Application Report
AN01287 – December 2011

# Stellaris® Graphics Library Display Drivers

**ABSTRACT**

The Stellaris Graphics Library (grlib) offers a compact yet powerful collection of graphics functions allowing the development of compelling user interfaces on small monochrome or color displays attached to Stellaris microcontrollers. This document describes how to support a new display device in the Stellaris Graphics Library.

**Contents**

1 Introduction ..... 1
2 Library Structure ..... 1
3 Display Driver API ..... 3
4 Conclusion ..... 8
5 References ..... 8

## 1 Introduction

The Stellaris Graphics Library (grlib) is included in all StellarisWare Firmware Development Packages supporting evaluation or development kits which include color displays. The Graphics library provides a collection of graphics functions that allow users to develop user interfaces on small monochrome or color displays attached to Stellaris microcontrollers. This list includes ek-lm4f232, ek-lm3s3748, dk-lm3s9b96, rdk-idm, rdk-idm-l35 and rdk-idm-sbc. StellarisWare releases for these kits can be downloaded from http://www.ti.com/stellarisware.

Following installation of your StellarisWare package, the graphics library source can be found in the C:\StellarisWare\grlib directory (assuming you installed in the default location) and various example applications using the library can be found in the C:\StellarisWare\boards\<your board name> directory. The example applications provided vary from board to board, but a version of “grlib_demo” is included in all kits supporting the graphics library.

This application note provides information on the lowest layer of the Stellaris Graphics Library architecture — the display driver. This layer acts as the interface between the common code of the graphics library and the display controller in use on a particular board. A new or modified display driver is required to support the Stellaris Graphics Library on a new board design or when using a graphics display controller other than those supported on existing Stellaris development kits.

Source code for the display driver for each supported board can be found in the C:\StellarisWare\boards\<your board name>\drivers directory.

## 2 Library Structure

The Stellaris Graphics Library is a layered library offering four main application programming interfaces (APIs) which an application developer may choose to use. Figure 1 shows the organization of the Stellaris Graphics Library.

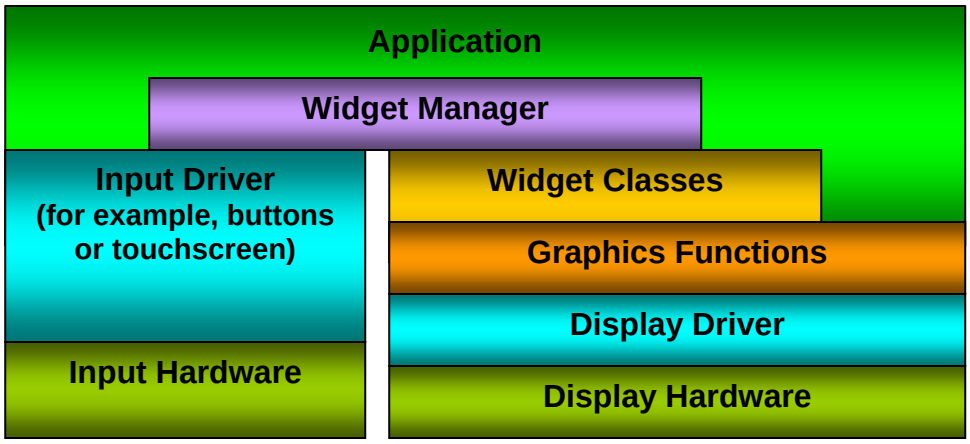


**Figure 1. Organization of Stellaris Graphics Library**

The right side of the figure shows the layers of the graphics stack. APIs are provided at the widget level (the Widget API), the graphics function level (the Low-Level Graphics API) and at the display driver level (the Display Driver API). Additionally, a standard user input driver interface (the Input Driver API) is also provided. Depending upon the requirements of a given application, some portions of the library may be omitted if their functions are not required. All type definitions, labels, macros, and function prototypes for the graphics functions and display driver layers can be found in the grlib.h file. Definitions relating to the higher level widget library can be found in widget.h and individual headers such as canvas.h and pushbutton.h, contain definitions for each supported widget class.

## *2.1 Display Driver Overview*

The display driver layer provides a standard programming interface to the graphics library code allowing it to draw actual pixels on the display. The API is very simple (draw horizontal and vertical lines, copy a line of pixels to a position on the screen, plot a single pixel) and is not typically accessed directly by the application since it is missing many of the graphics primitives that an application is likely to require such as slanted lines, rectangles, circles, text, and image support.

