AN214879

WICED Smart™ Hardware Interfaces

Associated Part Family: CYW20732

Cypress Wireless Internet Connectivity for Embedded Devices (WICED; pronounced "wicked") Smart™ SDKs are shipped with the BCM20732TAG-Q32-02 evaluation board. This document provides design notes and samples for using the hardware interfaces on the CYW20732 and the SDK.

Contents

1 Introduction

The CYW20732 uses several defined hardware blocks to provide interfaces to sensors, microcontrollers and other peripherals. This document describes the capabilities, configuration, and sample code to use for each of these interfaces.

1.1 Cypress Part Numbering Scheme

Cypress is converting the acquired IoT part numbers from Broadcom to the Cypress part numbering scheme. Due to this conversion, there is no change in form, fit, or function as a result of offering the device with Cypress part number marking. The table provides Cypress ordering part number that matches an existing IoT part number.

Table 1. Mapping Table for Part Number between Broadcom and Cypress

Broadcom Part Number	Cypress Part Number
BCM20732	CYW20732

1.2 Acronyms and Abbreviations

In most cases, acronyms and abbreviations are defined upon first use. For a more complete list of acronyms and other terms used in Cypress documents, go to: http://www.cypress.com/glossary.

2 IoT Resources

Cypress provides a wealth of data at http://www.cypress.com/internet-things-iot to help you to select the right IoT device for your design, and quickly and effectively integrate the device into your design. Cypress provides customer access to a wide range of information, including technical documentation, schematic diagrams, product bill of materials, PCB layout information, and software updates. Customers can acquire technical documentation and software from the Cypress Support Community website (https://community.cypress.com/)

3 Serial Peripheral Interfaces

CYW20732 uses two physical serial peripheral interface (SPI) blocks:

- SPI1 (aka SPIFFY1)
- SPI2 (aka SPIFFY2)

The CYW20732 SPI bus uses four logic signals:

- Clock (SCLK)
- Master out (MOSI)
- Master in (MISO)
- Chip select (CS)

Typically, SPIFFY1 is used internally by the CYW20732 logic to access NVRAM. The SPIFFY2 hardware block is available for use by the application to communicate with peripheral devices such as sensors, storage, and external microcontrollers.

When SPIFFY2 is used in master mode, multiple slaves can be connected to the same bus, with an application-controlled slave/chip select signal assigned to each slave device. A unified Application Programming Interface (API) enables configuration and control of both SPI interfaces. This is included in the WICED Smart SDK. The API only offers services for clock control, mode control, and data transfer operations (half/full duplex). The application generates the slave/chip select signal(s) using the GPIO driver API available in the SDK.

3.1 Capabilities

SPIFFY1 and SPIFFY2 interfaces are fundamentally the same, with the exception of the differences described in the sections below.

1. The interfaces provide support for all four SPI clock modes (0, 1, 2, and 3).
2. The maximum transaction size is 16 bytes (transmit only/receive only/exchange).
3. Both interfaces allow the application to control the Chip Select (CS) line using the GPIO driver API to optimize transactions that require more than 16 bytes to be exchanged in a transaction.

3.2 SPIFFY1

The SPIFFY1 interface supports only the master mode of operation which is dedicated to accessing non-volatile (NV) storage items such as configuration, patches, application code, and application data. When using I^2C for NV storage (with an I^2C EEPROM), SPIFFY1 is not available for application use.

For NV access, the CYW20732 uses P32 as MISO and P33 as the CS line to the serial flash (SF). These two pins are not available for application use. The application may also use SPIFFY1 to connect to other peripheral devices using the same MOSI, MISO, and SCLK lines as those used for the NV device, by driving a different CS line using the functionality of the GPIO API.

The maximum SCLK speed supported by SPIFFY1 is 12 MHz in all clock modes. This assumes that the IO supply rail can be guaranteed to be ≤ 2.4V. When the IO supply is ≤ 2.4V, then the SPIFFY1 SCLK is limited to 6 MHz operation. Attempting to run the clock at higher speeds will cause undefined behavior.

3.3 SPIFFY2

The SPIFFY2 interface supports either master or slave modes of operation. The application has full control of this interface. Several options are available for bringing out the SPI interface to the physical pins on the chip (see descriptions below). In master mode, the application controls when the CS pin is asserted and de-asserted using the GPIO driver API. In slave mode, the application configures the physical pin, with the hardware in full control of the pin.

The maximum SCLK speed supported by SPIFFY2 is 6 MHz at all IO supply voltages.

3.3.1 SPIFFY2 Master

To configure SPIFFY2 in master mode, the following options (shown in Table 2) are available for configuring the various SPI signals. The slave/chip-select line is under application control using the GPIO driver API available in WICED-Smart-SDK/include/Drivers/gpiodriver.h.

