PARALLAX SEMICONDUCTOR

www.parallaxsemiconductor.com
sales@parallaxsemiconductor.com
support@parallaxsemiconductor.com
phone: 916-632-4664 • fax:916-624-8003

Application Note AN004

GUI & Graphics Series — Getting Started with VGA and Terminal Output

Abstract: Each of the multicore P8X32A's eight cogs includes a video generator. This first tutorial in the GUI & Graphics series covers basic VGA character graphics, console terminals, mousing and simple text input with the Propeller P8X32A microcontroller. Example demos make use of available drivers to illustrate how effortless it can be to implement a simple VGA terminal application.

Introduction

One of the Propeller chip's most powerful features is its ability to implement peripherals through software drivers loaded at run time, instead of requiring fixed hardware. For example, RS-232 serial, SPI, I^2C, VGA generation, sound, keyboard, and mouse peripheral interfaces all can be implemented with software drivers.

This freedom is also one of the Propeller chip's biggest challenges from a programming perspective. There is no "correct" way to do something; rather, there are numerous methods, drivers, and approaches possible. A developer must decide what's right for the situation and constraints of a particular application, piece together a solution to that problem with a set of one or more drivers, and then add software on top to arrive at a final program that performs as desired.

With that in mind, there are a number of VGA drivers for the Propeller that tend to be "stock" or "default" drivers used by many programmers and applications. Therefore, instead of going to the Parallax Object Exchange and finding the latest and greatest VGA text driver, we are going to use one of the tested and more reliable Parallax internally developed VGA drivers for this application note. This has the advantage that many developers have used the driver; it is well understood and tested.

The driver selected for this application note is the "VGA High Res Text Driver." There are a few versions of it floating around and on the object exchange, but we are going to stick with version 1.0: VGA_HiRes_Text_010.Spin.

Reviewing the source code for VGA_HiRes_Text_010.Spin, the features of the driver are:

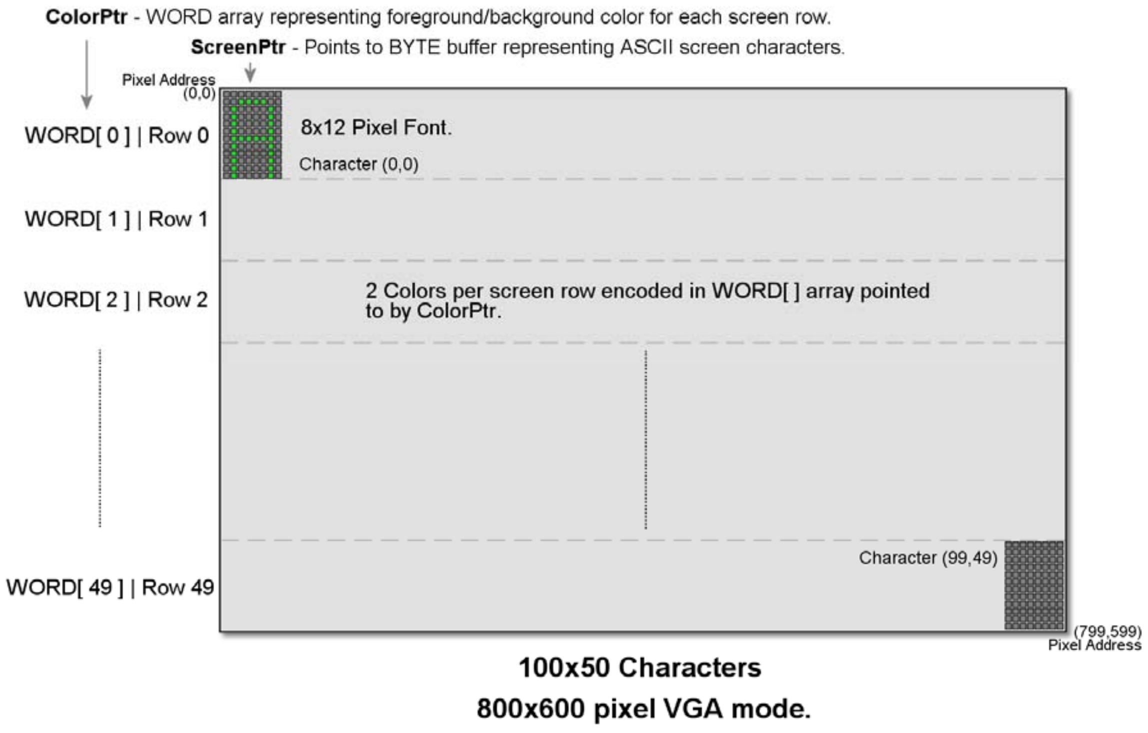
- Supports multiple resolutions such as 640x480, 800x600, and 1024x768.
- Supports 2 unique colors per row
- Requires only 2 processors (cogs) to run
- Very short, only a couple hundred lines of assembly language
- Uses a simple control interface; pointer to character memory and color words
- Includes a clean classic 8x12 pixel "IBM PC" style console font
- Two individual screen cursors (used for mousing or text input tracking)

The VGA Driver Features in Detail

The VGA driver itself doesn't know anything about text, strings, numbers, consoles, user input, and so forth. It simply reads a buffer (that you provide it) that represents the ASCII character codes of the VGA screen matrix. This matrix has a maximum size of 128 columns by 64 rows (128x64) = 8192 bytes for the 1024x768 pixel mode.

For example, if you set the driver to 800x600 pixels, considering the font is 8x12 means the text screen will be 800/8 x 600/12 = 100x50 characters and require only 5 KB. Figure 1 illustrates this as well as the other mechanics of the VGA driver setup.

Figure 1: Details of a 800x600 Pixel VGA Screen with 8x12 Character Font


ColorPtr - WORD array representing foreground/background color for each screen row.
ScreenPtr - Points to BYTE buffer representing ASCII screen characters.
Pixel Address (0,0)
WORD[0] | Row 0
8x12 Pixel Font.
Character (0,0)
WORD[1] | Row 1
WORD[2] | Row 2
2 Colors per screen row encoded in WORD[ ] array pointed to by ColorPtr.
WORD[49] | Row 49
Character (99,49)
(799,599) Pixel Address
100x50 Characters
800x600 pixel VGA mode.

