Graphical User Interfaces
Using Objects in a Real-time Environment

A master thesis

by

Stefan Zivkovic

November 1999

ABSTRACT

The objectives of this master project are to design a Graphical User Interface (GUI). The design was made right from the lowest level of hardware interaction up to the point were windows and graphical objects exists. Since this is a very large task, the work has been limited to match the work of a general master thesis. The master thesis covers the design of the GUI and will discuss some of the choices that have to be made during the work. The result of this work is a complete GUI system running in OSE a Real-time operating system from OSE Systems/Enea Data AB. The project has been divided into three parts: the hardware interface using OSE BIOS driver system, the application program's interface (API) with all the functionality that the programmer needs, and finally an object-oriented engine with a set of classes that can be used to build a graphical user interface. The master project should also result in a working prototype running on a PowerPC 821 with a color screen (TFT display).

The result of the master project runs both on the hardware above and in an emulator using the Win32 API. It is both easy to port to new hardware and easy to extend with new functionality.

TABLE OF CONTENTS:

PREFACE

I want to thank some people that have helped me with this master thesis and that also have supported me during my work. First of all I would like to thank Måns my child who was born during the work of this master thesis and Linda Andersen my girlfriend for her support and her understanding during the late evenings. Other important persons are my instructors Jan Eric Larsson (Department of Information Technology), Ola Nilsson (Enea Data AB, IS Malmö RTOS) and Jonas Kallmén (Enea OSE Systems AB). During my work I have come in contact with different persons during different circumstances and I would like to thank them (in alphabetic order):

Adrian Leufvén (Enea OSE Systems AB)
Anders Carlsson (Enea Data AB, IS Malmö RTOS)
Anders Holmung (Enea Data AB, IS Real Time Systems)
Andreas Ronge (Enea Data AB, IS Malmö Open Systems)
Carl-Johan Weiderstrand (Enea Data AB, IS Malmö Embedded Systems)
Daniel Jönsson (Student LTH)
Erik Persson (C-Tech)
Erik Sundström (Enea Data AB, IS Malmö Open Systems)
Håkan Persson (Enea Data AB, IS Malmö Embedded Systems)
Jan-Erik Malmquist (Student LTH)
Jonas Hjelmström (Student LTH)
Lars-Gunnar Lundgren (Student LTH)
Martin Andersson (DORO)
Peter Ekström (Enea Data AB, Software Components)
Per Böckman (Enea Data AB, Software Components)
Per Sigurdsson (Enea Data AB, Software Components)
Stefan Burström (Student LTH)
Stefan Bylund (Enea OSE Systems AB)
Thomas Goréus (Enea Data AB, IS Malmö)
Viveca Selander (Enea Data AB, IS Malmö Embedded Systems)

And also a big thanks to all other persons that has helped me with my task.

1 Introduction

The objective of this master project is to design a Graphical User Interface (GUI) and to implement a prototype running on a PowerPC with a color display (TFT). The design was made right from the lowest level of hardware interaction up to the point were windows and graphical objects exists. Since this is a very big task the work has been limited to match the work of a general master thesis.

The master thesis covers the design of the GUI and will discuss some of the choices that have to be made during the work. The result of this work is a complete GUI system running in OSE a Real-time operating system from OSE Systems/Enea Data AB witch is divided into three parts: the hardware interface, the application program interface (API) and the object part. On top of this is a set of objects that define the GUI.

The GUI is called Ginger after a small vegetable extensively used in cooking. The name is no acronym or anything similar. It was chosen since I used much ginger in my cooking at the time of the project start.

1.1 The structure of this report

In this report I have decided to divide the chapters in the same way as the solutions. This means that in this chapter ("Introduction") the task is analyzed and an overall solutions is structured. The sub solutions are discussed in each chapter. In the chapter "Defining the Ginger BIOS" the low-level hardware functions are defined. In the next chapter "Defining the Ginger application program interface" the design of programmers interface is described. In the next two chapters "Defining the Ginger class engine" and "Defining the Ginger classes" the object-oriented model and the classes are described. Last there is an evaluation and a conclusion chapter. In the appendix there is an Object Reference section supplying more info about the defined classes, and the last appendix is a reference list.

1.2 What is a real-time operating system?

The operating system (OS) is the program that runs on your computer in the background acting as an interface to some of the hardware. A real-time OS is an OS that will solve your problem in a well-defined time or as it says in the OSE Kernel user guide:

> "A real-time system is a program that must respond to external events within a limited time. A real-time operating system is a platform suitable for supporting real-time applications." [1]

This means that in a real-time OS there exists worst case timelines that really are worst case scenarios. A usual assumption about real-time operating systems is that they have to be fast. This is not the case. The important issue is that there exists a worst case scenario that uses a well-defined time. This is the big difference from a non real-time OS. For example if you save a file in a non real-time OS and it takes longer time then expected you probably would not even react, but if the same delay occurred in an airplane during landing the result of the delay could be fatal. I have based my work on the real-time kernel OSE from OSE System (Enea Data AB). This is a message-based kernel that is very effective.

1.3 What is a graphical user interface?

A user interface is the contact between the user and the machine. It is from the GUI that the human receives all his or her input, mostly visual or sound in today's computers. When the input is in a visual form the interface is called a graphical user interface or shorter GUI. Usually the GUI is what handles the visual parts of your display, showing windows, lines and boxes, etc.

1.4 Analyzing the task

Since the objective of this master project is to design a Graphical User Interface (GUI) and there are no available designs. It has to be designed right from the lowest level of hardware interface up to the point were windows and graphical objects exists.

Defining a GUI from scratch is a big task so the problem has to be limited in some way. The first decision to make is to define what should be handled and what should not be handled by the GUI. The requirements of the GUI have to be defined, and here are some of the design goals that were discussed in an early state of the project.

- The code should be portable.
- Draw primitives should support dots, lines and rectangles.
- Colors should be supported.
- The GUI should have limited support for bitmaps and fonts.
- It should be possible to have overlapping areas (windows).
- GUI Objects should support buttons, dragbars/scrollbars, windows, and checkboxes.
- The GUI should be resizable.

To make the GUI portable, all hardware dependent actions should be made via a set of functions that are well defined, separated from the rest of the GUI and easy to port. This means that there should be some sort of hardware abstraction layer defining all hardware dependent actions, a BIOS driver. The visual objects can also be divided into two different areas, primitive objects like dots, lines and rectangles, and other objects that are a compounded of primitive objects, like buttons, dragbars and checkboxes. The problem is now split up in three parts: to implement the BIOS driver, the primitive objects, and finally the compound objects.

The division into these parts makes it natural to use a 3-layer solution with the already mentioned BIOS driver closest to the hardware. On top of the BIOS is an application program interface (API) layer with basic drawing functionality like drawing dots, lines, rectangles, fonts and bitmaps. It is a API layer. Finally there is the GUI object layer with GUI objects like buttons, checkboxes and dragbars, see Figure 1.

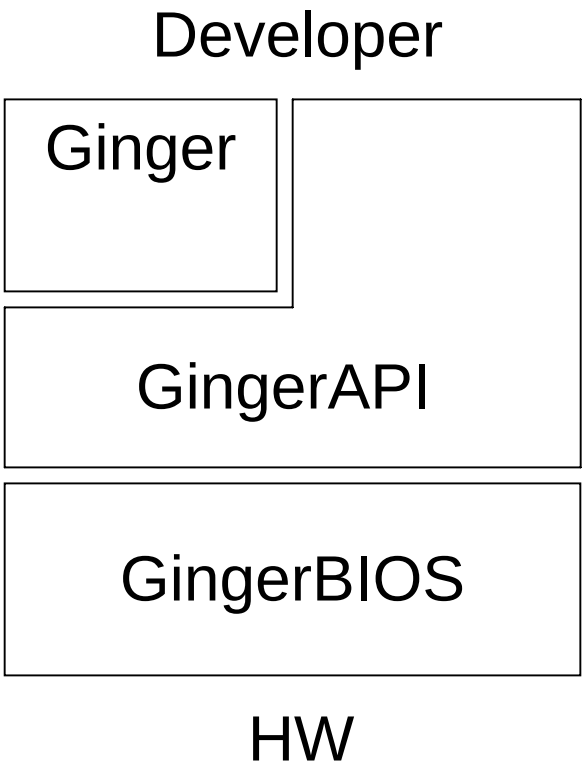

Figure 1. Structuring the solution.

These definitions take care of some of the design goals discussed earlier but leave us with the problem of positioning the window into the correct layer. The window is a primitive but an object at the same time so the position of it in the above 3-layer solutions is not in any way clear, without a new design goal. This new goal is that the user should be able to use the API layer alone, without the GUI objects defined in the object layer. Simple window routines are put in the API layer handling a window without any graphic like title and boarders.

1.4.1 Ginger BIOS driver

The lowest level of the GUI is the hardware level. Since GUI should be able to run on different hardware, there must be some code separating the hardware from the graphics routines. In this project I have used a BIOS driver that handles the display and made it to a simple and generic BIOS driver. This makes it possible to adopt the GUI to different displays. All that this layer knows is the display. All coordinates are according to the display, i.e., 0,0 is the top left corner of the display.