Note: All options may not be available in the package. See the module-specific documentation for more information. Combinations shown in red text are not generally available in the BCM20732 TAG. Additional options may be unavailable if they are being used for the peripheral UART serial interface

Table 2. SPIFFY2 Master GPIO Options

SCLK	MOSI	MISO
P03	P00	P01
P03	P00	P05
P03	P02	P01
P03	P02	P05
P03	P04	P01
P03	P04	P05
P03	P27	P01
P03	P27	P05
P03	P38	P01
P03	P38	P05
P07	P00	P01
P07	P00	P05
P07	P02	P01
P07	P02	P05
P07	P04	P01
P07	P04	P05
P07	P27	P01
P07	P27	P05
P07	P38	P01
P07	P38	P05
P24	P00	P25
P24	P02	P25
P24	P04	P25
P24	P27	P25
P24	P38	P25
P36	P00	P25
P36	P02	P25
P36	P04	P25
P36	P27	P25
P36	P38	P25

3.3.2 SPIFFY2 Master Sample Code

The following code snippet shows how to initialize SPIFFY2 in the master role, and how to write and then read a byte from a SPI slave.

```
#include "spiffydriver.h"

#include "gpiodriver.h"

#include "bleappconfig.h"

// use P15 for CS

#define CS_PORT 0

#define CS_PIN 15

// Use 1M speed

#define SPEED        1000000

// CS is active low

#define CS_ASSERT 0

#define CS_DEASSERT 1

// Initializes SPIFFY2 as a SPI master using P24 for SCLK,

// P4 for MOSI, P25 for MISO and P15 for CS.

void spiffy2_master_initialize(void)

{

// Use SPIFFY2 interface as master

spi2PortConfig.masterOrSlave = MASTER2_CONFIG;

// pull for MISO for master, MOSI/CLOCK/CS if slave mode

spi2PortConfig.pinPullConfig = INPUT_PIN_PULL_UP;


// Use P24 for CLK, P4 for MOSI and P25 for MISO

spi2PortConfig.spiGpioConfig = MASTER2_P24_CLK_P04_MOSI_P25_MISO;

// Initialize SPIFFY2 instance

spiffyd_init(SPIFFYD_2);

// Define this to the Port/Pin you want to use for CS.

// Port = P#/16 and PIN = P# % 16

// Configure the CS pin and deassert it initially.

// If enabling output, you only need to configure once. Use gpio_setPinOutput to toggle value
being o/p

gpio_configurePin(CS_PORT, CS_PIN,

GPIO_OUTPUT_ENABLE | GPIO_INPUT_DISABLE, CS_DEASSERT);
```

```
// Configure the SPIFFY2 HW block
spiffyd_configure(SPIFFYD_2, SPEED, SPI_MSB_FIRST,
SPI_SS_ACTIVE_LOW, SPI_MODE_3);
}
// Sends one byte and receives one byte from the SPI slave.
//  byteToSend - The byte to send to the slave.
//  Returns the byte received from the slave.
//
UINT8 spiffy2_master_send_receive_byte(UINT8 byteToSend)
{
BYTE byteReceived;
// SPIFFY2 is now ready to be used.
// Assert chipselect by driving O/P on the GPIO
gpio_setPinOutput(CS_PORT, CS_PIN, CS_ASSERT);     // Assert chipselect

// Tx one byte of data
spiffyd_txData(SPIFFYD_2, 1 ,&byteToSend);         // Send a byte
// Rx one byte of data
spiffyd_rxData(SPIFFYD_2, 1, &byteReceived);       // Read 1 byte
// Deassert chipselect
gpio_setPinOutput(CS_PORT, CS_PIN, CS_DEASSERT);   // Deassert chipselect
return byteReceived;
}
```

3.4 SPIFFY2 Slave

To configure SPIFFY2 for slave mode operation, the following options (shown in Table 3) are available for configuring the different SPI signals.

Note: All options may not be available within the package. See the module-specific documentation for details. Combinations shown in red text are not generally available in the BCM20732 TAG. Additional options may be unavailable if they are being used for the peripheral UART serial interface.

Table 3. SPIFFY2 Slave GPIO Options

CS	SCLK	MOSI	MISO
P02	P03	P00	P01
P02	P03	P00	P05
P02	P03	P00	P25
P02	P03	P04	P01
P02	P03	P04	P05
P02	P03	P04	P25
P02	P07	P00	P01
P02	P07	P00	P05
P02	P07	P00	P25
P02	P07	P04	P01
P02	P07	P04	P05
P02	P07	P04	P25
P06	P03	P00	P01
P06	P03	P00	P05
P06	P03	P00	P25
P06	P03	P04	P01
P06	P03	P04	P05
P06	P03	P04	P25
P06	P07	P00	P01
P06	P07	P00	P05
P06	P07	P00	P25
P06	P07	P04	P01
P06	P07	P04	P05
P06	P07	P04	P25
P26	P24	P27	P01
P26	P24	P27	P05
P26	P24	P27	P25
P26	P24	P33	P01
P26	P24	P33	P05
P26	P24	P33	P25
P26	P24	P38	P01
P26	P24	P38	P05
P26	P24	P38	P25
P26	P36	P27	P01
P26	P36	P27	P05
P26	P36	P27	P25
P26	P36	P33	P01
P26	P36	P33	P05
P26	P36	P33	P25
P26	P36	P38	P01
P26	P36	P38	P05
P26	P36	P38	P25

Table 3. SPIFFY2 Slave GPIO Options (Continued.)

