GUI SERVER PROGRAMMING MANUAL

INTRODUCTION:

The GUIServer implements a GUI interface for applications continuing the legacy of the various types of GUI that are available. e.g. windows or x-windows. This server is intended for systems that require a small yet efficient implementation of a GUI without too many frills or resource hungry requirements.

This manual introduces the programmer to the methods and ways of writing applications for this GUI.

SYSTEM REQUIREMENTS:

Hardware: -- A VESA compatible video card with minimum 1MB of VRAM is sufficient to support the GUIServer screen resolution of 640 x 480 with 256 colors (This is the resolution we have used).

This package has been tested successfully on the following
Video cards:

SiS 62.. with 4MB VRAM
Cirrus Logic GD5421 with 1MB VRAM .
SiS 6215c with 1MB VRAM

Software:

The GUI library uses the *SVGAlib* package as the underlying graphics primitive. Before the server is run it should be ensured that the SVGAlib library is setup properly by editing the '***/etc/vga/libvga.config***' file.

(The Guiserver system is developed with SVGALIB 1.4.2)

To configure the '*SVGALIB*' work with a VESA compatible video card following changes are to be made in the file '***/etc/vga/libvga.config***'

1. Open the file in any text editor .
2. Go to the Chipset section.
3. Uncomment the line containing the string 'chipset VESA'
4. Comment any other line containing other chipset specification.
5. Near the end of the file and uncomment the line containing VesaText .
6. Save the file and exit the editor .

[Information about svgalib may be found at www.svgalib.org]

HOW DOES IT WORK ?

The GUI or Display Server is an independent process that runs in the background and waits for inputs from its clients or from the foreground input (keyboard) process. Hence the system can be viewed in the following way.

a) The display server:

This is a 'window'-ing program that has control over the monitor and which responds to various 'screen' related events < lines, texts, window creation etc > . This program responds appropriately to the user's input too and forwards them to proper client processes for whom these inputs are meant. System related messages are also sent to the appropriate client in response to events < e.g. screen refresh or redraw etc >

The display server communicates with the outside world via 2 queues.

i. ***REQUEST QUEUE***: This queue is where the clients place their requests to the display server.
ii. ***OUTPUT QUEUE***: The application processes or clients receive feedbacks (in response to certain requests) or input notification, or any other event triggered by the user < e.g. clicking a button, or switching the active window > from this queue.

b) The foreground keyboard process :

This process has control over the keyboard and places the user's keystrokes in the INPUT QUEUE for the display server to pick up.

c) Application or Client processes:

These processes logon to the display server and utilize the capacities of the display server in a manner fit for the application. Requests are sent via the REQUEST QUEUE while output from the server comes from the OUTPUT QUEUE.

Thus the three functional components run independently of one another ... only communicating via messages.

AVAILABLE SCREEN COMPONENTS:

The GUI server has the following components. They are

1. *Window*: A window consists of a title bar , a cross-button and its body .

2. *Button*: Having a text label.

3. *CheckBox*: Contains a square area for check/uncheck mark and a text label

4. *Label*: Displays a label.

5. *Canvas*: A rectangular area on which texts, lines, circles, points , bars (in desired colors) can be drawn .

6. *Scrollable Canvas*: This is also a canvas with the above mentioned facilities but only a part of it can be seen at a time. One can navigate the whole canvas by using the arrow keys.
   This should be used when the canvas size required is greater than the maximum allowable size that can shown on the screen.

7. *EditBox*: This is a data entry box through which one can take text input from the user.

8. *Password Box:* This component is used to enter passwords.

9. *Menu*: Implements the standard pulldown menu with hotkeys.

Please Note:

1.The maximum number of colors available are 256 numbered 0-255 as defined by the palette in VESA mode 640x480x256. This is the mode in which the server is programmed.

2. Note that this GUI system supports keyboard only.

SCREEN COMPONENTS AND WINDOW DETAILS:

COMPONENT AND WINDOW IDENTIFIER INFORMATION:

- The screen is set to a resolution of 640x480x256. i.e. the width of the screen is 640 (0-639) ,height 480 (0- 479) and supports 256 colors.

- For the creation of a component it's parent window must be created first. No component can be created without defining a window to contain it.

- More over if the boundary of any component exceeds that of it’s parent window then it is appropriately clipped.

- The various windows and their components are accessed using identifiers. These identifiers are in the form of unsigned integers.