1.4.2 Ginger application program interface

The Ginger API level is the link between the user and the BIOS driver. This will take care of drawareas, bitmaps, fonts, etc. This layer relies only on the Ginger BIOS layer, and will not address the hardware directly in any way. In this way only the Ginger BIOS has to be ported between implementations. All drawing is done inside the drawareas and coordinates are defined relative to the drawarea, i.e., 0,0 is top-left corner of the drawarea.

1.4.3 The Ginger class engine

In my work I have also created a class engine that makes it possible to inherit classes and overload their methods and also add new data fields. It is possible to create custom classes based on old classes. The engine handles classes with their methods and objects with their data.

1.4.4 The Ginger classes

With the class engine I have also made a few GUI objects like dragbars, buttons, checkboxes, etc. It is with the GUI object that the applications GUI's are build.

2 Defining the Ginger BIOS

In this chapter the hardware interface functions are discussed. The functionality should be implemented and how to interface the hardware routines in a hardware independent way is also discussed.

2.1 Functionality

The hardware functions are the functions that will put the pixels on the display by accessing the hardware. The BIOS driver should be easily ported to different processors and different displays. The functionality should also allow hardware-accelerated functions like line drawing and memory copying. With these things in mind I made the decision to only handle dots, lines and rectangles (both filled and not filled ones). This will keep the BIOS driver small without limiting it's usefulness.

2.2 Interfacing the BIOS driver

To be portable the GUI need some code interfacing the hardware. This piece of code should easily be replaced with new code to allow the GUI to be ported to different display environments, like different graphics card. During the design I also had to keep in mind that the hardware interface probably is protected by the processors memory management hardware. This means that code accessing the hardware has to be run in kernel mode were the hardware can be accessed [1]. To accomplish this I investigated a few possible solutions, using a function library or a standardized entry point and using a function table.

A *function library* is a set of predefined functions that are specified and put together in a pre-linked library. This is a very static solution were nothing can be added or changed after the design is done. All code has to be run in kernel mode since the functions access the hardware.

An *entry point* function use a specific function that is called with a special argument that defines the action to take, this entry point function then executes the action or call another function. This solution could use some special instruction to switch to kernel mode so that only the entry point function have to be put in kernel mode.

Using a *function table* means that the function pointers are put in a static table where they can be accessed. This also prevents a kernel separation and all code has to be executed in kernel mode. This solution is hard to maintain and add new functions to in future version of the interface.

- **Function library**
  Hard to use since the layer above is precompiled.
  Hard to maintain.
  Fast since no extra code is required.
  All code has to be run in kernel mode.
- **Entry point**
  Adds an extra switch statement.
  Consistent, this is how other devices are used in OSE today.
  Could be made running in kernel mode, so memory areas for graphics chips could be put in mmu-preserved memory.
- **Function table**
  Hard to maintain.
  Hard to glue together with the rest of the code.
  Fast since no extra code is required.
  All code has to be run in kernel mode.

Based on these facts I chose to use the entry point method since this makes it possible to have the driver access restricted memory. This method also makes the driver very similar to how other drivers in OSE works, e.g., the serial and ethernet drivers. This basically means that a function is supplied that will handle all calls to the device. The function is then placed behind a TRAP function [1] (which will switch over to kernel mode). Arguments to the function are passed via the OS since it is hardware dependent how function migrates to the kernel side of the OS. The code to switch over to kernel mode is already defined in OSE and is used in other drivers like serial or ethernet drivers. The entry point function in OSE has seven arguments. The first argument is a function code selecting what driver function to call. The rest of the arguments are passed over as arguments to the function on the other side of the kernel barrier. This limits the arguments to the BIOS function to six but this is more than enough and should cause no problem.

2.3 Choosing the color model

The driver should support colors, but how should colors be defined? The color model affects the portability since the displays usually have one of two different color schemes, true-color or paletted colors. These color schemes work quite differently.

The true-color method uses a color value defined by the number of bits per color component. The most used color components in computer displays are red, green and blue. This means that if yellow should be defined in a 8-bit per sub-color system using red, green and blue the color value is **`0x00ffff7f`** (read is **`0xff`**, green **`0xff`** and blue **`0x7f`**).

In a paletted system there is a color palette that translate the color from a pen number to the color value. Defining yellow in this system means that, first the yellow color value is put behind a pen number, e.g., pen 5 uses color **`0x00ffff7f`** (as before). Later the pen number is all that is used to access the color, e.g., 5 instead of **`0x00ffff7f`**.

The different color models have different limitations. For example if a limited amount of colors are used, the paletted system saves a lot of memory, i.e., if 256 colors are used a paletted system only a 1/3 of the true-color system memory. Using a paletted system has the disadvantage that if the display supports true-color the table will be enormous and millions of pens have to be allocated. True color is also used by most of the industry in newer hardware. Based on these facts the BIOS will only handle true color in the format red, green and blue, using 8-bits per color-component in the format **`0x00RRGGBB`**. Thus it is up to the BIOS to convert the color to paletted color if it is needed by the hardware.

2.4 Defining the BIOS functions

Looking at different graphic chips and comparing them, I discovered that most chips/LCD drivers handle pixels. Some of the chips handle lines and rectangle filling, and a few of them handle 3D primitives (triangles). I looked at what would be required by the Ginger BIOS driver and my conclusion was that it should be portable to different hardware, and it should be possible to use special hardware for line drawing etc. The smallest primitive of the BIOS driver is the pixel. To make it simple to port and to make it simple I decided that the BIOS driver should only handle pixels, lines and rectangles. Everything else should be left to higher layers. This means that there will be no support for fonts, bitmaps and 3D drawing in the BIOS driver. Higher layers must handle this.

Power saving can be used by embedded systems to save battery when the display is not active if the hardware allows it. This is also a good function to add. Supporting power save functions means that the driver should support that the display could be turned on or off and put in standby mode this means that the BIOS driver must handle at least these three modes **`DISPLAY_OFF`**, **`DISPLAY_ON`** and **`DISPLAY_POWERSAVE`**. I have also added a **`DISPLAY_NOTSTARTED`** to reflect the fact that everything is not initialised. This is the initial state for the BIOS driver after a power on.

Another problem is that the framebuffer access on LCD displays can be very slow to access. The best way to solve this is to write everything to an extra buffer, in faster memory and when done, copy the entire buffer (or the parts of the framebuffer that has changed) to the LCD driver. This problem should be solved by driver implementation.

Here follows a list of functions that I have implemented in the BIOS driver to solve the problems discussed above:

`DisplayBiosInit()`
Makes all necessary initialization of the BIOS driver and the hardware. When it returns the display is in **`DISPLAY_OFF`** mode (from **`DISPLAY_NOTSTARTED`**) and should be ready to power up.

`DisplayBiosExit()`
Sets the display in **`DISPLAY_NOTSTARTED`** mode and frees up anything the driver has allocated. This should be called if the driver is to be removed from the system or if it is going to extreme save mode, all used resources are removed.

`DisplayBiosGetDisplayData(*width,*height,*depth,*xRes,*yRes,*status)`
This function is used to get info from the driver, like the display width, height, resolution and status (power status). Each input variable is a pointer, were an unsigned 32-bit value is written. If any of the pointers are zero no data is written. In this way you only get the data that you needs.

DisplayBiosPowerStatus(pstatus)
This function is used to change the display power status to one of **DISPLAY_OFF**, **DISPLAY_ON** or **DISPLAY_POWERSAVE** as mentioned before.

DisplayBiosFlush()
Since most LCD hardware has extremely slow framebuffer access, most drivers will probably draw in a separate area before copying to the display. This is also a useful method to avoid flickering of the output. If the driver has an extra buffer, it is copied to the hardware framebuffer when this method is called.

DisplayBiosSetColor(color)
Change the color to use. The color value is in the standard 24-bit format 0x00RRGGBB.

DisplayBiosSetDrawMode(drawmode)
Change the drawmode to one of **DRAWMODE_NORMAL**, **DRAWMODE_DITHER** or **DRAWMODE_XOR**.

DisplayBiosSetClipRect(x1,y1,x2,y2)
The BIOS drawing routine controls all drawing against the display borders. This command will shrink this area so that a smaller area is used. This allows the same clipping routine to be used on windows also since the borders can be shrunken to match a window borders.

DisplayBiosLine(x1,y1,x2,y2)
Draws a line between (x1,y1) and (x2,y2), clipping against the clipping area.

DisplayBiosDrawRect(x1,y1,x2,y2)
Draws a frame of a rectangle between (x1,y1) and (x2,y2), clipping against the clipping area.

DisplayBiosFillRect(x1,y1,x2,y2)
Draws a filled rectangle between (x1,y1) and (x2,y2), clipping against the clipping area.

DisplayBiosSetPixel(x,y)
Sets a pixel on the screen at the (x,y), clipping against the clipping area.

These few routines form the basic support for all other routines. Since most objects are built of pixels all other drawing can be built on top of this simple interface.