CS	SCLK	MOSI	MISO
P32	P24	P27	P01
P32	P24	P27	P05
P32	P24	P27	P25
P32	P24	P33	P01
P32	P24	P33	P05
P32	P24	P33	P25
P32	P24	P38	P01
P32	P24	P38	P05
P32	P24	P38	P25
P32	P36	P27	P01
P32	P36	P27	P05
P32	P36	P27	P25
P32	P36	P33	P01
P32	P36	P33	P05
P32	P36	P33	P25
P32	P36	P38	P01
P32	P36	P38	P05
P32	P36	P38	P25
P39	P24	P27	P01
P39	P24	P27	P05
P39	P24	P27	P25
P39	P24	P33	P01
P39	P24	P33	P05
P39	P24	P33	P25
P39	P24	P38	P01
P39	P24	P38	P05
P39	P24	P38	P25
P39	P36	P27	P01
P39	P36	P27	P05
P39	P36	P27	P25
P39	P36	P33	P01
P39	P36	P33	P05
P39	P36	P33	P25
P39	P36	P38	P01
P39	P36	P38	P05
P39	P36	P38	P25

3.4.1 SPIFFY2 Slave Sample Code

The following code snippet shows how to initialize SPIFFY2 in the slave role, and how to read and then write a byte to a SPI master.

```
#include "spiffydriver.h"

#include "gpiodriver.h"

#include "bleappconfig.h"

// Initializes SPIFFY2 as a SPI slave using P03 for SCLK,

// P00 for MOSI, P01 for MISO and P02 for CS.

void spiffy2_slave_initialize(void)

{

// To use SPIFFY2 as slave

spi2PortConfig.masterOrSlave = SLAVE2_CONFIG;

// To pull for MISO for slave, MOSI/CLOCK/CS if we are slave mode

spi2PortConfig.pinPullConfig = INPUT_PIN_PULL_DOWN;

// To use P3 for CLK, P0 for MOSI and P1 for MISO in SLAVE mode

spi2PortConfig.spiGpioConfig = SLAVE2_P02_CS_P03_CLK_P00_MOSI_P01_MISO;

// DO NOT CONFIGURE CS_PORT and CS_PIN in SLAVE mode - the HW takes care of this.

// There is no need to configure the speed too – the master selects the speed.

// Initialize SPIFFY2 instance

spiffyd_init(SPIFFYD_2);

// Configure the SPIFFY2 HW block

spiffyd_configure(SPIFFYD_2, SPEED, SPI_MSB_FIRST,

SPI_SS_ACTIVE_LOW, SPI_MODE_3);

}
```

```
// Receives a byte from the SPI master, increments the byte and sends it back

void spiffy2_slave_receive_send_byte(void)

{

BYTE byteReceived;

// Rx one byte of data

spiffyd_rxData(SPIFFYD_2, 1 ,&byteReceived);   // Send a byte


// Send back receivedByte + 1

byteReceived++;


// Tx one byte of data

spiffyd_txData(SPIFFYD_2, 1, &byteReceived);    // Read 1 byte

}
```

3.5 Restrictions

When using either SPIFFY1 or SPIFFY2 in master or slave mode, please note the following restrictions on the use of GPIO configurations for the SPI signals:

1. If a GPIO that supports multiple functions (such as peripheral UART, IR, and PWM) is allocated for a function other than SPI, then that GPIO cannot be used for SPIFFY.
2. If a peripheral UART is also being used by the application, the SPIFFY2 signals and peripheral UART RX signal must be on the lower pad bank (P0 through P7) or on the upper pad bank (P24 through P39). Thus, SPIFFY1 or SPIFFY2, and peripheral UART RX, cannot be on two different pad banks.
3. On some packages, two or more GPIOs may be bonded together. If a bonded GPIO is to be used for SPI, then the other GPIO must be configured with the input and output disabled using the GPIO driver API before configuring SPIFFY. See the documentation that accompanies the BCM20732 module for more information on bonded GPIOs.
4. If the application requires use of the I^2C interface (for NV storage or to interface with other peripherals), SPIFFY1 cannot be used.
5. When the CYW20732 is in sleep or deep sleep, no data can be received over either SPI interface. Use a GPIO configured as an interrupt source to wake the chip and then receive bytes from the peer device. See the GPIO section for sample code that configures a GPIO as a wake/interrupt source.

3.6 SPIFFY Driver and API Documentation

APIs are used to configure, control, and exchange data using the SPIFFY interfaces. A list of the APIs is available in the WICED-Smart-SDK/include/Drivers/spiffydriver.h. Documentation for the SPIFFY driver is available under WICED-Smart-SDK/Doc/API.html.

