Serial Controlled Motor Driver (SCMD) Datasheet and Manual

This motor driver is intended to control small DC motors from microcontrollers and other small computers. It can be used alone or with other masters on a shared bus, or attached slaves. By controlling only the master, slaves receive data simplifying multi-motor designs.

The User Port is selectable to many modes by jumper, with additional settings available by register control.

The SCMD is designed to emulate a typical register-based device in operation. I2C, SPI, or UART can be used to directly operate on the registers, or, either the Arduino library or built-in UART parser can be used to deliver higher-level motor control.

Features

- 1.5 A peak drive per channel, 1.2 A steady state.
- Operates from 3 to 11 volts with 12v absolute max.
- 3.3v default VCC and logic.
- Max VCC in of 5.5v.
- 127 levels of DC drive strength.
- Controllable by I2C, SPI, or TTL UART signals.
- Direction inversion on a per motor basis.
- Global drive enable.
- Expansion port utilizing I2C, allows 16 additional drivers.
- Exposed TO-220 heat sink shape.
- Several I2C addresses, default UART bauds available.
- Bridgeable outputs.
- Optional fail-safe and diagnostics available.
- Configurable expansion bus bit rate to 50, 100, or 400 kHz.
- Configurable expansion bus update rate from 1ms to 255ms, or by command only.

Index

Electrical Ratings

The data in Table 1 reflects a combination of the components on the SCMD.

Symbol	Parameter	Conditions	Data	Units
Vin	Motor supply		-0.3 to 12	V
Vcc	Logic supply	REG disabled, ext power to VCC pin	-0.3 to 6	V
Vio_3.3	GPIO voltage	REG enabled, VCC = 3.3V	-0.3 to 3.8	V
Vio_ext	GPIO voltage	REG disabled, ext VCC	-0.3 to VCC+0.5	V
IVCC	Max Vcc output current	REG enabled, VCC pin output, VIN = 11V	50	mA
Im	Max continuous current per motor		1.2	A
Im_peak	Peak current per motor	1.2 A before current limiting	1.5	A
Tj_PSoC	PSoC junction		100	Deg C
Tj_DRV8835	DRV8835 Junction Temperature		150	Deg C

Table 1

Pin Functions

The 0.1” PTH on the board are described here.

				Function / Connection		
Group	Name	Direction	Description	UART	I2C	SPI
User Port	RX,SCL,MOSI	I	Multi-function Serial	Data In (RX)	SCL	MOSI
	TX,SDA,MISO	IO	Multi-function Serial	Data Out (TX)	SDA	MISO
	GND	-	Ground	Ground	Ground	Ground
	NC,SCL	I	SPI clock	NC	NC	SCL
	NC,CS	I	SPI chip select	NC	NC	CS
Expansion Port	GND	-	Ground	Slave bus Ground		
	SDA	IO	I2C Data line	Slave bus SCL		
	SCL	I	I2C Clock line	Slave bus SDA		
	In	I	Config In. Slave aquire-address/enable	Connects to upstream slave*		
	Out	O	Config Out. Enable next slave	Connects to downstream slave*		
Motor Port	A1	O	Winding of first addressable location	Motor A winding		
	A2	O	Winding of first addressable location	Motor A winding		
	B1	O	Winding of second addressable location	Motor B winding		
	B2	O	Winding of second addressable location	Motor B winding		
Power	GND	I	Main system ground (two pads)	Supply ground		
	MAX 11V	I	Motor driver raw voltage, regulator in (two pads)	Supply power		
	VCC	IO	Regulator output or user supplied VCC	NC		

Table 2: Pin functions

Concepts

Communication Perspective

A single driver board is capable of being expanded by adding one or more slaves. Illustration 1 shows a master connected to a slave, with the user controlling by microcontroller or computer.