The VGA driver has no knowledge or functionality of text, strings, printing, etc. Its single job is to render the VGA screen, drawing the characters using the 8x12 font in the colors selected along with the two overlay cursors. All this functionality is achieved via three different pointers passed to the VGA driver itself. They are named `ScreenPtr`, `ColorPtr` and `CursorPtr` respectively.

`ScreenPtr` — A pointer to up to 8192 bytes containing ASCII codes for each of the various screen resolutions:

- 640x480 pixels = 80x40 characters
- 800x600 pixels = 100x50 characters
- 1024x768 pixels = 128x64 characters.

Each byte's most significant bit (MSB) controls color inversion (inverse video) while the lower seven bits provide the actual ASCII code. VGA screen memory is arranged left-to-right, top-to-bottom. For example, the ASCII code for “A” is 64; in binary this is

%**0**_1000001. But, if the MSB is inverted like this; %**1**_1000001 then the character will be drawn with reversed colors, commonly referred to as "inverse video."

ColorPtr — A pointer to up to 64 words that define the foreground and background colors for each text row. The lower byte of each word contains the foreground RGB color for that row, while the upper byte contains the background RGB color for that row. The RGB data in each byte is in 2-bits-per-channel format in the following order:

%RRGGBB00 = RGB [2:2:2:0], where the lower 2 bits are always 0.

Each color has 4 possible intensity levels, so there are 4*4*4=64 possible colors for each row's foreground and background color. For example, the color word %%0020_3300 would be yellow on blue.

CursorPtr — A pointer to 6 contiguous bytes which controls the two overlay cursors. These cursors are drawn on top of the VGA character display and can be used for anything you wish—to show a mouse pointer, text input cursor, or whatever. The cursor control bytes are laid out in the following fashion for the 6 bytes in order:

- Bytes 0, 1, 2 represent Cx, Cy, and MODE of cursor 0.
- Bytes 3, 4, 5 represent Cx, Cy, and MODE of cursor 1.
- Cx, Cy represent the (x, y) cursor position on the screen and are in terms of screen characters, not pixels. So, in a 100x50 text mode (Cx, Cy) range from Cx = (0 to 99) and Cy = (0 to 50) respectively.

Finally, the 3rd byte in each record: MODE sets the cursor visibility and format. Only the lower 3 bits are used. The format of MODE's 3-bit encoding are:

```
%x00 = cursor off
%x01 = cursor on
%x10 = cursor on, blink slow
%x11 = cursor on, blink fast
%0xx = cursor is solid block
%1xx = cursor is underscore
```

For example, setting one of the cursors to (0, 0, %011) would set it at the upper hand corner of the screen at (0, 0) as a solid block, blinking fast.

Terminal Services and Console Support

With the VGA driver in hand, the next step is to develop a custom terminal console. A terminal/console driver is a piece of software the mimics "standard out" text display on a PC, much like using print commands from C or BASIC on a PC, or a VT-100 terminal. When text is printed to the screen, if it extends off the right hand side it wraps around, and when text reaches the bottom of the screen the text scrolls. These features, along with others, are part of "terminal/console" functionality.

There are terminal drivers available on the Parallax Object Exchange[1], but many are limited for simpler applications. Additionally, we would have to "connect" the terminal driver to the VGA driver, interface them, and re-write much of the terminal driver and/or VGA driver to make the connection because the terminal driver has to work within the constraints of the VGA driver's architecture.

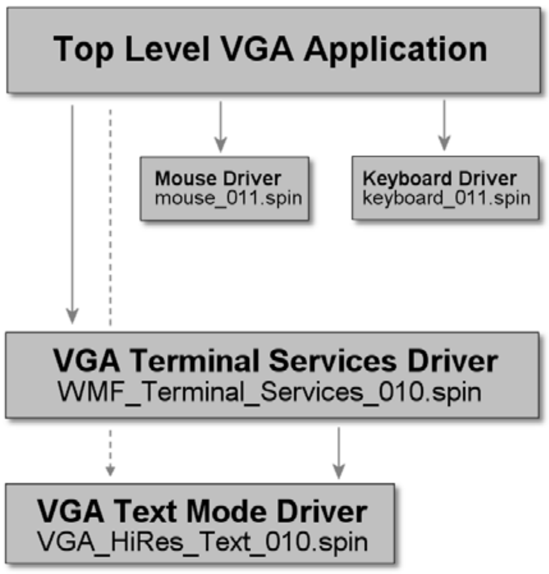
The goal here is not only to get you up and running with VGA, but to create a framework to build upon, a robust and feature-rich terminal services driver. Written from scratch for this purpose, the driver has the following features:

- Direct character and text printing to the VGA screen memory buffer for high speed applications
- Terminal emulation that allows standard printing of character, strings, numbers (hex, decimal, binary), and more with support for screen wrap, scrolling, etc.
- Complex rendering methods to draw shadowed boxes, frames, and outlines at high speed to facilitate drawing controls, buttons, lists, menus, and other higher level GUI objects
- Number conversion methods and string/character manipulation built in, emulating common Unix C library functions *atoi(), itoa(), strupper()* and more
- Simple time-delay methods

The driver WMF_Terminal_Services_010.spin has too many methods to cover here; for an exhaustive listing please review the source file. The demos below cover key functionality and features and illustrate how to get things up and running. Finally, Figure 2 below illustrates the relationship among all the pieces of any VGA application.

Note: The “WMF” prefix stands for Propeller “Window Manager Framework” and will be used to denote software modules that relate to graphics, windows, GUIs, and other similar drivers.