- Every component that is visible on the screen has a unique id <an unsigned integer> i.e. the identifier value is unique to the concerned window.

- Every component (except the window) should be a child of a unique window <whose identifier is yet another unsigned integer>. Every window has its own unique identifier.

- Also every application that uses the GUI Server sends its process-id with every communiqué. Thus the triad of values

  <component-id>:<window-id>:<process-id>

  is used to identify any screen object uniquely.

Due to optimizations(after version 2.7) that have been made on the server only the component-ids for canvases or scrollable canvases must be unique in a process i.e. if CANVAS1 id is used for a particular canvas then it cannot be reused for any other canvas or scrollable canvas in the process.

- *The regions (rectangular areas) occupied by the different components **MUST NOT** overlap.*

- *Assignment of identifier values is arbitrary excepting when menus are created (discussed later)*

**In all subsequent discussions expressions like WINDOW_1, BUTTON_2 etc refer to the
values (unsigned int) of the respective identifiers.

COMMUNICATION WITH THE GUI SERVER:

- ***Logging On*** : Each client needs an access token to interact with the GUI server . This access token is typically the process-id of the client. This value is returned by the function

```
unsigned GUILogIn().
```

This value is used for future interaction with the server (e.g. message looping and API)

```
e.g. const unsigned hproc = GUILogIn();
```

This hproc is kept for future use.

- ***Communication with the GUI Server*** :
  The communication with the Server is a two way process :

  a. Applications or clients send requests to the Server via the provided API's discussed later.

  b. The server sends back success (SS_SUCCESS) or failure (SS_FAILURE) wherever applicable through the return values of these API's.

  c. The server informs the applications of any change in the windows (or components in the windows) in response to user input or screen updation. The Update message is sent to the client.

The GUI output format on the output/feedback queue is as follows:

```
struct smt{
   unsigned int type A,B,C,D;
};
extern struct smt GuiM;
```

The types of feedback or output (returned via GuiM.type) are:

i)SS_SYSTEM_EVENT or SYSTEM_EVENT : This type denotes that some component or the window itself is sending a message to the client in response to some input
e.g.
checking or unchecking a check-box,
selection of a menu-item etc.

The values returned through variables A,B,C,D are :

A: The id of the window in which the event occured.

B: The id of the component in which the event occured.

C & D : The event .....

e.g.

a)Consider a check box with id CHECKBOX_1 in a window whose identifier is WINDOW_1.

Then if it is checked or unchecked the following Message is recieved

type = SS_SYSTEM_EVENT
A = WINDOW_1
B = CHECKBOX_1
C = SS_COMPONENT_USED

b)If however a window shown to the screen is updated and a particular window is affected then we will have

type = SS_SYSTEM_EVENT
A = WINDOW_1
B = WINDOW_1
C = SS_WINDOWUPDATE or SS_WINDOWSHOW
D = (x,y) coordinates of the left top corner of the window in the higher and lower 16 bits respectively.

c)For a Menu with id MENU_1 in the window WINDOW_1 we would have in response to a user selection.

type = SS_SYSTEM_EVENT
A = WINDOW_1
B = MENU_1
C = Id of the menu item choosen.

ii) SS_FEEDBACK :

This is generally sent when a new window is requested or deleted or a new window component is created or its state is requested to be altered by client processes. This is intercepted by the API calls and the result SS_SUCCESS or SS_FAILURE is returned as the return value of these APIs.

iii) SS_INPUT : This is generated when the focus is implicit on a canvas or scrollable canvas and the user keys in something.

type = SS_INPUT
A = Window Identifier value
B = Category i.e.
SS_NORMAL for ascii ,
SS_CURSOR for cursor keys.

C = Control i.e.
SS_ALT if the Alt keys were pressed
D = keyboard value i.e.
Value of the key hit

e.g. for an combination Alt+A hit in canvas CANVAS_1
in window WINDOW_1 we would have received
type=SS_INPUT
A = WINDOW_1
B = SS_NORMAL
C = SS_ALT
D = 65 (ASCII value of A)

This is also disabled by default. See section on Canvas and Scrollable Canvas

iii. **FOCUS_EVENT : (disabled by default in versions >2.7)** This is generated when focus shifts from
one control to another in the same window.
type = FOCUS_EVENT
A = Window Id .
B = Object Id concerned.
C = SS_GAINFOCUS/SS_LOSEFOCUS
Usually two messages are sent. One with the Object Id
which loses the focus and another with the Object Id
which gains focus.

