CYW20719 MIPI Display Interface Guide

Associated Part Family: CYW20719

Doc. No.: 002-23078 Rev. **

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com

Contents

1 Introduction

CYW20719 has a Mobile Industry Processor Interface (MIPI) Display Bus Interface (DBI) Type-C interface capable of driving embedded displays to create high-quality user interfaces (UIs). Moreover, CYW20719 does this while maintaining very low current consumption and simultaneously performing its regular Bluetooth/Bluetooth Low Energy functions. The open source U8glib library is used to provide an interface between the MIPI DBI-C hardware and the developer in Cypress WICED™ Studio. The library has been modified to optimize performance on the CYW20719 platform and is provided as part of WICED Studio.

This document will, first, provide a brief guide to get the developer off the ground quickly with our demo application. Next, a brief walkthrough of the code will familiarize the developer with the steps necessary to carry out their own design. Finally, the document will address in depth the RAM usage and expected frames-per-second (FPS) of their display implementation on CYW20719.

2 Quick Start with Display Demo on CYW20719

This section seeks to get the first-time user up and running quickly, using the sample code provided in the SDK. We will be using the Cypress CYW920719Q40EVB-01 evaluation board paired with the Freetronics 128x128 OLED (https://www.freetronics.com.au/products/128x128-pixel-oled-module), which uses the SSD1351 controller and is the default supported display/controller combination used by the WICED Studio demo application:

Figure 2-1. CYW920719Q40EVB-01

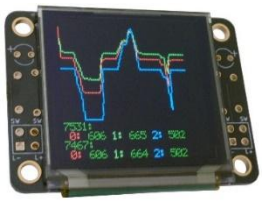


Figure 2-2. Freetronics 128x128 OLED

Step 1. Demo Hardware Setup

Connect the Freetronics OLED to the CYW920719Q40EVB-01 board using jumper wires according to the pin connections shown below in the table and image. In a later section, we will walk through the code that sets the pin numbers on CYW20719. We are currently using the default values as the demo code is written, but the pins can be muxed to any available IO.

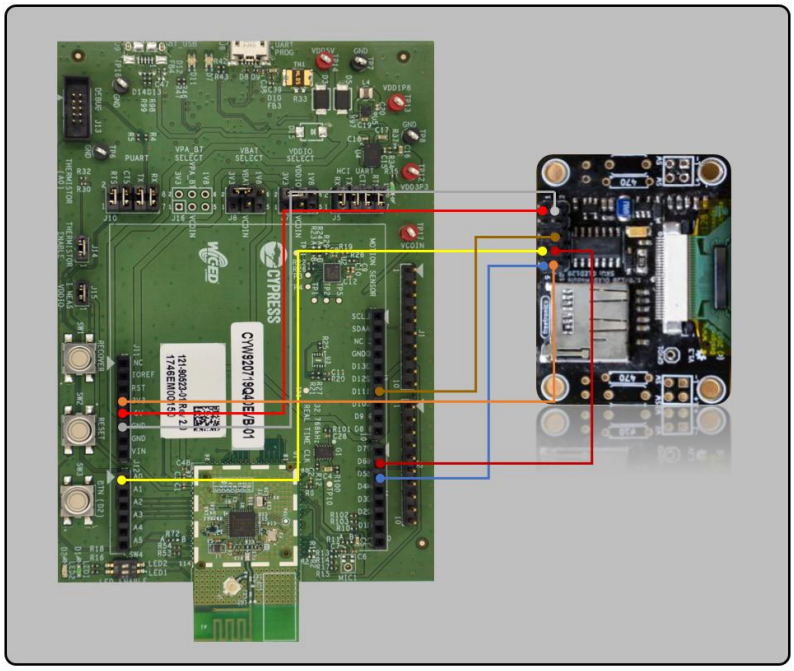


Figure 2-3. Default Pin Connections for CYW920719Q40EVB-01

	Function	CYW920719Q40EVB-01	OLED
SCK	P10	J12, Pin 1 (A0)	7
MOSI	P28	J3, Pin 7 (D11)	6
CS	P2	J4, Pin 2 (D6)	8
A0/DNC	P26	J4, Pin 3 (D5)	9
RESET	3.3V	J11, Pin 4	10
5V	5V	J11, Pin 5	1
GND	GND	J11, Pin 6	2