2.5 Defining the entry point

Defining the entry point function from these sets of functions is quite simple. All that has to be done is to define a function code that map to each function and make an entry point function that checks the function code and finally make a call to the right function. There are also an extra entry in the magic numbers that is used to check if this really is a Ginger BIOS driver by returning a predefined key. Here follows an example of how it is implemented:

```
enum device_dbd {
      DEVICE_DBDDEVCLASS=0,
      DEVICE_DBDINIT,
      DEVICE_DBDEXIT,
      DEVICE_DBDGETDATA,
      DEVICE_DBDPOWERSTATUS,
      DEVICE_DBDFLUSH,
      DEVICE_DBDLINE,
      DEVICE_DBDDRAWRECT,
      DEVICE_DBDFILLRECT,
      DEVICE_DBDSETPIXEL,
      DEVICE_DBDSETCOLOR,
      DEVICE_DBDSETDRAWMODE,
      DEVICE_DBDSETCLIPRECT
      };

long
DisplayBios_biosentry(unsigned long fcode, long arg1, long arg2, long arg3, long
arg4,long arg5, long arg6)
{
      switch (fcode)
      {
            case DEVICE_DBDDEVCLASS:
                  return 0xDBD0734a;   /*Is this a Display BIOS driver?*/
```

```
            case DEVICE_DBDINIT:
                  return DisplayBiosInit();
            case DEVICE_DBDGETDATA: return
                  DisplayBiosGetDisplayData(arg1,arg2,arg3,arg4,arg5,arg6);
            case DEVICE_DBDPOWERSTATUS:
                  return DisplayBiosPowerStatus(arg1);
            case DEVICE_DBDFLUSH:
                  return DisplayBiosFlush();
            case DEVICE_DBDLINE:
                  return DisplayBiosLine(arg1,arg2,arg3,arg4);
            case DEVICE_DBDDRAWRECT:
                  return DisplayBiosDrawRect(arg1,arg2,arg3,arg4);
            case DEVICE_DBDFILLRECT:
                  return DisplayBiosFillRect(arg1,arg2,arg3,arg4);
            case DEVICE_DBDSETPIXEL:
                  return DisplayBiosSetPixel(arg1,arg2);
            case DEVICE_DBDSETCOLOR:
                  return DisplayBiosSetColor(arg1);
            case DEVICE_DBDSETDRAWMODE:
                  return DisplayBiosSetDrawMode(arg1);
            case DEVICE_DBDSETCLIPRECT:
                  return DisplayBiosSetClipRect(arg1,arg2,arg3,arg4);
            default:
                  return DEVICE_EUNKNOWN;
      }
}
```

In OSE the **`biosInstall()`** functions will install the BIOS driver for you.

```
biosInstall("DISPLAYDriver", DisplayBios_biosentry, 0);
```

This will install the BIOS driver and you can later access it via a handler that can be fetched via the **`biosOpen()`** function like this:

```
handle = biosOpen("DISPLAYDriver");
```

To call the code, all you have to do is use the **`biosCall()`** function like this:

```
biosCall(handle, DEVICE_DBDSETPIXEL , x, y);
```

2.6 Comments

The Ginger BIOS driver is very simple. This allows the driver to be easily ported to new hardware and new displays. But simplicity has a price, and it is paid in speed. The simple design and the fact that the TRAP instruction adds some complexity, all drawing that uses more then a dot, line or rectangle are forced to use dots, and each dot has to switch from user mode to kernel mode which costs a lot of time. This means that live video, images, and text have to use many TRAP instructions before the result is finished. This issue is easily fixed in a few working days just by adding more functionality to the driver. There is no support for 3D but the model is easily expanded with more functions.

In the next chapter the layer between the programmer and this BIOS driver is discussed.

3 Defining the Ginger application program interface

The definition of the Ginger application program interface (API) is that this is what the application programmer will interface to.

The API layer will use the BIOS layer. If all access to the hardware goes via the BIOS driver from the API layer, the Ginger GUI will be easy to port since the programmer only needs to write a new BIOS driver for the new hardware.

3.1 Functionality

The API layer is the programmers interface. This means that the API in its simplest version would only be a wrapper around the BIOS functions. The API layer will then be hard to use and does not add any extra functionality and therefor only slowing down all function calls. It is in the API layer that things not handled in the BIOS layer is handled (up to the level of the simple windowing system). This means that the API layer will handle drawareas (windows), fonts and bitmaps in addition of the already discussed pixel, line and rectangle drawing routines in the BIOS layer.

3.2 Drawarea

The drawarea is an extremely simple version of the traditional window. It can be overlapped and has no graphics like title and borders. It is simply an empty window. The drawarea has a position and a size. Inside the drawarea all drawing is performed, e.g., placing a dot, line or rectangle, or drawing a bitmap or printing some text. The color and the drawing attribute (Normal or XOR mode) can be changed and should not conflict between drawareas, e.g., if one drawarea changes its color to red, drawing in the other drawareas should not change to red but use their old colors.

When drawing in the drawarea the same color and attribute are usually used for a series of drawing function calls. The color and drawing attribute should therefor be sticky for the drawarea. This means that the color and the drawing method are first saved in each drawarea and then set with a BIOS call. When a drawing command is executed, the API layer has to check if the drawing is in a new drawarea or if the drawing is done in the last used drawarea. If it is new the BIOS driver has to be initialized with the area color and drawing method used by the new drawarea.

All the drawing function uses coordinates that have to be relative to something. The placement of origo (the 0,0 coordinate position) can be the same for all drawareas, e.g., by using upper left of display, or be different for each drawarea see Figure 2. Using the same origo for each drawarea means that a coordinate will change if a window is moved. This make it very hard to write applications, but can easily be solved by using an origo locked to the drawarea position. The most common way is to use the upper left corner as the origo and using a positive y-axis downwards. This is also the way that I choose to implement the drawarea.

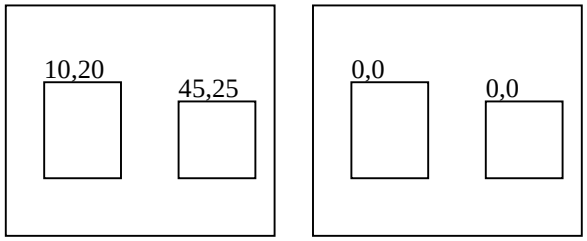

Figure 2. Display relative coordinates and drawarea relative coordinates.

Creating a drawarea is simple. All that is needed is the position and the size of the drawarea to be created. To remove the drawarea only the drawarea handle has to be supplied.

Here I will discuss some of the functions in the interface:

```
DrawArea * AllocDrawArea(int dx, int dy, int sizeX, int sizeY);
void FreeDrawArea(DrawArea *DA);
```

The drawarea is put on top of the others drawareas and can be moved and resized with a few function calls. The **`GetDrawAreaData()`** supplies pointers to 32 bits memory positions where the data is stored.

```
void GetDrawAreaData(DrawArea *DA,int *dx,int *dy,int *width,int *height);
void SetDrawAreaData(DrawArea *DA,int  dx,int  dy,int  width,int  height);
```

To rearrange the drawareas I have added functions to bring the drawarea to the front and to bring it to the back.

```
void BringToBack(DrawArea *DA);
void BringToFront(DrawArea *DA);
```

To alter the color and to change the draw attribute there are also two different functions:

```
void SetColor(DrawArea *DA,unsigned int Color);
void SetDrawMode(DrawArea *DA,int drawmode);
```

3.2.1 Drawing inside the drawarea

Drawing inside the drawarea should be like drawing on a smaller display. The coordinate (0,0) is the top-left position of the drawarea and the size is equal to the size of the drawarea. The drawing functions from the BIOS driver are mapped to work with the drawarea, to do this they have to convert the coordinates to the display before executing the equal BIOS function. These functions are converted BIOS driver function:

```
void SetPixel(DrawArea *DA,GCoord x1,GCoord y1);
void Line(DrawArea *DA,GCoord x1,GCoord y1,GCoord x2,GCoord y2);
void DrawRectangle(DrawArea *DA,GCoord x1,GCoord y1,GCoord x2,GCoord y2);
void FillRectangle(DrawArea *DA,GCoord x1,GCoord y1,GCoord x2,GCoord y2);
```

3.3 Font

Since the scope of this master project is restricted to implement limited support for fonts, the font handler is quite simple. First we need a function to choose a font. In this simple implementation fonts are defined to magic numbers like:

```
#define Helvetica_8  0x00000001
#define Helvetica_12 0x00000002
#define Courier_8    0x00000003
```

Fonts are opened with call to **`GetFont()`**. Like this:

```
GFont *fnt = GetFont(Helvetica_12);
```

This function call will fetch the font associated with the magic number from the system. The returned handle is used later in the text output call:

```
void WriteText(DrawArea *DA,GFont *fnt,char *Text,U32 len,GCoord x1,GCoord y1);
```

The **`WriteText()`** function will use a drawarea to write in, a font and the length of the text. If the lenght is zero the function will rely on a zero terminated string. And the last two arguments describe the coordinates of the baseline of the text. Here is an example typing "hello, world"[3] on position 10,10 in the drawarea:

```
WriteText(DrawArea,fnt,"hello, world",0,10,10);
```

3.4 Bitmap

Since the scope of this master project is to implement limited support for bitmaps, the bitmaps in Ginger are quite simple. There is only one type of bitmap and this is the monochrome bitmap using one bit per pixel. When drawing a bitmap it has to be converted to a format that then is used by the display. Since most bitmaps are drawn more then once I have divided the drawing into two functions instead of one. The first converts the bitmap to a native format and the second will draw the converted bitmap.

```
BitmapData *AllocBitmapData(GData format,GCoord sizeX,GCoord sizeY,void *bitmap);
```

This function will convert a bitmap and return a handle to it, which can be used later when drawing the bitmap. The first argument is the format the bitmap is using. Right now only monochrome bitmaps are supported. After this the size of the bitmap and a pointer to the memory area where the bitmap is stored are passed as arguments.

```
void FreeBitmapData(BitmapData *BM);
```
This function removes the handle and frees the memory used by the bitmap.

```
void DrawBitmap(DrawArea *DA,BitmapData *BM,GCoord x,GCoord y);
```
This function will draw the bitmap in the drawarea at the supplied coordinates.