The various messages are listed in "message.h"

- ***<u>Message Loop:</u>***

The communication with GUI server (as discussed before)
is two fold. Hence there must be a means of communication
between the server and applications (usually for
notification of user input events)...

This is achieved via a non-blocking message loop with a
call to the function

```
GUIOutput(GMChannel,<hproc>,<struct smt *>);
```

This function returns 1/true if any output is available and 0/false otherwise.

If any message is received it is loaded in to the structure passed.

e.g. consider the following code fragment

```
while(1){
   Set_Text_Colors(CANVAS_1,WINDOW_2,hproc,count%255,
                                  backcolor, rand()%MAX_FONT_NO);
     sleep(1);
         Draw_Text(CANVAS_1,WINDOW_2,hproc,100,100,
                                                   " Hello World");

     if(GUIOutput(GMChannel,hproc,&gm))
      if(gm.C==SS_DESTROYWINDOW
                                          &&gm.A==WINDOW_2
                                          &&gm.type==SS_SYSTEM)
        break;
   }// end message loop
```

This code snippet draws "Hello World" in random colors and fonts at position 100,100 in canvas whose id is CANVAS_1, in a window whose id is WINDOW_2 till the user hits the cross button .

It is assumed the GUI server is running when ever an user application is launched...

DETAILS ABOUT VARIOUS SCREEN OBJECTS WITH API DESCRIPTION

Please Note:

-- All co-ordinates unless specified are in terms of screen co-ordinates.

- The variable procid or hproc is used to denote the access token of the client application as discussed earlier .
- **This value is returned by GUILogin() called at the start of the client program only and saved for later use.**

Window

Description:

A window may be thought of a container that contains components like Buttons,Labels,Edit Boxes etc. It contains a cross button and a title bar and has a body that serves as a background for its components.

Upto 100 windows are supported simultaneously.

The window manipulation APIs are as follows:

Window Creation:

The following API is used to create a window on the screen.

int Create_Window(unsigned id,unsigned pid ,unsigned short procid, int x,int y,int w,int h,int stat =1);

where:

- id refers to the number identifying the particular window.

- pid refers to the parent of the window which should be 0 for normal windows ,and the parent window for modal windows.
- procid is the value returned by GUILogin().
- x,y,w,h denote the starting x ,y positions ,width and height repectively.

- stat =1 if a title is reqired and 0 otherwise

Return Value:

- The function returns SS_SUCCESS on success else SS_FAILURE.

Note: - No part of the window created should lie beyond the maximum width and height of the screen.

The other window related APIs are:

Displaying the Window:

To display the created window which is hidden by default we have the following API.

void Show_Window(unsigned id, unsigned short procid);

This will also show the child window (if any)

Hiding the Window:

To hide a window we have the following API.

void Hide_Window (unsigned id, unsigned short procid);

Changing the Window Title:

To change the window title which is 'window' by default use the following API

void Set_Window_Title(unsigned id,unsigned pid ,unsigned short procid,char *text);

Note: The window title must be less than 80 characters

Deleting the Window:

To delete the window from the server the following API is used

<u>**int Delete_Window(unsigned id , unsigned short procid);**</u>

for deleting the window. The API returns SS_SUCCESS or SS_FAILURE.(All child windows should be deleted before the parent.)

Note- Each client should delete all the windows it creates before it terminates otherwise they would remain as zombie windows on the screen which would consume vital resources.

Label

A label is used to display a static text on a window.

Creation:

To create a label the following function is used.

short Create_Label (unsigned id, unsigned pid, unsigned short procid ,int x,int y , int w, int h,char *text);

Here

id:	identifier of the label
pid:	identifier of the parent window
procid:	value returned by GUILogin();
text:	the string (<80 char) that is to be displayed.

Changing the text:

To change the text we use the following API.

int Set_Text_Label (unsigned id, unsigned pid, unsigned short procid, char *text);

Button

Description:

Buttons are window components whose responses are used to trigger events when they are selected and ENTER is pressed.

Creating a Button:

To create a button the following API is used .

short Create_Button(unsigned id,unsigned pid,unsigned short procid , int x,int y , int w, int h,char *text);

where

id : identifier of the Button.
pid :identifier of the window in which the button is to be created.
procid : the value returned by GUILogin();
x,y,w,h :denote the starting x ,y positions ,width and height repectively.
text : Label of the button (must be < 80 characters).

