

htmltty

This started out as my covid project (in stead of bike touring somewhere in the world). I had developed a version of Business Basic (BBX/Basic Four compatible) from 1977 thru about 2000. The last version (call SM32) was a windows version that made use of a Microsoft provided HTML view. That windows versions could "CONNECT", acting as a terminal server, to a unix version of SM32 giving the HTML ability on the unix side.

So when I took a look java script and when I saw it had added sockets (it is call WebSockets) I thought I could create a terminal server that would act like my windows version and connect to an SM32 server (windows and linux) and called it SM32Terminal. I got that running to my satisfaction. Along comes more waves of covid.

Since most Business Basic programmers have departed us - I thought there might be some character based unix apps that could benefit from some gui. htmltty adds a VT100 terminal to the SM32Terminal. SM32Terminal was packet based. htmltty adds module Xterm.js which is a state machine to interpret stream data from a pseudo terminal.

I pulled the WebSocket stuff from my SM32Server and created htmlttyd. This is like a telnet server there is no encryption like sshd. So htmltty (java script) with htmlttyd (C++) creates a vt100 connection to a linux logon.

My last addition was to have the build merge all the java script from htmltty into a single html (Single Page App). This htmltty SPA is data to htmlttyd during the build. Htmlttyd acts as a mini web server and serves up htmltty which in turn connects via WebSockets back to htmlttyd.

On my lubuntu box I start htmlttyd with:

```
$  
$ sudo su  
#htmlttyd 5001 htmlttyd.log &  
# exit  
$
```

To start htmltty:

I have a desktop short cut on my windows 10 box - target=
"[C:\Program Files\(x86\)\Google\Chrome\Application\chrome.exe](C:\Program Files(x86)\Google\Chrome\Application\chrome.exe)"
--app=<http://192.168.1.72:5001>

On my lubuntu box I have a shell script:

```
/usr/bin/google-chrome --app=http://127.0.0.1:5001 2> /dev/null &
```

At the moment htmltty runs with chrome or chromium.

If your looking for a quick look, maybe bring up 1 of the demos in demo/demos say demo3 and then go to demo/final and bring up alldemof to try a few demos from the menu.

Ken Yerex

Demo

The demo directory contains a Makefile that should build some simple demo programs written in C. There are 4 sub-directories: windows, demos, final and common. All the demos use curses and then extends the VT100 capabilities with SM32 Mnemonics.

The makefile in the demo directory should build:

common/out.a - archive with common routines

Character Window Demos

windows/wdemo1	SM32 window mnemonic
windows/wdemo1x	SM32 window mnemonic [title decorated]
windows/wdemo1y	SM32 window mnemonic [no title]
windows/wdemo2	SM32 fwindow mnemonic
windows/wdemo2x	SM32 fwindow mnemonic [adding style]
windows/wpop	Used to pop demo windows

Demos using SM32 window mnemonic

demos/helloworld	displays html/helloworld.html with 1 button
demos/demo1	displays html/demo1.html and inputs data
demos/demo2x	displays html/demo2.html and adds modifying the html at runtime using a simple replace
demos/demo3x	displays html/demo3.html and adds modifying the html at runtime using SETouterHTML
demos/demoiframe	displays html/demoiframe.html then loads html/left.html and html/right.html into their iframes
demos/menuhw	Displays html/helloworld.html added a 1 item menu

Demos using SM32 fwindow mnemonic

demos/demo2	displays html/demo2.html and adds modifying the html at runtime using a simple replace
demos/demo3	displays html/demo3.html and adds modifying the html at runtime using SETouterHTML
demos/democss	This demo show how to use data uri's.

Demos using SM32 dialog_open

demos/demo2y	displays html/demo2.html and adds modifying the html at runtime using a simple replace
demos/demo3y	displays html/demo3.html and adds modifying the html at runtime using SETouterHTML

protocol 8 demos

demos/window_open	displays html/windows_open.html then based on button click displays html/windows_open_page(1 or 2).html as a pop up (disconnected window)
demos/jsData	sends windows_open_page(1 and 2).html to htmltty as data. Then displays jsData.html which contains a javascript function that displays the 2 pages.

