Application Note

LVGL Use On MSPM33C

ABSTRACT

The Light and Versatile Graphics Library (LVGL) is an open source graphics library that has been integrated into the MSP M33 Software Development Kit (SDK). This software solution creates a middleware layer that is hardware independent and highly customizable. With LVGL and the MSPM33, users can easily create embedded Graphical User Interfaces (GUI) that can be easily written for and ported to any combination of MSPM33 microcontroller and supported display. This application note describes several programming examples to illistrate how LVGL can be conmfigured to create GUI's for several End Equipment (EE). The objective is to help users understand GUI development with the LVGL library.

The code examples were tested on the MSPM33C321A, however, the examples can be easily adpated to run on any MSPM33 device. Most of the examples use either the Serial Peripheral Interface (SPI) or Quad Serial Peripheral Interface (QSPI) module to send the LVGL pixel data to the display, and the Inter-Integrated Cirtual (I2C), SPI, or QSPI, module to read touch input from the display. This requirement can be met by selecting a display with an integrated display driver supporting either SPI or QSPI, and an integrated touch driver that supports either I2C, SPI, or QSPI. An osciliscope with enough digital lines for decoding SPI/QSPI signals is essential for debugging.

Table of Contents

List of Figures

Trademarks

1 Overview

LVGL provides a lightweight, open-source graphics library optimized for embedded systems. With high-performance rendering, flexible UI components, and broad hardware compatibility, LVGL enables seamless, modern GUI development for resource-constrained devices. Texas Instruments customers benefit from LVGL's efficient memory use, hardware acceleration support, and ease of integration, allowing them to create fast, visually appealing interfaces with minimal development effort.

LVGL has been integrated into the MSP M33 SDK as an optional middleware library. To utilize LVGL, the following main considerations must be made:

- Project Setup
- Configuration
- Initalization
- Graphics Output
- Graphics Input
- Graphics Update

This architecture is descibed in Figure 1-1.

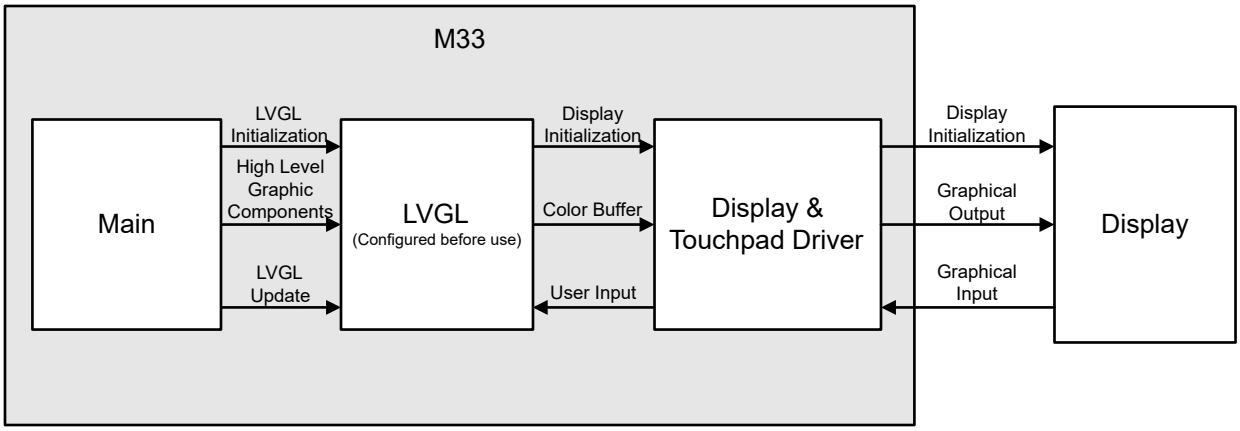

Figure 1-1. M33 LVGL Architecture

1.1 LVGL Project Setup

LVGL has been integrated into the M33 SDK, however, the library must be linked into the project. The following steps can be followed to perform this linking:

- Right click on the project and select "Add Files/Folders..."
- In the "Add Files/Folders" window, select the "+" sign to add the LVGL library
- From the "Select files/folders" window, change the dropdown from "Select files to add" to "Select folders to link"
- Then select the base path of the SDK LVGL library by clicking on the "..." icon near "Path".
  - The SDK LVGL library is located at "C:\ti\mspm33_sdk_XX\source\third_party\lvgl" where "XX" is the latest SDK version.
- Click "Ok" on the "Select files/folders" window
- Finally, click "Ok" on the "Add Files/Folders" window
- The SDK LVGL library should now be included in the project