4 Peripheral UART

The CYW20732 has two UART interfaces. One is the factory or HCI UART that is used for programming and factory testing. The other is the peripheral UART, which is available for application use to output debug messages/tracing or to interface with a peripheral device/microcontroller. The following sections describe the peripheral UART only. The HCI UART is not available for application use.

4.1 Capabilities

The peripheral UART interface supports a standard two-wire serial protocol with or without hardware flow control. The maximum baud rate supported by this interface is 115200 bps (other standard baud rates such as 19200, 38200, 57600 are also supported). The maximum transmit and receive hardware buffer size is 16 bytes. When hardware flow control is not used, the application is responsible for servicing the transmit FIFO before it is empty, and the receive FIFO before it is full. Not servicing the FIFOs may lead to lost bytes when receiving and gaps during transmission.

4.2 Peripheral UART Multiplexing Options

The peripheral UART can be muxed out to a number of physical pins on the module, with some restrictions. The combinations of pins shown in Table 4 and Table 5 on page 10 are the only valid options for the four UART signals (Receive - RXD, Transmit - TXD, Clear to Send - CTS and Ready to Send - RTS). All four signals must be selected from the same group. Hardware flow control is optional, in which case, CTS and RTS signals will not be configured by the driver API.

Note: All options may not be available within the package. See the module-specific documentation for details. Combinations shown in red text are not generally available in the BCM20732 TAG. GPIOs allocated by the application for other functions, such as SPI, IR and Touch Sense, is also unavailable for peripheral UART.

Table 4. Group 1 Peripheral UART Signal Muxing Options

RXD	TXD	CTS	RTS
P2	P0	P3	P1
P4	P5	P7	P6
	P24		P30
	P31		
	P32		

Table 5. Group 2 Peripheral UART Signal Muxing Options

RXD	TXD	CTS	RTS
P25	P0	P35	P1
P33	P5		P6
P34	P24		P30
	P31		
	P32		

4.2.1 Peripheral UART Sample Code

The CYW20732 includes two sets of APIs that are available for the application: the low-level driver API and the high-level profile API.

The following code snippet shows how to initialize the peripheral UART using the low-level driver API available in the WICED SDK.

```
#include "puart.h"
// Use P33 for peripheral uart RX.
#define PUART_RX_PIN       33
// Use P32 for peripheral uart TX.
#define PUART_TX_PIN       32

// Application initialization function.
void application_init(void)
{
    extern puart_UartConfig puart_config;
    // Do all other app initializations.

    // Set the baud rate we want to use. Default is 115200.
    puart_config.baudrate = 115200;

    // Select the uart pins for RXD, TXD and optionally CTS and RTS.
    // If hardware flow control is not required like here, set these
    // pins to 0x00. See Table 1 and Table 2 for valid options.
    puart_selectUartPads(PUART_RX_PIN, PUART_TX_PIN, 0x00, 0x00);

    // Initialize the peripheral uart driver
    puart_init();

    // Since we are not configuring CTS and RTS here, turn off
    // hardware flow control. If HW flow control is used, then
    // puart_flowOff should not be invoked.
    puart_flowOff();

        /* BEGIN - puart interrupt
                   The following lines enable interrupt when one (or more) bytes
                   are received over the peripheral uart interface. This is optional.
                   In the absence of this, the app is expected to poll the peripheral
                   uart to pull out received bytes.
        */

        // clear interrupt
        P_UART_INT_CLEAR(P_UART_ISR_RX_AFF_MASK);

        // set watermark to 1 byte - will interrupt on every byte received.
        P_UART_WATER_MARK_RX_LEVEL (1);

        // enable UART interrupt in the Main Interrupt Controller and RX Almost Full in the
    UART
    // Interrupt Controller
        P_UART_INT_ENABLE |= P_UART_ISR_RX_AFF_MASK;

        // Set callback function to app callback function.
        puart_bleRxCb = application_puart_interrupt_callback;

        // Enable the CPU level interrupt
        puart_enableInterrupt();

        /* END - puart interrupt */
```

```
        // print a string message assuming that the device connected
    // to the peripheral uart can handle this string.
    puart_print("Application initialization complete!");
}
// Sends out a stream of bytes to the peer device on the peripheral uart interface.
//  buffer - The buffer to send to the peer device.
//  length - The number of bytes from buffer to send.
// Returns The number of bytes that were sent.
UINT32 application_send_bytes(UINT8* buffer, UINT32 length)
{
    UINT32 ok = length;

    // Need to send at least 1 byte.
    if(!buffer || !length)
            return 0;
    // Write out all the given bytes synchronously.
    // If the number of bytes is > P_UART_TX_FIFO_SIZE (16)
    // puart_write() will block until there is space in the
    // HW FIFO.
    while(length--)
    {
            puart_write(*buffer++);
    }

    return ok;
}
// Attempts to receive data from the peripheral uart.
//  buffer - The buffer into which to read bytes.
//  length - The number of bytes to read.
// Return The actual number of bytes read.
UINT32 application_receive_bytes(UINT8* buffer, UINT32 length)
{
    UINT32 number_of_received_bytes = 0;

    // Need to receive at least 1 byte.
    if(!buffer || !length)
            return 0;
    // Try to receive length bytes
    while(length--)
    {
            if(!puart_read(buffer++))
            {
                    // If the FIFO is empty, break out, no
                    // more bytes are available.
                    break;
            }
            number_of_received_bytes++;
    }

    // Return the actual number of bytes read.
    return number_of_received_bytes;
}
// Application thread context uart interrupt handler.
// unused - Unused parameter.
void application_puart_interrupt_callback(void* unused)
{
    // There can be at most 16 bytes in the HW FIFO.
    char  readbytes[16];
    UINT8 number_of_bytes_read = 0;
```

```
    // empty the FIFO
    while(puart_rxFifoNotEmpty() && puart_read(&readbytes[number_of_bytes_read]))
    {
            number_of_bytes_read++;
    }
    // readbytes should have number_of_bytes_read bytes of data read from puart. Do something
    with this.
    // clear the interrupt
    P_UART_INT_CLEAR(P_UART_ISR_RX_AFF_MASK);

    // enable UART interrupt in the Main Interrupt Controller and RX Almost Full in the UART
    Interrupt Controller
    P_UART_INT_ENABLE |= P_UART_ISR_RX_AFF_MASK;
}
```