Table 2-1. Pin-to-Pin Mapping For Default Configuration in Demo App

Step 2. Power On and Reset the Hardware

For CYW920719Q40EVB-01:

1. Connect the CYW920719Q40EVB-01 board to your computer via USB.
2. Press and hold SW1 (“RECOVER”).
3. Press and release SW2 (“RESET”).
4. Release SW1.

Note: This reset may not need to be performed depending upon your version of WICED Studio. However, the reset does not affect the download procedure.

Step 3. Building and Downloading the Display Demo Application

Note: If you have not yet installed WICED Studio, see the CYW920719Q40EVB-01 kit user guide.

1. Edit the existing make target for the display demo so that it is able to find your board. Use the following make target to download to your board, where ‘x’ is the COM port of your device as found in your Device Manager (for Linux/Mac, find the device with the `ls /dev/` command):

   For CYW920719Q40EVB-01:

- Windows: `snip.hal.display- CYW920719Q40EVB_02 download UART=COMx`
- Linux/Mac: `snip.hal.display- CYW920719Q40EVB_02 download UART=/dev/tty.usbserial-x`

2. Double-click the make target to compile and download the project to your board.

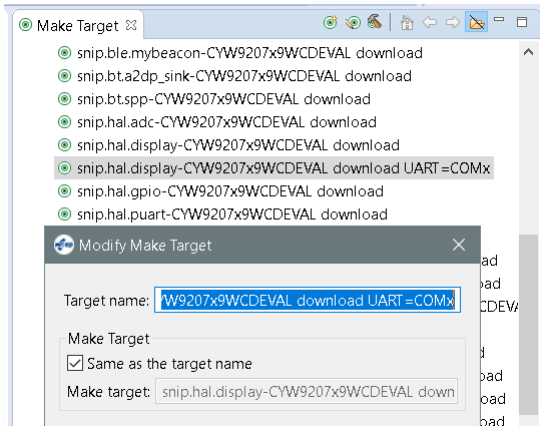


Figure 2-4. Make Target Editing to Input COM Port

Step 4. Press Reset on your board (SW1). The demo boots and then switches between screens:

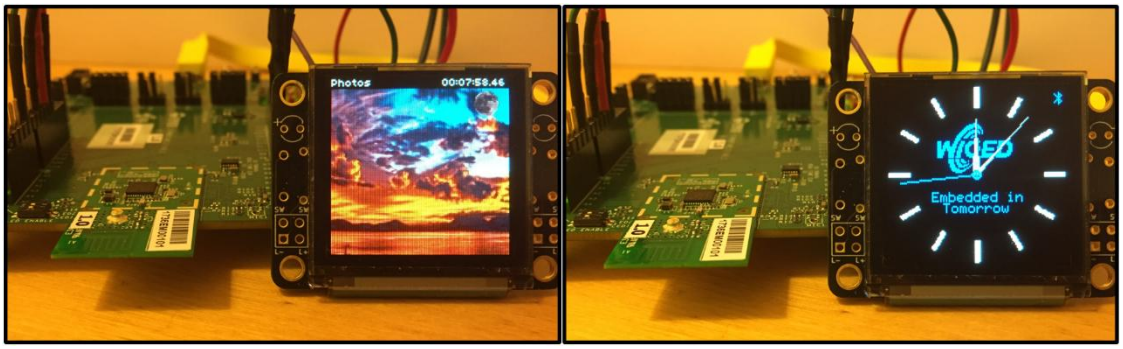


Figure 2-5. Display Demo Expected Result

Note: To avoid burn-in of the OLED pixels, avoid leaving the same image on the OLED for extended periods (either power down or switch the image periodically). Burn-in will cause the image that was left for an extended duration to be burned into the pixels permanently—leaving a faint outline no matter what is written to the pixels.

