VERY PRELIMINARY DOCUMENTATION for Version 2.4 Beta

NOTE: When this version is finally released into production it will be renamed to Version 3.0 signifying a major revision change over the existing 2.3.X

PLEASE SEND ALL FEATURE REQUESTS / BUG REPORTS / ETC to beta@reachtech.com

OVERVIEW OF CHANGES IN NO PARTICULAR ORDER

1. Macro improvements

A macro can now contain button handlers as well as standard commands.
This allows macros to be self-contained.

Example:

```
#define Example 2 /*
// the :xincl directive tells the macro interpreter that it
should look for button / hotspot directives
// not just start executing the macro from the top when a button
is pushed
:xincl
// the following gets executed when the macro is invoked by the
"m" command
// clear screen and define test buttons
z
bd 3 100 100 1 "ON" 5 5
bd 4 100 200 1 "OFF" 5 5
// tell the buttons to run this macro when pushed
xm 3 2
xm 4 2
// end of section
:quit
// define button handler for button #3
:x3
t "ON pushed   " 150 100
:quit
// define button handler for button #4
:x4
t "OFF pushed" 150 100
:quit
#end */
```

2. Button text

A button's text can now be longer than 20 characters total.

In Build 32, text space is allocated when a button is defined, but is not released when a button is redefined. Therefore when drawing new screens, one of the "z" commands should be used to reset all buttons and clear the button character storage area.

Example: (text is bigger than the button graphic, but this illustrates the concept)

```
>bd 1 10 10 1 "This is a very long text that\npreviously could
not be used" 5 5
```

3. Top level variables

Variables are now provided at the top level.
There are two types: signed 32 bit integers and double precision floating point.
These can be assigned, read, and used in commands with a syntax similar to macro variable substitution.
Integer variable names must start with an 'i' and floats with an 'f'. Variable names are at most 32 characters long.

Examples:

➔ declare an integer variable:

```
>int iMyValue
```

➔ set the variable to 100:

```
>= iMyValue 100
```
or
```
>=iMyValue 100
```

➔ read the value: (value is followed by a <return>

```
>=myValue
100
>
```

➔ use the variable in a command:

```
>t "The value of iMyValue is \"`iMyValue`\"" 10 10
```

➔ use the variable with specific formatting. Note that the format specifier is per the 'C' convention for the printf() function:

```
>t "The value of iMyValue with leading zeros\nand fixed 3 digits
is `iMyValue%03`" 10 10
```

➔ use the variable as a numeric argument:

```
>bv `iMyValue`
```

➔ floating point variable declaration and use. Note that the decimal point is required for floating point values:

```
>float fMyValue
>=fMyValue 123.456
>out "fMyValue is `fMyValue%7.2`"
fMyValue is  123.45
>
```

➔ declare a string variable, 40 characters long:

```
>string 40 sMyString
```

➔ set the variable:

```
>. "hello world" sMyString strcpy
```

➔ read it:

```
>= sMyString
hello world>
```

➔ display variable on screen

```
>z
>t "string = `sMyString`" 10 10
```

NOTE: setting strings don't work right using the '=' at the top level.

4. New "clear" commands.

Clear screen but instead of blanking, display a bitmap (here we use bitmap #2)

```
>zs 2
```

Do everything that the "z" command does, but leave screen "as-is". Buttons are cleared, levelbars, charts, sliders are initialized

```
>zs
```

5. New BMPload program an file format

A new BMPload program and file format is required with 2.4. The reason is that transparent bitmaps are now supported. This works by adding an extra line at the top of the bitmap with a special sequence as follows. The extra line is not displayed when the bitmap is drawn.

"Transparent" marker line, width N pixels (top right corner pixel is N)
Pixel 1 (top left corner) = index value that will not be drawn = transparency color
Pixel 2 (to the immediate right of the top left) = index value 1
Pixel 3 = index value 2
Pixel 4 = index value 3
.... don't care ...
Pixel N-3 = index value 252
Pixel N-2 = index value 253
Pixel N-1 = index value 254
Pixel N = don't care

Example of a transparent bitmap, assuming the system palette is txprnt_arrow.bmp

Note that custom palettes are supported as well with transparency.