Windows Directory

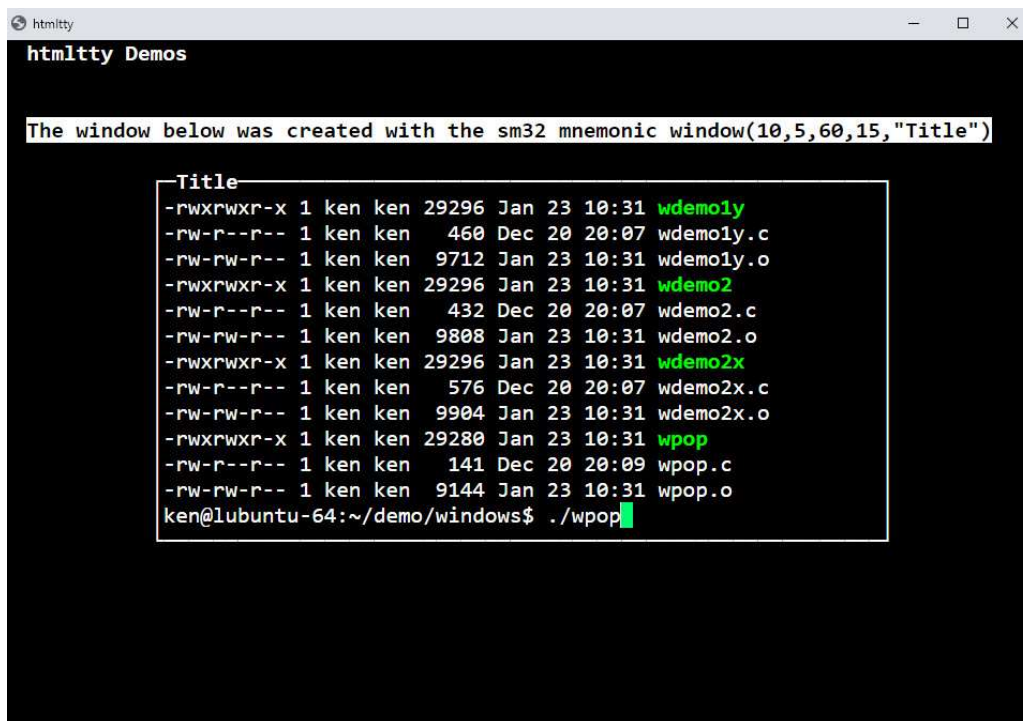
Looking at the windows directory first. The windows demo directory demonstrates the 2 main types of windows used to display HTML. Both of these types of windows are modal windows. The initial window (the VT100 window) is 24 rows by 80 columns. This window cannot be used to display HTML. It is window #0 for mnemonics that require a window number. All other windows can contain HTML. The windows demos display windows in character mode. To get back to window #0 do a `./wpop`. I am not expecting these windows to be used in character mode but just to demonstrate how they are created.

`window(col,row,cols,rows,"Some Title","", "window name")`

Creates a fixed position window that is locked into the col,row of window 0. The title and name are optional.

Add some bash commands and end with `./wpop` - The following screen shots have done a `ls -l` and then typed `./wpop` without an enter.

`./wdemo1`



```
htmltty Demos

The window below was created with the sm32 mnemonic window(10,5,60,15,"Title")

Title
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo1y
-rw-r--r-- 1 ken ken  460 Dec 20 20:07 wdemo1y.c
-rw-rw-r-- 1 ken ken  9712 Jan 23 10:31 wdemo1y.o
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo2
-rw-r--r-- 1 ken ken  432 Dec 20 20:07 wdemo2.c
-rw-rw-r-- 1 ken ken  9808 Jan 23 10:31 wdemo2.o
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo2x
-rw-r--r-- 1 ken ken  576 Dec 20 20:07 wdemo2x.c
-rw-rw-r-- 1 ken ken  9904 Jan 23 10:31 wdemo2x.o
-rwxrwxr-x 1 ken ken 29280 Jan 23 10:31 wpop
-rw-r--r-- 1 ken ken  141 Dec 20 20:09 wpop.c
-rw-rw-r-- 1 ken ken  9144 Jan 23 10:31 wpop.o
ken@lubuntu-64:~/demo/windows$ ./wpop
```

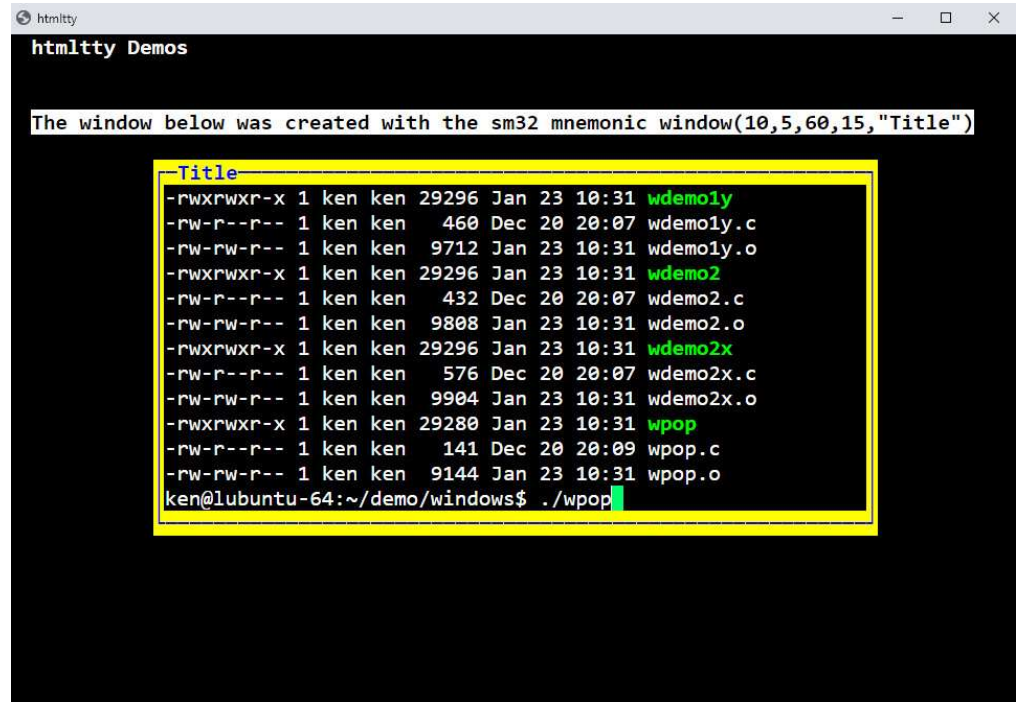
If you enter some bash commands, you will see that bash is stuck inside this window. Bash gets along okay but something like vim not so much. Also note that the usable size of the window is 58 columns by 13 rows. If there had been no title the usable window would have been 60 columns by 15 rows (see `wdemo1y` below).