This application note provides additional information on the display driver API and offers suggestions on how to develop a new display driver for your particular board and display controller.

## *2.2 Low-Level Graphics API Overview*

The first API which is intended for application use is the low-level graphics API. This gives access to functions which draw the major graphics primitives – lines, rectangles, circles, text, and images. In addition, functions and macros are provided to perform coordinate checking and rectangle processing (for example, checking for intersection and overlap, determining whether a point lies within a rectangle, and so on.).

The low-level graphics API is concerned only with drawing to the display and has no knowledge of user input or any high-level controls.

## *2.3 Widget API Overview*

Above the low-level graphics API and the input driver, the widget API offers a high-level interface that allows users to build a complex user interface that includes individual controls such as buttons, sliders, checkboxes, and other high-level widgets (controls).

This layer ties the graphical display to the user input system. It manages the input and updates the displayed widgets according to button or touch screen presses made by the user. Application interaction with widgets is via callback functions provided during initialization. These callbacks are specific to the type of widget but would include functions called when a button is pressed or a slider is moved.

### 2.4 Input Driver Overview

The input driver, like the display driver, is responsible for managing a block of hardware and translating user interaction into a standard format that the widget manager can understand. An application will not typically call the input driver other than during startup when a call is made to initialize the device.

Input devices may include touch screens or navigation buttons.

## 3 Display Driver API

The lowest level of the graphics library stack is the display driver. Although the display driver API is specified by the graphics library, the source is specific to the board and display hardware and can be found in the C:\StellarisWare\boards\<your board>\drivers directory. The driver source file name is typically derived from the display manufacturer, supported display controller part, display resolution, and bit depth. For example, the display driver for the rdk-idm is named formike240x320x16_ili9320.c since it supports a Formike display using an ILI9320 controller, and offers 240x320 resolution at 16 bits per pixel.

The display driver's responsibility is to translate calls made to the standard display driver API into orders to draw pixels or lines on the display. The interface to the driver is intended to offer the absolute minimum subset of drawing orders required to support the main graphics library and, as a result, make it extremely straightforward to develop a driver for a new display very quickly.

The display driver API includes the following basic functions which must be supported by every display driver:

```
void PixelDraw(void *pvDisplayData, long lX, long lY,
               unsigned long ulValue);

void PixelDrawMultiple(void *pvDisplayData, long lX, long lY,
                       long lX0, long lCount, long lBPP,
                       const unsigned char *pucData,
                       const unsigned char *pucPalette);

void LineDrawH(void *pvDisplayData, long lX1, long lX2,
               long lY, unsigned long ulValue);

void LineDrawV(void *pvDisplayData, long lX, long lY1,
               long lY2, unsigned long ulValue);

void RectFill(void *pvDisplayData, const tRectangle *pRect,
              unsigned long ulValue);

unsigned long ColorTranslate(void *pvDisplayData,
                             unsigned long ulValue);

void Flush(void *pvDisplayData);
```

The actual names of these functions are not important since they are provided to the graphics library by means of a function pointer table. This table can be found in the tDisplay structure that the display driver exports and which the application uses when calling the graphics API function. This structure is defined in grlib.h.

Additionally, the display driver typically provides an initialization function that the application is expected to call prior to initializing the graphics library. This call is used to initialize the underlying graphics hardware and clear the screen.

You will notice that the driver API contains significantly fewer graphics primitives than the low-level graphics API. Most graphics primitives are broken down by the higher level code and passed to the driver in pieces. For example, an unfilled rectangle is drawn using two calls to the LineDrawV function and two calls to the LineDrawH function. Similarly, text is rendered using multiple calls to the PixelDraw and LineDrawH functions. This model works well with small, low-cost displays which do not typically include any graphics acceleration hardware but do often include the ability to choose drawing direction and copy lines of pixels.

The other higher level feature carried out by the low-level graphics layer on behalf of the display driver is clipping. No coordinates that are outside the bounds of the display are ever passed to the display driver since this is checked for and handled in the layer above. Using this approach, it becomes quick and easy to produce a new graphics driver since only a small number of simple functions need to be developed.

## 3.1 Off-Screen Display Drivers

Although most display drivers are intended to allow specific hardware displays to be used with the Stellaris Graphics Library, three special drivers are included within the library itself. These drivers are intended for off-screen graphics rendering in 1 bpp (bit per pixel), 4 bpp, and 8 bpp formats and are typically used in combination with a driver which supports the physical display. These drivers support the standard display driver interface and may be used alongside other display drivers.