3.5 Clipping

Since drawareas can overlap there has to be some way to clip the drawareas against each other. I have solved with a list of visible rectangles. This means that the display is made up of visual rectangles that are used whenever something is drawn. When a line is drawn it is drawn several times but with different visual rectangles set in the BIOS driver. In this way it is only visible inside the visual rectangles see Figure 3.

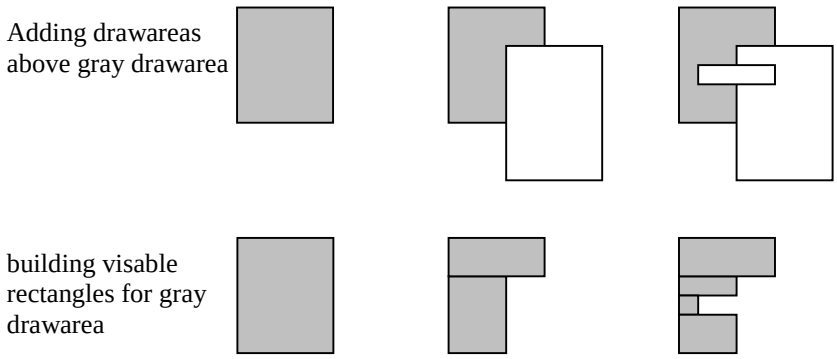

Figure 3. Generating the visual rectangles for the gray drawarea when new drawareas are added above it.

3.6 Comments

The Ginger API is only a simple implementation of the functionality of the display but works for the usual drawing primitives like dots, lines, rectangles, text, and images. The clipping is not very fast during the setup. For each drawarea that is created all other old drawareas clipping lists has to be updated, and this take some time.

In the next chapter the design of an object-oriented solution for the GUI objects is discussed.

4 Defining the Ginger class engine

In this section the reasons for and implementation of an object model are discussed. How to create and invoke methods and how to change attributes on objects is also discussed. Everything is done using the non object-oriented language C.

4.1 Functionality

The functionality of the class engine is that it should act like an object-oriented language. It should be possible to make new classes based on old classes, override, and add new methods and access attributes. Also, methods should be polymorphic.

4.2 Using an object-oriented solution

The implementation of the visual objects has in my experience, shown that an object-oriented approach is to be preferred. One of the most common object oriented examples is a GUI with its visual objects.

Creating an object-oriented environment for a GUI is easily done in some of the object-oriented languages, like Java, Simula, or even C++. But using an object-oriented language is not always possible. For this argument I will divide the solution into four different areas using:

- Simula, Smalltalk or other object oriented languages, except Java and C++.
- Using Java.
- Using C++.
- C using my own object-oriented environment.

During the following discussion the runtime aspects should be kept in mind. The GUI should be running in a real-time environment, running on different targets (with targets I mean hardware configurations, sometimes even with different processors), often in a hardware-wise cost-effective way. The memory is often limited.

4.2.1 Using a object oriented language, except Java and C++

The first approach that uses Simula, Smalltalk or other object-oriented languages, except Java and C++ are not practical for a number of reasons. There are not compilers for all targets, meaning that the compilers have to be ported. All of the system functions are interfaced via C or Assembler. A few of these languages require some code that during runtime fixes things, like garbage collection. This could be a big cost to the system if no other code is using the runtime environment.

4.2.2 Using Java

The second approach uses Java. The Java virtual machine has to have a runtime part to handle the garbage collection. The Java virtual machine does not exist for all targets so almost the same arguments are valid here as in the prior section. Another way to use Java is to use a Java compiler compiling not to Java byte code but to the target processor. There exists at least two such compilers, Diab, a commercial compiler, and gcc (gcj), a freeware compiler. Java has a Java native interface (JNI) to handle standard function outside the Java environment. Using Java means that the language of the GUI is limited to Java.

4.2.3 Using C++

The third approach uses C++ and has very few flaws. Compilers are no problems and interfacing C is not a problem either. Interfacing the C++ code on the other hand is a bigger problem, but C wrappers could be created solving this problem. Using C++ means that the language used to write programs under the GUI is locked to C++.

4.2.4 Using C

The final approach is using C and making a new object-oriented environment. This demands more effort but will result in a GUI that can be interfaced both from Java and C++ but also requires neither of them. BOOBSI for Amiga OS works in this way se [5].

4.2.5 Conclusions

From the facts above I draw the conclusion that using Simula, Smalltalk or other object oriented languages is out of the question, because the problems are too big. To use Java would have been nice but not very practical since the rest of the system is in C. Compiled Java is a better solution but the compilers are still in beta stage, and the benefits using this solution compared to the problems makes this a big risk. Using C++ is a nice solution but using C is better since most projects in the embedded area still uses C instead of C++. Using C++ limits the compiled code to one single compiler solution, while the C solution is usable from all other languages. The reason for this is that the object model is not locked during compile time and the linking phase is well defined. Therefore I have chosen the C approach, which means that I had to construct an object-oriented environment in C that implement basic object-oriented features.

4.3 Designing a basic objecte-oriented environment in C

What object-oriented features is the object-oriented environment expected to have? It is obvious that the environment must handle classes and their instantiated objects. The basic functionality to create classes and objects and trigger the object functions is also required. An object should also have the ability to contain data. Since there could be many instances of the class this must also be handled by the model.

Another feature that is one of the main reasons for the decision to use an object-oriented environment is the ability to base new classes on older by just adding or changing some functionality (polymorphism).

To summarize I have listed the discussed features below. These are the main features that are on the wish list. Another feature that is nice to have is data abstraction. Features like multiple inheritance are not features that I think will make the environment better. Usually a design can be made in a better way without using multiple inheritance. Thus I have decided to handle at least these features:

- Classes and object instances of these classes.
- Class inheritance.
- Polymorph methods.
- Data abstraction.

4.4 The class and object model

To be able to handle objects I need some sort of structures that (behind the scene of the user/programmer) will handle all the things that the model should support, like creating classes, creating objects, invoking methods and so on. Since everything is supposed to work from a C environment, everything has to be done in runtime therefore functions have to be added to support this. Usually the compiler handles this in object-oriented language. These functions will operate on special structures that are hidden from the user/programmer. During the design phase I studied a few ideas on how to build the object-oriented environment. The binding and method calling preformed runtime is similar in how Smalltalk and Objective-C solves the object-oriented schemes.

For this discussion I will begin with defining the minimal configuration needed. Since the class should be able to be access its parent class, the minimum configuration is:

```
struct GingerClass
{
      struct GingerClass      *Parent;
};
```

Since classes should handle different sizes of data there must be some way to determine the size of data each class uses:

```
struct GingerClass
{
      struct GingerClass      *Parent;
      unsigned int            DataSize;
};
```

Then we create the object. The minimum configuration is that the object should contain the data and be able to find its class:

```
struct GingerObj
{
      struct GingerClass*     Class;
      unsigned char           data[<datasize>];
};
```

There are two things to discuss at this stage about the data field of the object. First of all is how to define the data in the model. This could be handled in a few different ways:

- One data pointer for each class struct
  `unsigned char     *data[<number of parent classes>];`
- One big chunk of data, like
  `unsigned char     data[<size of all parent classes data>];`

I have chosen the later since it is not only faster to access the data but also consumes less overhead, both in terms of speed and memory control code.

Secondly, since I use C, the size of the data is not known at compile time. The size can only be calculated at runtime since the number of parent classes is not known. This is easily handled when the object is created by just adding all the parent classes **`DataSize`** fields, then adding the **`sizeof(struct GingerClass*)`** and finally allocating the amount of memory needed.

Using this solution there must be a way to index and step through the data space. For this I have to add a field in the class that will point out were in the data space of the object data space this class object data is.

```
struct GingerClass
{
      struct GingerClass      *Parent;
      unsigned int            DataOffset;
      unsigned int            DataSize;
};
```

This simple model handle classes, which could be in a tree structure and to make objects of the classes that contain data. With this we can handle data in objects, but not methods.

4.5 Adding methods to the model

Since the classes and objects are built in runtime instead of compile time there must be a way to access a method. I have chosen the easiest way by just defining a magic number for each method. There are a few ways to add methods to the object:

- One big function table, like C++ compilers often do.
- A dynamic function table.
- An entry point function with a switch/case like statement.

A big function table means that the numbers of the methods are used to index a function pointer array. This is not in any way desirable. Since the number of entries is locked it is impossible to add extra methods after the limit is reached. If the method numbers are not packed (for example using methods numbers 1,2,3,4,37,100) it will consume more memory then necessary.