Return Value:

The API returns SS_SUCCESS on success and SS_FAILURE on Failure

For example to create BUTTON_1 in WINDOW_1 at 222,350 labelled 'Exit'

Create_Button(BUTTON_1,WINDOW_1,hproc,222,350,120,25,"Exit");

Note: All the coordinates are given in terms of absolute screen coordinates.

Setting the button text:

int Set_Text_Button(unsigned id,unsigned pid,unsigned short procid,char * text);

Check Box

For Check Boxes we have the following API

CheckBox creation:

To create a new checkbox at absolute coordinates (x,y) with width ‘w’ and height ‘h’ we have the API

short Create_CheckBox(unsigned id,unsigned pid,unsigned short procid ,int ,int y , int w, int h,char *text);

The manipulation routines are:

Getting and Setting the check box status:

short Get_CheckBox_Status(unsigned id,unsigned pid,unsigned short procid, short *pStatus);

short Set_CheckBox(unsigned id,unsigned pid,unsigned short procid, short Status);

Variable description:

Here

id: refers to the identifier of the checkbox.
pid: refers to the identifier of its parent window.
Status: refers to the state 1 for checked 0 for Unchecked states.
Text: refers to label (should be <80 char) associated with the check box

All functions return SS_FAILURE on failure and SS_SUCCESS otherwise

Edit Box

Description:

The Edit Box is used to take in user text information (max 256 characters) .

Creation:

To create an Edit Box we use

int Create_EditBox(unsigned id,unsigned pid,unsigned short procid,int x,int y,int w,int h);

The coordinates 'x' and 'y' are absolute screen coordinates, while the values 'w' and 'h' refer the width and height respectively.
'pid' is the parent window of the edit box.

Return Value:

The function returns SS_SUCCESS on success and SS_FAILURE otherwise.

Setting the Edit Box Text:

It is possible to set the displayed text of the edit box by

int Set_Text_EditBox(unsigned id,unsigned pid,unsigned short procid, char * text);

Retrieving the text in the Edit Box:

To get the text from the edit box to the client application use

int Get_Text (unsigned id, unsigned pid, unsigned short procid, char * Buffer);

Variables:

-Here id and pid refer to the identifiers denoting the edit box and its parent window.

- 'text' and 'Buffer' are the buffers (<256 char) from which text is set and kept respectively.

All functions return SS_FAILURE on failure and SS_SUCCESS otherwise

Password Box

This component is used to accept passwords from the user through a data entry box on the screen. Only the standard editing facilities are available and the text entered is replaced by asterisks.

Creation:

To create a Password Box we use the API

int Create_PasswordBox(unsigned id,unsigned pid,unsigned short procid,int x,int y,int w,int h);

The coordinates ‘x’ and ‘y’ are absolute screen coordinates, while the values ‘w’ and ‘h’ refer the width and height respectively.

Return Value:

The function returns SS_SUCCESS on success and SS_FAILURE otherwise.

Setting Text:

To set the password box to an initial value.

int Set_Text_PasswordBox (unsigned id,unsigned pid,unsigned short procid,char * text);

where ‘text’ refers to the text to be set

Retrieving the text in the Password Box:

To get the text from the password box to the client application use

int Get_Text (unsigned id, unsigned pid, unsigned short procid, char * Buffer);
where ‘Buffer’ refers to the array in which the text is to be kept.

‘pid’ is the parent window of the password box.
‘procid’ is the value returned by GUILogin();

Canvas

The canvas is the most versatile of all the components. It is possible to draw lines, text, bars, circles etc in the desired colors here.

Creation:

A canvas is created using the following API.

int Create_Canvas(unsigned id,unsigned pid,unsigned short procid,int x,int y,int w,int h);

id: refers to the identifier of the canvas.
pid: refers to the parent window id.
x,y: are the Left and Top screen coordinates of the canvas.
w,h: are the width and height of the canvas.

Return value: The function returns SS_SUCCESS /SS_FAILURE as usual.

Shape Drawing:

The following API are used to draw various lines and shapes in a canvas.

