Celoxica

DK2

Handel-C VGA graphics output

Authors: SB

Document number: 1

Customer Support at http://www.celoxica.com/support/

Celoxica in Europe
T: +44 (0) 1235 863 656
E: sales.emea@celoxica.com

Celoxica in Japan
T: +81 (0) 45 331 0218
E: sales.japan@celoxica.com

Celoxica in Asia Pacific
T: +65 6896 4838
E: sales.apac@celoxica.com

Celoxica in the Americas
T: +1 800 570 7004
E: sales.america@celoxica.com

Contents

Conventions

A number of conventions are used in this document. These conventions are detailed below.

Warning Message. These messages warn you that actions may damage your hardware.

Handy Note. These messages draw your attention to crucial pieces of information.

Hexadecimal numbers will appear throughout this document. The convention used is that of prefixing the number with '0x' in common with standard C syntax.

Sections of code or commands that you must type are given in typewriter font like this:

```
void main();
```

Information about a type of object you must specify is given in italics like this:

```
copy SourceFileName DestinationFileName
```

Optional elements are enclosed in square brackets like this:

```
struct [type_Name]
```

Curly brackets around an element show that it is optional but it may be repeated any number of times.

```
string ::= "{character}"
```

1 Tutorial: Handel-C and VGA graphics output

The following examples illustrate how to use Handel-C to generate simple VGA graphics and respond to user input. Three examples are used, each building on the previous one to add new features. The **TutorialVGA** workspace contains the code for each of the examples. A basic knowledge of Handel-C is assumed, and some knowledge of digital electronics and design techniques will also be helpful.

New users are recommended to work through the following topics in order:

Generating VGA graphics (see page 4)

Responding to user input (see page 7)

Adding mouse input (see page 12)

2 Generating VGA graphics

The **GraphicsDemo1** project in the **TutorialVGA** workspace contains the code for this example. The first step in generating VGA graphics using DK and Handel-C is to set up a PAL workspace for one or more targets. This has already been done in the **GraphicsDemo1** project for **Simulation** and **RC200**, but the procedure is explained fully in Setting up a PAL workspace.

In the main function, a macro is defined which returns the *PalHandle* representing the optimal video mode for the chosen clock rate, and the version of PAL and the resources required are specified:

```
macro expr VideoOut = PalVideoOutOptimalCT (ClockRate);
PalVersionRequire  (1, 0);
PalVideoOutRequire (1);
```

The next step is to run the video driver in parallel with the code which will generate the graphics to be displayed, in this case a macro called `RunOutput`. Note that the video output must also be enabled. The `ClockRate` macro should be defined to return the actual clock rate of the system. In **GraphicsDemo1** the clock rate is `PAL_ACTUAL_CLOCK_RATE`.

