 - F	6 (000000 to FFFFFFFF)	\$FF000080C0CC

In the following example, the file is defined using the **G16** color space, instead of the **C16** space used in the other examples. The program draws a shaded, diagonally-running "wire".

```
FOR y = 1 TO 148
  DEBUG "MV", DEC 100 + y / 2, ",", DEC y, "PX$0123456789ABCDEFDB97531", CR
NEXT
```

Here's what it looks like:



Label Direction

LD0 or LD90

Ser2ftp can place horizontal and vertical text labels into the image. The Label Direction takes one argument, specifying the direction of subsequent labels. The valid values are 0 (horizontal) and 90 (vertical, i.e. rotated 90 degrees counter-clockwise).

Label Size

LS<size>

Ser2ftp's labels can be any of five monospaced font sizes, ranging from 0 (tiny) to 4 (extra large), with sizes 2 and 4 being boldface fonts. The Label Size command takes a single argument: the size of the font. See the example in the next section for samples of the various fonts. The font metrics are summarized in the following table:

Font Size	Font Weight	Character Size (W x H)
0	Normal	5 x 8
1	Normal	6 x 12
2	Bold	7 x 13
3	Normal	8 x 16
4	Bold	9 x 15

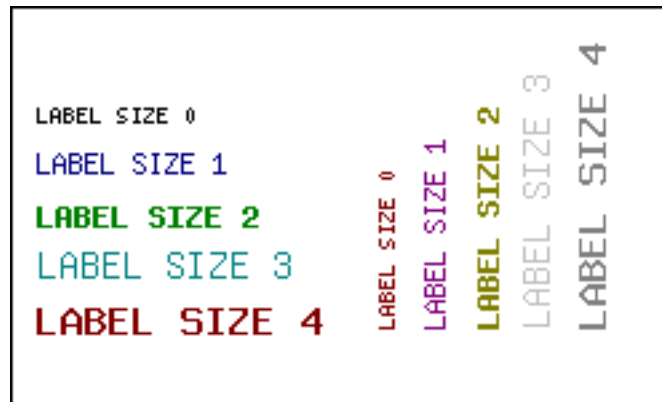
Label

LB<text>^C_R

This is the command that draws the text labels. Because a label can consist of any printable characters, the Label command will "gobble up" the entire rest of the command line as its single parameter. Therefore, **LB** has to be the last command on any given line. To draw multi-line labels, embed a newline (**\n**) in the text to move to the next line. When the **LB** command has finished executing, the current position is located after the last character plotted (or at the beginning of the next line if the last character was a newline). The following example prints labels in various sizes and orientations:

```
FOR Size = 0 TO 4
  DEBUG "LS", DEC Size, CR
  FOR Dir = 0 TO 1
    IF (Dir) THEN
      DEBUG "MV", DEC 145 + (Size * 20), ",120"
    ELSE
      DEBUG "MV10,", DEC 45 + (Size * 20)
    ENDIF
    DEBUG "SP", DEC Dir * 4 + Size, CR
    DEBUG "LD", DEC Dir * 90, "LBLabel Size ", DEC Size, CR
  NEXT
NEXT
```


Here's what the output looks like:



You will note that all lower-case letters have been converted to upper-case. Ser2ftp only does upper-case labels.