Figure 2: Components of a VGA Application

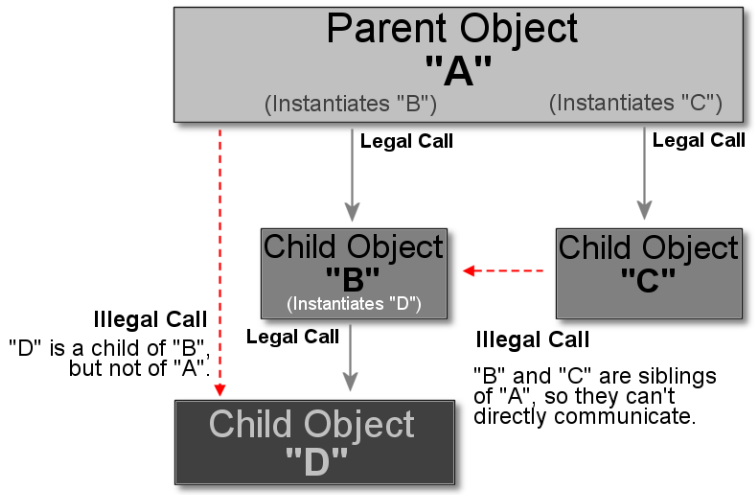


Referring to Figure 2, all applications have a top-level file that runs the application and instantiates the terminal services driver WMF_Terminal_Services_010.spin along with the mouse and keyboard driver. Furthermore, the terminal services driver includes the VGA driver itself.

Selecting Sub-Object Locations in the Hierarchy

Spin does not allow direct communication between child objects, nor can a child object directly call a parent object's methods. Take care when making the organizational decisions of where to put objects hierarchically. For example, there is no way for the top-level application to make calls to the VGA driver directly since the VGA driver is instantiated by the terminal services driver. Therefore, if the top level object wants control of the methods in the VGA driver (if any) or any child object, an interface layer must be made that passes method calls from top level object to child object to sub-child object. The call graph of Figure 3 below shows this: direct parents can call children's methods, but siblings can't call each other and parents can't call through to grandchildren objects.

Figure 3: Legal Calls from Parent to Child Objects



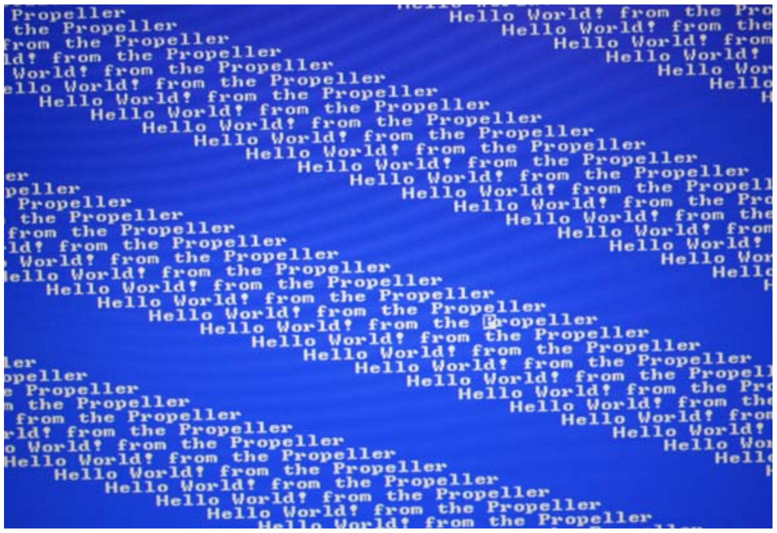
There are solutions to this dilemma such as using global memory mapped message passing, but that can become messy as well. Therefore, sometimes, it's just better to "drag" a child object up the hierarchy to a top-level parent object, even when it makes more sense as a child object, and then instantiate it directly and pass downward its state information to sub-objects and children that need it. This is the thought process for the mouse and keyboard.

It is cleaner from an abstraction point of view to make the mouse and keyboard objects part of the terminal services. However, if we did this, the application couldn't make calls to the mouse or keyboard driver without creating pass-through methods in the terminal services driver. Ultimately, we will want everything to be abstracted away and tucked into a single driver, but for now having the keyboard and mouse at the application level will serve our needs.

Example 1: Hello World

Load the sample application top-level file WMF_HelloWorld_010.Spin. It requires a Propeller development board with VGA, PS/2 mouse and keyboard peripherals. Change the base pin group for the VGA, mouse and keyboard in the CON block of the code as needed. Also, the source assumes a 5 MHz crystal as well, so change as required. Figure 4 below shows the demo running on a standard VGA enabled LCD TV.

Figure 4: The “Hello World” Output on VGA at 800x600 in 100x50 Character Mode



Tip: Rather than buying a new LCD computer monitor, use an LCD TV. It may not support the ultra-high-resolution modes such as 2000x1600 that some LCD monitors do, but most LCD TVs can support at least 1280x768. Not only do you get a monitor that you can use for your Propeller experiments, but you get a built in tuner and can use the TV for other purposes other than the computer.

The demo begins by entering the first `PUB Start` which is shown below (with some comments removed for brevity).

```
PUB Start

  ' first step create the GUI itself
  CreateAppGUI

  ' MAIN EVENT LOOP - this is where you put all your code in an infinite loop...
  repeat
    ' get mouse state which is being tracked by VGA driver to move
    ' the virtual cursor(s) as well
    gMouseCursX   := mouse.bound_x
    gMouseCursY   := mouse.bound_y
    gMouseButtons := mouse.buttons

    ' main code goes here...............

    ' hello world just print some text to the "terminal"
    ' which will scroll for us automatically
    WMF.StringTerm(string("Hello World! from the Propeller                     "))

    ' make a call to delay 10 ms just to slow things down, so you can see them...
    WMF.DelayMilliSec( 10 )

' end PUB -------------------------------------------------------------------
```

This is the recommended software pattern for developing GUI applications. The main entry point is entered, then a call to some initialization method is made (`CreateAppGui` in this case), and then an infinite loop is entered that gets user input, prints on the terminal, gets user input, etc. In this case, the user input is read by the mouse and keyboard drivers which are assigned to the globals; `gMouseCurs*`. Additionally, `gTextCurs*` is updated the other “text” cursor is needed to track user input or show a text input focus. However, in this demo only the mouse cursor is updated.