*** In each of the following the first 3 parameters are**

i. **id value of the Canvas.**
ii. **id value of the parent Window.**
iii. **procid returned by the GUILogin().**

*** The coordinates that are used to draw are relative to the area in the Canvas.**

The various shapes and their corresponding API are:

Line:

To draw a line from the point (x1,y1) to (x2,y2) use:
void Draw_Line(unsigned id,unsigned pic ,unsigned short procid, int x1,int y1,int x2,int y2, int color);

Bar:

To draw a bar from the point (x,y) with a width ‘w’ and height ‘h’ use:
void Draw_Bar(unsigned id,unsigned pic ,unsigned short procid,int x,int y,int width,int height,int color);

Pixel:

To plot a point with color ‘color’ at (x,y) on the canvas
void Draw_Pixel(unsigned id , unsigned pid ,unsigned procid, int x,int y,int

color);
Circle:
To draw a circle use:
void Draw_Circle(unsigned id , unsigned pid ,unsigned procid, int x,int y,int radius ,int color);
Text:
To draw text on the canvas with selected font we have
void Draw_Text (unsigned id,unsigned pid ,unsigned short procid,int x,int y ,char *text);

In the above 'shape' functions x,y or(x1,x2,y1,y2) refer to the coordinates relative to the left upper corner of the canvas.

Text Colors:
To change the textcolors (fc = foreground, bc = background)
and the current font of the canvas we use

Set_Text_Colors(unsigned id,unsigned pid,unsigned short procid , unsigned fc,unsigned bc,unsigned fno);

The number of fonts available are MAX_FONT_NO = 30 and they are

Font No -	**0**	**to**	**5**	**are**	**“times Roman”**	**pts 10, 12, 14, 16, 20, 24**
	6	**to**	**11**	**are**	**“times (Italic)”**	**pts 10, 12, 14, 16, 20, 24**
	12	**to**	**16**	**are**	**“courier”**	**pts 10, 12, 16, 20, 24**
	17	**t o**	**20**	**are**	**“ arial (Italic)”**	**pts 10, 14, 16, 24**
	21	**to**	**24**	**are**	**“impact”**	**pts 12, 16, 20, 24**
	25	**to**	**29**	**are**	**“ arial”**	**pts 12, 14, 6, 29, 24**

The fonts are identified by numbers 0 through 29, and the text colors are set in the range 1 – 255.

Shifting APIs:
The following APIs are used to shift the contents of a Canvas by a desired amount to the left, right, top or bottom.
void Shift_Up_By (unsigned id, unsigned pid, unsigned short procid, unsigned dy, unsigned color);
void Shift_Down_By (unsigned id,unsigned pid,unsigned short procid,unsigned dy, unsigned color);

void Shift_Right_By (unsigned id,unsigned pid,unsigned short procid,unsigned dx, unsigned color);
void Shift_Left_By (unsigned id,unsigned pid,unsigned short procid,unsigned dx, unsigned color);

Here dx, and dy are the amounts by which the canvas contents are to be shifted horizontally and vertically respectively. The cleared portion is filled up by the color value in the parameter ‘color’.

To set the keyboard messages on/off (default is off)

The following API enables/disables keyboard messages from the server when focus is on the canvas.

void Set_Canvas_Key_State(unsigned id,unsigned pid, unsigned short procid,int keyState);

keyState = 1 enables while keyState = 0 disables the key flow

Scrollable Canvas

This component behaves much like a canvas (excepting the font selection which is fixed). However it defines a logical display area which is larger than the one which is available.
The user will have to move around using the arrow keys.

Creation:

The scrolling canvas is created by:

int Create_ScrollableCanvas (unsigned id, unsigned pid, unsigned short procid, int x, int y, int w, int h, int vw, int vh);

Here

id and pid refer to the identifiers of the canvas and its parent window.
x,y, are the Left and Top screen coordinates of the scrollable canvas
w,h are the width and height of the canvas.
vw,vh are the virtual width and height of the scrollable canvas.
procid is the value returned by GUILogin().

Drawing.

The various shapes possible are

Line:

void Draw_LineSc(unsigned id,unsigned pid ,unsigned short procid, int x1, int y1, int x2, int y2, int color);

Bar:

void Draw_BarSc(unsigned id,unsigned pid ,unsigned short procid, int x,int y,int width,int height,int color);

Point:

void Draw_PixelSc(unsigned id , unsigned pid ,unsigned procid,int x,int y,int color);

Circle:

void Draw_CircleSc(unsigned id , unsigned pid ,unsigned procid, int x,int y,int radius ,int color);

Text:
void Draw_TextSc(unsigned id,unsigned pid ,unsigned short procid, int x,int y ,char *text);

In the above 'shape' functions x,y or(x1,x2,y1,y2) refer to the coordinates relative to the left upper corner of the canvas considering from the start of the virtual screen.