The dynamic function table is much nicer since it will solve the problem with adding methods and not having to care if the numbers of the methods are packed. One problem with this dynamic function table is that it will consume more memory than an optimal version of "one big function table", since it also has to keep track of the method numbers.

The last method that uses an entry point will also work nicely since each class will have its own entry point function to take care of its own methods. But the entry point solution has a few drawbacks. All method calls will go via all classes entry point functions, instead of as in the dynamic function table where the functions that are not defined in subclasses go directly to the right function. Using the dynamic function table should be the best choice. Defining the dynamic function table is right now done in the easiest way using a dynamic table:

```
struct GingerMethodMatrixEntry
{
      unsigned int MethodVal;         // method magic number
      void *       GMethodFunc;       // function to use
};

struct GingerClass
{
      struct GingerClass              *Parent;
      unsigned int                    DataOffset;
      unsigned int                    DataSize;
      unsigned int                    nbr_of_methods;
      struct GingerMethodMatrixEntry  *MethodMatrix;
};
```

As the case was with the object **datasize** that was counted and allocated during runtime this is also true with the size of the **MethodMatrix** array. Since the method is mapped to a generic function prototype, the number of arguments has to be fixed or a vararg list. But since the vararg list is not very easy to use and it is locked to the C language, using a standard number of arguments is much easier both for the class implementers and the users of the class.

To determine the number of arguments needed I set up a few use cases and the following was determined. In most cases one argument is enough, in some cases you need two, and the rest of the cases you need many. Using many variables every time is not an option since it would be a great overhead those times it is not needed, and it could be simulated with a struct instead. At least two is needed since two pointers are needed in a set-method (see below) which is a vital part of the object-oriented data system used. This means that I choose two arguments.

Using this way of solving the function problem also leads to the positive effect that functions that was not even thought of during the design could easily be added at runtime by expanding the function table and adding the function. This means that classes could add extra methods to themselves during the execution phase.

4.6 Standard methods

To get the object model to work I also needed a few standard methods that could be defined. These are the methods known from C++ as constructor/destructor, setting data, and getting data. Since most objects have to be initialized before they are usable I have choose to split up the initialization in two parts. First the memory used by the structures are allocated, and filled with zero. Then a **GM_NEW** method is called so default values that are not zero can be initialized. After this the user can initialize all the data needing to be initialized, and finally a **GM_OPEN** method is called, in which the object specific data has to be initialized. When an object should be removed the **GM_CLOSE** is first called making the opposite of **GM_OPEN** and last **GM_DISPOSE** that should do the opposite of **GM_NEW**. Below you will find a detailed description of the standard methods:

GM_NEW	This method is the constructor. It is called first, right after the object structure and data is allocated. The object data area is always cleared (filled with zeros) when it is allocated. In **GM_NEW** every data member that should not be zero should be initialized.
GM_DISPOSE	This method is the destructor. It is called last in the object lifetime. In this method all data allocated in **GM_NEW** should be released.
GM_SET	Use this method to set data in the data area. The first argument is a magic number defining the field to set, the second is the value.
GM_GET	Use this method to fetch the object data. The first argument is the magic number defining the field to get, same as in **GM_SET**.
GM_OPEN	After the object is allocated and **GM_NEW** is called, **GM_OPEN** is called to startup the object.
GM_CLOSE	Close the object, this is the opposite of whatever **GM_OPEN** does.

4.7 Support functions

Since the object oriented class hierarchy has to be built during the execution of the system instead of during the compilation stage, most things that are done during compile time in traditional object-oriented languages have to be done at runtime. This means that the class hierarchy and the structures discussed above have to be built. To solve this, some support functions where created. Obvious functions are the class/object maintainer functions like adding and remove classes/objects, and functions to invoke the methods on the objects:

```
GClass *CreateGingerClassA(GClass *Parent,U32 DataSize,DevTag *tagList);
```

This function is used to create new classes. The first parameter is the super class to use, and the second parameter is the size of data that the class will use. The last parameter is a taglist that should be filled with the method numbers and function pointers that should be added/override against the super class. A taglist is an array of unknown size with usually a number of two elements in pair. Using a magic number at the first element position ends the taglist. Often a taglist is constructed so that first element is some sort of magic number defining how the second element should be used.

E.g., to create a class **`MyClass`** that inherits from **`MyParentClass`** and has a data size of 4 bytes, and new methods for **`GM_NEW`** and **`GM_SET`**.

```
unsigned int MyClassTagList[] = {
      GM_NEW,     <pointer to new function>,
      GM_SET,     <pointer to set function>,
      <tag end magic number> };
// ...
GClass *MyParentClass,*MyClass;
// ... code that set/create MyParentClass
MyClass = CreateGingerClassA(MyParentClass,4,MyClassTagList);
```

```
void DisposeGingerClass(GClass *GO);
```

This removes the class from the class hierarchy. Be careful when using this function, since there could be classes defined on this class. If so they have to be removed first.

```
GObj * CreateGingerObjA(GClass *gclass,DevTag *tagList);
```

This function creates new objects. The first parameter is the class to make an object from, the second parameter is a taglist defining what should be initialized in the object. Most objects are different in some way, like a string object should contain different strings or a window object could be on different screen positions. I have found it very useful to be able to set values in the object during the creation of it. This means that the process of creating the object works like this. First the structures are allocated, and the data zeroed. After this the method **`GM_NEW`** is invoked just before the taglist is traversed and a series of the method **`GM_SET`** is invoked on the object. After this **`GM_OPEN`** is invoked. Let us create an object **`MyObject`** from the class **`MyClass`**:

```
unsigned int MyObjectTagList[] = {
      GA_VALUE_2,       37, //Or whatever it should be
      GA_VALUE_1,       54,
      <tag end magic number> };

// ...
GObj *MyObject;
// ...
MyObject = CreateGingerObjA(MyClass,MyObjectTagList);
```

This example will result in the following method calls on the object created, and invoked by the **`CreateGingerObjA()`** function:

```
DoMethod(MyObject,GM_NEW,0,0);
DoMethod(MyObject,GM_SET,GA_VALUE_2,37);
DoMethod(MyObject,GM_SET,GA_VALUE_1,54);
DoMethod(MyObject,GM_OPEN,0,0);
```

For the **`GM_SET`** method to work the class needs to define it so that it handles **`GA_VALUE_1`** and **`GA_VALUE_2`** it is up to the creator of the class to define this.

```
void DisposeGingerObj(GObj *GO);
```

This function removes a ginger object allocated with **`CreategingerObjA()`**

```
U32 DoMethod(GObj *Obj,U32 method,U32 arg1,U32 arg2);
```
This function invokes a method on the object, in C++ it would look like:
```
Obj->method(arg1,arg2);
```
As discussed before there are always two arguments.

To make thing more easily maintained there is also a variable argument (va_arg [3]) version of the create-class and create-object functions:

```
GClass *CreateGingerClass(GClass *Parent,U32 DataSize, ...);
GObj *CreateGingerObj(GClass *gclass, ...);
```

Using these functions makes programming the interface mush easier especially when objects are used inside other objects like in most applications. Object can then be created dynamically in one big function call. Even the simple example above becomes easier:

```
GClass *MyParentClass,*MyClass;

// ... code that set/create MyParentClass

MyClass = CreateGingerClassA(MyParentClass,4,
      GM_NEW,     <pointer to new function>,
      GM_SET,     <pointer to set function>,
      <tag end magic number> );
GObj *MyObject;
MyObject = CreateGingerObjA(MyClass,
      GA_VALUE_2,       37, //Or whatever it should be
      GA_VALUE_1,       54,
      <tag end magic number> );
```

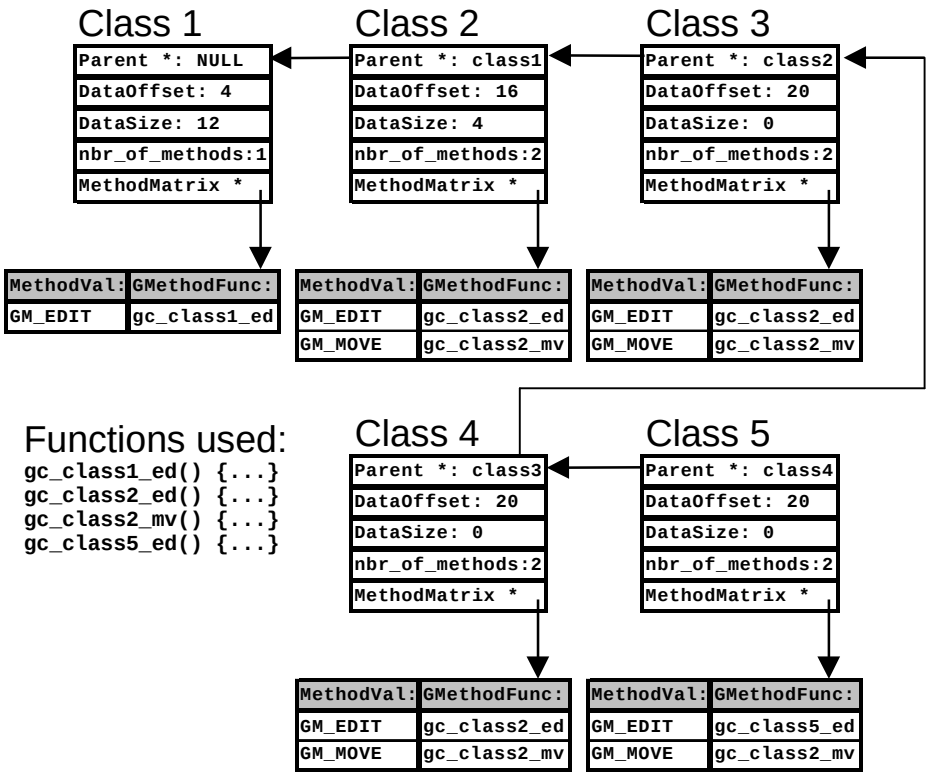

Figure 4. Overriding methods, method `GM_EDIT` is changed in Class 2 and Class 5. Method `GM MOVE` is added in Class 2.