3 Walk-through of Display Demo Sample Code

3.1 Basic Display Loop Use

For a more extensive walk-through of the picture loop implemented by u8g, see the documentation provided in the SDK:

`20719-B1_Bluetooth/doc/u8g/u8glib-pictureloop.htm`

The picture loop is the core component of any code you will write to operate the display. Hardware initialization must occur once prior to calling the loop. Hardware initialization will essentially attach your u8g_t instance to the physical hardware. From that point forward, any APIs you use from the u8glib will require that you pass a pointer to this instance as an argument so that the library knows to which physical hardware to perform the operation:

```
static u8g_t u8g;
u8g_t * u8g_p = &u8g;
u8g_InitSPI_cy(u8g_p, &u8g_dev_ssd1351_128x128_hicolor_65k_hw_spi, 10, 28, 2, 26, 255);
```

Figure 3-1. U8G Initialization

Following the hardware initialization, you must register a function to be executed periodically. This could be as simple as a periodic app timer or a button interrupt handler (if you only want the screen to change on a button press). In the display demo code, however, the function is registered as an idle-time loop. This means that whenever the system scheduler determines that it will remain idle after a time threshold, it will call your function. This is ideal for running a display because the rate at which the function is called is maximized without affecting other components of the system:

```
u8g_RegisterIdleLoopHandler(display_loop);
```

Figure 3-2. Register Idle-Time Loop

Put your picture loop within the function registered as an idle time loop. The picture loop is the core component of the operation of the display – you perform the actual display writes in this loop. A basic loop that writes the whole screen will start with setting the cursor to the first page. Then, it enters a loop performing all drawing operations for each page of the display buffer until there is none left. When this function exits this loop, it must call u8g_ScreenRefresh if it is registered as an idle-time loop:

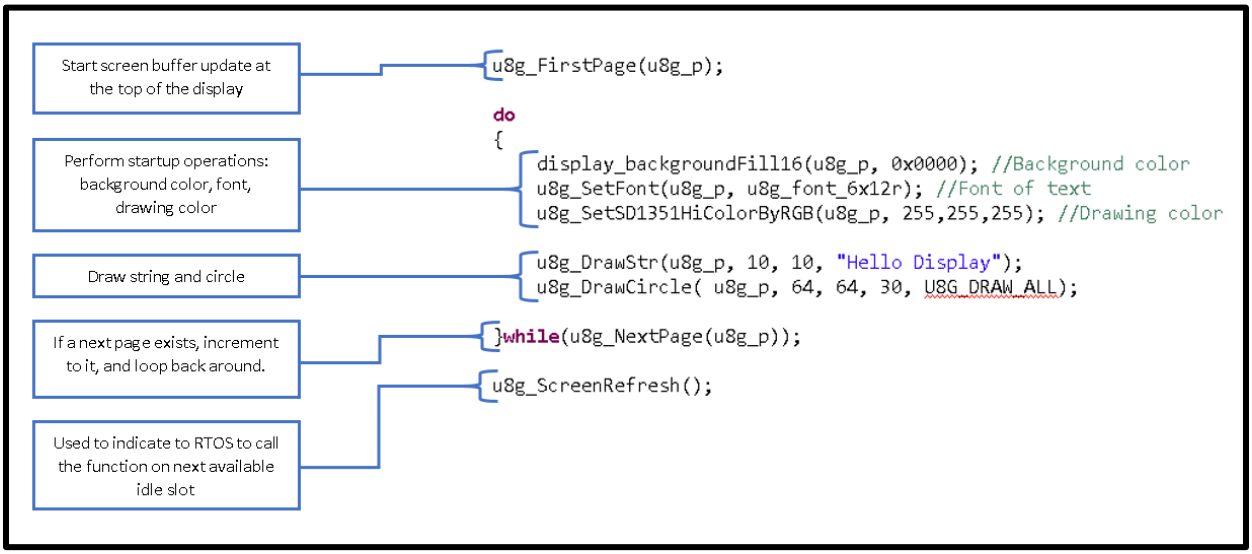
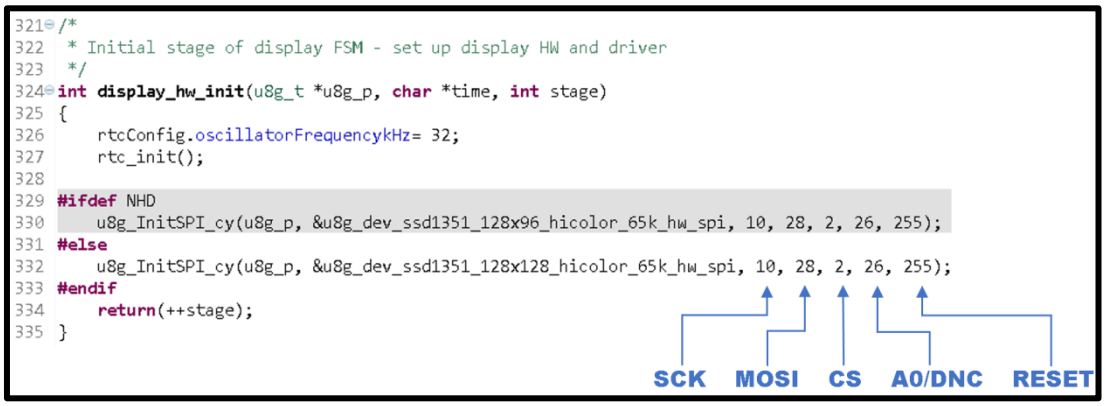