The interesting thing about these globals is they are linked to the VGA driver itself during the initialization call to the driver. However, we don’t make this call directly, since the terminal services driver instantiates the VGA driver. Therefore, when the call is made to the terminal services driver to start it up, that’s where the VGA pins and the base address of the global cursor variables are (6 bytes are sent). This is shown in the code listing below `CreateAppGUI` which is called by the previous `Start` method to perform initialization and setup (comments abridged to save space).

```
PUB CreateAppGUI | retVal
' This method creates the entire user interface for the application and performs
' any other static initialization you might want.
  ' text cursor starting position and as blinking underscore
  gTextCursX     := 0
  gTextCursY     := 0
  gTextCursMode  := %110

  'set mouse cursor position and as solid block
  gMouseCursX    := VGACOLS/2
  gMouseCursY    := VGAROWS/2
  gMouseCursMode := %001

  ' start the mouse
  mouse.start( MOUSE_DATA_PIN, MOUSE_CLK_PIN )

  ' set boundaries
  mouse.bound_limits(0, 0, 0, VGACOLS - 1, VGAROWS - 1, 0)

  ' adjust speed/sensitivity (note minus value on 2nd parm inverts the axis as well)
  mouse.bound_scales(8, -8, 0)

  'mouse starting position
  mouse.bound_preset(VGACOLS/2, VGAROWS/2, 0)

  ' start the keyboard
  kbd.start( KBD_DATA_PIN, KBD_CLK_PIN )

  ' now start the VGA driver and terminal services
  retVal := WMF.Init(VGA_BASE_PIN, @gTextCursX )

  ' rows encoded in upper 8-bits.
  ' columns in lower 8-bits of return value, redundant code really
  ' since we pull it in with a constant in the first CON section, but up to you!
  gVgaRows := ( retVal & $0000FF00 ) >> 8
  gVgaCols := retVal & $000000FF

  ' VGA buffer encoded in upper 16-bits of return value
  gVideoBufferPtr := retVal >> 16

  '---------------------------------------------------------------------------
  'setup screen colors
  '---------------------------------------------------------------------------

  WMF.ClearScreen( PWM#CTHEME_ATARI_C64_FG, PWM#CTHEME_ATARI_C64_BG )

  ' return to caller
  return

' end PUB ---------------------------------------------------------------------
```

Reviewing the method, it starts the mouse, keyboard, and then makes a call to `WMF.Init(VGA_BASE_PIN, @gTextCursX )`. This call sets the proper VGA base pin group as well as binding the base address of the globals that are used as an information passing mechanism to control the positions of the two virtual cursors the VGA driver supports. The VGA driver could have required method calls to update the positions, but this is a little more elegant and transparent. Whatever happens to the globals in the top-level object is reflected almost instantly in the VGA driver's cursor rendering code.

A few key variables are extracted to track rows, columns, and the video buffer. Finally, a call to `WMF.ClearScreen(PWM#CTHEME_ATARI_C64_FG, PWM#CTHEME_ATARI_C64_FG)` is made. This is one of the terminal services driver's methods that clears the VGA screen as well as sets every row of the color control words to the sent foreground and background colors. In this case, it is a classic Atari 800/C64 white text on blue background. As noted before, the VGA driver refers to a screen buffer that holds the ASCII codes of the character matrix `ScreenPtr`, but there is also pointer to the colors for each row `ColorPtr`. This gives you the ability to create colorful GUI arrangements where at least every character row can have 2 colors per character.

The "#" syntax `PWM#CTHEME_ATARI_64_FG` might not be familiar to you. In general, to access a constant in a child object, use the syntax *child_object#constant_name*. This is a great way to pull in constants from a child object without having to re-define them locally.

The method `ClearScreen` sets the entire screen to the same foreground/background pair. To update only a single row's colors, make a call to this method:

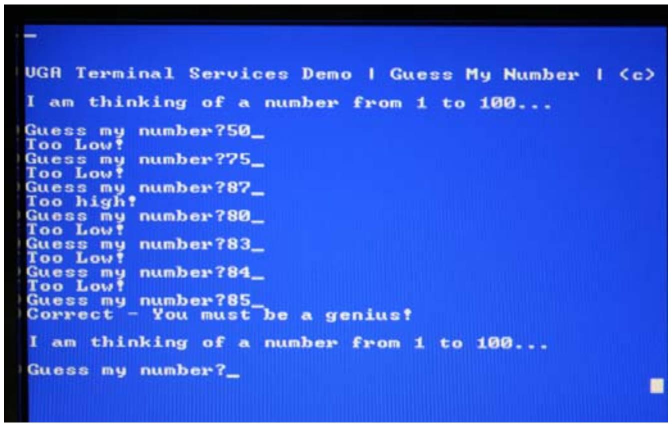
```
PUB SetLineColor( pRow, pFGroundColor, pBGroundColor )
```

...with the desired row, foreground, and background color where the colors are in RGB(2:2:2:0) format. There will be more about color selection in the third example demo.

Example 2: Text Input with "Guess My Number?"

Load the sample application top-level file WMF_GuessMyNumber_010.Spin. It requires a Propeller development board with VGA, PS/2 mouse and keyboard peripherals. Change the base pin group for the VGA, mouse and keyboard in the `CON` block of the code as needed. Also, the source assumes a 5 MHz crystal as well, so change as required. Figure 5 below shows the demo running on a standard VGA-enabled LCD TV.