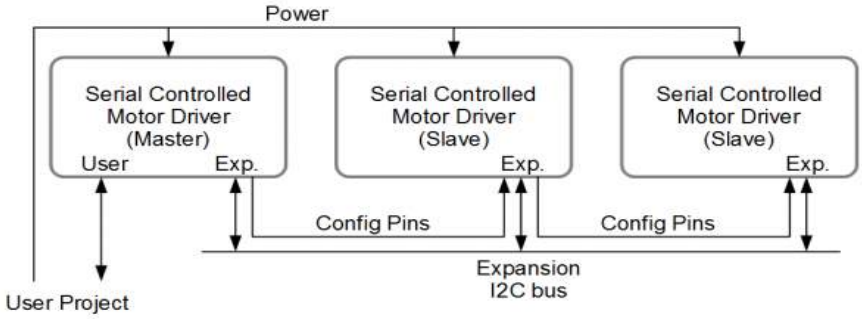


Illustration 1: Basic topology

To avoid confusion, a few terms are defined here.

User Port – Connects the user's project to the master SCMD and is unused on slaves.

Expansion Port – Connects a single master and up to 16 slaves, and should never be connected to the user's project.

Master – The SCMD which is connected to the user's project. Also, it is the I2C master on the expansion port

Slave – A SCMD which is only connected to a master SCMD by way of the expansion port.

Note: The config in and out pins of the expansion port may sometimes be connected to the user's project for error checking and mitigation purposes.

Motor Polarity – The driver and this document often omit motor polarity. This is because polarity to spin direction is not standardized, and it is assumed the user will attach them backwards 50% of the time anyway. Each motor channel is independently configurable for what is thought of as 'forward' spin by command. It is assumed that the user will first attach the motors, then decide which channels need to be inverted and issue inversion settings at boot time.

Numbering – Each SCMD has two channels, A and B. When they are bridged the B channel becomes a duplicate A, and the drive strength is doubled.

When somethings refers to a 'motor number' the following scheme is used. The motor attached to the master at position 'A' will always be motor 0, and the 'B' position, motor 1. The first slave device attached will have motors 2, and 3, at positions 'A' and 'B' respectively. Slave 2 will have motors 4, and 5, and so on.

When a SCMD is designated as bridged mode, it loses whatever motor is attached to the ‘B’ position, and any information sent to control the ‘A’ position will control both outputs synchronously, such as inversion or drive strength.

This is not to be confused with ‘driver number’, which indicates which SCMD in the chain is being referenced. Illustration 2 shows how drivers and motors are referenced in a chain.

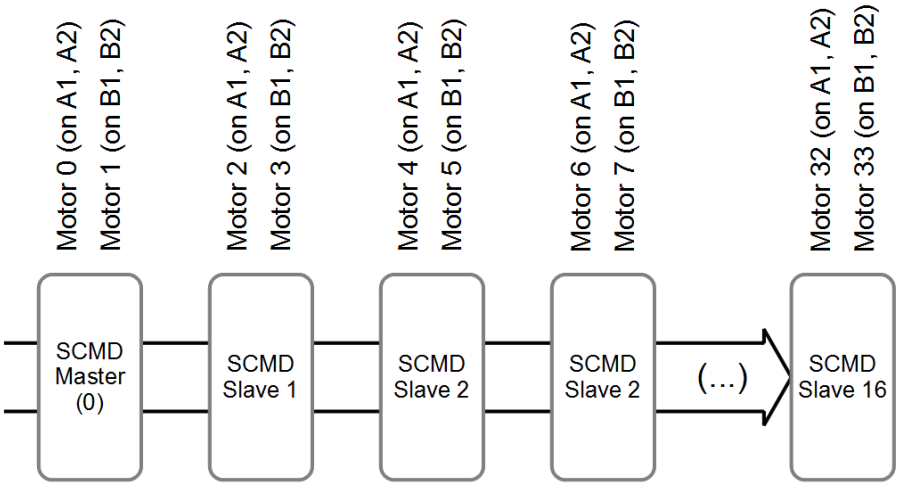
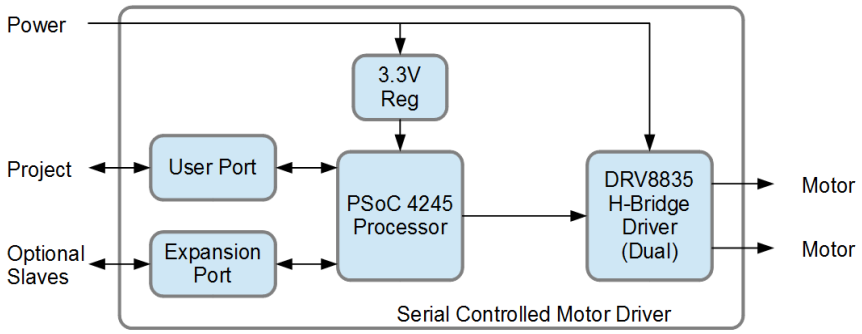