Figure 3-3. Basic Picture Loop

3.2 Setting I/O Mux for MIPI DBI-C Pins

CYW920719 is able to mux the MPI-DBI pins onto any GPIO exposed on the evaluation board. To customize pin numbers, you can reference the demo code on line 324 of *display_demo.c*. In this function, the demo initializes the hardware interface with the default pins. You can change these pin numbers to whichever best suit your design with a few limitations:




```
321 /*
322  * Initial stage of display FSM - set up display HW and driver
323  */
324 int display_hw_init(u8g_t *u8g_p, char *time, int stage)
325 {
326     rtcConfig.oscillatorFrequencykHz= 32;
327     rtc_init();
328
329 #ifdef NHD
330     u8g_InitSPI_cy(u8g_p, &u8g_dev_ssd1351_128x96_hicolor_65k_hw_spi, 10, 28, 2, 26, 255);
331 #else
332     u8g_InitSPI_cy(u8g_p, &u8g_dev_ssd1351_128x128_hicolor_65k_hw_spi, 10, 28, 2, 26, 255);
333 #endif
334     return(++stage);
335 }
```

Figure 3-4. Changing the Pins

Note: The reset pin is not always necessary to use. If it is not used, it can be set to 255 as it is in the demo code.

3.3 U8G Lib APIs and Documentation

The u8glib is an open-source library that has been interfaced to the CYW920719 platform. Most of the APIs provided are documented and fully compatible with the use of the standard open source documentation of the library. Documentation modified for Cypress-specific changes to the library is available at the following location in the SDK:

20719-B1_Bluetooth/doc/u8g/u8glib.htm

4 Using Your Own Display Controller

4.1 Switching to a Supported Driver

To see which display controllers are already supported, navigate to *20719-B1_Bluetooth/apps/snips/hal/display/drivers/*

Every controller currently supported by u8glib is listed in this folder. These drivers are not fully functional for immediate use; they must be modified to be compatible with the Cypress implementation of u8glib. To see an example of the modification necessary, compare the following files:

20719-B1_Bluetooth/apps/snips/hal/display/drivers/u8g_dev_ssd1351_128x128.c

20719-B1_Bluetooth/apps/snips/hal/display/drivers/u8g_dev_ssd1351_128x128_cy.c

Note: See the documentation in WICED Studio for further details on the changes that must be made: *20719-B1_Bluetooth/doc/u8g/u8g_display_driver_design.htm*.