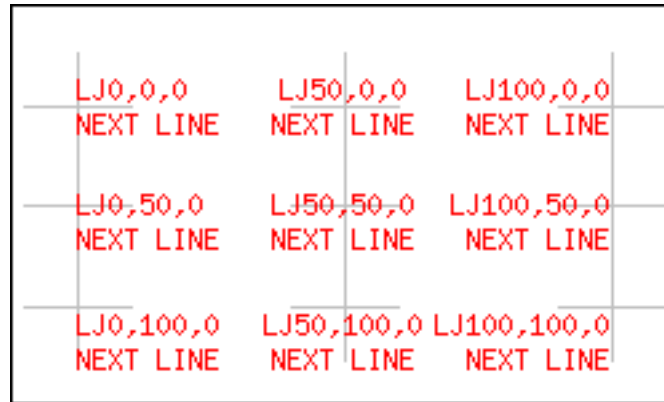
Label Justify

LJ<x percent>,<y percent>,<leading percent>

Unless otherwise specified, a label begins with its lower-left corner at the current point, thereby left- and bottom-justifying it. Also, the spacing between lines of multi-line labels will be the natural spacing, without any additional "leading" (a printer's term, referring to the lead slugs placed between lines of type to spread them out vertically). But the justification and leading can be changed with the Label Justify command. It takes three parameters: horizontal justification, vertical justification, and leading. Each is a percentage. Horizontal justification tells where in a line of characters the starting point lies. At 0, it begins at the far left (left-justified); 50 in the middle (centered); and 100, at the right (right-justified). Vertical justification determines the same thing on the vertical axis: 0 is bottom-aligned, 50 is middle aligned, and 100 is top aligned. A leading value of 0 means normal spacing; 100, double spacing; 200, triple spacing; and so on. In the following example, the starting point of each label is shown with a gray cross-hatch:

```
DEBUG "TS20LS1", CR
FOR x = 0 TO 100 STEP 50
  FOR y = 0 TO 100 STEP 50
    DEBUG "MV", DEC x * 2 + 25, ",", DEC y * 3 / 4 + 38, CR
    DEBUG "SP7TT0TI+0,+0TT1TI+0,+0SP12"
    DEBUG "LJ", DEC x, ",", DEC y, ",0", CR
    DEBUG "LBLJ", DEC x, ",", DEC y, ",0\nNext Line", CR
  NEXT
NEXT
```

And here's the output:



Scale Image

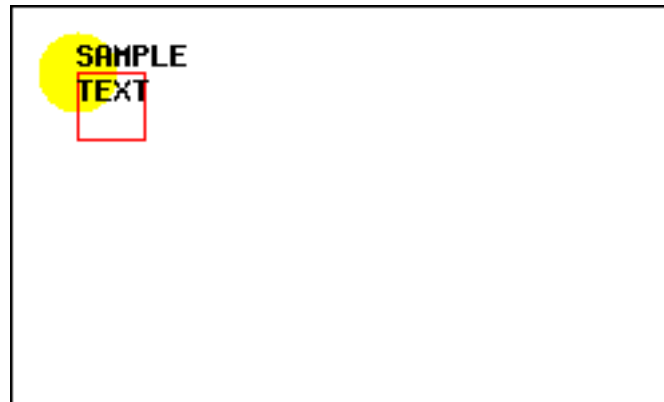
SC<old extent>,<new extent>

The Scale Image command magnifies the contents of the image by the scale factor given by the ratio of its two arguments: <new extent> / <old extent>. If this ratio is less than 1, a value of 1 is used. (i.e. you can't shrink the contents of an image with this command.) The overall dimensions of the image don't change when you scale it – only the size of the graphics within it. So, for an image size of 250 x 150 and a scale ratio of 2.5 (e.g. **SC100,250**), only graphic elements drawn in the upper-left rectangle sized 100 x 60 will be shown. And these will be blown up to fill the entire 250 x 150 image area. The actual zooming is done just before the image file is written or transferred, but **SC** can be executed anytime during the drawing of the file. Only the last ratio issued is used for the actual zoom.

In the following program, a few things are drawn in the upper left corner of the image:

```
DEBUG "SP14MV25,25FC15SP12ER50,50SP0LBSample\nText", CR
```

Here's the image produced:



Now, here's the same program, but this time using the **SC** command:

```
DEBUG "SC100,250SP14MV25,25FC15SP12ER50,50SP0LBSample\nText", CR
```

And here's the blown-up result:



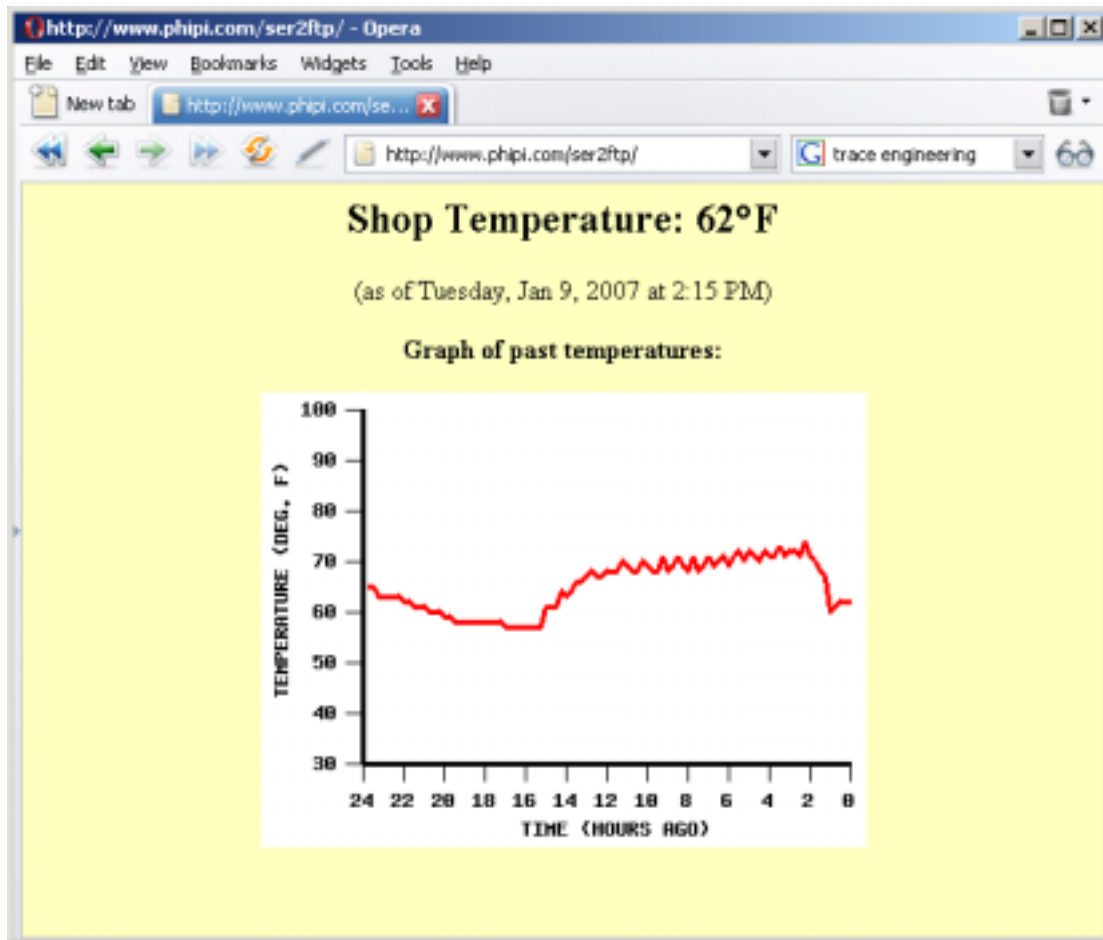
Note that the outline drawn in the template code got truncated on the bottom and right-hand sides. Also note that everything gets magnified: text, line thicknesses, even individual pixels.

An Actual Application

This section attempts to pull all of the foregoing together into an actual application that can be viewed on the web (www.phipi.com/ser2ftp). This site consists of two pages: **index.html**, and **images/temps.gif**. Both are produced by a MoBoStamp-pe (http://www.parallax.com/detail.asp?product_id=28300) to which is interfaced a DS1620 temperature sensor (http://www.parallax.com/detail.asp?product_id=604-00002), and which is connected to a PC running ser2ftp. The main page, index.html, displays the latest date, time, and temperature. It also includes an link to images/temps.gif. That file shows a graph of temperatures. A typical source for index.html would look like this (with extra newlines and tabs thrown in for readability):

```
<html>
  <body bgcolor="#ffffc0">
    <center><h2>Shop Temperature: 62&#176;F</h2>
    (as of Tuesday, Jan 9, 2007 at 2:15 PM)
    <h4>Graph of past temperatures:</h4>
    
  </body>
</html>
```

And this would display a page that looks like this in a web browser:



The PBASIC code that produces all this is as follows:

```
' {$STAMP BS2pe}
' {$PBASIC 2.5}

' Program to read a temperature from the DS1620 temperature sensor,
' write it to a circular buffer in EEPROM, then draw a graph from
' the recorded data for upload to the internet using the
' PC program ser2ftp.exe.

' The DS1620 interface portion of this program was lifted wholesale
' from the Parallax DS1620 appnote "AppKit DS1620 Digital Thermometer"
' available from the Parallax website.

' ===== Define Pins and Variables =====

DQ      CON 8          ' Pin 8 <=> DQ.
CLK     CON 9          ' Pin 9 => CLK.
RST     CON 12         ' Pin 12 => RST (high = active).

DSdata  VAR Word       ' Word variable to hold 9-bit data.
```

```

Sign    VAR DSdata.BIT8    ' Sign bit of raw temperature data.
T_sign  VAR Bit            ' Saved sign bit for converted temperature.
TempPtr VAR Byte          ' Pointer to next temperature in EEPROM
Temp    VAR Byte
i        VAR Word

' ===== Define DS1620 Constants =====

' >>> Constants for configuring the DS1620

Rconfig CON $AC            ' Protocol for 'Read Configuration.'
Wconfig CON $0C            ' Protocol for 'Write Configuration.'
CPU      CON %10           ' Config bit: serial thermometer mode.
NoCPU    CON %00           ' Config bit: standalone thermostat mode.
OneShot  CON %01           ' Config bit: one conversion per start request.
Cont     CON %00 ' Config bit: continuous conversions after start.

' >>> Constants for serial thermometer applications.

StartC   CON $EE           ' Protocol for 'Start Conversion.'
StopC    CON $22           ' Protocol for 'Stop Conversion.'
Rtemp    CON $AA           ' Protocol for 'Read Temperature.'

' ===== Begin Program =====

LOW 4                                ' Turn off serial echo.
DEBUGIN WAIT("EV"), DECL i          ' Get the event number.
IF (i = 0) THEN                      ' Event 0 (reset)?
    DEBUG "!WDT:60", CR              ' Yes: Set watchdog timer to 60 seconds.
    DEBUG "!EV1:*/15 * * * *", CR    ' Schedule event 1.
    GOTO KeepAlive
ENDIF

DEBUG "!LTC", CR                    ' Event 1 (read temperature): Capture local time.
LOW RST                             ' Deactivate '1620 for now.
HIGH CLK                             ' Put clock in starting state.
PAUSE 100                           ' Let things settle down a moment.
HIGH RST                             ' Activate the '1620 and set it for continuous..
SHIFTOUT DQ,CLK,LSBFIRST,[Wconfig,CPU+Cont] ' ..temp conversions.
LOW RST                             ' Done--deactivate.
PAUSE 50                             ' Wait for the EEPROM to self-program.
HIGH RST                             ' Now activate it again.
SHIFTOUT DQ,CLK,LSBFIRST,[StartC]    ' Send start-conversion protocol.
LOW RST                             ' Done--deactivate.
PAUSE 500
HIGH RST                             ' Activate the '1620.
SHIFTOUT DQ,CLK,LSBFIRST,[Rtemp]     ' Request to read temperature.
SHIFTIN DQ,CLK,LSBPRES,[DSdata\9]    ' Get the temperature reading.
LOW RST