The library has the most basic configuration and no low level drivers. To configure the library for a specific application and add low level drivers for a specific display, five files must be added to the project and adjusted:

- lv_conf.h: The main configuration file for LVGL
- lv_port_disp.h: The header file for the LVGL display driver
- lv_port_disp.c: The source file for the LVGL display driver
- lv_port_indev.h: The header file for the LVGL input device driver
- lv_port_indev.c: The source file for the LVGL input device driver

1.2 Configuration

LVGL is highly configurable, with the configuration of the library being done in the *lv_conf.h* file. There are several fields in this file that must be update so that the library works properly. These fields are:

- MY_DISP_HOR_RES: Used to set the horizontal size of the internal color buffer, dependent on the display being used
- MY_DISP_VER_RES: Used to set the vertical size of the internal color buffer, dependent on the display being used
- LV_COLOR_DEPTH: Used to set the color format for the color buffer, dependent on the display being used

- LV_MEM_SIZE: Used to set the amount of dynamic memory to use for storing objects and animations, typically 48 kB
- LV_DRAW_COMPLEX: Used to enable complex shapes, such as rounded corners, circles, and arcs

There are, also, several *lv_conf.h* sections that can be adjusted to reduce the applications memory footprint. These fields are:

- Fonts: Disable unused fonts to reduce the required flash size
- Widgets: Disable unused widgets/demos to reduce the required flash size
- Demos: Disable unused demos to reduce the required flash size

1.3 Initialization

When creating an embedded GUI using LVGL, the display being used must be initalized and LVGL must be configured to work with the display. This initalization and configuration is done through the LVGL global function *lv_port_disp_init*. This function is broken into three sections:

- Display Initalization
  - LVGL includes a global function, *disp_init*. This function is called in *lv_port_disp_init* and can be used to trigger the initalization sequence for the display being used.
- Color Buffer Allocation and Refresh Mode Selection
  - LVGL uses a color buffer to store the pixel data for a rendered image. This buffer, lv_disp_draw_buf_t, is allocated in *lv_port_disp_init*. There are two typical buffering configurations:
    - Single Buffer: A single buffer is allocated and passed into *lv_disp_draw_buf_init*. The size of this buffer is typically ten times the horizontal resultion (10 rows of pixels). This buffer is solely populated with the pixel data for a rendered image and the CPU is responsible for transfering the color buffer's pixel information to the communication interface used to communication with the display.
    - Double Buffer with DMA: Two buffers are allocated with the same amount of memory and passed into *lv_disp_draw_buf_init*. The size of both buffers is typically 10 rows of pixels. LVGL populates one buffer with part of the pixel data for a rendered image and then makes the data available for the DMA to begin transfering data to the communication interface.
  - There are, also, two modes to configure how LVGL performs image refreshes:
    - Partial Refresh: In partial refresh, only the portion of the rendered image that is being updated is written to the color buffer. This is enabled by default. This mode allows for faster refreshes and a smaller color buffer, however, can result in residual display artifacts if the display is partial to tearing or banding.
    - Full Refresh: In full refresh, the entire rendered image is written to the color buffer. This is enabled by setting the *full_refresh* field in the *lv_disp_drv_t* display driver. This mode elimiates the issue of residual display artificacts, but is much slower and requires a color buffer that is equal to the resolution of the screen.
- Display Driver Configuration
  - LVGL provides a display driver structure that must be configured so that LVGL properly renders the display. The first step is initializing the display driver by calling *lv_disp_drv_init* with a new instance of the driver, *lv_disp_drv_t*
  - After the display driver has been initalized, the main values that must be configured are:
    - The horizontal and veritical resolutions: The horizontal and vertical resolutions, *hor_res* and *ver_res*, are members of the display driver and can be set with the previously defined MY_DISP_HOR_RES and MY_DISP_VER_RES
    - The flush callback: The display driver must be configured with the location of the flush callback, *flush_cb*. The flush callback is the method that is called when LVGL is to flush the color buffer that has been populated with the pixel data of the rendered image. This callback method is discussed further in this application note
    - The draw buffer: The display driver must be configured with the location of the color buffer that was previosuly initalized, *draw_buf*
    - The refresh mode: If using the full refresh mode, the display driver must be configured with *full_refresh* being set to 1
  - Finally, the display driver must be registered with LVGL, by calling *lv_disp_drv_register*

LVGL has support for several types of input devices, including a touchpad, mouse, keypad, encoder, or button. All of these input devices must initalized and configured with LVGL. This is performed in the LVGL global function *lv_port_indev_init*, and is broken into two sections:

- Input Device Initalization
  - LVGL includes several global functions that can be used to initalize the input device. For example, when using a display that has either resistive or capacitve touch, the global function *touchpad_init* can be called to perform the touchpad initialization.
- Input Device Driver Configuration
  - LVGL provides an input device driver structure that must be configured so that LVGL properly processes inputs. The first step is initializing the input device driver by calling *lv_indev_drv_init* with a new instance of the driver, *lv_indev_drv_t*
  - After the input device driver has been initalized, the main values that must be configured are:
    - The input device type: The type of input device being used: pointer (touchpad or mouse), keypad, button, and encoder
    - The input read callback: The input device driver must be configured with the location of the input read callback, *read_cb*. The input read callback is the method that is called when LVGL is to read for any input from the input device. This callback method is discussed further in this application note
  - Finally, the input device driver must be registered with LVGL, by calling *lv_indev_drv_register*

1.4 LVGL Output

When LVGL is ready to flush the color buffer, LVGL calls the flush callback method that was registered in the configuration step. The flush callback is where the color buffer is processed and sent to the display specific communication interface. Typical processing methods are:

- Individual Pixel Flush: Pixel data is writen to communication interface one-by-one and sent to the display. This method is the most simple, but is also the slowest due to the required processing time by the CPU
- Full Buffer Flush: Pixel data is transfered to the communication interface through DMA and sent to the display. This method is more complex than the previous method due to the DMA, however, the DMA usage allows LVGL to begin writing the next part of the rendered image to a second color buffer, making this method much faster.

After the color buffer has been processed, LVGL must be notified that the color buffer can be flushed by calling *lv_disp_flush_ready*.

1.5 LVGL Input

When LVGL is ready to read for input from the input device, LVGL calls the input read callback method that was registered in the configuration step. The input read callback is where the input device is polled for input information. The typical flow for reading from the input device is as follows:

1. Check if the input device has been interacted with and update the input device driver state with the corresponding interaction
2. Retrieve the input information from the input device. This is typically be the x/y location for a touchpad or the key number for a keypad. Set the input device driver fields corresponding to the type of input device with the retrieved input information

1.6 LVGL Update

For LVGL to properly perform animations and other tasks, LVGL requires a system tick, periodic timer triggers, and sleep management. Additional information on these requirements is listed below:

- System Tick: LVGL requires that it be notified of time elapsed. This is done by periodically calling the lv_tick_inc functional with the amount of time elapsed in milliseconds. This can be done either in a timer interupt or in the main while loop.
- Periodic Timer Triggers: LVGL requires that it have dedicated CPU time to handle timers and tasks. This is done by calling lv_timer_handler periodically in a similar fashion to the system tick.

- Sleep Management: In certain cases, LVGL requires that it be notified when the MCU goes into sleep mode. If using a timer interupt to call lv_tick_inc or lv_timer_handler, this interupt must be stopped so that LVGL returns to the same state when the MCU wakes up.

2 LVGL Example

The following LVGL example showcases the LP-MSPM33C321A connected to a TFT display, which utilizes the ST7796 display driver and DFT6636U capacitive touch driver. The ST7796 display driver communicates with the LP-MSPM33C321A through a SPI and the DFT6636U touch driver communicates with the LP-MSPM33C321A through an I2C interface.

2.1 Hardware Connections

The connections that need to be made between the LP-MSPM33C321A and the TFT display are documented in Table 2-1.

Table 2-1. LVGL Example Hardware Connections

LP-MSPM33C321A	TFT Display
PA8	SPI SCLK
PA9	SPI PICO
PA10	RESET
PB3	PWM
PB6	LCD SCS
PB30	LCD SDC
PC5 (Not used)	INT
PA0	SDA
PA1	SCL
PC4	RST

2.2 Software

The following describes the software configurations that were made for the LVGL example:

- Configuration
  - MCU: The following values are specifically configured for this example:
    - GPIO: Several GPIO pins are configured in this example to be initally set high or low, as defined in Table 2-2. These values are dependent on the requirements from the display and touch drivers.