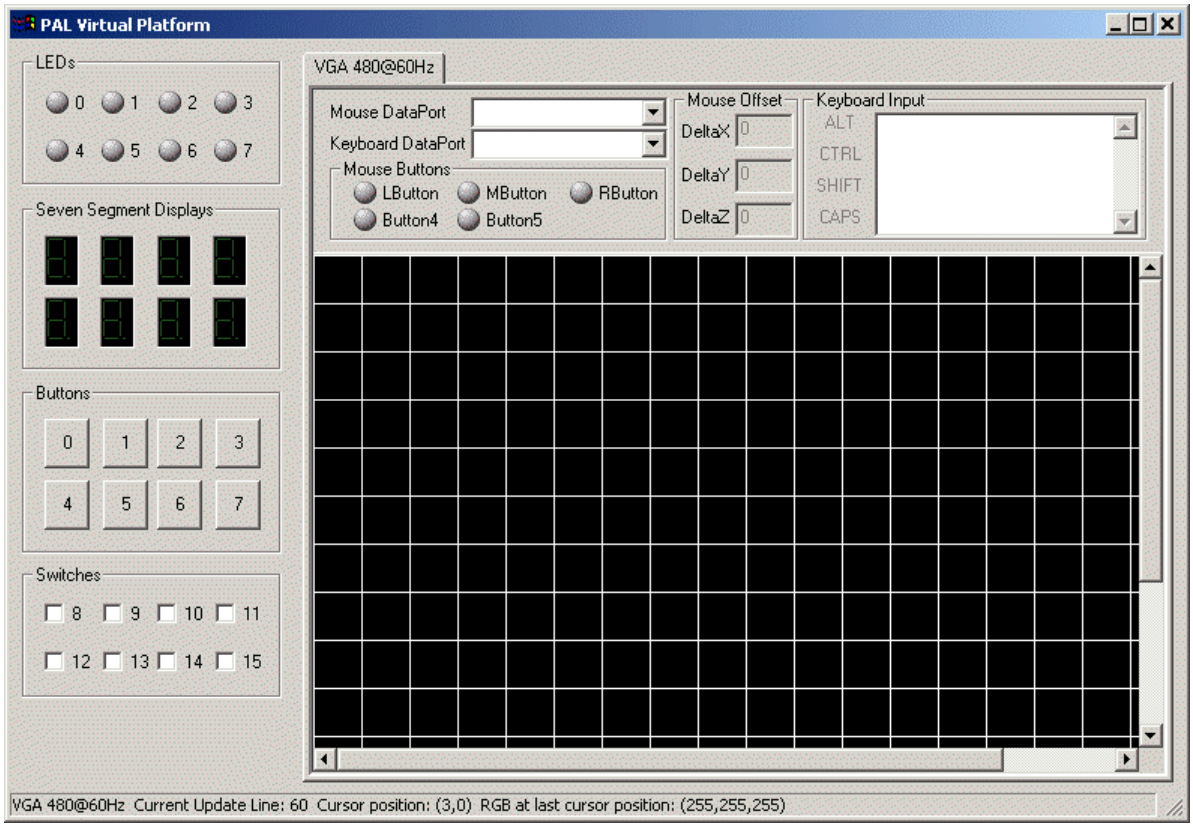
```
par
{
    PalVideoOutRun (VideoOut, ClockRate);
    seq
    {
        PalVideoOutEnable (VideoOut);
        RunOutput (VideoOut);
    }
}
```

In order to display graphics, the `RunOutput` macro will need to know what the current VGA scan position is and have some predefined colours to write to the screen. PAL uses 24-bit RGB colour format, which is then reduced to the colour depth supported by the target device. To determine the current VGA scan position, a pointer to the video PalHandle is passed into `RunOutput`, and the standard PAL video macros are used. The code sample below shows the definitions of the colours and two macro expressions to give quick access to the current VGA scan position.

```
macro expr White = 0xFFFFFF;
macro expr Black = 0x000000;
macro expr Red   = 0xFF0000;
macro expr Green = 0x00FF00;
macro expr Blue  = 0x0000FF;

macro expr ScanX = PalVideoOutGetX (VideoOut);
macro expr ScanY = PalVideoOutGetY (VideoOut);
```

Having defined these simple macro expressions, it is now possible to make the `RunOutput` macro display graphics on a VGA output. The example in **GraphicsDemo1** draws a white grid on a black background. This is achieved by taking the lowest five bits of ScanX and ScanY, and drawing a white pixel whenever these bits are equal to zero. All other pixels are drawn as black, resulting in a grid of white lines one pixel wide, and spaced by 32 pixels vertically and horizontally. The code to generate this grid is shown below, and a screenshot of the output in simulation is shown in the figure below. Note that the code to generate the grid graphics is inside a `while(1)` loop, so it will run forever, and it also executes in a single cycle, as this is the rate at which pixels must be sent out to the VGA display.

```
while (1)
{
    if ((ScanY <- 5 == 0) || (ScanX <- 5 == 0))
        PalVideoOutWrite (VideoOut, White);
    else
        PalVideoOutWrite (VideoOut, Black);
}
```



PALSIM RUNNING GRAPHICSDEMO1

To run the example yourself, open the **TutorialVGA** workspace, set **GraphicsDemo1** as the active project, set the Active Configuration to Sim, then build and run the project. For a Celoxica RC200 board with a VGA monitor connected, set the Active Configuration to RC200, rebuild, then use the Place and Route tools to generate a bitfile to download to the board.