The special case: `window(-1,-1,-1,-1)`

will create a window that totally covers window 0. You should resize the main window to see it tracks window 0 - as you resize watch the T in Title it will always track the e in below. Also you should notice that if `htmltty` has focus this window has the focus.

./wdemo1x

The second last option to the window mnemonic was specified as “”. It is the attributes for the title. However it is difficult to use from c. So wdemo1x.c shows how to change the attributes before the window mnemonic and change them back after the window mnemonic.



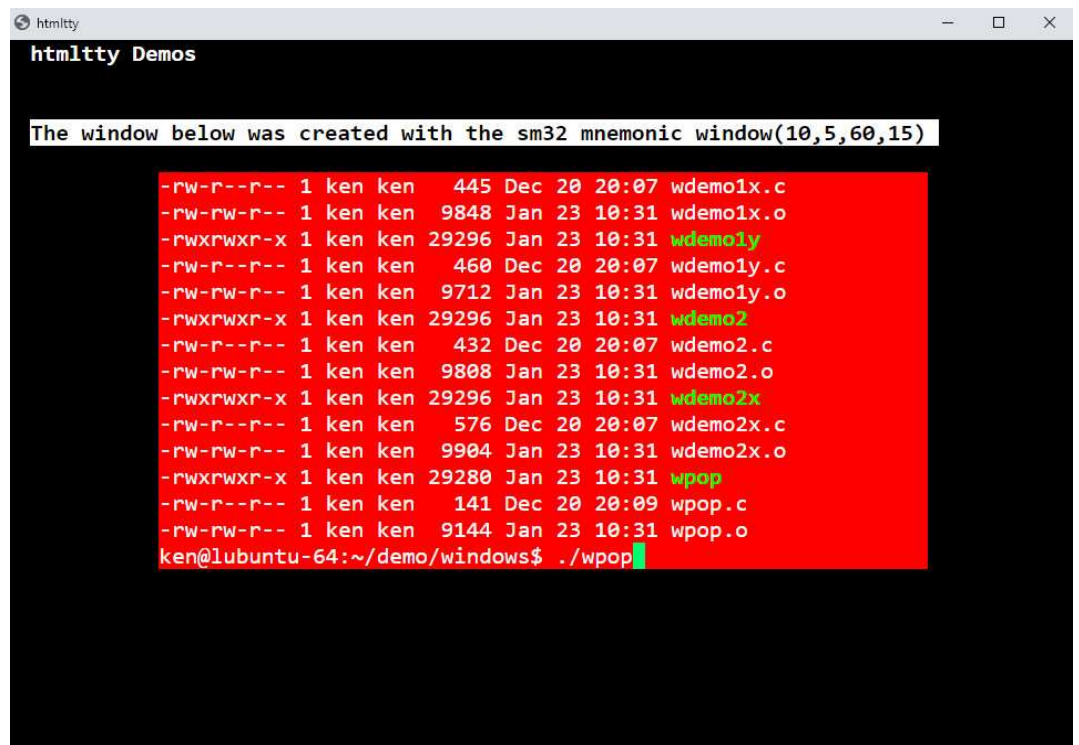
```
htmltty Demos

The window below was created with the sm32 mnemonic window(10,5,60,15,"Title")

Title
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo1y
-rw-r--r-- 1 ken ken 460 Dec 20 20:07 wdemo1y.c
-rw-rw-r-- 1 ken ken 9712 Jan 23 10:31 wdemo1y.o
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo2
-rw-r--r-- 1 ken ken 432 Dec 20 20:07 wdemo2.c
-rw-rw-r-- 1 ken ken 9808 Jan 23 10:31 wdemo2.o
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo2x
-rw-r--r-- 1 ken ken 576 Dec 20 20:07 wdemo2x.c
-rw-rw-r-- 1 ken ken 9904 Jan 23 10:31 wdemo2x.o
-rwxrwxr-x 1 ken ken 29280 Jan 23 10:31 wpop
-rw-r--r-- 1 ken ken 141 Dec 20 20:09 wpop.c
-rw-rw-r-- 1 ken ken 9144 Jan 23 10:31 wpop.o
ken@lubuntu-64:~/demo/windows$ ./wpop
```

./wdemo1y

wdemo1y drops the title and changes the background to red and foreground to white (mostly so we can see the window). You might want to change window mnemonic to have argument of -1,-1,-1,-1 to see window #0 fully covered as you may want in your HTML. Also you may want to try 0,0,80,24 so you see the difference - you may have to resize the window.