Table 2-2. LVGL Example GPIO Pin Configuration

Pin Name	Direction	Initial Value	Assigned Pin
LCD_SCS	Output	Set	PB6
LCD_SDC	Output	Cleared	PB30
LCD_RESET	Output	Set	PA10
LCD_PWM	Output	Cleared	PB3
CTP_RST	Output	Set	PC4
CTP_INT	Input	N/A	PC5

    - SYSCTL: The SYSCTL module has been configured to use an external high frequency clock input to serve as the input to the MCLK. The HFXT is set to run at 40MHz
    - I2C: The I2C peripheral selected, UC3, has been configured to operate as a Controller and run in Fast Mode (400kHz), with SDA and SCL lines assigned to PA0 and PA1 respectively
    - SPI: The SPI peripheral selected, UC2, has been configured to operate as a Controller, run at 40Mhz, use CS1 as the Chip Select, and run in Motorola 4-wire mode, with SCLK, PICO, POCI, and CS1 being assigned to PA8, PA9, PB19, and PB6

    - TIMER: The Timer module selected, TIMG8_0, has been configured to be sourced by the BUSCLK, prescaled by a factor of 40, and setup in Periodic Down Counting at a period of 1ms. An interupt has been confiogured to be triggered at the Zero event of the timer
  - LVGL:
    - In this example, LVGL has been configured with the specifications of the TFT display being used. Many of the optional fonts and widgets are enable as the LVGL demo utilizes many different modules. The demo configuration value, LV_USE_DEMO_WIDGETS, is also specifically enabled so that the LVGL demo is compiled and run.

```
#define MY_DISP_HOR_RES        480
#define MY_DISP_VER_RES        320
...
#define LV_COLOR_DEPTH          16
...
#define LV_MEM_SIZE   (48U * 1024U) // [bytes]
...
#define LV_DRAW_COMPLEX 1
...
#define LV_USE_DEMO_WIDGETS 1
```

- Initalization
  - The first thing that needs to be initalized are the peripherals configured previously through SysConfig
  - Next, the timer interupt that is used to trigger the system tick needs to be initalized
  - After the peripherals and timers, the LVGL specific items need to be initalized
    - LVGL Initalization: LVGL needs to be initalized by calling *lv_init*
    - Output Initialization: The LVGL display driver is initalized by calling *lv_port_disp_init*
      - In this example, *lv_port_disp_init* follows the initalization steps defined in Section 1.3.The flush callback is defined later in this section

```
void lv_port_disp_init(void)
{
    disp_init(); // Initialize your display

    static lv_disp_draw_buf_t draw_buf_dsc_1;
    static lv_color_t buf_1[MY_DISP_HOR_RES * 10];                          /*A buffer
for 10 rows*/
    lv_disp_draw_buf_init(&draw_buf_dsc_1, buf_1, NULL, MY_DISP_HOR_RES * 10);   /
*Initialize the display buffer*/

    static lv_disp_drv_t disp_drv;                          /*Descriptor of a display
driver*/
    lv_disp_drv_init(&disp_drv);                    /*Basic initialization*/

    disp_drv.hor_res = MY_DISP_HOR_RES;
    disp_drv.ver_res = MY_DISP_VER_RES;

    disp_drv.flush_cb = disp_flush;

    disp_drv.draw_buf = &draw_buf_dsc_1;

    lv_disp_drv_register(&disp_drv);
}
```

    - Input Initialization: The LVGL input device driver is initalized by calling *lv_port_indev_init*
      - In this example, *lv_port_indev_init* follows the initalization steps defined in Section 1.3. The input read callback is defined later in this section

```
void lv_port_indev_init(void)
{
    static lv_indev_drv_t indev_drv;

    touchpad_init(); //Initialize your touchpad

    lv_indev_drv_init(&indev_drv); //Register a touchpad input device
```

```
    indev_drv.type = LV_INDEV_TYPE_POINTER;
    indev_drv.read_cb = touchpad_read;
    indev_touchpad = lv_indev_drv_register(&indev_drv);
}
```

  - Finally, the system tick timer is started

```
int main(void)
{
    SYSCFG_DL_init();

    NVIC_EnableIRQ(TIMER_0_INST_INT_IRQN);

    lv_init(); // init LVGL
    lv_port_disp_init();
    lv_port_indev_init();

    DL_TimerG_startCounter(TIMER_0_INST);
```

- Output
  - LVGL sends out pixel data by calling the flush callback method previously defined in the initalization section. In this example, *disp_flush* calls the low level driver code, *lvgl_LCD_Color_Fill*, which handles the SPI communcation, and then notifies LVGL that the flush is ready

```
static void disp_flush(lv_disp_drv_t * disp_drv, const lv_area_t * area, lv_color_t * color_p)
{
    lvgl_LCD_Color_Fill(area->x1, area->y1, area->x2, area->y2, color_p);
    lv_disp_flush_ready(disp_drv); //Inform the graphics library that you are ready with the
flushing
}
```