```

```

T_sign = Sign                                ' Save the sign bit of the reading.
DSdata = (DSdata */ $e6)                     ' Multiply by 0.9.
IF T_sign THEN DSdata = DSdata | $FF00
DSdata = DSdata + 32                         ' Complete the conversion.
READ 0, TempPtr                             ' Get the pointer to the next available EEPROM slot.
WRITE TempPtr, DSdata.LOWBYTE                ' Write the temp to EEPROM.
TempPtr = TempPtr // 96 + 1                  ' Increment the pointer, wrapping around to slot #1.
WRITE 0, TempPtr                             ' Write the new pointer value.

' >>> Transfer data to the internet, beginning with temperature graph:

DEBUG "!FTP:images/temps.gif:C16:X358:Y270:end", CR      'Begin capture of images/temps.gif.
                                                    'It's 358 x 270 pixels, with a 16-color palette.
DEBUG "BG15SP0LJ50,100,0LD90MV5,110,LBTemperature (Deg. F)", CR      'Label the Y-axis.
DEBUG "LJ50,0,0LD0,MV210,265LBTime (Hours Ago)", CR      'Label the X-Axis.
DEBUG "LJ100,50,0TT0TS5"                                'Put tick marks and values on Y-axis.
FOR i = 30 TO 100 STEP 10
    DEBUG "TI55,", DEC 310 - (i * 3), "MV-10,+0LB", DEC i, CR
NEXT
DEBUG "LJ50,100,0TT1"                                'Put tick marks and values on X-axis.
FOR i = 0 TO 24 STEP 2
    DEBUG "TI", DEC 348 - (12 * i), ",,225MV+0,+10LB", DEC i, CR
NEXT
DEBUG "SP12LT3", CR
FOR i = 0 TO 95                                         'Plot up to 96 temperature values.
    TempPtr = TempPtr - 1
    IF (TempPtr = 0) THEN TempPtr = 96
    READ TempPtr,Temp                                  'Read next previous temperature.
    IF (Temp = 255) THEN EXIT                            '255 = undefined, so we're done.
    Temp = 310 - (Temp * 3)                             'Compute Y-axis position.
    IF (i = 0) THEN
        DEBUG "MV348,", DEC Temp, "LI"                  'If first point, need to move there.
    ENDIF
    DEBUG DEC 348 - (i * 3), ",", DEC Temp, ",",          'Draw a line from prev point to current one.
NEXT
DEBUG CR, "SP0MV60,10,LI+0,+210,+288,+0", CR, "end", CR      'Draw the axes, then upload the file.

' >>> Continue with updated html file that shows current temp and date/time.

DEBUG "!FTP:index.html:end", CR                        'Begin capture of index.html.
DEBUG "<html><body bgcolor=\q#ffffc0\q><center><h2>Shop Temperature: ", DEC Dsdata
DEBUG "&#176;F</h2>"                                '\q means insert "
DEBUG "(as of \W, \M \d, \Y at \H:\u \P)", CR '\? means, "Insert last-read date and time values."
DEBUG "<h4>Graph of past temperatures:</h4><img src=\qimages/temps.gif\q>", CR
                                                    'Include the graph.
DEBUG "</body></html>", CR, "end", CR                  'Upload the updated file.

KeepAlive:

```

```

DO
    DEBUG "!NOP", CR                                'Keep the watchdog timer happy.
    SLEEP 30                                          'Sleep for 30 seconds.
LOOP                                                'Lather, rinse, repeat...

InitEEPROM:
    WRITE 0, 1
    FOR TempPtr = 1 TO 96
        WRITE TempPtr, 255
    NEXT
RETURN

```

We'll go through the ser2ftp-specific stuff step-by-step to see how it works. First comes the reset code. This is what gets executed when the program starts up or is interrupted by ser2ftp:

```

LOW 4                                          ' Turn off serial echo.

```

This command is specific to the MoBoStamp-pe and turns off the serial echo.

```

DEBUGIN WAIT("EV"), DECL i                    ' Get the event number.

```

After ser2ftp resets the Stamp, it waits 0.5 seconds (as programmed in the ini file by the **resetdelay** setting), then sends a character string starting with **EV**, followed by a single digit, **0** through **4**. The digits are event numbers. Event **0** is a watchdog or startup reset; events **1** through **4** can be scheduled from the Stamp itself. The **DEBUGIN** statement just waits for the event message, then records the event number in the variable **i**.

```

IF (i = 0) THEN                                ' Event 0 (reset)?
    DEBUG "!WDT:60", CR                        ' Yes: Set watchdog timer to 60 seconds.
    DEBUG "!EV1:*/15 * * * *", CR              ' Schedule event 1.
    GOTO KeepAlive
ENDIF

```

If the event is **0**, we can assume we're starting from scratch, so we need to set up a few things:

1. First, we need to set the watchdog timer so if we get disconnected, ser2ftp will start us up again. We'll set it to 60 seconds.
2. Next, we want to schedule an event that will be used to read the temperature and upload the updated files to the web server. We'll use event **1** for this, and schedule it every 15 minutes, on the 15-minute mark.
3. Finally, we'll jump to a keep-alive section of code that just sends **NOPs** to ser2ftp to keep the watchdog timer from expiring and triggering another event **0**.

```

DEBUG "!LTC", CR                                ' Event 1 (read temperature): Capture local time.

```

If the event is not **0**, we can assume it's our scheduled event **1**. The first thing to do is record the local time with the **LTC** command. Then we can read the temperature. The portion of code that does this is documented elsewhere by Parallax, so we'll skip over that part, and assume we've obtained a successful reading in the variable **Dsdata**.

```

DEBUG "!FTP:images/temps.gif:C16:X358:Y270:end", CR 'Begin capture of images/temps.gif.