Once the driver is properly adapted, modify the application makefile to compile the new driver instead of the default:

`20719-B1_Bluetooth/apps/snips/hal/display/makefile.mk`

4.2 Adding a Driver for an Unsupported Controller

See the following documentation in WICED Studio for an extensive guide on adding a driver for an unsupported display controller: *20719-B1_Bluetooth/doc/u8g/u8g_display_driver_design.htm*

4.3 Limitations on Display Size

Due to the size of the MIPI DBI-C FIFO of CYW20719, the size of displays that can be used is limited. The maximum width of display supported is a function of the color depth. Table 4-1 lists the limitations per depth:

Color Depth	Max Display Width
8-bit	1024 pixels
16-bit	512 pixels
24-bit	341 pixels

Table 4-1. Display Horizontal Pixel Limitation

5 RAM Usage

The RAM usage that results from the implementation of a display in your design consists of two components: (1) the memory utilized by the u8glib itself, and (2) the application-specific implementation.

In summary, the library consumes 10–20 KB of RAM, while the application can consume negligible amounts of RAM if properly implemented using vector graphics. The following sections explore these points in depth.

5.1 Display Library

The display library itself can consume *up to* approximately 20 KB of RAM. The library is loaded dynamically depending upon the application use of specific functions. A small “Hello World” display application consumes approximately 10 KB of RAM as a result of u8glib usage (i.e., not including the application code). As more and more complex drawing functions are used, the size of the library included in RAM gets closer to 20 KB, but never exceeds it.

5.2 Application-Level Display Implementations: Fonts

The number and type of fonts used has a large impact on the RAM utilization of your display application. Individual fonts can consume anywhere from 200 bytes to over 5 KB of RAM. Most use cases require only a single font, in the range of 1–2 KB.

5.3 Application-Level Display Implementations: Vector Graphics

The application largely determines the overall RAM usage. Specifically, the decision to use vector graphics rather than bitmaps is the key to lowering the application’s RAM usage to very low levels. Vector graphics trade off a small increase in CPU usage for a large reduction in RAM usage; instead of storing every image in the form of bitmaps, vector graphics allow us to dynamically generate the image at the cost of additional computation.

The use of vector graphics is what makes the u8glib practical for small embedded processors. *For individual vector graphics, you can often consider the RAM usage negligible* (even if it is a full screen shape), because it requires just a few added instructions to call functions that already exist in u8glib.

Using bitmaps, on the other hand, results in significant RAM usage. The usage can be computed logically as a function of screen size and color depth. For example, a bitmap with a 16-bit depth to cover an entire 128x128 display will consume 128*128*2 = ~33 KB of RAM. Given system memory constraints, it is clear that this is not a sustainable way to code every screen in your embedded application.

Furthermore, many applications benefit from a combination of vector graphics and bitmaps because some shapes are difficult to create through vector graphics alone. However, the bitmaps used will often be very small and can often be implemented as binary bitmaps (one bit per pixel, unicolor). For example, to draw a blue circular icon with a white Bluetooth logo inside, you can use vector graphics to draw the circle and overlay a small binary bitmap of the Bluetooth logo on top of the circle. This allows drawing a logo with two colors. An equivalent bicolor icon that is only a bitmap requires at least 1 byte per pixel and must cover the square image of the entire circle instead of only the small logo inside the circle.

5.4 Storage of Bitmaps

If bitmaps are used, it may be beneficial to move them into on-chip flash (OCF) to save SRAM. CYW20719 has 1 MB of OCF, which is partially available for the application developer to store constant values like bitmaps. The slowdown in write speed (FPS) is considered to be negligible as a result of reading the bitmaps from OCF.

6 Display Update Speed: FPS

The actual FPS achieved by CYW20719 depends on several implementation-specific factors. Consequently, there is no blanket statement that can be made regarding expected FPS. The following sections explore the major factors.