- Input
  - LVGL attempts to read for user input by calling the input read callback method previously defined in the initalization section. In this example, *touchpad_read* calls the low level driver code, which handles the I2C communcation, to check if the touchpad has been pressed, and then processes the location of the press if the press occurred

```
static void touchpad_read(lv_indev_drv_t * indev_drv, lv_indev_data_t * data)
{
    static lv_coord_t last_x = 0;
    static lv_coord_t last_y = 0;

    if(touchpad_is_pressed()) {
        touchpad_get_xy(&last_x, &last_y); //Save the pressed coordinates and the state
        data->state = LV_INDEV_STATE_PR;
    }
    else {
        data->state = LV_INDEV_STATE_REL;
    }

    data->point.x = last_x; //Set the last pressed coordinates
    data->point.y = last_y;
}
```

- Update
  - The LVGL system tick was triggered via the TIMG8_0 timer interupt configured earlier. This interupt triggered every millisecond.

```
void TIMER_0_INST_IRQHandler(void)
{
    switch (DL_TimerG_getPendingInterrupt(TIMER_0_INST)) {
        case DL_TIMER_IIDX_ZERO:

            lv_tick_inc(1);     //LVGL Heart Beat

            break;
        default:
            break;
```

```
    }
}
```

  - The LVGL timer triggers are being handled in the main while loop of the example. There is a delay of 320 cycle to not hog all of the CPU processing.

```
while (1) {
        // update LVGL task
        lv_task_handler();

        delay_cycles(320);
    }
```

- Main Demo
  - The main demo is defined in *lv_demo_widgets*, which is included in the LVGL library. This demo showcases the primarily widgets that are included in LVGL. To use this demo, the header file for the demo was included in the main source file for this demo.

```
#include "lv_demo_widgets.h"
...
int main(void)
{
...
    lv_demo_widgets();
...
}
```

2.3 LVGL Example Summary

After the hardware connections have made between the LP-MSPM33C321A and the TFT display, and the software changes have been compiled and flashed on the M33 device, the M33 and the TFT display communicate with each other to run the LVGL demo. In this demo, the display can be interacted with showing several widgets, including buttoms, charts, and images.

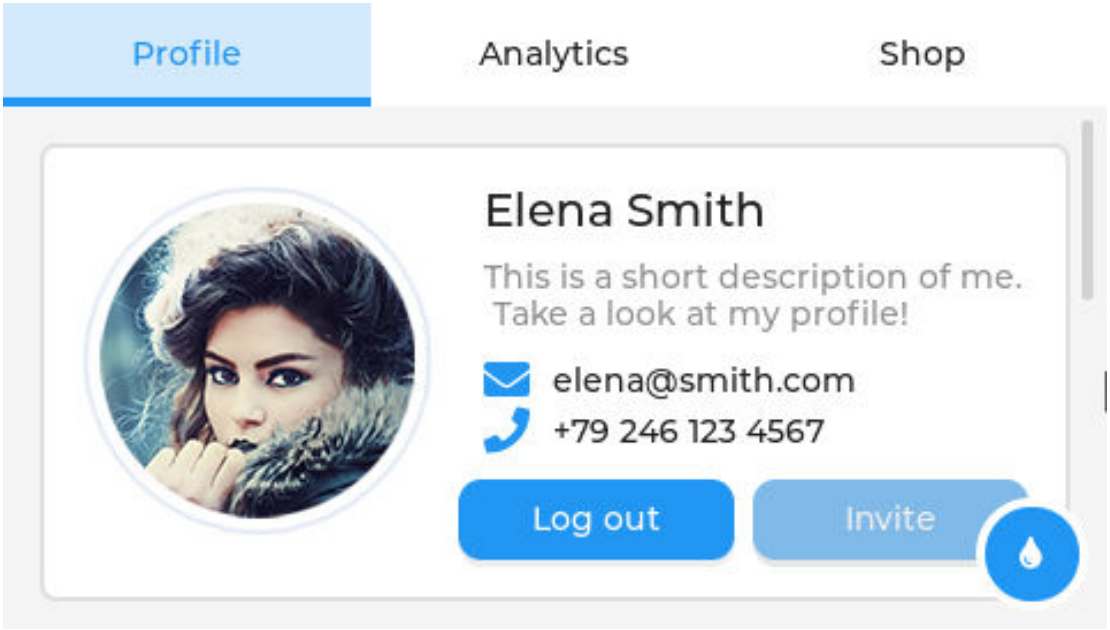

Figure 2-1. LVGL Demo User Interface

3 Summary

This document begins with an overview of LVGL and the considerations that need to be made to properly integrate it into a MSPM33 embedded GUI. The application note then describes an example of LVGL being used with a SPI display driver and I2C touchpad driver to replicate the base LVGL demo.

4 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

DATE	REVISION	NOTES
December 2025	*	Initial Release

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025