```

Once the time and temperature have been acquired, we can build and send the two files. The first is the image file, named **temps.gif** in the **images** subdirectory. This image has a size of 358 x 270 pixels and uses the **C16** (DOS) color space. The file will be sent by ser2ftp after we send it a line containing the single string, **end**, as specified in the **FTP** command.

```
DEBUG "BG15SP0LJ50,100,0LD90MV5,110,LBTemperature (Deg. F)", CR 'Label the Y-axis.
```

The first order of business is to label the X- and Y-axes, Y-axis first. Here are the commands and their arguments used to do that:

1. **BG15** – This changes the background color from **0** (black) to **15** (white).
2. **SP0** – Select pen **0** (black)
3. **LJ50,100,0** – Set X justification to **50** (centered), Y justification to **100** (top), and leading to **0** (no leading).
4. **LD90** – Set the label direction to **90** degrees (vertical).
5. **MV5,110** – Move the pen to location (**5, 110**). This is where the center-top (as specified by **LJ**) of the label will be.
6. **LBTemperature (Deg. F)** – Draw the label.

```
DEBUG "LJ50,0,0LD0,MV210,265LBTime (Hours Ago)", CR 'Label the X-Axis.
```

1. **LJ50,0,0** – Set X justification to **50** (centered), Y justification to **0** (bottom), and leading to **0** (no leading).
2. **LD0** – Set the label direction to **0** degrees (horizontal).
7. **MV210,265** – Move the pen to location (**210, 265**). This is where the center-bottom (as specified by **LJ**) of the label will be.
3. **LBTime (Hours Ago)** – Draw the label.

```
DEBUG "LJ100,50,0TT0TS5" 'Put tick marks and values on Y-axis.
```

1. **LJ100,50,0** – Set X justification to **100** (right justify), Y justification to **50** (centered), and leading to **0**.
2. **TT0** – Set the tick type to **0** (horizontal line).
3. **TS5** – Set the tick size to **5**.

```
FOR i = 30 TO 100 STEP 10
  DEBUG "TI55,", DEC 310 - (i * 3), "MV-10,+0LB", DEC i, CR
NEXT
```

We're going to put marks and labels on the Y-axis every 10 degrees Fahrenheit from 30 to 100. The variable **i** will hold that number.

1. **TI55, 310 - (i * 3)** – Put a horizontal tick at location (**55, 310 - (i * 3)**). The Y-axis will lie along the X = 60 line. The tick mark is size 5, which means its center needs to be 5 pixels to the left of that for the ticks to end on the axis. The vertical tick spacing is 30 pixels. That's the reason for the **i * 3**, since **i** increases in increments of 10.
2. **MV-10+0** – Move 10 pixels to the left of the current position. (The + and – signs indicate relative movement.)
3. **LB i** – Print the number **i** as a label, right justified, as specified by the **LJ** instruction.

```
DEBUG "LJ50,100,0TT1" 'Put tick marks and values on X-axis.
```

1. **LJ50,100,0** – Set X justification to **50** (center), Y justification to **100** (top), and leading to **0**.
2. **TT1** – Set the tick type to **1** (vertical line).

```
FOR i = 0 TO 24 STEP 2
  DEBUG "TI", DEC 348 - (12 * i), ",225MV+0,+10LB", DEC i, CR
```


NEXT

We're going to put marks and labels on the X-axis every 2 hours from 0 on the right to 24 on the left. The variable **i** will hold that number.

1. **TI 348 - (12 * i),225** – Put a vertical tick at location **(348 - (12 * i), 225)**. The X-axis will lie along the **Y = 220** line. The tick mark is size 5, which means its center needs to be 5 pixels lower than that for the ticks to top out on the axis. The horizontal tick spacing is 24 pixels. That's the reason for the **12 * i**, since **i** increases in increments of 2.
2. **MV+0+10** – Move 10 pixels down from the current position. (The **+** and **-** signs indicate relative movement.)
3. **LB i** – Print the number **i** as a label, centered, as specified by the **LJ** instruction.

DEBUG "SP12LT3", CR

1. **SP12** – Select pen 12 (red in the **C16** colorspace).
2. **LT3** – Set the line thickness to 3 pixels.

```
FOR i = 0 TO 95                                'Plot up to 96 temperature values.
  TempPtr = TempPtr - 1
  IF (TempPtr = 0) THEN TempPtr = 96
  READ TempPtr,Temp                            'Read next previous temperature.
  IF (Temp = 255) THEN EXIT                     '255 = undefined, so we're done.
```

Variable **i** is just a counter here. The variable **TempPtr** points to the next available position in EEPROM after the latest temperature reading. We're going to read back through these readings in reverse chronological order. So we subtract **1** from **TempPtr** each time through the loop. When **TempPtr = 0**, we've gone too far, so cycle back to position 96.