4.3 Restrictions

The following restrictions apply to the use of GPIOs when using the peripheral UART:

1. When both SPIFFY2 and peripheral UART interfaces are used by the application, the SPIFFY2 signals and peripheral UART RX signal must be on the lower pad bank (P0 through P7) or on the upper pad bank (P24 through P39). Thus, SPIFFY2 and the peripheral UART RX cannot be on two different pad banks.
2. If hardware flow control is required, all GPIOs for the peripheral UART must be selected from the same group (see Table 4 and Table 5).
3. If hardware flow control is not required, the application must ensure that the receive FIFO does not overflow (16 bytes maximum) at the configured baud rate by reading out the received bytes before the overflow.
4. When the CYW20732 is in sleep or deep sleep mode, no data can be received over the peripheral UART. Use a GPIO configured as an interrupt source to wake the chip. Once the chip is awake, then bytes can be received from the peer device.

Since GPIOs can be configured for different functions, there can be no contention between GPIOs that are defined for peripheral UART operations or other functions.

5 I²C

The CYW20732 supports the I²C-compatible specification master-only interface that enables communication with peripheral devices over the standard two-wire I²C bus, Serial Clock (SCL) and Serial Data (SDA).

5.1 Capabilities

1. The I²C interface supports three types of transfers - read, write, and combination write then read. In the combined operation, the I²C hardware block generates a repeated START condition between the two parts of the transaction.
2. A maximum transaction length of 16 bytes.
3. Both low-speed and fast-mode slave devices, up to a maximum SCL speed of 4000 kbps. If the slave is capable of clock cycle stretching, the maximum SCL speed is limited to 2400 kbps.
4. The SCL and SDA lines always operate on fixed physical pins on the BCM20732 TAG or module. See the module documentation to determine pin assignments.
5. The I²C interface does not support the multi-master bus mode. Thus, the CYW20732 should be the only master on the bus.
6. The I²C interface supports only 7-bit slave addresses.

Note: The SCL speed may be limited to less than the above speeds by other factors such as transmission time due to the external pull-up.

5.2 I²C Sample Code

The following code snippet shows how to initialize I²C in the master role, and how to write, read and use the combo write-then-read transaction using the driver API available in the WICED SDK.