3 Responding to user input

The **GraphicsDemo2** project in the **TutorialVGA** workspace contains the code for this example. This example takes **GraphicsDemo1,** which drew a white grid on the screen, and adds a red box, drawn underneath the white grid. The size of the box can be varied using the switches in the PalSim Virtual Platform and on the Celoxica RC200 board.

To enable the use of switches for user input, they should be required at the start of the program, at the same time as requesting video output and PAL version. In this case a minimum of two switches are requested, as shown below. Switches do not require a Run macro (like the video output does), as they are simple devices and can be accessed directly.

```
PalVersionRequire  (1, 0);
PalVideoOutRequire (1);
PalSwitchRequire   (2);
```

The `RunOutput` macro must first be modified to draw a box as well as the white grid, so some additional macros are defined to help in this task, as shown below. `MaxX` and `MaxY` return the maximum number of pixels visible, `XWidth` and `YWidth` return the bit width required to hold the X and Y VGA scan variables, and `XPos` and `YPos` are used to mark the center of the box which will be displayed.

```
macro expr MaxX   = PalVideoOutGetVisibleXCT (VideoOut, ClockRate);
macro expr MaxY   = PalVideoOutGetVisibleYCT (VideoOut);
macro expr XWidth = PalVideoOutGetXWidthCT (VideoOut);
macro expr YWidth = PalVideoOutGetYWidthCT (VideoOut);
macro expr XPos   = MaxX/2;
macro expr YPos   = MaxY/2;
```

As the size of the box to be drawn will be changed according to user input, it needs to be a variable with an initial value assigned:

```
static unsigned YWidth BoxSize = 20;
```

To actually draw the box, the display output code must be changed to detect when the VGA scan position is within the box region:

```
while (1)
{
    if ((ScanY <- 5 == 0) || (ScanX <- 5 == 0))
        PalVideoOutWrite (VideoOut, White);
    else if((ScanX > (Xpos - BoxSize)) && (ScanX < (Xpos + BoxSize)) &&
            (ScanY > (YPos - BoxSize)) && (ScanY < (YPos + BoxSize)))
        PalVideoOutWrite (VideoOut, Red);
    else
        PalVideoOutWrite (VideoOut, Black);
}
```