Figure 5: "Guess My Number?" Demo on VGA at 800x600 in 100x50 Character Mode



```
VGA Terminal Services Demo | Guess My Number | (c)
I am thinking of a number from 1 to 100...
Guess my number?50_
Too Low!
Guess my number?75_
Too Low!
Guess my number?87_
Too high!
Guess my number?80_
Too Low!
Guess my number?83_
Too Low!
Guess my number?84_
Too Low!
Guess my number?85_
Correct - You must be a genius!
I am thinking of a number from 1 to 100...
Guess my number?_
```

The "Guess my Number" application illustrates the use of the keyboard to input text. This is a challenging part of programming: when there aren't built-in input commands and methods, you have to write them yourself. Moreover, creating a single line "editor" can be challenging. With that in mind, the demo uses the template from the previous "Hello World" demo and computes a random number, enters a loop, waits for user input, and tests against the user's input number. If the number entered is correct the user wins, if it is too high or too low the loop continues. The initialization code in `CreateAppGui` is nearly the same, however there is a bit of code at the end of it that computes a random number seed:

```
  ' seed the random number generator with something random such as signals on inputs
  ' this works quite well!
  repeat index from 0 to 1024
    gRandSeed += INA
```

This code generates a random number to "seed" Spin's built-in pseudo-random number generator operator which is based on a LFSR (linear feedback shift register) algorithm. The operator is the question mark "**?**" and performs a forward or reverse shift using the following syntax:

```
' forward-shift of variable x, refer to Propeller Reference Manual pg. 159
?x

' reverse-shift of variable x
x?
```

However, with a given seed, the pseudo-random numbers will always have the same numerical sequence. Thus, one needs to "seed" the initial value of **x** with something random to begin with. This can be quite challenging, but there usually are some common strategies available. One is to start a counter, and when the user does something, that count becomes the seed. However, if there is no initial user interaction, that won't work.

Another strategy is to pick up noise in memory or I/O pins, which is random by nature. This is the strategy used in the code snippet above: the random noise on all 32 I/O lines is summed up and then the resulting 32-bit number generated becomes the seed. Of course, if you set all inputs to a known state before calling this algorithm, then the results wouldn't be very random.

The call to `CreateAppGui` initializes the application. Once complete, the execution returns to the Start method and enters into the main event loop which is the game of "Guess my Number."

```
PUB Start | randNumber, guessNumber

  ' first step create the GUI itself
  CreateAppGUI

    ' set terminal cursor initial position a little from top
    WMF.GotoXYTerm( 0, 3)
    WMF.StringTermLn(string("VGA Terminal Services Demo | Guess My Number | (c) ←
                                        Parallax 2011"))


  ' MAIN EVENT LOOP - this is where you put all your code in an infinite loop...
  repeat

    ' main code goes here................
    ' Guess my number game to illustrate user input and string to numeric conversion
    WMF.NewlineTerm
    WMF.StringTermLn(string("I am thinking of a number from 1 to 100..."))

    ' get a new random number from generator
    ' || is absolute value operator, ? uses the seed as the value for a
    ' LFSR (linear feedback shift register)
    ' refer to Propeller Manual for more info on the ? operator
    randNumber := 1 + ||(?gRandSeed // 100)

    ' start guess off incorrectly
    guessNumber := -1

    ' enter into loop, once in here, the code blocks waiting for user
    ' input and the mouser cursor doesn't update anymore
    ' this is to be expected, and purposely shown here.
    repeat while ( guessNumber <> randNumber )
      WMF.NewlineTerm
      WMF.StringTerm(string("Guess my number?"))

      ' get user input with local method
      GetStringTerm( @gStrBuff1, 4)

      ' convert input from string to decimal
      ' (here you might want to do input validation as well)
      ' this atoi is very smart, it can convert decimal, hex $, binary % numbers!
      guessNumber := WMF.atoi( @gStrBuff1, 4 )

      ' test input
      if ( guessNumber > randNumber )
        WMF.NewlineTerm
        WMF.StringTerm(string("Too high!"))
      elseif ( guessNumber < randNumber )
        WMF.NewlineTerm
        WMF.StringTerm(string("Too Low!"))
      else
        WMF.NewlineTerm
        WMF.StringTermLn(string("Correct - You must be a genius!"))

' end PUB ----------------------------------------------------------------------
```

The code is straightforward, so let's focus on the user input elements. There are two primary methods. One is `GetStringTerm(@gStrBuff1, 4)` which gets the user's input and stores it in the global character array `gStrBuff1` while limiting the user input to 4 characters. The other is the method call that converts and validates (to a degree) the user input: `guessNumber :=WMF.atoi(@gStrBuff1, 4)`. The method call to `atoi` mimics the standard C/Unix library ASCII-to-integer function and tries to convert the sent string to an integer. This integer will be compared against the pre-computed random number.

The code for `atoi` can of course be found in the terminal services driver WMF_Terminal_Services_***.spin. However, the `GetStringTerm` is a local method since it requires the keyboard and the keyboard is instantiated locally, thus makes the method a little easier to code. Later you might decide to push the `GetStringTerm` method down into the terminal driver, it's up to you. But, for now, the code is local and a copy is listed below for review:

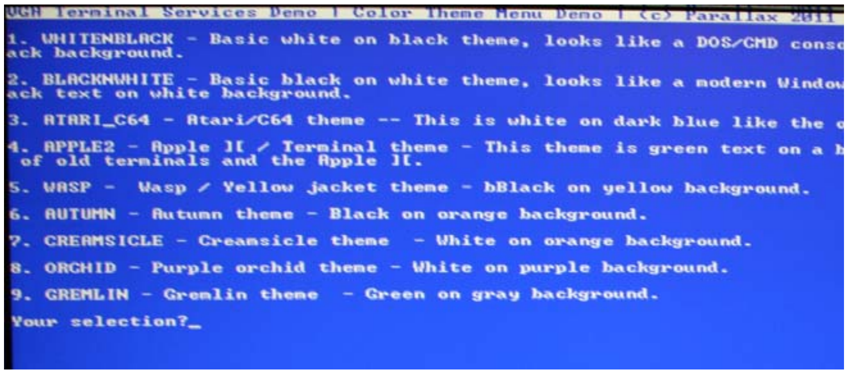
```
PUB GetStringTerm(pStringPtr, pMaxLength) | length, key
{{
DESCRIPTION: This simple method is a single line editor that allows user to enter keys from
the keyboard and then echoes them to the screen, when the user hits <ENTER> |
<RETURN> the method exits and returns the string. The method has simple editing and allows
<BACKSPACE> to delete the last character, that's it! The method outputs to the terminal.

PARMS: pStringPTr - pointer to storage for input string.
       pMaxLength - maximum length of string buffer.

RETURNS: pointer to string, empty string if user entered nothing.
}}

  ' current length of string buffer
  length := 0

  ' draw cursor
  repeat

    ' draw cursor
    WMF.OutTerm( "_" )
    WMF.OutTerm( $08 )

    ' wait for keypress
    repeat while (kbd.gotkey == FALSE)

    ' user entered a key process it

    ' get key from buffer
    key := kbd.key

    case key
       ASCII_LF, ASCII_CR: ' return

        ' null terminate string and return
        byte [pStringPtr][length] := ASCII_NULL

        return( pStringPtr )

       ASCII_BS, ASCII_DEL, ASCII_LEFT: ' backspace (edit)

         if (length > 0)
           ' move cursor back once to overwrite last character on screen
           WMF.OutTerm( ASCII_SPACE )
           WMF.OutTerm( $08 )
           WMF.OutTerm( $08 )

           ' echo character
           WMF.OutTerm( ASCII_SPACE )
           WMF.OutTerm( $08 )

           ' decrement length
           length--

       other:    ' all other cases
         ' insert character into string
         byte [pStringPtr][length] := key

         ' update length
         if (length < pMaxLength )
           length++
         else
           ' move cursor back once to overwrite last character on screen
           WMF.OutTerm( $08 )

         ' echo character
         WMF.OutTerm( key )

' end PUB ----------------------------------------------------------------------
```