Text Colors:
short Set_Text_Colors_Sc (unsigned id, unsigned pid, unsigned short procid , unsigned fc, unsigned bc);

may be used to set the foreground (fc) and background (bc) colors of the font.

Navigation:
On selecting the scrollable canvas the arrow keys may be used to move around the scrollable canvas.

To set the keyboard messages on/off (default is off)
The following API enables/disables keyboard messages from the server when focus is on the canvas.
void Set_Canvas_Key_State_Sc(unsigned id,unsigned pid, unsigned short procid,int keyState);
keyState = 1 enables while keyState = 0 disables the key flow

Menu

The Menu that is available with the GUI server is like a standard menu supporting hotkeys and various levels of pulldown menus containing text labels .

Each menu is a static, i.e. menu items are not added or deleted dynamically.

It is assumed that the menu-bar is to be placed near the top of the window and there is sufficient space for each popup along the width within each window. This would obviously depend on the text length of the menu-item descriptors.

<u>About assigning identifier values</u>:
Suppose there is a menu with identifier value MENU_1 .If the depth ie maximum level of pull-down submenus of MENU_1 be 'n' then the minimum difference between the ids of MENU_1 and any other component in the same window should be at least ‘n+1’.

Creating a Menu:

Every menu is defined by passing the name of the file that contains the description of the menu through the following API.

short Create_Menu(unsigned id,unsigned pid,unsigned short procid ,int x,int y , int w, int h,char *filename);

Here id and pid refer to the identifier values of the menu and its parent window.
x,y refer to te absolute screen coordinates of the upper left corner of the Menu.
proid refers to the value returned by GUILogin().
'filename' refers to the file containing the menu definition.

Format of the Menu Descriptor file

The format for a menu is explained by a typical menu example:
Suppose the menu definition file "sample.md" contains:

```
MENU: [5];
"Main" [4]{
        "File" [4]{
      "New " [101];
           "Open" [102];
           \SEPARATOR;
           "Exit" [103];
           }
      "Edit" [4]{
                "Cut" [201];
                "Copy" [202];
                \SEPARATOR;
                "Special" [2]{
                    "Zap File" [401];
                    "Reload from disk" [402];
                }
      \SEPARATOR;
      "Talk" [10]{
                "Send 1" [501];
                "Receive 2 "[502];
                "Send 3" [503];
                "Receive 4 "[504];
                "Send 5" [505];
                "Receive 6 "[506];
                "Send 7" [507];
                "Receive 8 "[508];
                "Send 9" [509];
```

```
                 "Receive 10 "[510];
                 }
  }
```

This definition defines the following menu

```
File          Edit                              Talk
Send 1
Receive 2
Send 3
Receive 4
Send 5
Receive 6
Send 7
Receive 8
Send 9
Receive 10
Cut

Copy
-----------------------------------------------------------------


Special >
New

Open
-----------------------------------------------------------------

Exit







Zap File
Reload from Disk
```

Every menu starts with the signature MENU: followed by the total number of menu pages (including the horizontal menu bar). In the example there are 5 menu pages namely main, file, edit, special and talk.

Each page is defined in the following format

```
"<name of page >" [<number of items in the page>]{
      "<name of menu item >" [<IDENTIFIER VALUE >];
      "<name of menu item >" [<IDENTIFIER VALUE >];
      "<name of menu item >" [<IDENTIFIER VALUE >];
      "<name of menu item >" [<IDENTIFIER VALUE >];
            .
        .
        .
       }
```

The < IDENTIFIER VALUE> is an integer value and is returned by the menu to the application process when the corresponding menu item is selected by the user .

The value <number of items in the page> value denotes the number of items in the page

e.g.

The page "File" has 4 items in it, so the corresponding entry is

```
"File" [4] {
        ...
        ...
}
```

If sub-menu pages are required then they are defined recursively. Then the menu-item name is the name of the sub menu and the number in its identifier value is the number of items in its page (as given in the example, Special menu-item has a menu page of its own).