The main use for an off-screen display driver is to support applications which require smooth animation or which render an image slowly. In these cases, an image is drawn into a memory buffer using the off-screen display driver and, once the image is completed, it is transferred to the physical display in one operation. Since the rendering of the image takes many steps and may include erasing the entire buffer before starting to redraw, using an off-screen display driver allows flicker-free operation. The physical display continues to show the previous image until a new one is ready for display at which point the image is updated so quickly that the user does not see any of the intervening graphics operations that were required to generate the new image.

For an example of the use of the off-screen display driver, see the source for the qs-scope example application from the ek-lm3s3748 StellarisWare release. This application implements a simple oscilloscope and renders the waveform into an off-screen buffer before updating the actual display, resulting in smooth, flicker-free display updates.

Source for the off-screen display drivers can be found in the offscr1bpp.c, offscr4bpp.c, and offscr8bpp.c files in the C:\StellarisWare\grlib directory.

## 3.2 Individual Display Driver Functions

This section describes the individual display driver functions in detail. Note that the first parameter to each function, pvDisplayData, is a pointer that the driver itself provides in the tDisplay structure it exports. The driver does not need to use this parameter, but it is provided to support drivers which must maintain state data.

### 3.2.1 Init

The prototype for the driver initialization function is driver-specific. An application calls this function directly prior to initializing the low-level graphics API layer and the function then initializes the display hardware and blanks the screen in preparation for receiving other calls.

### 3.2.2 ColorTranslate

```
unsigned long ColorTranslate(void *pvDisplayData,
                             unsigned long ulRGBColor);
```

The higher level graphics driver APIs make use of a standard 24-bit RGB color description with the color described in a single, unsigned long value with the red component in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Different displays, however, describe color in different ways so the ColorTranslate function allows the graphics library to obtain a representation of a given RGB24 color in the native format supported by the display.

For monochrome displays, the returned value should represent the brightness of the supplied RGB color. For color displays, the returned value should represent the original color as closely as possible given the constraints of the display. If the display supports 16-bit RGB, for example, the returned color value truncates, masks, and shifts the supplied 8-bit R,G, and B samples into a correctly packed 16-bit value.

All other calls to the display driver will be passed pre-translated colors (that is, colors which have been returned from a previous call to this ColorTranslate function), so the overhead of color translation is kept to a minimum.

### 3.2.3 PixelDraw

```
void PixelDraw(void *pvDisplayData, long lX, long lY,
               unsigned long ulDispColor);
```

The simplest function that a display driver must support is the ability to plot a single pixel at a given position on the display. This function plots a pixel using color ulDispColor at position (lX, lY) on the screen. Note that the color passed has already been translated into the display-dependent format using a previous call to the ColorTranslate function.

### 3.2.4 PixelDrawMultiple

```
void PixelDrawMultiple(void *pvDisplayData, long lX, long lY,
                       long lX0, long lCount, long lBPP,
                       const unsigned char *pucData,
                       const unsigned char *pucPalette);
```

The PixelDrawMultiple function is used when displaying images. A block of pixel data representing a given horizontal span is passed to the display driver which renders the pixel data onto the display at the specified position. In this case, the driver must support 1 bpp, 4 bpp, and 8 bpp pixel formats. Displays supporting 16 bpp formats may also support native 16 bpp pixels (since these are output by, for example, the JPEG image decoder included in the StellarisWare release for dk-lm3s9b96 and rdk-idm-sbc).

For the 1 bpp pixel format, the pucPalette points to a 2-entry array containing pre-translated colors for background and foreground pixels.

For 4 bpp and 8 bpp formats, the pucPalette parameter points to a color table containing RGB24 colors which the driver must translate to the native color format during the drawing process.

The palette is ignored for the 16 bpp formats since it is assumed that the pixels passed are in the native color format of the display.

When using 1 bpp and 4 bpp formats, the lX0 parameter indicates where the first pixel to draw is within the first byte of supplied pixel data. For 1 bpp, valid values are 0 through 7 and for 4 bpp, values 0 or 1 may be used. In each case, pixels are packed with the leftmost pixel in the most significant bit or nibble of the byte. Taking 1bpp as an example, if lX0 is 5 this indicates that we skip the 5 leftmost pixels in the first byte passed and will draw 3 pixels from that byte. These will be taken from bits 2 , 1 and 0 of the byte.