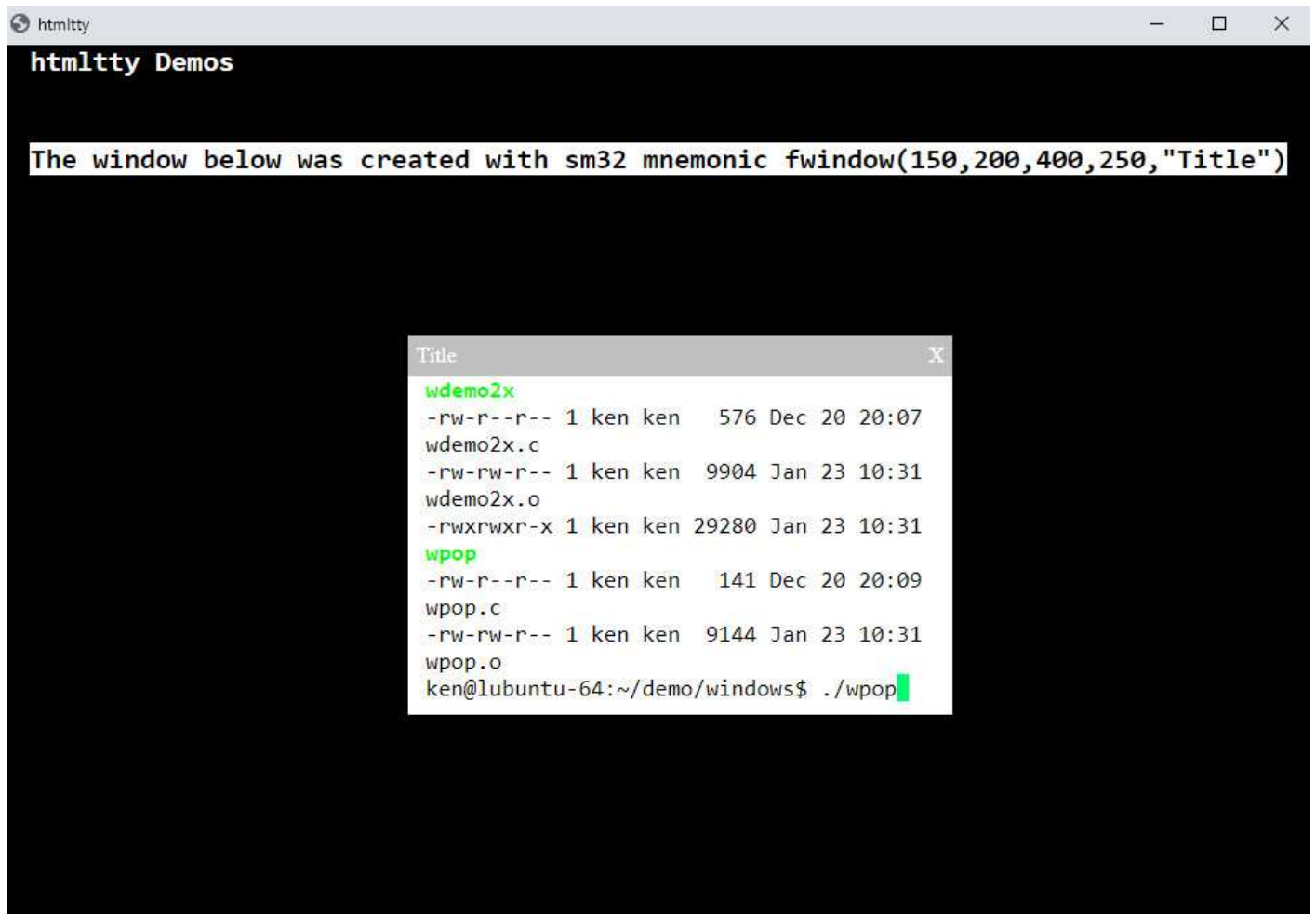
```
htmltty Demos

The window below was created with the sm32 mnemonic window(10,5,60,15)

-rw-r--r-- 1 ken ken 445 Dec 20 20:07 wdemo1x.c
-rw-rw-r-- 1 ken ken 9848 Jan 23 10:31 wdemo1x.o
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo1y
-rw-r--r-- 1 ken ken 460 Dec 20 20:07 wdemo1y.c
-rw-rw-r-- 1 ken ken 9712 Jan 23 10:31 wdemo1y.o
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo2
-rw-r--r-- 1 ken ken 432 Dec 20 20:07 wdemo2.c
-rw-rw-r-- 1 ken ken 9808 Jan 23 10:31 wdemo2.o
-rwxrwxr-x 1 ken ken 29296 Jan 23 10:31 wdemo2x
-rw-r--r-- 1 ken ken 576 Dec 20 20:07 wdemo2x.c
-rw-rw-r-- 1 ken ken 9904 Jan 23 10:31 wdemo2x.o
-rwxrwxr-x 1 ken ken 29280 Jan 23 10:31 wpop
-rw-r--r-- 1 ken ken 141 Dec 20 20:09 wpop.c
-rw-rw-r-- 1 ken ken 9144 Jan 23 10:31 wpop.o
ken@lubuntu-64:~/demo/windows$ ./wpop
```