Please note that

i. The number of items in a page comprises of menu-items ,submenu-headers, and separators.
ii. The entire submenu if any contributes only 1 to the total count of menu pages (to be inserted after the signature 'MENU:' at the beginning) .
iii. Separators if any are defined by the keyword \SEPARATOR and these are also taken into account when stating the number of items in the page.

iv. Every menu-page must be enclosed in curly braces i.e. '{}'
 There is no ';' after the closing '}'
v. Every number must be in the format "[<number>]" e.g. [2] or [1098] etc without spaces .

vi. Each label must be enclosed in double quotes containing at least one character .

vii. By default the first character of each label is taken as the hotkey. However this can be overridden by entering in the label string an '&' before the desired character
e.g. if we want Fi<u>l</u>e instead of <u>F</u>ile in the menu we should enter the label for File as "Fi&le".

To set a particular menuitem active or deactive:
void Set_Menu_Item(unsigned id,unsigned pid,unsigned short procid, unsigned Value,short State);
e.g. Suppose the menu in the example is MENU_1 in window WINDOW_1.To deactivate "File"->"New" we would give

Set_Menu_Item (MENU_1,WINDOW_1,hproc,101,0);
since 101 is the identifier value of "New".

To set a particular page active or deactive:

**void Set_Menu_Page(unsigned id,unsigned pid,unsigned short procid, unsigned Value,
short State);**
However to do this we must supply two numbers to the corresponding page entry **without spaces** in between them.

e.g.
If we wish such access for the "File" menu-page in the example then we will have to modify the entry as

```
"File" [4][1234]{
        "New " [101];
            "Open" [102];
            \SEPARATOR;
            "Exit" [103];
  }
```

and the call
Set_Menu_Page (MENU_1,WINDOW_1,hproc,1234,0);
would deactivate the "File" menu.

Value returned to the application:

Suppose "New" <value 101> is selected by the user .Then this is informed to the user by the message structure (explained previously) where

type=SS_SYSTEM

A= WINDOW_1

B= MENU_1

C= 101, i.e. the value associated with “New”

Table Editor

Table Editor is a built in editor that can be used for editing data stored in rows and columns .

The API to create the table editor is as follows

Creation:

short Create_Table_Editor(unsigned id , unsigned pid, unsigned short procid, int x, int y, char *rcfilename);

id :	identifier of the Table Editor.
pid :	identifier of the window in which the button is to be created.
x, y:	denotes the starting x, y screen coordinates of the table editor in the parent window .
rcfilename :	This is the resource filename which contains the necessary information regarding the table editor's structure (i.e. number of columns , rows and their specification)

Saving:

To save the table editor data in a data file the following API should be used :

short Save_Table_Editor(unsigned id, unsigned pid,unsigned short procid, char *datafilename);

“datafilename” is the string containing the name of the datafile in which table editor data would be stored .

Rules for writing the table editor script:

To write a table editor script the following rules should be followed .Failing to do this will produce a script error and the requested table editor will not be created by the GUI Server process.

1. The very first line of the table editor script should contain the signature TABLEEDITOR only.
   (Note :- the script is case sensitive).

1. On the second line the number of rows and columns of the table editor should be specified. The numbers should be separated by a blank.

2. Now the column descriptions should follow. Each column

description has three fields

1. its type (A for alphanumeric N for Numeric)
2. its length .
3. Whether it is locked or not . If the field is locked(Not editable) then the field should contain Y otherwise it should contain N(i.e the column is editable).

Each field should be separated by a blank.

3.a Each line should contain description of one column only.

4. Following the column descriptions the optional loadfilename should be specified as follows.

LOADFILE=loadfilename with fullpath .

If no load file is specified the editor will be filled with blank entries.

Example:

Here is a sample table editor script file.

The table editor has 10 rows 4 columns. First two columns are numeric type of length 10 and 4 respectively .First column is not editable. Last two columns are of type alphanumeric and of length 8 and 25 respectively.

The load file is mydatafile.dat situated at /usr/gui.

```
TABLEEDITOR
10 4
N 10 Y
N 4  N
A 8  N
A 25 N
LOADFILE=/usr/gui/mydatafile.dat
```

<u>**Note :**</u> The data file form which the table editor reads the data
Should be a text file. Each entry must be separated by one space only . Otherwise data read error will generated.
In case of any error while reading the data file (including the above mentioned one) the table all the table editor entries will be blank.
If a numeric entry in the data file contains non numeric characters that cell in the table editor will contain either "ERR" or "-"(depending on the size of the field).