6.1 Hardware Capability: MIPI DBI-C Bus Speeds, Buffers, DMA

The MIPI DBI-C hardware has a maximum speed of 10Mbit/second, which means that transferring an 8-bit 128x128 display buffer takes approximately 13 ms. However, this number merely represents a physical limitation of the hardware. *In reality, this number has little practical impact on calculating the expected FPS.* For example, pushing entire display buffers is almost never necessary, leading to improved update speeds. On the other hand, we are simultaneously slowed by factors like vector graphics, which require computation in between page writes to the display. Therefore, a true representation of the FPS needs to consider these factors.

Aside from the raw speed of the bus, the use of a DMA paired with ping-pong buffers ensures smooth writes to the display. The DMA frees the CPU from performing the data transfers. Further, if the speed of the writes to the display exceeds what a single buffer can handle, ping-pong buffers will come into play to ensure that one buffer is being computed while the previous is being written out to the display.

Note: The DMA is strictly utilized by low-level firmware. A developer using the WICED SDK does not have control over the DMA.

6.2 Idle Loop: Dependence on Other Applications and BT Activity

Because most implementations rely on the use of an idle loop, the FPS depends partly on what is happening in other parts of the system. For example, if the system is handling an influx of Bluetooth activity, the idle loop is the one that will go unexecuted until the system can allocate the slots for it.

6.3 Controller-Specific Functionalities

The apparent FPS of a display implementation can be greatly affected by controller-specific features. For instance, if you are implementing some type of icon menu with a scrolling effect, the use of a controller with a scroll buffer will allow you to send a single command to scroll the screen without having to rewrite the entire display buffer with a slight shift. The improvement in apparent FPS gained from the use of a scroll buffer in this instance would be several orders of magnitude (in one test, the use of a scroll command turned tens of milliseconds into tens of microseconds).

6.4 Update Area Optimizations

The display experiences the slowest update speed when an entire screen buffer must be written with each change to the screen. However, this is almost never necessary for the use cases of embedded displays. For instance, for an unread mail alert to appear in the upper corner of an icon, it is only necessary to update the few pages of the screen that cover the small alert. The update speed of the display increases proportionally as the area of the update decreases: if the unread mail alert in this example takes up one-tenth of the height of the display, you can expect this update to occur approximately ten times faster than that of a whole screen update.

Partial display updates are implemented in the demo app in the SDK: it is carried out by setting the cursor at a specific page number where we want the update to begin, instead of going to the first page every time. Similarly, we also stop the picture loop early if we have updated the required area before reaching the last page of the display.

6.5 Vector Graphics

The use of vector graphics as discussed above in section “RAM Usage,” will have a negative impact on update speed. The effect is minimal and creates disproportionally beneficial savings in RAM. *The reduced update speed is a result of the additional computation required to generate a vector graphic.* While the update speed could be improved by strictly using bitmaps, it would be detrimental to RAM usage.

Number of Elements: When using vector graphics, the number of separate elements drawn must be taken into account. While the update speed of a bitmap is agnostic to what the bitmap contains (only depends on size and color depth), the

speed of a vector graphics display depends entirely on the number of elements drawn to the screen because every element requires additional computation to produce.

6.6 Color Depth and Display Size

The data transfer rate will depend on both the number of pixels that must be written, as well as the color depth per pixel. As a result, a larger display will have a slower update speed. Similarly, the color depth has a large impact on the total data that must be transferred. For example, an 8-bit color depth will require one byte of information per pixel; while a 16-bit color depth will double that size and approximately double the time it takes to write to the controller.

Document Revision History

Document Title: CYW20719 MIPI Display Interface Guide

Document Number:002-23078

Revision	ECN	Issue Date	Description of Change
**	6071778	02/16/2018	Initial release

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© Cypress Semiconductor Corporation, 2018. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. No computing device can be absolutely secure. Therefore, despite security measures implemented in Cypress hardware or software products, Cypress does not assume any liability arising out of any security breach, such as unauthorized access to or use of a Cypress product. In addition, the products described in these materials may contain design defects or errors known as errata which may cause the product to deviate from published specifications. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.