### 3.2.5 LineDrawH

```
void LineDrawH(void *pvDisplayData, long lX1, long lX2,
               long lY, unsigned long ulDispColor);
```

This function draws horizontal lines using the supplied, display-dependent color. Note that the line drawn includes both the first and last pixels specified by parameters lX1 and lX2 which means that the number of pixels written is ((lX2 – lX1) + 1). The graphics library ensures that lX2 is always greater than lX1, so no parameter sorting is required in the display driver.

### 3.2.6 LineDrawV

```
void LineDrawV(void *pvDisplayData, long lX, long lY1,
               long lY2, unsigned long ulDispColor);
```

This function draws vertical lines using the supplied, display-dependent color. As for LineDrawH, the line drawn includes both the first and last pixels specified by parameters lY1 and lY2 meaning that the number of pixels written is ((lY2 – lY1) + 1). The graphics library ensures that lY2 is always greater than lY1 so no parameter sorting is required in the display driver.

### 3.2.7 RectFill

```
void RectFill(void *pvDisplayData, const tRectangle *pRect,
              unsigned long ulDispColor);
```

This function fills a rectangle on the display with the solid color provided in the ulDispColor parameter.

Note that the tRectangle type uses a bottom-right inclusive definition so the width of the rectangle to draw is given by ((pRect->sXMax – pRect->sXMin) + 1) and the height is ((pRect->sYMax – pRect->sYMin) + 1). This is different from Windows and various other graphics libraries which use a bottom-right exclusive rectangle definition.

### 3.2.8 Flush

```
void Flush(void * pvDisplayData);
```

The flush function is provided to support display hardware which does not contain an integrated frame buffer and where the display driver must keep the display contents in a local RAM buffer. In this model, the drawing functions provided by the driver update the contents of the RAM buffer instead of updating the display. These changes are flushed to the actual display using the Flush API.

In drivers which update the display on each call to any of the driver drawing APIs, this call can be a stub which returns without performing any action.

Note that the widget classes do not currently make use of the Flush() driver API so, if you plan to use a driver which performs off-screen rendering with the widget layer, you must update the widget classes you are using to call Flush() at appropriate points in their paint functions.

## *3.3 Existing StellarisWare Display Drivers*

The following is a list of the graphics library-compatible display drivers provided in StellarisWare and information on the type of hardware (if any) they support and the connection method to the Stellaris microcontroller.

**Table 2. StellarisWare Graphics Library-Compatible Display Drivers**

| Driver Name | Board(s) | Part Number | Conn. | Type |
|---|---|---|---|---|
| offscr1bpp.c | All | N/A | N/A | 1bpp off-screen rendering driver within grlib. |
| offscr4bpp.c | All | N/A | N/A | 4bpp off-screen rendering driver within grlib. |
| offscr8bpp.c | All | N/A | N/A | 8bpp off-screen rendering driver within grlib. |
| cfal96x64x16.c | ek-lm4f232 | Crystalfontz CFAL9664B-F-B1 | SSI | 96x64 16bpp OLED display. SSD1332 controller. |
| formike128x128x16.c | ek-lm3s3748 | KWH015C04-F01 | GPIOx8 | 128x128 16bpp color CSTN display. |
| formike240x320x16_ili9320.c | rdk-idm | KWH028Q02-F03/F05 | GPIOx8 | 320x240 16bpp TFT display with touch screen. ILI9320 and ILI9325 controllers. |
| kitronix320x240x16_ssd2119.c | rdk-idm-l35 | K350QVG-V1-F | GPIOx16 | 320x240 16bpp TFT display with touch screen. SSD2119 controller. |
| kitronix320x240x16_ssd2119_idm_sbc.c | rdk-idm-sbc | K350QVG-V1-F | GPIOx8 | 320x240 16bpp TFT display with touch screen. SSD2119 controller. |
| kitronix320x240x16_ssd2119_8bit.c | dk-lm3s9b96 | K350QVG-V1-F | GPIOx8 + EPI HB8 | 320x240 16bpp TFT display with touch screen. SSD2119 controller. EPI HB8 connection used when optional SRAM/Flash/LCD or FPGA expansion board is installed. |

**Table 1. StellarisWare Graphics Library-Compatible Display Drivers (continued)**

| Driver Name | Board(s) | Part Number | Conn. | Type |
|---|---|---|---|---|
| kitronix320x240x16_fpga.c | dk-lm3s9b96/ dk-lm3s9d96 + FPGA expansion board | K350QVG-V1-F + DK-LM3S9B96-FPGA | EPI (GP mode) | 320x240 TFT display with touch screen. Display interface via control registers implemented in the expansion board FPGA. Contains extensions for video image mixing. |

In addition to the grlib-compatible drivers, other drivers for lower resolution monochrome displays are also included. Although these do not operate with the Stellaris Graphics Library, they may be of use as examples if developing a driver for a similar display or using a similar hardware connection. Table 2 shows these sample drivers.