Illustration 2: Motor and driver numbering scheme

Operational Description

General

The main elements of the motor driver are a microcontroller and dual H-bridge driver, with a voltage regulator for the logic end of things.



The SCMD has been designed to emulate a typical I2C or SPI peripheral. Inside, there is a set of memory locations called registers. By writing into some registers, the SCMD takes action. Other registers have data that can be used, but writing does not cause action within the SCMD. The SCMD idles in a remote update loop and waiting for changes to registers, then performs actions based on new data if applicable.

During boot, the SCMD captures the contents of the config jumpers into a register and uses that value to determine which serial configuration to choose. They are set by entering a number from 0 to 15 in 4 bit binary, with a solder jumper indicating a '1'.

The SCMDs are equipped with an expansion I2C bus, where they can be ganged to form a group of motors that can all be controlled by talking to the master only.

For I2C buses, pull-up resistors are provided and default disabled. The slave bus requires a single set enabled for any number of slaves. If I2C is used, the user pull-up jumper must be closed, or pull-up resistors supplied externally.

The SCMD has been released with an Arduino library that performs high level functions by writing the correct registers, and can ease development. The library also has functions to read or write the registers directly.

Status LED

There is a single LED on the motor driver that blinks to indicate status, and blips to indicate serial port activity.

A normally operating blinks once with much space inbetween while an error state will be displayed as a fast group of flashing.

Reception of data will cause a slight blip in the LED. When a steady stream of data is being received, the LED will display dim with the other status message overlaid. The fail safe watchdog is also tickled at this time, so the LED can be used as a visual indication of a satisfied watchdog.

The Status Register

The main register, the status register, contains two bits that can be read to determine when it is OK to send more commands. The LSB indicates a zero while the firmware is booting and a 1 when enumeration is complete. The 2nds LSB indicates a 1 when the SCMD is busy with another task.

The enumeration complete bit should be checked before beginning to configure the SCMD and write motor data.

The busy bit is valid 30us after a triggering register is written, and should be checked to insure the previously written command is not overwritten before it has a chance to execute.

Register access safeguards

Certain registers are designated "User unlockable". These registers contain configuration data for the current application. To prevent accidental writes a user lock register exists. When the user lock contains the user key byte, the user lock registers can be written to. The default is unlocked.

Registers that are designated read-only are actually also writable, to no documented end. A global lock exists to allow writes to these addresses. Note that the global lock is protected by the user lock, so both are needed to allow write access.

Slave operation

Multiple SCMDs can be connected to create a group of controllable motors. In this case, certain registers within the master cause changes to the slaves.

Writing to the drive registers in the master do not instantly update the drive levels. Instead, the master saves the new drive levels and sends them to the slaves on a periodic loop (configurable rate or by manual request). There are 34 registers that contain drive levels for each of the possible motor numbers. Writing to any of these registers in the master causes that motor in the slave chain to match

the drive level after the next transmission.

Writing to the remote configuration or system configuration registers causes immediate transfer when necessary. The register contents will be applied locally and then sent out to the slaves.

H-Bridge driver control

The H-Bridge driver is essentially a dual, two leg totem pole drive with control logic. The default power-on state is to have the drives completely disabled with all 4 outputs at zero. Configuring the SCMD causes the driver to connect to two PWM generators, choosing polarity and source. They can be configured such that both outputs will be time and duty synchronized, allowing bridging of the output terminals for double current.

Power-on enumeration

When power is first applied, the master operates at the selected serial configuration and address, and all slaves exist on an address that is not used. When the master is ready, it triggers the first driver to move to a polling address, accept an assigned address, and signal the next slave. In turn, each slave receives an address in this manner.