In an object-oriented language you can call your super method from a method. To do this runtime I have to traverse the class tree backward from the class you started at to find the previous function. In my environment all classes will have a full set of function tables. All method that is implemented in at least one of the parent classes will exist in the final class function table. This means that a function that has been overridden in class number 2 and 5 will have function pointer #1 in class 1, function pointer #2 in class 2,3,4 and function pointer #5 in class 5,6... Se Figure 4.This means that when a parent method is to be called at least these data has to be supplied to make my model work: starting class or last function pointer, the object pointer and the method to call. Since the last called method is saved, only the object pointer and the function pointer or the class pointer is enough. The class pointer is to prefer, it is faster to use since the object engine can start looking directly on the class instead of first finding the class via the object. It's also easier to use by the programmer implementing classes. This result in a **`DoParentMethod()`** that will have these arguments:

```
U32 DoParentMethod(GClass *gclass,GObj *Obj, U32 method,U32 arg1,U32 arg2);
```

These functions are all that is needed to maintain, use, and interact with the classes and objects.

4.8 Object maintenance functions

Setting up the object model in a user environment defining GUI object will need some standard classes and a standard entry object. The standard classes are defined in a big struct defining a magic number to each of the classes and adding a function that will change the magic number to a **`GClass`** pointer like:

```
GClass *GetStandardClass(U32 GClassID);
```

This means that the internal magic number class array has to be initialized with a series of **`CreateGingerClass()`** calls and this is done during the initialization.

To handle the object there has to be a standard object entry point in the system and this is solved with a global object. There should only exist one instance of this class. The global object will contain the rest of the objects in the GUI model, e.g., the global object will contain some windows, and the windows will contain new objects like dragbars, textboxes and so on.

There has to be a way to add new objects to the standard object and removing them, e.g., like adding window and removing them from the system.

```
void AddGingerObj(GObj *Obj);
void SubGingerObj(GObj *Obj);
```

4.9 Comments

The class engine created in this chapter is very general and could easily be used in other contexts then graphics. One drawback though, is that it is in C so everything has to be created at runtime. This takes slightly more time then a complied solution. Another drawback is that it is harder to understand than a compiler only solution.

In the next chapter the classes are defined.

5 Defining the Ginger classes

The visual GUI components are discussed in this chapter. With visual GUI components I mean windows, dragbars, checkboxes and similar objects. The classes will be discussed in detail and their methods. The classes will also have to handle drawing and layouting the objects into the display.

5.1 GUI object basic functionality

Since the object-oriented model is used the visual object model can be quite simple. One of the design goals was to have a resizable GUI. This means that the GUI objects have to maintain information of their position and size. Since the objects should be drawn in a drawarea a pointer to the corresponding drawarea (from Ginger API) is also needed:

```
#define GA_AREA_DRAWAREA      0x00000100  /* Used to get and set the data */
#define GA_AREA_POS_X         0x00000101
#define GA_AREA_POS_Y         0x00000102
#define GA_AREA_SIZEX         0x00000103
#define GA_AREA_SIZEY         0x00000104

struct GC_Area_Data
{
      DrawArea *DA;
      int   Pos_x, Pos_y, SizeX, SizeY;
};
```

The object should be painted, so a method to handle this is required, **GM_PAINT**. This method will draw the object with the right size at the object's coordinates inside the drawarea like this:

```
U32 GC_Area_Paint(GClass* gc,GObj *Obj,struct GC_Area_Data *data,
                  U32 method ,U32 arg1, U32 arg2)
{
      /* Draws a big x inside the object */
      SetColor(data->DA,0x000000);
      Line(data->DA,data->Pos_x, data->Pos_y, data->SizeX, data->SizeY);
      Line(data->DA,data->Pos_x, data->SizeY, data->SizeX, data->Pos_y);
      return True;
}
```

5.2 Layouting the objects

To layout the objects I have added special objects that can contain one or more objects and layout them in a special way. These object containers will, when they are being layouted, also layout the compounded objects. This means that some methods like setting the container position, size and drawarea in some cases has to act on the object inside the container.

5.2.1 Horizontal and vertical containers

The horizontal containers will layout all objects it contains on a row, and the vertical in a column. Application can be built using containers inside other containers see Figure 5. If we want to build a grid of objects then we start with a vertical container and use several horizontal containers, one for each row. Inside these containers we put the actual GUI objects.

5.2.2 Priority

Without expanding the container layout we can only put things on a line or in a grid with equal size and equal space between the contained objects. This is a very limiting approach. The container layout has to be expanded to allow some way to perceptually divide the space between the objects. To solve this I have added a priority value to each object. The priority is defined to be an integer between 0 and 100 to allow it to be added without risking an overflow (2^32 is quite big). If all objects has the same priority the objects is layouted to get equal space. If some object has a zero priority and the rest has something bigger like 30. Then the zero sized ones are as small as possible and the rest of the objects divide the rest of the space between them see Figure 5. If an application should have some sort of file list on the left side and an open text area on the right (like Visual Studio) the two objects simply get priorities like 20 and 80 in a horizontal container. The whole layout process will be done using the method **GM_SET_SIZE** with no arguments.

To avoid problems with zero-sized objects and to allow objects to have a fixed size there has to be some way to set the smallest size and the biggest size an object can get. This has to be set per object since there are objects that have different minimum/maximum sizes depending of the data, like text strings and containers. Fixed size objects are implemnted by setting both the minimum and maximum sizes to the fixed size. An example of a fixed size object is a checkbox that always should use the same size. The method that will calculate the minimum and maximum size is called **GM_SET_MINMAX** and uses no arguments. The new area structure will look like this.

```
struct GC_Area_Data
{
      DrawArea *DA;
      int   x,y,SizeX,SizeY;
      int   Priority;
      int   minSizeX,minSizeY;
      int   maxSizeX,maxSizeY;
};
```

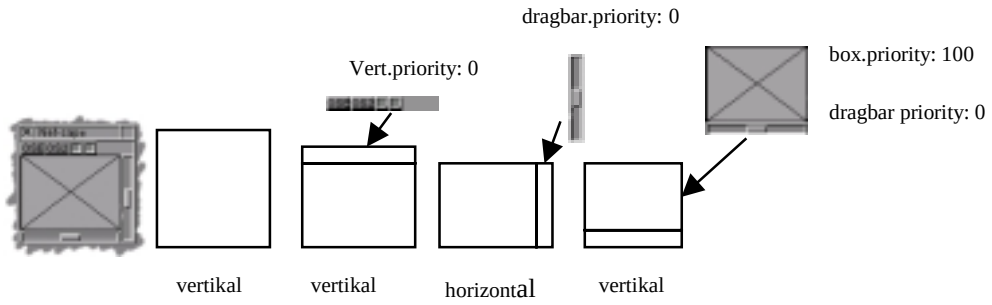

Figure 5. This illustrates how a window can be build with the containers and by setting priority.

5.3 Defined classes

In this section I will give a short explanation of the classes used and their function. In figure 6 the class chart is shown. A more detailed description is given in the "Appendix object reference".

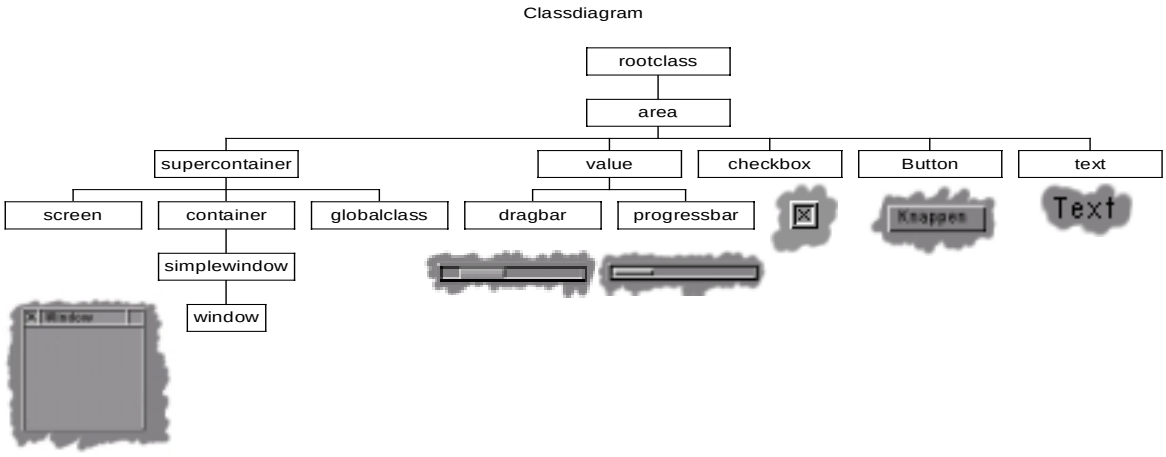

Figure 6. A diagram over the defined classes and how they are connected.