6. New command - hotspot that reports touch position

```
xxy <index> x0 y0 x1 y1

returns x<index> <x> <y>
```

Example (makes entire screen a hotspot)

```
>xxy 128 0 0 319 239
(touch screen in the middle)
x128 164 109
```

7. Window Save / Window Restore up to 64x64 areas

Save:
```
ws x0 y0 x1 y1
```

Restore:
```
wr x y
```

Example:

```
        .
>// display first bitmap
>xi 1 0 0
>// save a 20x20 part of it
>ws 30 30 49 49
>// set the rectangle white
>r 30 30 49 49 1 FFF
>// restore
>wr 30 30
```

8. New polygon commands

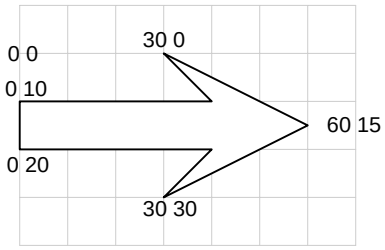
Note: all the polygon commands use the current pen width and foreground color for drawing. They also use the draw mode, normal or XOR. However due to the way lines are drawn, the XOR is not very useful for drawing as doing it twice leaves some pixel residue. If the polygon is to be erased, what is recommended instead is to draw the polygon on top of itself in the background color.

8.1 Draw polygon

`pg <x origin> <y origin> <x0> <y0> <x1> <y1> [ . . . <x9> <y9>`
This draws a polygon using the current pen width and the foreground color. The command takes a variable number of <x> <y> points, up to a maximum of 10 points total. The points are relative to the specified x and y origin.

Example; draw black arrow pointing to the right at origin 50, 100:

```
>z
>s 0 1
>pg 50 100 0 10 40 10 30 0 60 15 30 30 40 20 0 20
```



The same arrow at origin 150, 100:

```
>pg 150 100 0 10 40 10 30 0 60 15 30 30 40 20 0 20
```

Erase the above arrow by swapping foreground and background color:

```
>s 1 0
>pg 150 100 0 10 40 10 30 0 60 15 30 30 40 20 0 20
```

8.2 Draw polygon rotated

```
pgr <angle in degrees> <x origin> <y origin> <x0> <y0> <x1>
<y1> [ . . . <x9> <y9>
```

This is the same as “draw polygon” except that it is rotated counter colckwise by the positive degrees specified. The rotation point is the first x, y point

Example – draw the arrow pointing up:

```
>pgr 90 150 100 0 10 40 10 30 0 60 15 30 30 40 20 0 20
```

NOTE: due to rounding errors, the polygon is not correctly rendered at 90 degrees – this will be fixed.

8.3 Draw filled polygon

```
pf <x origin> <y origin> <x0> <y0> <x1> <y1> [ . . . <x9> <y9>
```

8.4 Draw polyline (connected points)

```
pl <x origin> <y origin> <x0> <y0> <x1> <y1> [ . . . <x9> <y9>
```

8.5 Draw polyline rotated

```
plr <angle in degrees> <x origin> <y origin> <x0> <y0> <x1>
<y1> [ . . . <x9> <y9>
```

9. Text Alignment

The “ta” command allows text to be left, center, or right aligned and top, center or bottom aligned with respect to the specified origin. This is useful with variable sized proportional fonts.

Single letter arguments:	L = left	T = Top
	C = Center	C = Center
	R = Right	B = Bottom

```
ta <L|C|R><T|C|B>
```

Example: left top align text to the top right corner of the screen:

```
>ta RT
>t "Top right" 319 0
```

This puts the phrase "Top Right" in the top right corner of the screen.

```
>ta CC
>t "Middle of Screen" 160 120
```

This puts the phrase in the middle of the screen.

If called with NO ARGUMENTS, the current alignment is returned:

```
>ta<return>
:LT
>
```

10a. Protocol-friendly prompts

The protocol prompts are now more protocol state-machine friendly. The general communication protocol is as follows.

1. After a command has been executed, there may be one or more information lines followed by either a success or failure prompt. Information lines are either computer or human readable. The number of retun lines varies with the command.

CHANGE THIS !!!
-> command responses (computer friendly) should be preceeded by an equals sign '='

-> information responses (human friendly) preceeded by a colon ':'
Host can throw away ALL responses that start with a colon and end with a return

10b. Custom Palette

Custom palettes are supported. To use this feature, make sure all bitmaps have the same palette. Then set the "Custom Palette" option in the BMPLoad program. The new palette will be loaded after BMPload has finished downloading. The screen may change color or go blank depending on the palette. Clear the screen to a know color after downloading a new custom palette for the first time.