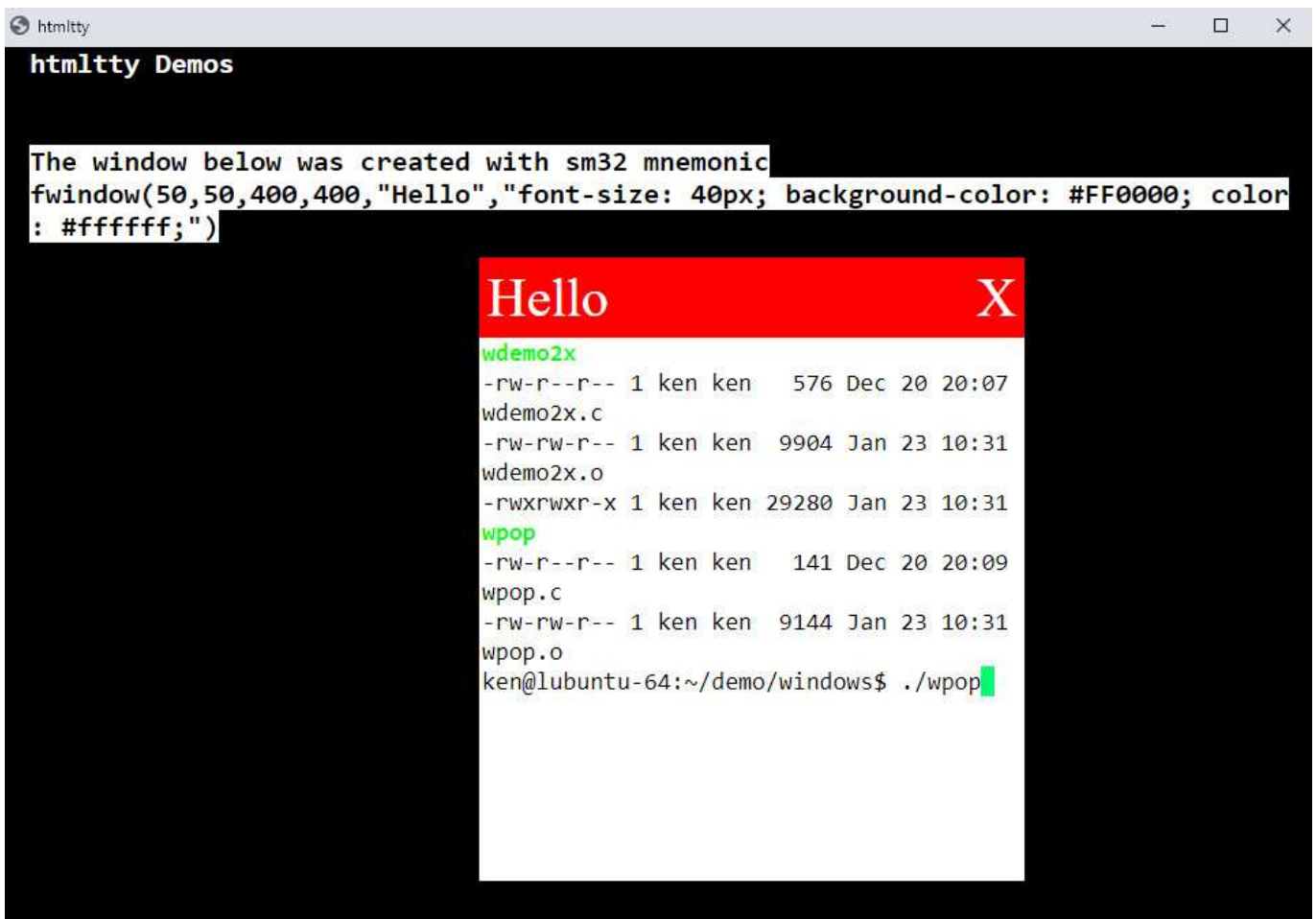
`fwindow(xpos,ypos,cx,cy,"Some Title","Title Bar Style","window name")`
Creates a window at xpos pixels,ypos pixels from the upper left of window 0. The title, style and name are optional.

`./wdemo2`



Although the title is optional, one of the feature of the "fwindow" is to allow the user to drag the window via the title bar. This window was never intended to be used as a character window but was implemented this way to be compatible with "window". In character mode there is always 40 columns by 12 rows. The X in the upper right expects html to be loaded. The size cx,cy does not include the title bar.

./wdemo2x



This demo adds a title bar style. Note that you cannot totally drag the Hello window out side window 0.

Windows Summary

1. You create windows with SM32 mnemonics.
2. window 0 cannot be used for HTML
3. window(-1,-1,-1,-1) completely covers window 0.
4. The last created window has the focus (modal)
5. The current window is deleted with a pop

Not shown here:

1. you can create a new window from any window.
2. you can change window focus with goto(win#) or goto(name)
3. you can delete a window with drop(win#) or drop(name)

Interface

Before looking at the demos a short explanation of the implementation of SM32 mnemonics and HTML display into the VT100 terminal emulation.