An input method is a bit tricky to write and you may need more functionality, but its main purpose is to get a single string of input from the Propeller keyboard and allow some modicum of editing. In this case, backspace and delete work; a more complex editor could be developed that allowed the arrow keys, and copy and paste as well. Of course this would take a lot more code, buffers, and state machines to track what's going on. Nonetheless, this little string input method works quite well with the VGA driver and terminal services driver and leverages the terminal services printing functionality to operate. Thus, porting to other terminals is very easy.

Example 3: The Art of Selecting Appropriate Colors for Text and GUI Applications

Load the sample application's top-level file WMF_ColorThemeDemo.Spin. It requires a Propeller development board with VGA, PS/2 mouse and keyboard peripherals. Change the base pin group for the VGA, mouse and keyboard in the `CON` block of the code as needed. Also, the source assumes a 5 MHz crystal as well, so change as required. Figure 6 below shows the demo running on a standard VGA enabled LCD TV.

Figure 6: "Color Theme Demo" Demo on VGA at 800x600 in 100x50 Character Mode



```
VGA Terminal Services Demo | Color Theme Menu Demo | (c) Parallax 2011
1. WHITENBLACK - Basic white on black theme, looks like a DOS/CMD conso
ack background.
2. BLACKNWHITE - Basic black on white theme, looks like a modern Window
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on dark blue like the c
4. APPLE2 - Apple ][ / Terminal theme - This theme is green text on a b
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on yellow background.
6. AUTUMN - Autumn theme - Black on orange background.
7. CREAMSICLE - Creamsicle theme  - White on orange background.
8. ORCHID - Purple orchid theme - White on purple background.
9. GREMLIN - Gremlin theme  - Green on gray background.
Your selection?_
```

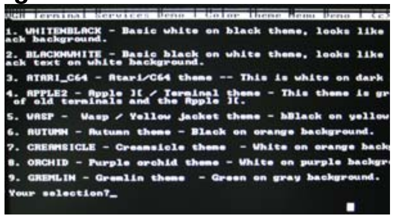
The demo uses the local Propeller keyboard to select one of the color schemes numerically. Simply enter the number of the theme and press <Enter>, and the theme will slowly update.

There are entire books written about GUI and color design; it's a complex field. It takes a a true artist's eye to select complementary colors, position, and design elements to make them useful, easy to understand, and visually pleasing. In other words, color takes a lot of creativity. Many programmers have trouble with colors and pick color schemes that they personally like, or just don't think about it and use ugly defaults. With that in mind, this driver includes a number of simple 2-color schemes to help you get started without wasting a lot of time trying out different colors. The schemes are denoted by two pairs of constants in the terminal services driver WMF_Terminal_Services_***.spin in the format shown below.

```
 ' basic white on black theme, looks like a DOS/CMD console terminal,
 ' white text on black background
CTHEME_WHITENBLACK_FG      = %%333
CTHEME_WHITENBLACK_BG      = %%000

CTHEME_WHITENBLACK_INFO_FG = %%000
CTHEME_WHITENBLACK_INFO_BG = %%333
```

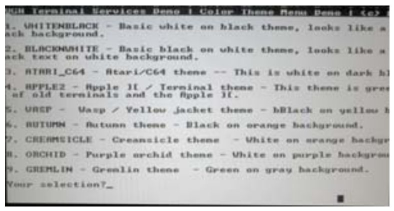
Figure 7 shows each of the colors schemes and what they would look like on the VGA monitor. Of course, the screen shots don't do them justice, but you get the idea.

Figure 7: Screen Shots of the Pre-defined Color Themes in the Terminal Services Driver



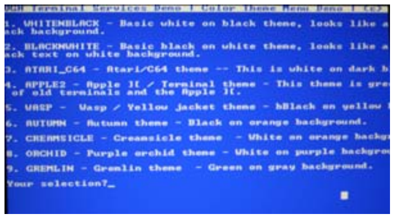
```
1. WHITENBLACK - Basic white on black theme, looks like
ack background.
2. BLACKNWHITE - Basic black on white theme, looks like
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on dark
4. APPLE2 - Apple ][ / Terminal theme - This theme is gr
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on yellow
6. AUTUMN - Autumn theme - Black on orange background.
7. CREAMSICLE - Creamsicle theme  - White on orange back
8. ORCHID - Purple orchid theme - White on purple backgr
9. GREMLIN - Gremlin theme  - Green on gray background.
Your selection?_
```

CTHEME_WHITENBLACK



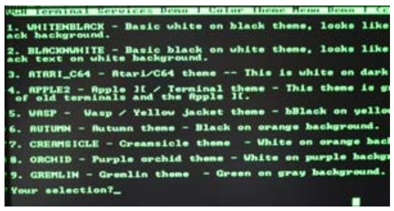
```
1. WHITENBLACK - Basic white on black theme, looks like a
ack background.
2. BLACKNWHITE - Basic black on white theme, looks like a
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on dark bl
4. APPLE2 - Apple ][ / Terminal theme - This theme is gree
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on yellow b
6. AUTUMN - Autumn theme - Black on orange background.
7. CREAMSICLE - Creamsicle theme  - White on orange backgr
8. ORCHID - Purple orchid theme - White on purple backgrou
9. GREMLIN - Gremlin theme  - Green on gray background.
Your selection?_
```