The description of the I²C API is available in WICED-Smart-SDK/Wiced-Smart/cfa/cfa.h.

```
#include "cfa.h"

// Use slave address 0x1A = (7'b0001101 << 1) | 1'bR|W.
#define I2C_SLAVE_ADDRESS            (0x1A)

// Read operation to the lower level driver is 0.
#define I2C_SLAVE_OPERATION_READ     0

// Write operation to the lower level driver is 1.
#define I2C_SLAVE_OPERATION_WRITE    1

// Reads data from the slave into the given buffer.
//  buffer        - The buffer into which to read. Should be allocated by caller.
//  length        - The number of bytes to read form the slave.
//  slave_address - The address of the slave device. Valid bits are the
//                  upper 7 bits, bit 0 must be 1'b0.
// Returns 1 if successful, else 0.
//
UINT8 i2cm_read_from_slave(BYTE* buffer, UINT32 length, UINT8 slave_address)
{
    // Assume failure.
    UINT8 return_value = 0;

    // Invoke the lower level driver. Non-combo transaction, so set offset parameters to NULL/
    0.
    CFA_BSC_STATUS read_status = cfa_bsc_OpExtended(buffer, length, NULL, 0, slave_address,
                                                    I2C_SLAVE_OPERATION_READ);
    switch(read_status)
    {
    case CFA_BSC_STATUS_INCOMPLETE:
```

```
            // Transaction did not go through. ERROR. Handle this case.
            break;
        case CFA_BSC_STATUS_SUCCESS:
            // The read was successful.
            return_value = 1;
            break;
        case CFA_BSC_STATUS_NO_ACK:
            // No slave device with this address exists on the I2C bus. ERROR. Handle this.
        default:
            break;
        }
        return return_value;
    }
    // Writes data to the slave from the given buffer.
    //  buffer        - The buffer that has the data to be written to the slave.
    //  length        - The number of bytes to write to slave.
    //  slave_address - The address of the slave device. Valid bits are the
    //                  upper 7 bits, bit 0 must be 1'b0.
    // Return 1 if successful, else 0.
    //
    UINT8 i2cm_write_to_slave(BYTE* buffer, UINT32 length, UINT8 slave_address)
    {
        // Assume failure.
        UINT8 return_value = 0;
        // Invoke the lower level driver. Non-combo transaction, so set offset parameters to NULL/
        0.
        CFA_BSC_STATUS read_status = cfa_bsc_OpExtended(buffer, length, NULL, 0, slave_address,
                                                        I2C_SLAVE_OPERATION_WRITE);
        switch(read_status)
        {
        case CFA_BSC_STATUS_INCOMPLETE:
            // Transaction did not go through. ERROR. Handle this case.
            break;
        case CFA_BSC_STATUS_SUCCESS:
            // The read was successful.
            return_value = 1;
            break;
        case CFA_BSC_STATUS_NO_ACK:
            // No slave device with this address exists on the I2C bus. ERROR. Handle this.
        default:
            break;
        }
        return return_value;
    }
    // Combo write-then-read transaction. Writes some bytes to slave,
    // then issues a repeated START followed by a read transaction.
    //  buffer_to_write - The buffer from which to write to the slave. If the number of
    //                bytes to read exceeds 16, the driver will update this value by the number
    //                of bytes read and start another new transaction until length_to_read number
    //                of bytes have been read. This API is available for devices that behave like
    //                EEPROMs, so the buffer has to point to an array of bytes with the 'offset'
    //                address in big endian format. See any I2C EEPROM specification for details.
    //  length_of_write - The number of bytes that buffer_to_write points to. This is
    //                 also the number of bytes in the first WRITE transaction before the repeated
    //                 start. Note that length_of_write + length_to_read has to be <= 16,
    //                 typically is 2 for EEPROMs because they use 16 bit offsets.
    //  buffer_to_read_into - The buffer into which the data is to be read during the READ
    //                        part of the combo transaction (after the repeated START condition).
    //  length_to_read      - The number of bytes to read during the READ portion of the combo
    //                  transaction. Note that length_of_write + length_to_read has to be <= 16,
```

```
//                          typically is 2 for EEPROMs because they use 16 bit offsets.
//  slave_address       - The address of the slave device. Valid bits are the
//                          upper 7 bits, bit 0 must be 1'b0.
// Returns 1 if successful, else 0.
//
UINT8 i2cm_write_then_read_from_slave(BYTE* buffer_to_write, UINT32 length_of_write,
                                      BYTE* buffer_to_read_into, UINT32 length_to_read,
                                      UINT8 slave_address)
{
    // Assume failure.
    UINT8 return_value = 0;

    // Invoke the lower level driver. This is a combo transaction.
    CFA_BSC_STATUS read_status = cfa_bsc_OpExtended(buffer_to_read_into, length_to_read,
    buffer_to_write,
                                                    length_of_write, slave_address,
                                                    I2C_SLAVE_OPERATION_READ);

    switch(read_status)
    {
    case CFA_BSC_STATUS_INCOMPLETE:
            // Transaction did not go through. ERROR. Handle this case.
            break;
    case CFA_BSC_STATUS_SUCCESS:
            // The read was successful.
            return_value = 1;
        break;
    case CFA_BSC_STATUS_NO_ACK:
            // No slave device with this address exists on the I2C bus. ERROR. Handle this.
    default:
            break;
    }

    return return_value;
}
```

5.3 Restrictions

When using the I^{2}C-compatible master, the following restrictions apply to use of the NV storage device:

1. If the SPI serial flash is used for NV storage, then the I^{2}C master device will be unavailable for application use.
2. If the I^{2}C EEPROM is used as the NV storage device, then the SPIFFY1 SPI interface will be unavailable (see the Serial Peripheral Interface section in the document).
3. If the I^{2}C EEPROM is used, then the 7-bit control word/slave address for the EEPROM device must be 0x50. Thus, the slave address generated for read/write for the EEPROM will always be 0xA1/0xA0 on the I^{2}C bus.