If a custom palette is loaded, the 'S' command no longer works since it cannot map RGB colors to an unknown palette. Similarly the 's' command no longer sets specific colors. Instead the 's' command syntax becomes:

```
s <foreground index> <background index>
```

where the index is the palette table index from 0 to 255. So, if index 0 is black and 255 is white, the command:

```
s 0 255
```

is equivalent to the "`s 0 1`" command when using the system palette.

11. Run arbitrary command on button press

The "xdo" command allows an arbitrary command to be executed when a button is pressed. This is an alternative to invoking a macro and is useful in conjunction with the interpreter (described in a separate document). The xdoq (quiet) option does not send button press notification on the serial port.

```
xdo "... valid SLCD command ..."

or

xdoq "... valid SLCD command ..."
```

A somewhat trivial example is as follows (buttons use internal bitmaps). When button 1 is pressed, the text ON is displayed, and when button 2 is pressed, the text "OFF" is displayed.

```
>bd 1 50 100 1 "ON" 15 15
>bd 2 150 100 1 "OFF" 12 15
>xdo 1 "t \"ON \" 100 50"
>xdo 2 "t \"OFF\" 100 50"
```

Note: all xdo commands are cleared by the "z" command.

12. Timed display tasks

The "dtask" command allows an arbitrary command to be executed at a timed interval. This does not happen "in the background" while other commands are executing, rather when each time interval has passed, the command is queued and runs when the currently executing command (macro, serial line) completes. This command is useful in conjunction with the interpreter (described in a separate document).

```
dtask <interval in milliseconds> "... valid SLCD command ..."
```

Example:

```
>int iVal       // declare an integer variable
>z              // clear screen
>// display value of iVal every 0.25 sec.
>dtask 250 "t \"iVal = `iVal`  \" 10 10"
>=iVal 2
// look at the screen
```

This causes the SLCD to continuously display the value of the iVal variable on the screen. Note that the dtask execution interval is not precise – other commands may delay the task. The dtask command is meant to handle display tasks, not real-time processing functions.

To list the currently running dtasks, use the command without arguments:

```
>dtask
:dtask list
:   250 "t "iVal = `iVal`  " 10 10"
>
```

To delete all dtasks, use the "dtask clear" command:

```
>dtask clear
>
```

Note: all dtask commands are also cleared by the "z" command.

ADD: DOUBLE ESCAPE KILLS DTASKS...

13. Protocol change – binary commands

The 2.3 version supported a single "binary" command – run macro. This has been removed. We plan to implement a universal escape mechanism for binary arguments to commands; that is, commands have the same format as present, except that any argument can be provided in binary form by preceding it with an appropriate escape sequence.

INTRODUCTION TO THE EMBEDDED INTERPRETER

1. Basic concepts

A stack-based interpreter has been added to this version. It uses "Reverse Polish Notation" or RPN to avoid compilation and make the code very compact and efficient. The trick is to remember that all operations are done on the stack. This means that instead of writing:

1 + 1

You write

1 1 +

This is interpreted as: push literal 1, push literal 1, add 2 top items on the stack, and replace them with the result. All operators such as +, -, /, etc. operate on the stack.

The interpreter has a very simple tokenizer, so all syntactic elements must have spaces before and after them. For example, "end-of-line" comments start with the double slash // and must be followed by a space. Also, the line:

1 1+ will look for a primitive "1+" and not see that the number and operator are separate.

2. Interactive mode

The interpreter can be run in interactive mode so the user can develop and test code interactively. This mode is entered with the "pgm" command, and a special prompt "->" appears. Use "x" to exit:

```
>pgm
-> x
>
```