5.3.1 Basic classes

Here the abstract classes are defined. These classes are the base for other classes and are not often used by themselves except the **areaclass** that could be used to define space between objects.

rootclass
The **rootclass** is the basic class for all classes. Right now this class is empty but if there is some code that should be common for all classes it should be added here.

areaclass
The **areaclass** is the basic class handling position and size of an area. It will also handle priority, layout weight, min and max size of the class. This is probably the superclass to most user-defined classes. This class is also useful to add if some space is needed between other GUI objects. Just add it and give it higher priority than the other object (they should use a priority of zero) and it will consume all the space between.

valueclass
The **valueclass** is the basic class for objects that will handle boundary-checked values. It uses a min-value, a max-value and a value that has to be between min and max value. It is based on the **areaclass**.

5.3.2 Container classes

The container classes are the GUI objects that have other objects inside. The base class of the container classes is the **supercontainer**.

supercontainer
The **supercontainer** will handle object inside other object. When a **supercontainer** is painted it will paint the internal objects also. The **supercontainer** is not painted itself and it is based on the **areaclass**.

globalobject
The **globalobject** is based on the **supercontainer** and is just the starting point of all GUI objects in the system. The whole system lies under this object container in a tree shaped format, first all screens then inside them the window objects and inside them containers, more containers and at last other GUI objects.

screen
The **screen** class is based on the **supercontainer** and acts like a virtual desktop in X11 window managers, or like the screen concept on Amiga. It acts like a window covering the whole display containing the windows inside it. The screen class will invoke the paint function and draw some lines on top of the display.

container
The **container** is a subclass of the **supercontainer** defining some new layout behavior. The container is layouted as discussed above, and will layout it's contents horizontally or vertically depending on how it is set.

simplewindow
The **simplewindow** is a subclass of the **container** defining a simple popup window without any graphics like window title and borders. It will allocate a drawarea in the ginger API and pass the drawarea to all is sub-components.

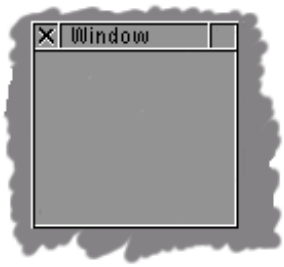

Figure 7. Window.

`window`
The **`window`** is a subclass of the **`simplewindow`** adding window borders.

5.3.3 GUI classes

Here all the visual objects are described, except the **`window`** and the **`screen`**. These are the objects that are used to create the visual parts of the GUI.

Figure 8. Progressbar.

`progressbar`
The **`progressbar`** is a subclass of the **`valueclass`**. It shows how many percent of the max-min the value of the **`valueclass`** is when it is painted.

Figure 9. Dragbar.

`dragbar`
The **`dragbar`** is a subclass of the **`valueclass`** adding the size of the dragbar. This object will draw a dragbar when it is painted. The dragbar can be moved with the mouse.

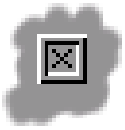

Figure 10. Checkbox.

`checkbox`
The **`checkbox`** is a subclass of the **`areaclass`**. It will show if the checkbox is on or off when painted. When clicked it will toggle its value.

Figure 11. Text.

`text`
The **`text`** class is a subclass of the **`areaclass`**, it is a simple string. This class is used to type text in the GUI.

Figure 12. Button.

`button`
The **`button`** is a subclass of the **`areaclass`.** It can be defined as a simple container using one or two objects. The **`button`** class will draw different frames depending on if it is pressed or not. Inside the frame a new object has to be added. Usually a **`text`** object or two is added. If one is added it is used for both a pressed and a not pressed button. If two are added different objects are used. The button can be made sticky and will then act as a checkbox but with customized objects. It will change state when clicked by with the mouse and change back when the mouse button is released.

5.4 Comments

The classes define the basic components that are required to build a GUI. What is lacking are more classes like a menu class, popup-window class and similar. Adding new classes is quite simple, using this classes a building block or writing completely new classes.

In the next section the solution is evaluated.

6 Evaluation

In this chapter the results of this master project are discussed and comment on how to improve or rework the solution to a better one is made. The overall solution works well and is very flexible and easy to extend. But nothing is so good that it can not be better. I will discuss enchantments of each part of these solutions in this chapter.

6.1 Ginger BIOS

The Ginger BIOS driver system uses a rather sparse approach to make it simple. This means that since only pixels, lines and rectangles are available, things like drawing images and typing text on the display results in many pixel drawing operations that has to cross the user to kernel space boarders. Since the BIOS driver is put in kernel space using a TRAP function or similar to reach it there is much overhead. To make it faster the BIOS driver should be expanded to be smarter when handling images and text so that fewer TRAP instructions is required. Another item that is not designed well is a system to display some sort of pointer/cursor on the display and some function to fetch the graphics in the display from the driver. All these changes are easy to add there is noting in the current model that prevents this.

6.2 Ginger application program interface

Using the application program interface (API) is quite straight forward and works the way it should. But it only provides a basic set of functions. More functionality specialy in the text and image areas are needed. Speedup are also needed in the same areas as discussed in the BIOS section above. The clipping is slow and could be cached, and some function to alert drawareas when they are destroyed are requrided, i.e., if a window on top of another window is moved the window below has to be notified so it can be painted again. All these changes are easy to add and there is noting in the current model that prevents this.

6.3 Ginger class engine

The class engine algorithms are quite fast. The class engine works well and after the implementation almost no changes had to be made to it to get the classes to run. A drawback right now is that all the classes only uses C. If it is used from C++ only the C functions are used. It would be nice if there was a C++ version that used the C++ object model.

6.4 Ginger classes

Right now the classes are very dynamical. Each time a size value changes the object recalculates its position and repaints it self. This leads to a very high numbers of size recalculations and repaints, especially during the setup. This solution is not good, If this is going to be improved this is one thing that needs attention. The classes are also programmed for a color system with a mouse-like pointing device. A better solution is to have different classes for different environments, i.e., having one set of classes for a black and white display using checked black and white instead of gray and one for touch displays with bigger buttons. There is also much work left expanding the classlibrary with classes for menus, popups, special textboxes, strings and some other classes.

7 Conclusions

In this chapter I will discuss the result and conclusions of this master thesis.

The major conclusion is that it is possible to make a GUI on top of the real-time operating system OSE. To solve the problem it was divided into three parts a BIOS driver, an application program interface (API) and an object-oriented engine with a set of class. This general solution works well and is easy to both port to new hardware and to extend with new functionality. Using the object-oriented solution makes the GUI very adaptable both to resizes and to look and feel, which can easily be modified by making new subclasses and overriding the paint functions. The resulting program is also very small in size. If the code is compiled with the optimize-size flag on the size is about 50-60 kb using Visual C++5.0 compiling for PentiumII, for the whole GUI with drives, API and class engine with classes.

A Appendix object reference

A.1 rootclass

Identification	**GC_ROOTCLASS**
Parent class	None
Description	This is the root class to all other classes, should not be used directly and should be considered as an empty object.
Include files	ginger.h
Attributes	None
Macros	None
Methods	None
Restrictions	Do not use this class directly
See also	

A.2 area

Identification	**GC_AREA**
Parent class	**rootclass**
Description	This is the root class to all gfx components,used to bulid most of the other components of. It handles only the basic stuff for the compunent like the size and the drawarea (an ginger API primitive).
Include files	ginger.h

Attributes

GA_AREA_DRAWAREA	The drawarea that the object uses for it drawings The drawarea does not exist befor OPEN is called and the object exist in or is a Window Object
GA_AREA_POS_X	x pos for the object in the drawarea
GA_AREA_POS_Y	y pos for the object in the drawarea
GA_AREA_SIZEX	the x size for the object
GA_AREA_SIZEY	the y size for the object
GA_AREA_PRIORITY	The layout priority 0-100, size is delivered to this priority

Macros

This macros is supplied to get som values used a lot, faster
use this to avoid the **DoMethod(Obj,GM_GET,<Attribute>,0);** call.

gc_area_get_drawarea(Obj)	Macro to get the value of **GA_AREA_DRAWAREA**
gc_area_get_pos_x(Obj)	Macro to get the value of **GA_AREA_POS_X**
gc_area_get_pos_y(Obj)	Macro to get the value of **GA_AREA_POS_Y**
gc_area_get_sizex(Obj)	Macro to get the value of **GA_AREA_SIZEX**
gc_area_get_sizey(Obj)	Macro to get the value of **GA_AREA_SIZEY**
gc_area_get_minsizex(Obj)	Macro to get the Minimum x-size
gc_area_set_minsizex(Obj)	Macro to set the Minimum x-size
gc_area_get_minsizey(Obj)	Macro to get the Minimum y-size
gc_area_set_minsizey(Obj)	Macro to set the Minimum y-size
gc_area_get_maxsizex(Obj)	Macro to get the Maximum x-size
gc_area_set_maxsizex(Obj)	Macro to set the Maximum x-size
gc_area_get_maxsizey(Obj)	Macro to get the Maximum y-size
gc_area_set_maxsizey(Obj)	Macro to set the Maximum y-size
gc_area_get_priority(Obj)	Macro to get the value of **GA_AREA_PRIORITY**

Methods

GM_SET	Handles the new Attributes arg1=Attribute arg2=value
GM_GET	Handles the new Attributes arg1=Attribute return value
GM_INSIDE	Check if (arg1,arg2) is inside the area
GM_SET_MINMAX	Fix Min size and Max size.