CTHEME_BLACKNWHITE



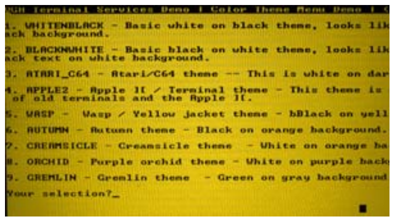
```
1. WHITENBLACK - Basic white on black theme, looks like a
ack background.
2. BLACKNWHITE - Basic black on white theme, looks like a
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on dark b
4. APPLE2 - Apple ][ / Terminal theme - This theme is gre
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on yellow
6. AUTUMN - Autumn theme - Black on orange background.
7. CREAMSICLE - Creamsicle theme  - White on orange backg
8. ORCHID - Purple orchid theme - White on purple backgro
9. GREMLIN - Gremlin theme  - Green on gray background.
Your selection?_
```

CTHEME_ATARI_C64



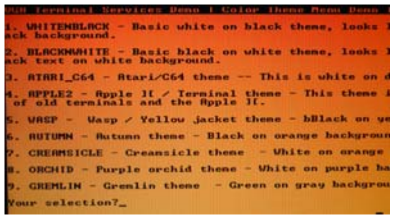
```
1. WHITENBLACK - Basic white on black theme, looks like
ack background.
2. BLACKNWHITE - Basic black on white theme, looks like
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on dark
4. APPLE2 - Apple ][ / Terminal theme - This theme is g
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on yello
6. AUTUMN - Autumn theme - Black on orange background.
7. CREAMSICLE - Creamsicle theme  - White on orange bac
8. ORCHID - Purple orchid theme - White on purple backg
9. GREMLIN - Gremlin theme  - Green on gray background.
Your selection?_
```

CTHEME_APPLE2



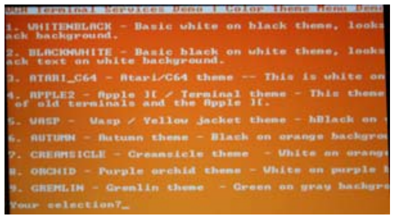
```
1. WHITENBLACK - Basic white on black theme, looks lik
ack background.
2. BLACKNWHITE - Basic black on white theme, looks lik
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on dar
4. APPLE2 - Apple ][ / Terminal theme - This theme is
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on yell
6. AUTUMN - Autumn theme - Black on orange background.
7. CREAMSICLE - Creamsicle theme  - White on orange ba
8. ORCHID - Purple orchid theme - White on purple back
9. GREMLIN - Gremlin theme  - Green on gray background
Your selection?_
```

CTHEME_WASP



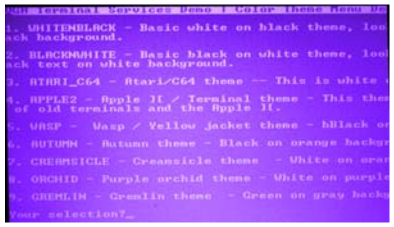
```
1. WHITENBLACK - Basic white on black theme, looks
ack background.
2. BLACKNWHITE - Basic black on white theme, looks l
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on d
4. APPLE2 - Apple ][ / Terminal theme - This theme i
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on ye
6. AUTUMN - Autumn theme - Black on orange backgroun
7. CREAMSICLE - Creamsicle theme  - White on orange
8. ORCHID - Purple orchid theme - White on purple ba
9. GREMLIN - Gremlin theme  - Green on gray backgrou
Your selection?_
```

CTHEME_AUTUMN



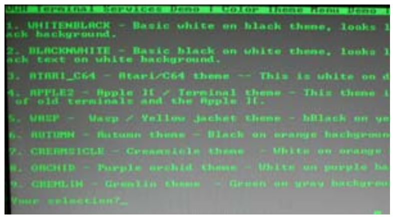
```
1. WHITENBLACK - Basic white on black theme, looks
ack background.
2. BLACKNWHITE - Basic black on white theme, looks
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on
4. APPLE2 - Apple ][ / Terminal theme - This theme
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on
6. AUTUMN - Autumn theme - Black on orange backgro
7. CREAMSICLE - Creamsicle theme  - White on orang
8. ORCHID - Purple orchid theme - White on purple
9. GREMLIN - Gremlin theme  - Green on gray backgr
Your selection?_
```

CTHEME_CREAMSICLE



```
1. WHITENBLACK - Basic white on black theme, loo
ack background.
2. BLACKNWHITE - Basic black on white theme, loo
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white
4. APPLE2 - Apple ][ / Terminal theme - This the
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack o
6. AUTUMN - Autumn theme - Black on orange backg
7. CREAMSICLE - Creamsicle theme  - White on ora
8. ORCHID - Purple orchid theme - White on purpl
9. GREMLIN - Gremlin theme  - Green on gray back
Your selection?_
```

CTHEME_ORCHID



```
1. WHITENBLACK - Basic white on black theme, looks l
ack background.
2. BLACKNWHITE - Basic black on white theme, looks l
ack text on white background.
3. ATARI_C64 - Atari/C64 theme -- This is white on d
4. APPLE2 - Apple ][ / Terminal theme - This theme i
 of old terminals and the Apple ][.
5. WASP -  Wasp / Yellow jacket theme - bBlack on ye
6. AUTUMN - Autumn theme - Black on orange backgroun
7. CREAMSICLE - Creamsicle theme  - White on orange
8. ORCHID - Purple orchid theme - White on purple ba
9. GREMLIN - Gremlin theme  - Green on gray backgrou
Your selection?_
```