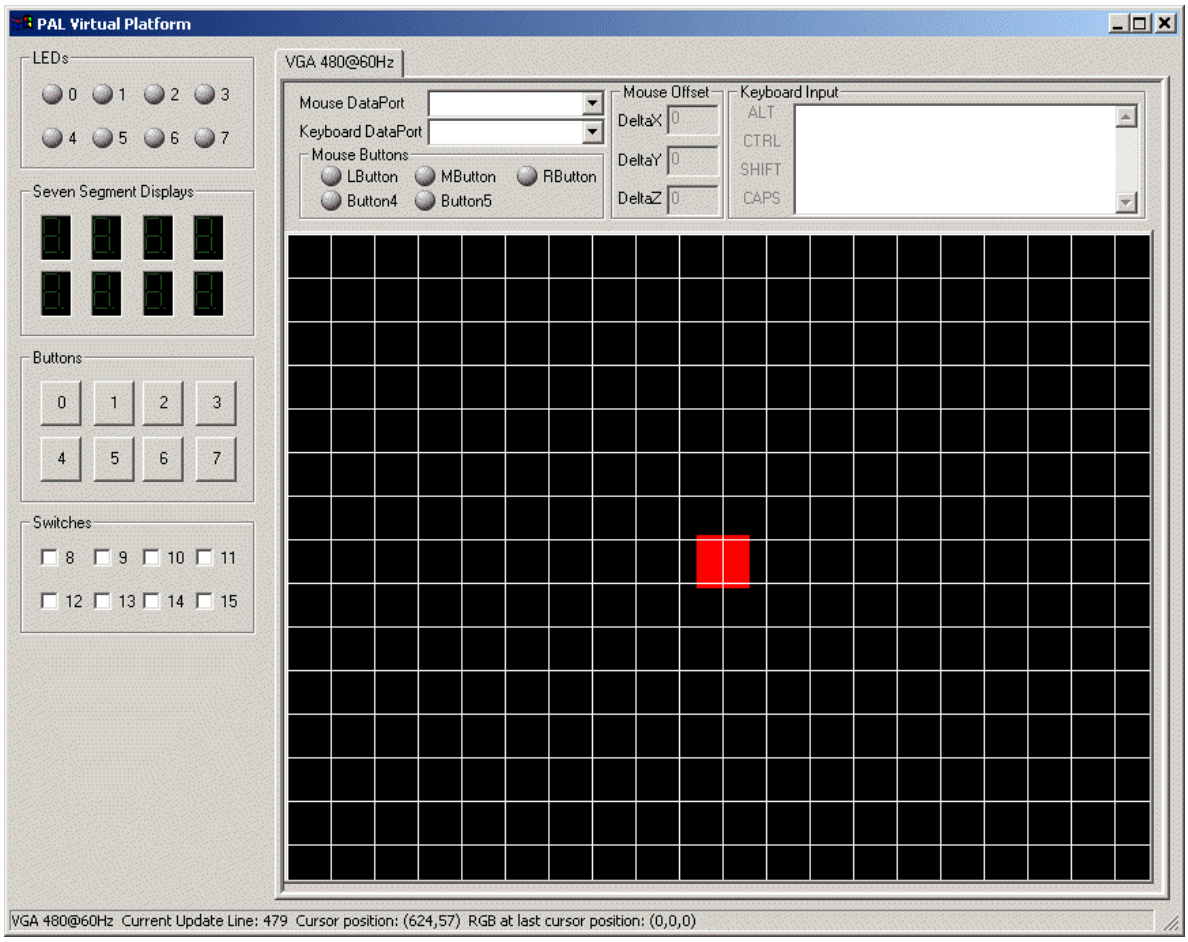
In parallel with the `while(1)` loop running the display output, there must be another `while(1)` loop which reads the switches and modifies the box size to account for any user input detected. The size of the box should be limited so that it does not go below zero, and does not go above the maximum visible number of pixels in the Y direction. This is necessary for the display output code shown to work correctly, as attempting to store a negative result in an unsigned number results in a large (incorrect) positive number. The code below shows how the user interaction is performed. Two calls are made in parallel to `PalSwitchRead` to get data from the two switches, and at the same time the data from the switches is checked and the box size updated. As Handel-C only updates variables at the end of a clock cycle, data read from the switches will not be checked until the following cycle, but this will not have any impact on the operation of this example.

```
while (1)
{
    par
    {
        PalSwitchRead (PalSwitchCT (0), &SwitchData[0]);
        PalSwitchRead (PalSwitchCT (1), &SwitchData[1]);

        if (SwitchData[0] == 1)
        {
            if (BoxSize != (MaxY / 2))
            {
                BoxSize++;
                Sleep (20);
            }
            else
                delay;
        }
        else if (SwitchData[1] == 1)
        {
            if (BoxSize != 0)
            {
                BoxSize--;
                Sleep (20);
            }
            else
                delay;
        }
        else
            delay;
    }
}
```