6 GPIO

The CYW20732 supports up to 40 GPIOs. The majority of these GPIOs have multiple functional modes (such as SPI2, peripheral UART, Keyscan input/output, and ADC input). When not used in any special function modes, the GPIOs are initialized with input enabled by default at boot up. The 40 GPIOs (P0 through P39) are arranged in three ports as follows:

- P0 through P15 on port0
- P16 through P31 on port1
- P32 through P39 on port2

6.1 Capabilities

1. All GPIOs can be input and output disabled (high-Z), input enabled, or output enabled.
2. When the GPIO is input-enabled, an internal pull-up or an internal pull-down can also be optionally configured.
3. A GPIO that is output-enabled and driven high or low will remain driven in sleep and deep sleep.
4. All GPIOs are capable of being configured for edge (rising/falling/both) or level interrupts. An application level interrupt handler can be configured to handle the interrupt in the application thread context.
5. Interrupts can additionally be configured to wake the system from sleep and deep sleep.
6. All GPIOs can source or sink up to 2 mA. P26, P27, P28 and P29 can sink up to 16 mA.

Note: All GPIOs may not be available in the package. Some of the GPIOs may be bonded together inside the package when not all 40 GPIOs are brought out. See the documentation accompanying the module you are using for more information. On BCM20732TAG boards, the following pairs of GPIOs are bonded together: P8 and P33, P11 and P27, P12 and P26, P13 andP28, P14 and P38. Only one of the bonded GPIOs from each pair may be used. The associated pin must be input and output disabled. Additionally, the following GPIOs are not brought out from the package: P5, P6, P7, P9, P10, P16, P17, P18, P19, P20, P21, P22, P23, P29, P30, P31, P34, P35, P36, P37 and P39.

6.2 GPIO Sample Code

The following code snippet shows how to use the GPIO driver API available in the WICED-Smart-SDK/include/Drivers/gpiodriver.h:

```
#include "gpiodriver.h"

// Configure Green LED on P14.
// Green LED is on Port 0. Port = 14 / 16.
#define APPLICATION_LED_GREEN_PORT    0

// GREEN LED is on Pin 14. Pin = 14 % 16.
#define APPLICATION_LED_GREEN_PIN     14

// GREEN LED is active low.
#define APPLICATION_LED_GREEN_ON      GPIO_PIN_OUTPUT_LOW

// GREEN LED is active low.
#define APPLICATION_LED_GREEN_OFF     GPIO_PIN_OUTPUT_HIGH

// Sets try to read the level on P24.
#define APPLICATION_GPIO_LEVEL_PORT   1
#define APPLICATION_GPIO_LEVEL_PIN    8

// Configure interrupt from external processor on P32.
// Interrupt from an external processor is on Port 2. Port = 32 / 16.
#define APPLICATION_EXTERNAL_INTERRUPT_PORT     2

// Interrupt from an external processor is on Pin 0. Pin = 32 % 16.
#define APPLICATION_EXTERNAL_INTERRUPT_PIN      0
```

```
/// Initialize the application.
void application_init(void)
{
    // Initialize the GPIO driver.
    gpioDriver_init();

    // Configure some app specific ports and pins.
    application_configure_some_pins();
}

/// Initializes the GPIOs required by this application.
void application_configure_some_pins(void)
{
    // There are three ports (port0, port1 and port2), each bit corresponds to
    // a GPIO in that port for which we want to register an interrupt handler.
    UINT16 interrupt_handler_mask[3] = {0, 0, 0};

    // Configure the Green LED. LED is active low. So init with off.
    gpio_configurePin(APPLICATION_LED_GREEN_PORT, APPLICATION_LED_GREEN_PIN,
        GPIO_OUTPUT_ENABLE, APPLICATION_LED_GREEN_OFF);

    // Another way to do the same is:
    // gpio_configurePinWithSingleBytePortPinNum((APPLICATION_LED_GREEN_PORT << 5) |
    //        APPLICATION_LED_GREEN_PIN, GPIO_OUTPUT_ENABLE, GPIO_PIN_OUTPUT_HIGH);

    // Configuring a GPIO for an interrupt is a two step process.
    // Step 1: Register an application level interrupt handler.
    // A single interrupt handler can handle interrupts from one or more GPIOs.
    // Once registered, there is no way to unregister. Interrupt handlers are expected
    // to be startup activities. To stop receiving interrupts, disable interrupts on that GPIO
    by
    // reconfiguring it with gpio_configurePin.
    // Since we will only be handling an interrupt from one external
    // microprocessor, we will set only one bit in the handler registration mask.
    interrupt_handler_mask[APPLICATION_EXTERNAL_INTERRUPT_PORT] |= (1 <<
        APPLICATION_EXTERNAL_INTERRUPT_PIN);

    // Now register the interrupt handler.
    gpio_registerForInterrupt(interrupt_handler_mask, application_gpio_interrupt_handler);

    // Step 2: Configure the GPIO and enable the interrupt. Output parameter is a don't care.
    Let
    // us also insert an internal pull-down so that we know it is in a known good state unless
    // the external processor is driving it.
    gpio_configurePin(APPLICATION_EXTERNAL_INTERRUPT_PORT, APPLICATION_EXTERNAL_INTERRUPT_PIN,
        GPIO_EN_INT_RISING_EDGE | GPIO_PULL_DOWN, GPIO_PIN_OUTPUT_LOW);
}

// Turns on the GREEN LED.
void application_turn_on_green_led(void)
{
    // When output has been enabled once, you only need to set output to drive the other
    way. There
    // is no need to configure the GPIO again.
    gpio_setPinOutput(APPLICATION_LED_GREEN_PORT, APPLICATION_LED_GREEN_PIN,
    APPLICATION_LED_GREEN_ON);
}

// Turns off the GREEN LED.
```

```
void application_turn_off_green_led(void)
{
    // When output has been enabled once, you only need to set output to drive the other way.
    There
    // is no need to configure the GPIO again.
    gpio_setPinOutput(APPLICATION_LED_GREEN_PORT, APPLICATION_LED_GREEN_PIN,
    APPLICATION_LED_GREEN_OFF);
}

/// The application level GPIO interrupt handler.
void application_gpio_interrupt_handler(void* parameter)
{
    // External microprocessor interrupted the app.
    // Do something now.
    // Lets try to read from P24. The value returned will be either a 0 or a 1.
    BYTE gpio_level = gpio_getPinInput(APPLICATION_GPIO_LEVEL_PORT,
    APPLICATION_GPIO_LEVEL_PIN);

    // Pending interrupt on this GPIO will automatically be cleared by the driver.
```