**Table 2. Sample Lower Resolution Monochrome Displays**

| Driver Name | Board(s) | Part Number | Conn. | Type |
|---|---|---|---|---|
| display96x16x1.c | ek-lm3s811 | RIT 9913701000 & OSRAM OS096016 | $I^2C$ | 96x16 OLED monochrome display. |
| rit128x96x4.c | ek-lm3s6965<br>ek-lm3s8962<br>ek-lm3s1968<br>ek-lm3s2965 | RGS13128096WH000 | SSI | 128x96 OLED display supporting 4bpp (16 grey levels). |
| osram128x96x4.c | ek-lm3s2965_revA<br>ek-lm3s6965_revA | OS128064PK10MG1B10 | SSI | 128x96 OLED display supporting 4bpp (16 grey levels). |

### *3.4 Writing Your Own Display Driver*

Working from an existing Stellaris Graphics Library display driver as an example and assuming that you already have an understanding of the new display hardware you are trying to support, you can typically develop a new display driver in a day or less. The recommended steps are:

1. Find an existing StellarisWare display driver that is as close as possible to the hardware you are trying to support.
2. Take a copy of the existing driver's C and H source files, renaming them to something suitable for your new hardware.
3. Search for and replace all instances of the previous driver's function name prefix with one suitable for your new hardware. Using the Formike 128x128 CSTN driver shipped with ek-lm3s3748 as an example, all its exported function names begin "Formike128x128x16".
4. Rename the global tDisplay structure containing the driver entry points and screen dimensions to something suitable for your new hardware.
5. Update the contents of the tDisplay structure to include the correct dimensions for your new display and ensure that the function pointers it contains have been correctly updated to reflect your newly named functions.
6. Replace the function bodies for each of the driver functions with code suitable for your new hardware. Depending on how closely your new hardware matches the device supported by the driver you are using as an example, you may be able to retain the low-level GPIO, SSI, or $I^2C$ communication functions in the driver. If this is the case, you must update the GPIO and peripheral pins used to match your own board.
7. Create a copy of the "hello" example application from a StellarisWare release and modify it to use your new driver. This should involve changing the function call which initializes the display driver and changing the name of the global tDisplay structure whose pointer you pass to GrContextInit. You may have to modify the coordinates of the various elements drawn by the application depending upon the dimensions of your new display compared to the previous one that the application was using.

8. Build your new “hello” version and debug your driver using this as a starting point. Once “hello” is running, move on to “grlib_demo” which is more complex but exercises a lot more of the graphics library and display driver including the widget layer.

## 4 Conclusion

With its simple API and sample source code for several existing displays, the Stellaris Graphics Library display driver layer provides an easy way to offer graphics support on your new board with minimal software development required to get a new display up and running.

## 5 References

The following related documents and software are available on the Stellaris web site at www.ti.com/stellaris:

- *Stellaris Graphics Library User’s Guide*, publication SW-GRL-UG
- Stellaris Graphics Library Standalone Package
- StellarisWare Driver Library Standalone Package

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

| **Products** | | **Applications** | |
|---|---|---|---|
| Audio | www.ti.com/audio | Communications and Telecom | www.ti.com/communications |
| Amplifiers | amplifier.ti.com | Computers and Peripherals | www.ti.com/computers |
| Data Converters | dataconverter.ti.com | Consumer Electronics | www.ti.com/consumer-apps |
| DLP® Products | www.dlp.com | Energy and Lighting | www.ti.com/energy |
| DSP | dsp.ti.com | Industrial | www.ti.com/industrial |
| Clocks and Timers | www.ti.com/clocks | Medical | www.ti.com/medical |
| Interface | interface.ti.com | Security | www.ti.com/security |
| Logic | logic.ti.com | Space, Avionics and Defense | www.ti.com/space-avionics-defense |
| Power Mgmt | power.ti.com | Transportation and Automotive | www.ti.com/automotive |
| Microcontrollers | microcontroller.ti.com | Video and Imaging | www.ti.com/video |
| RFID | www.ti-rfid.com | | |
| OMAP Mobile Processors | www.ti.com/omap | | |
| Wireless Connectivity | www.ti.com/wirelessconnectivity | | |
| | **TI E2E Community Home Page** | | e2e.ti.com |

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265