The calls to the Sleep() macro are required to avoid the box size changing too quickly, so that you can observe it happening. In this case a sleep period of 20ms is used, limiting the rate of change to 50 pixels per second. The code for the Sleep() macro is shown below, including a notional clock rate of 10000Hz for simulation.

```
static macro proc Sleep (Milliseconds)
{
#ifdef USE_SIM
    macro expr Cycles = (10000 * Milliseconds) / 1000;
#else
    macro expr Cycles = (ClockRate * Milliseconds) / 1000;
#endif
    unsigned (log2ceil (Cycles)) Count;

    Count = 0;
    do
    {
        Count++;
    }
    while (Count != Cycles - 1);
}
```

The figure below shows the **GraphicsDemo2** project running in simulation on the PalSim Virtual Platform. To run the example yourself, open the **TutorialVGA** workspace, set **GraphicsDemo2** as the active project, set the Active Configuration to Sim, then build and run the project. For a Celoxica RC200 board with a VGA monitor connected, set the Active Configuration to RC200, rebuild, then use the Place and Route tools to generate a bitfile to download to the board.



PALSIM RUNNING GRAPHICSDEMO2

4 Adding mouse input

The **GraphicsDemo3** project in the **TutorialVGA** workspace contains the code for this example. This example takes **GraphicsDemo2**, allows the red box drawn on the screen to be moved around using a mouse and changes the colour of the box when the mouse buttons are pressed.

To use the mouse under PAL, the `pal_mouse.hch` header must be included and the `pal_mouse.hcl` library added to the linker settings for both **Sim** and **RC200** targets. A pointer of type `PalMouse` must be created, and a PS2 port will be required to connect the mouse to. It is also useful to create a macro expression to provide quick access to the PS2 port, as shown in the code below:

```
macro expr PS2 = PalPS2PortCT (0);
PalMouse *MousePtr;
PalPS2PortRequire  (1);
```

The mouse driver must be run and enabled in parallel with the video driver and the `RunOutput` macro, the limits on the cursor position should be set and wrapping (what happens to the cursor at the edge of the screen) turned off, as show below. `MaxX` and `MaxY` are macro expressions returning the number of visible pixels.