CTHEME_GREMLIN

Even though this VGA driver has 2 colors per row, you can have any 2 colors in any row; and could build up a very colorful display/GUI. However, try to restrain yourself when it comes to *too* many colors. Less is always better. Some suggested guidelines are:

- Never use red and blue together as background and foreground.
- Never use two colors that have weak contrast, for example grey on white.
- Select colors based on the application; warm, inviting colors such as for an organic feel—browns, chocolate orange, etc.
- Use pastels with higher overall brightness settings for more feminine/friendly GUI displays.
- Use simple solid backgrounds such as black, blue, brown, white for general information display.
- Use more energetic colors such as red, yellow, and green backgrounds to grab attention; however, these colors will really tire and strain users' eyes over time.
- Use darker backgrounds with brighter foreground text to reduce eye strain. Black text on white background used by most Windows applications actually strains the eyes more. This is because the display is emitting light rather than reflecting. Thus, books are printed with black text on a white background since they reflect light, but using that theme on a computer does strain the eyes with a lot of constant brightness.
- Don't use your favorite colors, for example fluorescent green on purple won't look good to anyone, but you!
- Try your color selections on a few people and let them decide which is most pleasing and appropriate.
- When in doubt stick to black on white, white on black, white on dark blue, or for more of a old style terminal feel, green or red on black.

Finally, if you decide to design a display design that has an “information bar” or different regions vertically with different colors, select inverse colors for those regions and/or colors that still complement the larger color scheme. For example, don’t use a black background with green text then make a single row with yellow text on cyan background. Those have no relationship at all. All colors should look and feel complementary.

Summary

This application note has outlined a framework including terminal services and a display driver for building VGA textual applications. Additionally, user input via keyboard has been explored as well as color selection and some of the more artistic aspects of GUI design.

API Listing

The following is a complete API functional listing for WMF_Terminal_Services_010.spin. Look to the source code for more details and complete parameter descriptions.

Initialization Method(s)

`PUB Init(pVGABasePin, pTextCursXPtr)` — Initializes the VGA driver as well as basic terminal parameters.

Direct Frame Buffer Methods for General Rendering for Console and Controls

`PUB PrintString(pStrPtr, pCol, pRow, pInvFlag)` — This method draws a string directly to the frame buffer.

`PUB PrintChar(pChar, pCol, pRow, pInvFlag)` — Draws a character directly to the frame buffer avoiding the terminal system.

`PUB ClearScreen(pFGroundColor, pBGroundColor)` — Clear the screen at the memory buffer level, very fast.

`PUB SetLineColor(pRow, pFGroundColor, pBGroundColor)` — Sets the sent row to the given foreground and background color.

`PUB DrawFrame(pCol, pRow, pWidth, pHeight, pTitlePtr, pAttr, pVgaPtr, pVgaWidth)` — This method draws a rectangular "frame" with title and shadows.

Text Console Terminal Methods

`PUB StringTermLn( pStringPtr )` — Prints a string to the terminal and appends a newline.

`PUB StringTerm( pStringPtr )` — Prints a string to the terminal.

`PUB DecTerm( pValue, pDigits)` — Prints a decimal number to the screen.

`PUB HexTerm(pValue, pDigits)` — Prints the sent number in hex format.

`PUB BinTerm(pValue, pDigits)` — Prints the sent value in binary format with 0's and 1's.

PUB `NewlineTerm` - Moves the terminal cursor home and outputs a carriage return.

PUB `PrintTerm( pChar )` — Prints the sent character to the terminal console with scrolling.

PUB `OutTerm( pChar )` — Outputs a character to terminal. This is the primary interface from the client to the driver in "terminal mode."

PUB `SetColTerm( pCol )` — Sets terminal x column cursor position.

PUB `SetRowTerm( pRow )` — Sets terminal y row cursor position

PUB `GotoXYTerm( pCol, pRow )` — Sets the x column and y row position of terminal cursor.

PUB `GetColTerm` — Retrieves x column cursor position.

PUB `GetRowTerm` — Retrieves y row terminal cursor position.

String and Numeric Conversion Methods

PUB `StrCpy( pDestStrPtr, pSourceStrPtr )` — Copies the NULL terminated source string to the destination string and null terminates the copy.

PUB `StrUpper( pStringPtr )` — Converts the sent string to all uppercase.

PUB `ToUpper( pChar )` — Returns the uppercase of the sent character.

PUB `IsInSet(pChar, pSetStringPtr)` — Tests if sent character is in sent string.

PUB `IsSpace( pChar )` — Tests if sent character is white space, cr, lf, space, or tab.

PUB `IsNull( pChar )` — Tests if sent character is NULL, 0.

PUB `IsDigit( pChar )` — Tests if sent character is an ASCII number digit [0..9], returns integer [0..9]

PUB `IsAlpha( pChar )` — Tests if sent character is in the set [a...zA...Z].

PUB `IsPunc( pChar )` — Tests if sent character is a punctuation symbol !@#$%^&*()--+={}[]|\;:'",<.>/?.

PUB `HexToDec( pChar )` — Converts ASCII hex digit to decimal.

PUB `HexToASCII( pValue )` — Converts a number 0..15 to ASCII 0...9, A, B, C, D, E, F.

PUB `itoa(pNumber, pBase, pDigits, pStringPtr)` - "C-like" function that converts *pNumber* to string; decimal, hex, or binary formats.

PUB `atoi(pStringPtr, pLength)` — "C-like" function that tries to convert the string to a number, supports binary %, hex $, decimal (default).

Time Methods

`PUB DelayMilliSec( pTime )` — Delays *pTime* milliseconds and returns.

`PUB DelayMicroSec( pTime )` — Delays *pTime* microseconds and returns.

Resources

The following example Spin objects are available for download as a zip archive from this application note's web page: www.parallaxsemiconductor.com/an004.

WMF_HelloWorld_010.Spin
WMF_GuessMyNumber_010.Spin
WMF_ColorThemeDemo_010.spin

WMF_Terminal_Services_010.spin
VGA_HiRes_Text_010.spin
Mouse_011.spin
Keyboard_011.spin

References

1. Propeller Object Exchange: http://obex.parallax.com

Revision History

Version 1.0: original document.