Then we read the temperature value at that point. A value of 255 means that no value was recorded there, so we're done.

```
Temp = 310 - (Temp * 3)                        'Compute Y-axis position.
IF (i = 0) THEN
  DEBUG "MV348,", DEC Temp, "LI"                'If first point, need to move there.
ENDIF
DEBUG DEC 348 - (i * 3), ",", DEC Temp, ",",    'Draw a line from prev point to current one.
NEXT
```

Next we compute the Y-axis position in the image that corresponds to the temperature we just read and assign it back to **Temp**. Then, if this is the first point plotted, we have to move to that point. That's what the **MV348, Temp** does. Then we can issue the **LI** command and provide its arguments in each iteration through the loop. That's what the last **DEBUG** statement does: feed pairs of numbers as parameters to the already-issued **LI**.

```
DEBUG CR, "SP0MV60,10,LI+0,+210,+288,+0", CR, "end", CR 'Draw the axes, then upload the file.
```

After the last coordinate pair is sent, we issue a carriage return to terminate the line so the **MV** instruction and **LI** instruction with all those coordinate pairs can execute. We haven't drawn the X- and Y-axes yet, so we'll do that last:

1. **SP0** – Select pen **0** (black).
2. **MV60,10** – Move to location **(60, 10)**. This is the top of the Y-axis.
3. **LI+0,+210,+288,+0** – Draw a 210-pixel line down to the origin (Y-axis), then continue with another line 288 pixels to the right (X-axis).

After the terminating carriage return, we send the **end** established in the FTP command as our end-of-file, followed by another carriage return. At this point, ser2ftp will send the image file to the server.

```
DEBUG "!FTP:index.html:end", CR                                     'Begin capture of index.html.
```

Next, we need to update **index.html** with the current date, time, and temperature data. We'll use the same end-of-file string as we used before.

```
DEBUG "<html><body bgcolor=\q#ffffc0\q><center><h2>Shop Temperature: ", DEC Dsdata
DEBUG "&#176;F</h2>"                                             '\q means insert "
```

Now comes the actual HTML stuff. Notice that where quote marks are required in the HTML tags, we use the escape sequence **\q**. This is easier than interrupting a quoted string, *e.g.*: "**string1**", **34**, "**string2**", which would be the only other way to do it in PBASIC. Also note that the temperature is inserted directly into the HTML file using the variable **Dsdata**. And finally, the HTML literal, **°**, is the way to produce a degree symbol.

```
DEBUG "(as of \W, \M \d, \Y at \H:\u \P)", CR '\? means, "Insert last-read date and time values."
```

Here's where the time and date that we acquired just before reading the temperature comes into play. Remember all those fancy escape sequences that were described pages ago? Here's how they get used. Ser2ftp replaces each of them with their corresponding time and date strings before inserting into the file.

```
DEBUG "<h4>Graph of past temperatures:</h4><img src=\qimages/temps.gif\q>",CR 'Include the graph.
```

This is the line that displays the graph with the HTML file.

```
DEBUG "</body></html>", CR, "end", CR                               'Upload the updated file.
```

And, finally, we close out the HTML tags and close out the file with the end-of-file string.

```
KeepAlive:
DO
  DEBUG "!NOP", CR                                             'Keep the watchdog timer happy.
  SLEEP 30                                                    'Sleep for 30 seconds.
LOOP
```

After everything is done, all we have to do is burp once in awhile to let ser2ftp know we're still alive. This loop is adequate to keep the watchdog timer from interrupting things. Plus, with a timeout of only 60 seconds, if we get disconnected, ser2ftp will know about quickly.

```
InitEEPROM:
  WRITE 0, 1
  FOR TempPtr = 1 TO 96
    WRITE TempPtr, 255
  NEXT
RETURN
```

This section of code isn't reachable by the program as it stands. It is used to initialize the EEPROM to all undefined readings. It can be reached with a **GOSUB** and should be called once, before running the program with ser2ftp.