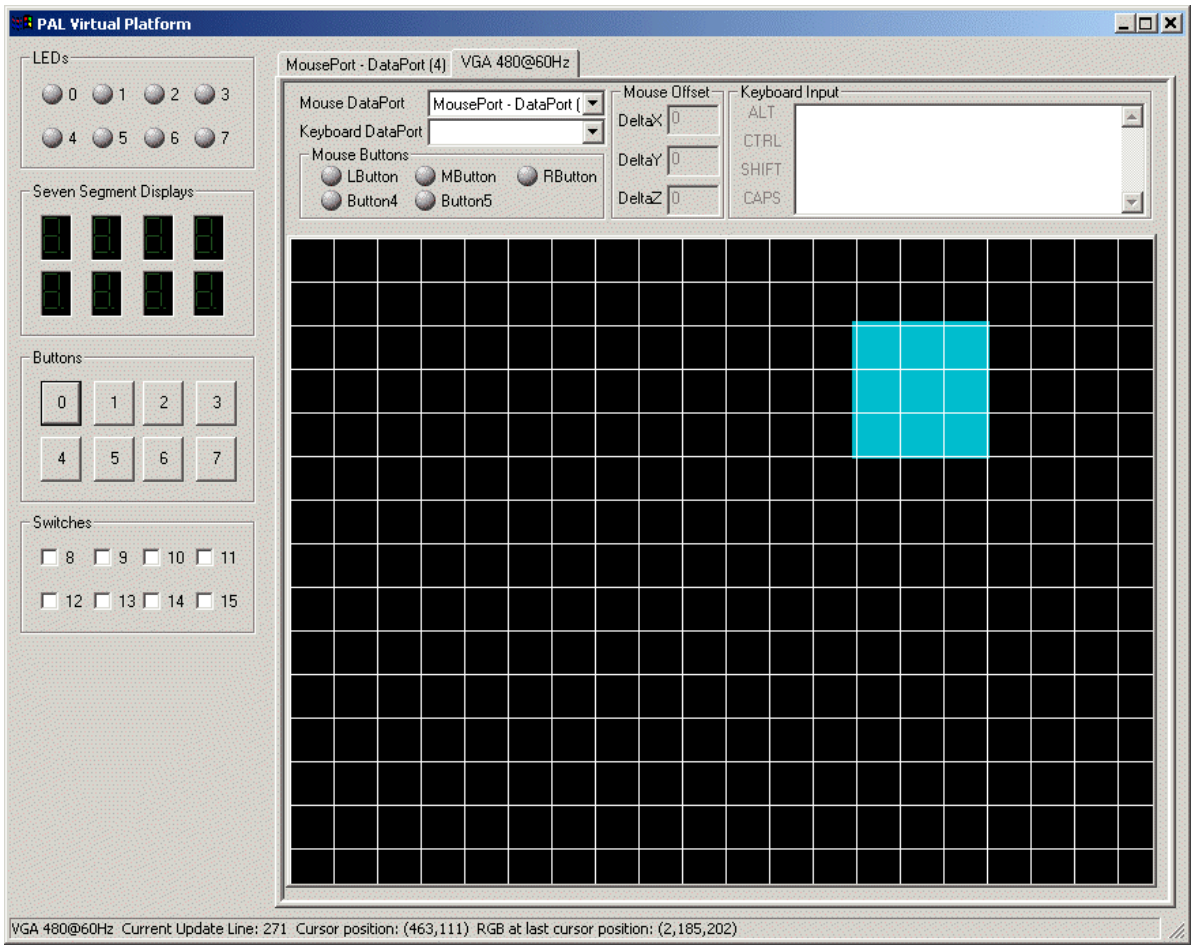
```
par
{
    PalVideoOutRun (VideoOut, ClockRate);
    PalMouseRun (&MousePtr, PS2, ClockRate);

    seq
    {
        par
        {
            PalVideoOutEnable (VideoOut);
            PalMouseEnable (MousePtr);
        }
        par
        {
            PalMouseSetMaxX (MousePtr, MaxX);
            PalMouseSetMaxY (MousePtr, MaxY);
            PalMouseSetWrap (MousePtr, 0);
        }

        RunOutput (VideoOut, MousePtr, ClockRate);
    }
}
```

The final code to add for this example takes the mouse input and uses it to control the position and colour of the box displayed on the VGA output. This code is in the `RunOutput` macro, running in parallel with the code reading the switches and updating the box size. The mouse coordinates are copied into the box position every cycle, if the left mouse button is pressed the 24 bit box colour is incremented, and if the right mouse button is pressed, the colour is reset to red. Two new macro expressions, `MouseX` and `MouseY` are created to provide easy access to the current mouse coordinates, and their use can be seen in the code below:

```
while (1)
{
    par
    {
        XPos = MouseX;
        YPos = MouseY;

        if (MouseL == 1)
            BoxColour++;
        else
            delay;

        if (MouseR == 1)
            BoxColour = Red;
        else
            delay;
    }
}
```

The code for updating the box position and colour can not go in the same `while(1)` loop as the code which reads the switches, as it needs to execute every cycle, and the switch code includes calls to `Sleep()`. Instead, separate `while(1)` loops are run in parallel within the `RunOutput` macro, allowing each to take different numbers of cycles simultaneously.

The figure below shows the **GraphicsDemo3** project running in simulation on the PalSim Virtual Platform. To run the example yourself, open the **TutorialVGA** workspace, set **GraphicsDemo3** as the active project, set the Active Configuration to Sim, then build and run the project. For a Celoxica RC200 board with a VGA monitor connected, set the Active Configuration to RC200, rebuild, then use the Place and Route tools to generate a bitfile to download to the board.



PALSIM RUNNING GRAPHICSDEMO3

5 Index