The following is a quick and dirty tutorial using the interactive mode:

```
>pgm
-> 1       // push the number 1 on the stack
-> .s      // display contents of stack
Stack: 1
-> clear   // clear the stack
-> .s      // display contents of stack; leave unchanged
Stack: Empty.
-> 1       // push 1 on the stack
-> .       // display the top of stack and discard it
1
-> .s
Stack: Empty.
-> // clear stack, push 1, push 2, add them, display the stack
-> clear 1 2 + .s
Stack: 3
->
-> // --------- other arithmetic operators ---------------
-> // for clarity we clear the stack before each operation
-> //
-> clear 4 1 - .s
Stack: 3
->
-> clear 4 2 * .s
Stack: 8
-> clear 12 3 / .s
Stack: 4
-> clear 12 10 mod .s
Stack: 2
-> clear 10 5 max .s
Stack: 10
-> clear 10 5 min .s
Stack: 5
-> clear 10 negate .s
Stack: -10
-> clear -2 abs .s
Stack: 2
-> clear
-> // calculate 10*2+3, display and drop from stack
-> 10 2 * 3 + .
23
-> .s
Stack: Empty.
->
-> // -------------- logic operators -----------
-> hex     // hex output mode – no "0x" in front
-> 0x33 0x11 and .s
Stack: 11
-> decimal .
Stack: 17
```

```
-> hex
-> clear
-> 0xf0 1 or .
F1
-> 0xf0 not .
FFFFFF0F              // all stack integers are SIGNED 32 bits
-> 3 1 xor .
2
-> 0x0f 4 << .
F0
-> 0xf0000000 24 >> .       // shift is UNSIGNED
F0
-> decimal
->
-> // ------------- stack manipulation -------------
-> clear
-> 1 .s                // push 1
Stack: 1
-> 2 .s                // push 2
Stack: 1 2             // 2 is on the top of the stack
-> drop .s             // drop top of stack
Stack: 1
-> dup .s              // duplicate top of stack
Stack: 1 1
-> clear .s
Stack: Empty.
-> 1 2 .s swap .s      // swap top 2 stack values
Stack: 1 2 Stack: 2 1
-> clear 1 2 3 over .s      // over pushes second from top
Stack: 1 2 3 2
-> clear 1 2 3  0 pick .s   // pick top of stack and push it
Stack: 1 2 3 3
-> clear 1 2 3 1 pick .s    // pick top of stack - 1 and push it
Stack: 1 2 3 2
-> clear 1 2 3 4 rot .s     // rotate top 3 items left
Stack: 1 3 4 2
->
-> // --------------- variables --------------------------
-> empty               // clear all word and variable definitions
-> int iValue         // integers should start with 'i' and
->                     // are initialized to zero
-> iValue .      // entering the variable name pushes its
0                     // value on the stack
->
-> 5 int= iValue2     // initialized variable
-> 2 to iValue        // assignment
-> iValue iValue2 + .
7
-> int iVALUE
IVALUE isn't unique.  // names internally convert to upper case
-> forget iVALUE      // kill iVALUE and all definitions after
                 // it was defined
```

```
-> float fVal         // declare float value (double precision)
-> fVal .s            // takes up 2 values on the stack
Stack: 0 0
-> f.                 // use f. as equivalent to .
0
-> .s
Stack: Empty.
-> 1.23 to fVal       // floating point literals MUST use decimal
-> fVal f.
1.23
-> clear              // clear stack
-> // BAD CODE:       // attempting to add integer literal
-> 1 fVal f+ f.

Stack underflow.
Walkback:

-> // GOOD CODE:
-> 1.0 fVal f+ f.
2.23
-> // float operators same as regular but preceeded with 'f'
-> 2.0 fVal f* f.
2.46
->
-> // string variables
-> 20 string sString1      // define sString1 20 characters long
-> "test" sString1 STRCPY
-> sString1 s.             // use s. as equivalent to .
test
-> " this" sString1 STRCAT // concatanate
-> sString1 s.             // display sString value
test this
-> hex                     // hex display
-> sString1 0 SPICK .      // pick 0th element push, display
74                         // 0x74 == 't'
-> sString1 STRLEN .       // string length
9
-> sString1 "test that" STRCMP .
-1
-> sString1 "test this" STRCMP .
0
-> 12 "=%5ld" s1 strform   // equivalent to sprintf; must use ld
                           // for integers
-> s1 s.
=   12
-> 12.345 "=%7.2f" s1 fstrform s.
=  12.35
-> 12.345 "=%9.1e" s1 fstrform s.
=  1.2e+01
->
->
```

```
->
-> // -------------- define new words ----------------
-> // all control elements (loops, if) must be inside
-> // word definitions. Definitions start with ':'
-> //
-> // FOR LOOP equivalent to:
-> // void forExample() { for(i=0;i<10;i++) printf("%d",i); }
-> : forExample 0 10 1 for i . endfor ;
-> // now that the new word has been defined we can use it
-> forexample
0 1 2 3 4 5 6 7 8 9
-> //
-> // IF STATEMENT
-> int iTest
-> : ifExample iTest 0 == if "zero" s. else "non-zero" s. endif ;
-> ifExample
zero
-> 1 to iTest
-> ifExample
non-zero
-> // comparison operators:
-> // ==, !=, <, <=, >, >= (integer)
-> // F==, F!=, F>, F>=, F<, F<=
->
```