7 ADC

The CYW20732 has an onboard 16-bit ADC that uses multiple input channels with 10 effective number of bits and better than +/- 2% of the input. The conversion time is typically 10 µs with a maximum sampling rate of ~187 KHz. The application may select from three different input voltage ranges: 0-1.2V, 0-2.4V and 0-3.6V. However, the input should only swing at most rail-to-rail.

7.1 ADC Input Options

Table 6 shows the ADC input channel GPIO options.

Table 6. ADC Input Channel GPIO Mapping

GPIO	ADC Channel
	VSS (GND) = 0
P38	1
P37	2
P36	3
P35	4
P34	5
P33	6
P32	7
P31	8
P30	9
P29	10
P28	11
P23	12
P22	13
P21	14
	Internal Bandgap reference = 15
P19	16
P18	17
P17	18
P16	19
P15	20
P14	21
P13	22
P12	23
P11	24
P10	25
P9	26
P8	27
P1	28
P0	29
	Vcore = 30
	Vio = 31

Note: On packages where GPIOs may be bonded, only one of the bonded GPIOs from each pair may be used as an ADC input channel (see GPIO section for information on bonded GPIOs). The other GPIO must be input and output disabled. GPIOs noted in red text in Table 6 are not available on BCM20732TAG boards. ADC channels 0, 15, 30 and 31 are not mapped to any GPIO. These are internal signals.

7.2 ADC Sample Code

```
#include "adc.h"

// ADC input is on P32.
#define APPLICATION_ADC_INPUT_CHANNEL   ADC_INPUT_P32

// Application initialization function.
void appliction_init(void)
{
    // Initialize and configure the ADC driver.
    adc_config();
}

/// Reads and returns the voltage seen on a known ADC input channel.
UINT32 application_read_adc_voltage(void)
{
    // ADC driver provides an API read and convert to voltage.
    return adc_readVoltage(APPLICATION_ADC_INPUT_CHANNEL);
}

/// Reads the voltage from any ADC channel when only the P# of the GPIO is known.
UINT32 application_read_adc_voltage_from_gpio(UINT8 gpio_number)
{
    // First converts the P# to an ADC channel and then reads the voltage.
    return adc_readVoltage(adc_convertGPIOtoADCInput(gpio_number));
}
```

8 References

The references in this section may be used with this document.

Note: Cypress provides customer access to technical documentation and software through the WICED website . Additional restricted material may be provided through the https://community.cypress.com.

Document (or Item) Name	Number	Source
Broadcom Items		
[1] WICED™ Quick Start Guide	WICED-Smart-QSG10x-R	wiced-smart

Document History Page

Document Title: AN214879 - WICED Smart™ Hardware Interfaces Document Number: 002-14879				
Rev.	**ECN No.**	**Orig. of Change**	**Submission Date**	**Description of Change**
**	-	-	12/04/2013	MMP920732HW-AN100-R Initial release
*A	-	-	01/15/2014	MMP920732HW-AN101-R Converted to Broadcom template
*B	5446302	UTSV	11/22/2016	Updated in Cypress template
*C	5834576	BENV	07/27/2017	Updated logo and copyright

Worldwide Sales and Design Support

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturers' 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

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Video | Blogs | Training | Components

Technical Support

cypress.com/support

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