The windows created above used SM32 Mnemonics. They were sent using sm32_out that simply warped the unused operating system command (OSC) 99 around the text of the Mnemonic. So the POP mnemonic to delete the current window and go to the previous window was the following stream of characters:

```
"\033]99;POP\033\\"
```

The html interface was designed to pass an associative array to/from the user program to/from java script code (htmltty). SM32 calls an associative array a keyed string array (KSA). The description that follows uses the KSA syntax.

For example `K!["key"]="data"`

All data is characters. The KSA is serialized before transmitting as follows:

```
key\0data\0key\0data\0.....key\0\data\0\0\0
```

to send the KSA to htmltty the following header is prepended to the serialized array:

```
"12351,0,0,0,0,9,"
```

This whole string is then base64 encoded (equivalent of JS btoa) and warped with the OSC 98:

```
"\033]98;base64 encoded string\033\\"
```

If the html command returns something it is a serialize KSA. (there is no header)

The code to handle the KSA received by htmltty is in HTMLwrite.js, in the JS object p.data . It gets checked for a "protocol" value and a "type" value to be valid. Currently there are only 3 protocols 0, 8 and 999 so the interface can easily be expanded. Protocol 0 is the main protocol – it puts HTML into a window. Protocol 8 implements a number of miscellaneous commands. To avoid hanging the window on an error there is also a return protocol of 999 indicating an error:

```
k!["protocol"]="999"  
k!["type"]="ERROR"
```

The currently implemented protocols-type are as follows:

K!["protocol"]="0" protocol 0 put html into a window

K!["type"]="load" or K!["type"]="loadform"
are the only 2 types.

K!["html"]=the whole html document.
K!["entry"]=optional entry point.
The document has an

For type loadform the html document must contain a form with the ID of "THEFORM"

if k!["html"]=" " the current document is left displayed in the window. This could be used after the user is informed of an error in input (via a dialog) and the input is left so the user can edit it.

K!["protocol"]="8"

k!["type"]="window_open"
K!["name"]="some name" (optional)
k!["html"]=the html to display in the window
k!["params"]="comma delimited window parameters"

Parameters are:

height=pixels The height of the window. Min. value is 100
width=pixels The width of the window. Min. value is 100
top=pixels The top position of the window.
left=pixels The left position of the window.

This is a pop up. It has no connections to htmltty once it is created. However, if you name the window, a second window_open command with the same name will replace the html in the existing window. The window can only be destroyed by the user X-ing it closed. See: ./window_open in the protocol8 directory.

K!["protocol"]="8"

k!["type"]="dialog_open"
k!["params"]="comma delimited window parameters"

Parameters are:

height=pixels The height of the window. Min. value is 100
width=pixels The width of the window. Min. value is 100
top=pixels The top position of the window.
left=pixels The left position of the window.

This is a pop up window but htmltty keeps it in focus so it becomes a modal dialog box. It can be lost behind other windows but is always on top if htmltty has the focus. Once created it can be used just like the SM32 window or fwindow – loaded with html via protocol 0. Its main disadvantage is you can not have another dialog from this dialog. See: ./demo2y

K!["protocol"]="8"
k!["type"]="dialog_close"

This closes the window created with a dialog_open


```
K!["protocol"]="8"
```

```
k!["type"]="delete_element"  
k!["ElementID"]="id"
```

This removes the element whose id is "id" from the active html page.

```
K!["protocol"]="8"
```

```
k!["type"]="GETscreenINFO"
```

Returns the following KSA:

```
k!["type"]="screenINFO"  
K!["availHeight"]="available height"  
k!["availWidth"]="available width"  
K!["height"]="current height"  
K!["width"]="current width"  
K!["innerHeight"]="inner height"  
K!["innerWidth"]="inner width"  
k!["outerHeight"]="outer height"  
K!["outerWidth"]="outer width"  
K!["screenX"]="screen X"  
K!["screenY"]="screen Y"
```

```
K!["protocol"]="8"
```

```
k!["type"]="GETouterHTML"  
k!["ElementID"]="id"
```

Returns:

```
k!["type"]="outerHTML"  
k!["Text"]="the full html of the element "id"
```

```
K!["protocol"]="8"
```

```
k!["type"]="SETouterHTML"  
k!["ElementId"]="id"  
k!["Text"]="the full html of the element "id"
```

Replaces the full html of element "id"

```
K!["protocol"]="8"
```

```
k!["type"]="GETvalue"  
k!["ElementID"]="id"
```

Returns:

```
k!["type"]="value"
```

```
k!["Text"]="The value contained in element "id"
```

```
K!["protocol"]="8"
```

```
k!["type"]="SETvalue"  
K!["ElementID"]="id"  
k!["Text"]="value"
```

Sets the value of element "id"

These jsData routines gives access to a javascript object [x.jsData]. The programmer can use this object to pass data between a javascript routine on an html page and the main program. See: ./jsData in the protocol8 directory

```
K!["protocol"]="8"
```

```
k!["type"]="GETjsData"  
k!["key"]="any key"
```

Returns:

```
k!["type"]="jsData"  
k!["str"]=x.jsData[k!["key"]]
```

```
K!["protocol"]="8"
```

```
k!["type"]="SETjsData"  
k!["key"]="any key"  
k!["str"]="any data"
```

Sets:

```
x.jsData[k!["key"]]=k!["str"]
```

Demos Directory

In the previous section it indicated that the HTML communication was implemented by serializing and de-serializing associative arrays. In the C demos the data was just left serialized but the access provided by html.c still looks familiar.