Restrictions

The **GA_AREA_DRAWAREA** value is not handled directly by this object and are initiated to NULL. This value will not get its values before the **GM_OPEN** phase done after the **GM_NEW** phase. And only if object is part of some structures defininig a drawarea like a simplewindow.

See also

A.3 supercontainer

Identification	**GC_SUPERCONTAINER**	
Parent class	**area**	
Description	The container class houses other objects in a few diffrent modes	
Include files	ginger.h	
Attributes	**GA_CONTAINER_ACTIVE**	The active sub object
	GA_CONTAINER_ADD_CHILD	This is not an attribute but a special case if its used in a **GM_SET** the method **GM_ADD_OBJ** is invoked
	CHILD	same as **GA_CONTAINER_ADD_CHILD**
Macros	gc_container_tunnel(gclass,Obj,method arg1, arg2)	this macro tunnles the method
Methods	**GM_DISPOSE**	Frees the internal linked list.
	GM_SET	Special value to set added **GA_CONTAINER_ADD_CHILD** (or **CHILD** (short version)) invokes metod **GM_ADD_OBJ**
	GM_GET	Get attribute
	GM_ADD_OBJ	Add object to container arg1=obj
	GM_SUB_OBJ	Remove object to container arg1=obj
	GM_SETSIZE	Calculate the size and updated the children.
	GM_PAINT	Tunnelled to the children
	GM_TUNNEL	Internal use only, Make the LastMethod on all children.
	GM_CONTAINER_FIRST	point out the first children object
	GM_CONTAINER_NEXT	point out the next children object
Restrictions	None	
See also	**area**, **container**	

A.4 GlobalRootClass

Identification	**GlobalRootClass**	
Parent class	**supercontainer**	
Description	The root container, add screen to it via **GM_ADD_OBJ**	
Include files	ginger.h	
Attributes	None	
Macros	None	
Methods	**GM_PAINT**	Draw only the current screen
	GM_SET	Handle the events
Restrictions	None	
See also	**area**, **supercontainer**, **simplewindow**	

A.5 screen

Identification	**GC_SCREEN**	
Parent class	**supercontainer**	
Description	Screens always as big as the whole area, contains the windows	
Include files	ginger.h	
Attributes	None	
Macros	None	
Methods	**GM_NEW**	Create the drawarea to fill the display
	GM_PAINT	Draw screen stuff
Restrictions		
See also	**area**, **container**, **simplewindow**	

A.6 container

Identification	**GC_CONTAINER**	
Parent class	**supercontainer**	
Description	The container class houses other objects. In a few different modes it will also execute methods on its children objects.	
Include files	ginger.h	
Attributes	**GA_CONTAINER_LAYOUT**	valid values right now is **HORIZONTAL** and **VERTICAL**, default is **HORIZONTAL**
	GA_AREA_DRAWAREA	Are snooped and tunnelled to the children
	GA_AREA_POS_X	Are snooped and **GM_SETSIZE** is called
	GA_AREA_POS_Y	Are snooped and **GM_SETSIZE** is called
	GA_AREA_SIZEX	Are snooped and **GM_SETSIZE** is called
	GA_AREA_SIZEY	Are snooped and **GM_SETSIZE** is called
Macros	None	
Methods	**GM_SET**	Sets the Layout, and snopps som layoutchanges
	GM_SETSIZE	Calculate the size and updated the children.
Restrictions	None	
See also	**area**	

A.7 simplewindow

Identification	**GC_SIMPLEWINDOW**	
Parent class	**container**	
Description	This is the root class to all popups/windows. It will allocate a drawarea for its context and will use it to draw the window and all its children objects	
Include files	ginger.h	
Attributes	**GA_WINDOW_ACTIVE**	
	GA_WINDOW_POS_X	x pos for the object in the display
	GA_WINDOW_POS_Y	y pos for the object in the display
	Monitores this to resize the window **GA_AREA_SIZEX** **GA_AREA_SIZEY**	
Macros	None	
Methods	**GM_SET**	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET	Handles the new Attributes arg1=Attribute return value
	GM_OPEN Will create the drawarea	
	GM_CLOSE	Will dispose the drawarea
	GM_INSIDE	Check if (arg1,arg2) is inside the window
Restrictions	None	
See also	**area**, **window**, **screen**	

A.8 window

Identification	**GC_WINDOW**	
Parent class	**simplewindow**	
Description	A standard window with title and borders.	
Include files	ginger.h	
Attributes	**GA_WINDOW_TITLE**	a char* the string is NOT copied
Macros	None	
Methods	**GM_SET**	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET	Handles the new Attributes arg1=Attribute return value
	GM_PAINT	Draws the windowborder and its childres objects
	GM_SETSIZE	Set the size of the children to this class
Restrictions		
See also	**area**, **simplewindow**, **screen**	

A.9 value

Identification	**GC_VALUE**	
Parent class	**area**	
Description	This is the root class to DragBar and ProgressBar. It contains useful values.	
Include files	ginger.h	
Attributes	**GA_VALUE**	Should be between min and max value
	GA_VALUE_MIN	Minimum Value
	GA_VALUE_MAX	Maximum Value
	GA_LAYOUT	**HORSONTAL** or **VERTICAL**
Macros	None	
Methods	**GM_SET**	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET	Handles the new Attributes arg1=Attribute return value
	GM_SET_MINMAX	Set Size
Restrictions	None	
See also	**area**, **dragbar**, **progressbar**	

A.10 dragbar

Identification	**GC_DRAGBAR**	
Parent class	**value**	
Description	A dragbar, nothing more and nothing less.	
Include files	ginger.h	
Attributes	**GA_VALUE_SIZE**	Used to make the dragbar proportional.
Macros	None	
Methods	**GM_SET**	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET	Handles the new Attributes arg1=Attribute return value
	GM_PAINT	Draws the dragbar
Restrictions	None	
See also	**value**, **progressbar**	

A.11 progressbar

Identification	**GC_PROGRESSBAR**	
Parent class	**value**	
Description	A progressbar showing the percentage that value is of the total size. (**GA_VALUE_MAX - GA_VALUE_MIN**)	
Include files	ginger.h	
Attributes	None	
Macros	None	
Methods	**GM_SET**	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET	Handles the new Attributes arg1=Attribute return value
	GM_PAINT	Draws the dragbar
Restrictions		
See also	**value**, **dragbar**	

A.12 checkbox

Identification	GC_CHECKBOX	
Parent class	area	
Description	A checkbox, marks ON or OFF.	
Include files	ginger.h	
Attributes	**GA_VALUE**	True or False
Macros	None	
Methods	**GM_SET**	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET	Handles the new Attributes arg1=Attribute return value
	GM_PAINT	Draws the checkbox
	GM_SET_MINMAX	Set Size
Restrictions	None	
See also	area	

A.13 button

Identification	**GC_BUTTON**	
Parent class	**area**	
Description	A Button. It can be pressed.	
Include files	ginger.h	
Attributes	**GA_VALUE**	True or False True=Pressed
	GA_NORMALOBJ	The object when not pressed
	GA_PRESSEDOBJ	The object when pressed, if one exist
Macros	None	
Methods	**GM_SET**,	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET,	Handles the new Attributes arg1=Attribute return value
	GM_PAINT	Draws the button
	GM_SET_MINMAX	Set Min-Max Size
	GM_SET_SIZE	Set Size
Restrictions	None	
See also	**area**	

A.14 text

Identification	**GC_TEXT**	
Parent class	**area**	
Description	This class handles text.	
Include files	ginger.h	
Attributes	**GA_TEXT** The string, its not copied	
	GA_COLOR	Color of the text.
Macros	None	
Methods	**GM_SET**	Handles the new Attributes arg1=Attribute arg2=value
	GM_GET	Handles the new Attributes arg1=Attribute return value
	GM_PAINT	Types the text
	GM_SET_MINMAX	Calculate the object size.
Restrictions	None	
See also	**area**	

B Appendix References

[1] OSE System, "OSE 4.0.1 Documentation",1998
[2] D. Hearn and M.P. Baker, "Computer graphics - C version", Prentice Hall, New Jersey 1997
[3] B. Kernighan and D. Ritchie, "The C programming language", Prentice Hall, New Jersey 1988
[4] C. Petzold, "Programming Windows 95", Microsoft Press, Washington 1996
[5] Amiga Technical Reference Series, "AMIGA ROM Kernel Reference Manual Libraries", Third edition, Addison-Wesley, Reading 1992