Navigation inside the table editor .

The table editor works in two modes i.e Navigation and editing modes . In the navigation mode ,the cells of the table editor can be explored using the arrow keys. A highlighted bar shows the currently selected cell. To enter into the edit mode the desired cell should be highlighted and ENTER key should be pressed. In the editing mode the cell entries can be altered. In the edit mode DEL, BACKSPACE keys have their usual functions. Pressing ENTER key in the editing mode aborts editing and the table editor switches to navigation mode. If the cell is locked the highlighted bar goes

RED. The locked cells are not editable.

THE OTHER APIs

void Set_Component_State(unsigned id,unsigned pid,unsigned short procid,short State);
This API is used to set the state of an component in a window (except a menu).

short Set_Focus_To(unsigned id,unsigned pid,unsigned short procid);
Sets the focus to the component whose identifier value is id.
Function return SS_SUCCESS/SS_FAILURE .

short Delete_Window_Component(unsigned id,unsigned pid,unsigned short procid);
Deletes the component whose identifier value is id.
Function returns SS_SUCCESS/SS_FAILURE

unsigned Get_Focussed_Component(unsigned pid,unsigned short procid);
returns the identifier value of the currently focussed component in the window having identifier pid.

short Get_Next_Component_Info(unsigned id,unsigned pid,unsigned short procid,unsigned *nextid,unsigned *nexttype);

returns the identifier value of the component next to the component ‘id’ .The last field is reserved.
Function returns SS_SUCCESS/SS_FAILURE

void Shutdown_GUI(void);

<u>shuts down the gui server should be used judiciously.</u>

NAVIGATION

-Pressing ALT+TAB selects different windows one after another.

-Pressing TAB shifts focus from one component to another within a window.

COMPILATION & LINKING:

The following files are required for compiling and linking the client application.

i) message.h & constants.h
ii) request.h & requestlist.h
iii)mesg.h
v) mesg.c
vi) request.cpp
viii. client program files.

The compiling and linking should be done by a C++ compiler.
e.g. Suppose the application program is “client1.cpp” , the desired output file is “client1” and we are using the GNU C++ compiler(g++),then the following statement would do the required.

```
$  g++ -o client1  client1.c mesg.c request.cpp
```

assuming the header (.h) files are present in the working directory.

This process may be run from the prompt by the following command:

```
$ ssgui < keyboard handler >  client1
```

for instance if ‘mq1’ is the keyboard handler then we would give the following command:

```
$ ssgui mq1  client1
```

Here we assume that ssgui,mq1 and client1 programs are in the current directory.

The source files of the GUI Server are (in alphabetical order).

* Function prototypes are defined in the .h files

* Function definitions are given in the corresponding .c or .cpp files.

box.c & box.h : Contains the code for various types of box drawing.

constants.h : Contains the defined constants for various screen object types.

*debug.c & debug.h : c*ontains the code for debug messages if DEBUG_PROG option is on.
event.c & event.h: Contains code 0 keyboard strokes and requests.
font.c & font.h: Contains code for drawing text.

font_bdf.c & font_bdf.h: Contains the bitmap of the supplied fonts *fontlist.dat:* Contains bitmaps of the fixed fonts.*guiout.cpp &*

guiout.h: Contains the code of the server output queue messages.

guiprocess.cpp & guiprocess.h: contains code for request arbitration and overall management.

guisetup.c & guisetup.h: Contains code for setting up the initial environment.

keyboardproc.h: Contains keyboard constants.

keymap.h & keyval.h : Contains keyboard mappings.

makefile: Makefile for the server

menu.cpp & menu.h: Contains the code for the menu.

mesg.c & mesg.h: Contains the code for queue handling.

message.h: Defines constants for the 0 system.

mq1.c: Sample keyboard handler.

processinput.cpp & processinput.h: Contains code for receiving input to the server.

readresource.cpp & readtbrc.cpp: Code for reading menu and table editor script files.

request.cpp & request.h:Request APIs from the client defined here.

requestlist.h: Defines constants for the request APIs.

ssgui.cpp: Main file of the GUI Server.

table.cpp & table.h: Contains code for table editor.

vgasetup.c & vgasetup.h: Contains code for setting up the graphics environment.

*winarea.cpp & winarea.h: c*ontains code for handling areas in the screen occupied by the various windows.

winobject.cpp & winobject.h: Contains code for the behaviour of the windows and window components.