Routine from html.c	SM32	javascript
html_init();	dim k!	let k = []
set_value("key","data");	k!["key"]="data"	k["key"]="data"
get_value("key",&v);	v\$=k!["key"]	let v=k["key"]

html.c provides:

void sm32_out(char *mnemonic)

warps the OSC 99 code around the sm32 mnemonic and places it in output stream.

void html_init()

initialize serial output.

void html_load(char *filename, char **hfile)

will load a complete file into memory. The returned data is in malloc'ed memory so the user must call free.

void html_load_base64(char *fname, char **h)

will load a complete file into memory then convert it to base64 encoding. The returned data is in malloc'ed memory so the user must call free.

void html_replace(char **hfile, char *str, char *str_replace)

will replace str with str_replace in the data *hfile. *hfile should be malloc'ed memory (from html load). *hfile will be returned as malloc'ed data.

void set_value(char *key, char *data)

place key, data in serial output

void html_out()

will do the base 64 encoding and warp the OSC 98 code around the serial output and places it in output stream.

void html_in()

input the base 64 encoded string, decode it and set up for subsequent get_value calls.

void get_value(char *key, char **value)

return pointer the value associated with the key or NULL.

out.c has curses routines and the character output used by html.c

helloworld

html/helloworld.html:

```
<html>
<head>
<style>
body {
    background-color: #C0C0C0;
}
</style>
</head>
<body>
<H1>HELLO world</h1>
<button ID="OKAY" type="button" onclick="x.Click(this)">OKAY
</button>
</body>
</html>
```

helloworld.c:

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "demo.h"

int main()
{
    char *hbp;
    out_open();
    sm32_out("window(-1,-1,-1,-1)");

    html_init(); //init output
    set_value("protocol","0");
    set_value("type","load");
    html_load("html/helloworld.html",&hbp); //load html file as a single
string
    if (hbp == NULL) {
        printf("HTML file not found\n");
        abort(); //can't happen
    }
    set_value("html",hbp); //place in output
    free(hbp); //html_load used malloc
    sm32_out("gwin(\"web\")");
    html_out();
    html_in();
    sm32_out("gwin(\"console\")");
    sm32_out("pop");
    out_close();
}
```

The out_open() and out_close() is just for curses.

```
sm32_out("window(-1,-1,-1,-1)");  
gives us a window that completely covers window 0.  
we then setup the 3 variables:  
    html_init(); // this also establishes the header  
    set_value("protocol","0");  
    set_value("type","load");  
    set_value("html",hbp);
```

The next 3 are important to understand. Until now the window was in character mode. We need the gwin("web") to allow us to execute the html commands.

```
sm32_out("gwin(\"web\")");  
html_out();  
html_in();
```

At this point the html page is displayed. It has **FULL Control**. The VT100 emulator in htmltty can do nothing. The html page has the keyboard it has the focus. You can verify this by hitting cntr C. The helloworld program is hung on a keyboard read – waiting for a serialized KSA.

In HTMLWrite.js the current window gets assigned a value x:
w.x=this

This gives the currently loaded html page access to the object that contains HTMLWrite routine. There is only 2 things of interest to the HTML code:

 x.jsData object see protocol8 jsData example

and

```
x.Click funtion which was used in helloworld.html  
the x.Click(this) caused a return of  
  
k!["type"]="hdata" (HTMLWrite set this when we did the "load")  
k!["OKAY"]="CLICKED" (x.Click got the "OKAY" key from this.id)
```

helloworld.c did not bother checking as this was the only way out of the html page. The next 2 are very important:

```
sm32_out("gwin(\"console\")");  
sm32_out("pop");
```

Until the "pop" htmltty does not have control of the keyboard.

demo1

demo1.c uses some SM32 mnemonics to set window 0's background to red. It then uses the protocol 0, type loadform to display html/demo1.html

```
<FORM ID="THEFORM">  
<B>Enter String:&nbsp;</b><input TYPE="Text" NAME="STRING" SIZE=40 VALUE="Enter something  
Here"><br><br>  
<B>Check or Uncheck this checkbox</b><br>  
<B>Check Box:&nbsp;</b><input type="Checkbox" value="CHECKED" name="CHECKBOX"><br>  
<br>  
<input type="submit" name="DOIT" value=" Do It ">&nbsp;&nbsp;&nbsp;<br>  
<input type="submit" name="CANCEL" value="Cancel">  
</FORM>
```

Note the form has an ID of "THEFORM" this is required. When one of the submit buttons is clicked the form will be returned as key/value the key being the name of the element:

```
k!["protocol"]="0"
k!["type"]="formdata";
k!["STRING"]="the contents of the text box"
k!["CHECKBOX"] = "CHECKED" (value=) if the box is checked
k!["DOIT"]="CLICKED" if it is the submitter
k!["CANCEL"]="CLICKED" if it is the submitter
```

demo2,demo2x and demo2y

These 3 demos are all the same except for the window that is used.

demo2 – uses fwindow

demo2x - uses window

demo2y - uses protocol 8 dialog_open

This demo shows adding run time data by updating the time using a simple string substitution before the html is displayed.

demo3, demo3x and demo3y

These 3 demos are all the same except for the window that is used.

demo3 - uses fwindow

demo3x - uses window

```
demo3y - uses protocol 8 dialog_open
```

This demo shows adding run time data by updating the time using protocol 8 - SETouterHTML.

democss

Since the `html` we send to `htmltty` must be complete this demo shows how to use `data uri`'s for things such as `css`. It loads 3 `.css` files for changing the style, one `html` file to embed and a sound file.

demo i frame

Shows how to have co-operating iframes.

MENU – STATUS LINE

menuhw

The menuhw demo adds a minimal menu (1 item) with the helloworld.html. The menu is built with sm32 the mnemonic **menu**. It also has a status line. When the mouse is over the menu item it can have a comment for the status line.

The status line is controlled by the following SM32 mnemonics:

gwin("statuson") and **gwin("statusoff")**
status("status line content")

'MENU' - Make Menus - from sm32 programmers manual.

The 'MENU' mnemonic gives the basic programmer the ability to maintain named menu's. The 1st parameter to 'MENU' is the command. All commands have at least 2 more parameters. The 2nd parameter is the menu name. If required the 3rd parameter is the position string to specify the position of the item or sub menu.

If required the 4th parameter is sub menu/item information:

for a sub menu "menu,menu name" eg. "menu,&File", for a menu item "item ID#,item name" eg. "item,34000,E&xit"

For items a 5th and 6th parameter can be specified:

5th is the message that appears in the status line if the mouse is over this item - this is "" if not specified

6th is an integer flag bit 1 = checked bit 2 = disabled:

0=enabled

1=checked and enabled

2=disabled

3=checked and disabled ???

"CREATE"

PRINT 'MENU'("CREATE","MYMENU")

Deletes "MYMENU" if it exists then Creates an empty menu named "MYMENU".

"DELETE"

PRINT 'MENU'("DELETE","MYMENU")

Delete the menu "MYMENU" - all its sub menus and items. If the position field is specified, only that Item or SubMenu is deleted.

"SHOW"

PRINT 'MENU'("SHOW","MYMENU")

Makes "MYMENU" the currently displayed menu. PRINT 'MENU'("SHOW","")

Turns the menu off.

"APPEND"

PRINT 'MENU'("APPEND","MYMENU","", "menu,&File")

Adds a submenu named File to MYMENU

PRINT 'MENU'("APPEND","MYMENU","0","item,34000,E&xit","Exit this Program")

appends an item to the 1st sub menu

PRINT 'MENU'("APPEND",,"MYMENU","0","separator")

appends a separator line.

"INSERT"

PRINT 'MENU'("INSERT",,"MYMENU","0,0","item,34001,&Open","Open a File")

inserts an item before the 1st item in the first sub menu

PRINT 'MENU'("INSERT","MYMENU","1,3,4","item,8002,&Save")

inserts an item before the 5th item in the sub menu that is the 4th element(it must be a sub menu) of the 2nd sub menu in MYMENU

"MODIFY"

PRINT 'MENU'("MODIFY","MYMEU","0,1","item,34000,E&xit","Exit This Program",2)

changes the Exit item to disable and grayed condition

"SETFLAG"

PRINT 'MENU'("SETFLAG","MYMENU","", "34000","New Status Message",1)

changes the Message displayed in the status line and checks the menu item that has an id of 34000 SETFLAG is similar to MODIFY but it changes the 6th and 7th parameters based on id number not position, the position string is ignored. If 6th parameter(the status line message) is "" it is not changed

There is another program in demos - `./alldemo`

This program contains a character style menu and a windows style menu that will execute the other demos. The windows menu and the status bar show how to use a menu while in character mode. Note that the menu is using window #0 the vt100 window.

In character mode the menu is implemented as follows:

a control X followed by a string representing the item number followed by cr. control X is disable on the keyboard so it can only come from the menu.

You can insure you do not get unwanted menu input by wrapping the menuon-menuoff sm32 mnemonics around your reads.

Although the MENU mnemonic has a lot of options it is fairly easy to created a menu only using the "append" option.

Note you can have multiple menus loaded and turn them off and on with the "show" option.

MENU("show","") will show no menu.

final directory

The final directory has 1 program `./alldemof`. This is more like a production program.

There is 1 c file in the demo directory - `loadhtml.c`. It gets built and executed in the main Makefile.

```
(make loadhtml)
@./loadhtml final/html
as final/html/html_files.s -o final/html/html_files.o
```

The `loadhtml` program takes a directory as a parameter, it then creates an assembly file that contains the data from all the files in the directory. The file is then assembled into the object file `html_files.o`.

The program `alldemof` links with this file so that once created `alldemof` is stand alone - it requires no data files.

`Alldemof` also get rid of the character menu uses the windows menu in html mode.