When the master can't get a response within a set number of tries, enumeration stops and the master is ready to accept control.

Fail safe operation

The default operation of the SCMD is to simply drive the motors at the last level indicated until another command come in. To prevent run-away situations a Fail safe is implemented. When configured, the drivers will remove power to the outputs if the serial bus remains inactive for a period of time, which can be set up to 2.5 seconds.

The config in pin on the master and slaves can also work as a reset line, or bus recovery attempt request. For the slaves, bringing the line low resets the device. For the master, pulsing the line can cause either low-level reboot, re-enumeration of the slaves, or reinitialization of either bus.

Using these two features, a system can be made to recover from many situations.

Thermal Information

The temperature rise of the driver IC is related to the current in the motors. To determine the load, attach motors directly to a power supply and apply torque to approximate the application conditions. Illustration 3 show measured steady state temperatures of three heat sink configurations at various loads. It can be used to approximate the operating temperature and determine acceptable cooling.

It is foreseeable that the SCMD will be attached to something with a great thermal mass or chassis, but not have adequate airflow or outward conduction paths. To test this, a SCMD is attached to a 3/4 x 1/2 x 3 inch aluminum bar and is set to drive 2 amps total output current. Illustration 4 is a graph of the IC temperature. It is an emulation of a worst case chassis-mounted or semi insulated uncooled application.

Illustration 5, Illustration 6, Illustration 7, and Illustration 8 show the configurations for the tests. For steady state tests the hood was used to stabilize the air surrounding the EUT only, and had a large opening.

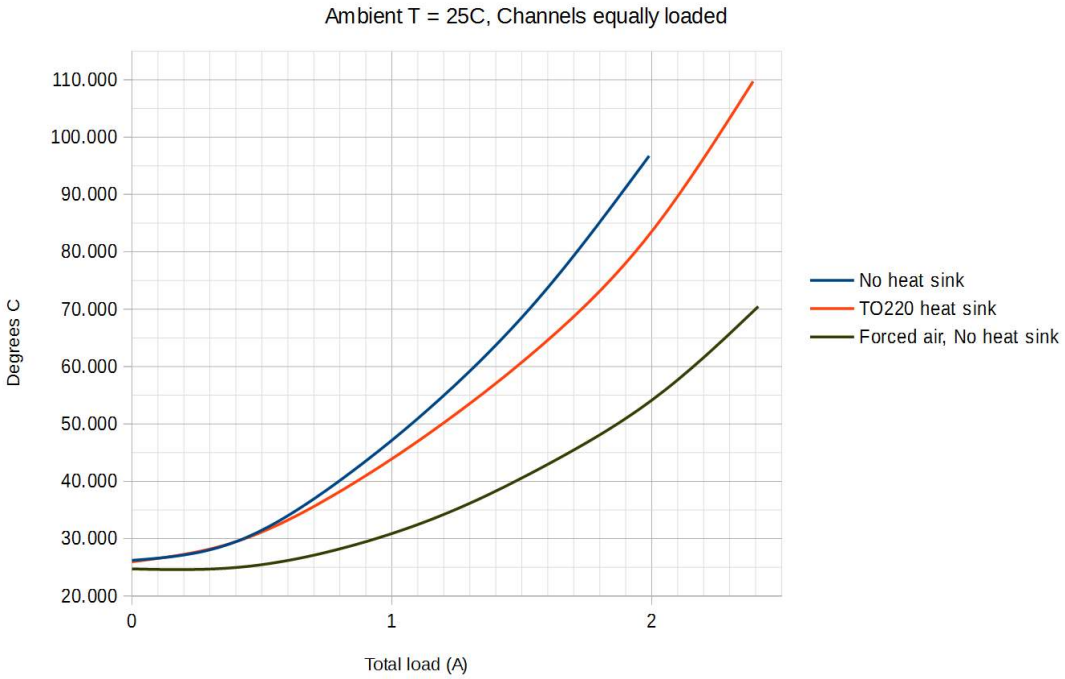


Illustration 3: Temperature rise as a function of total driver load

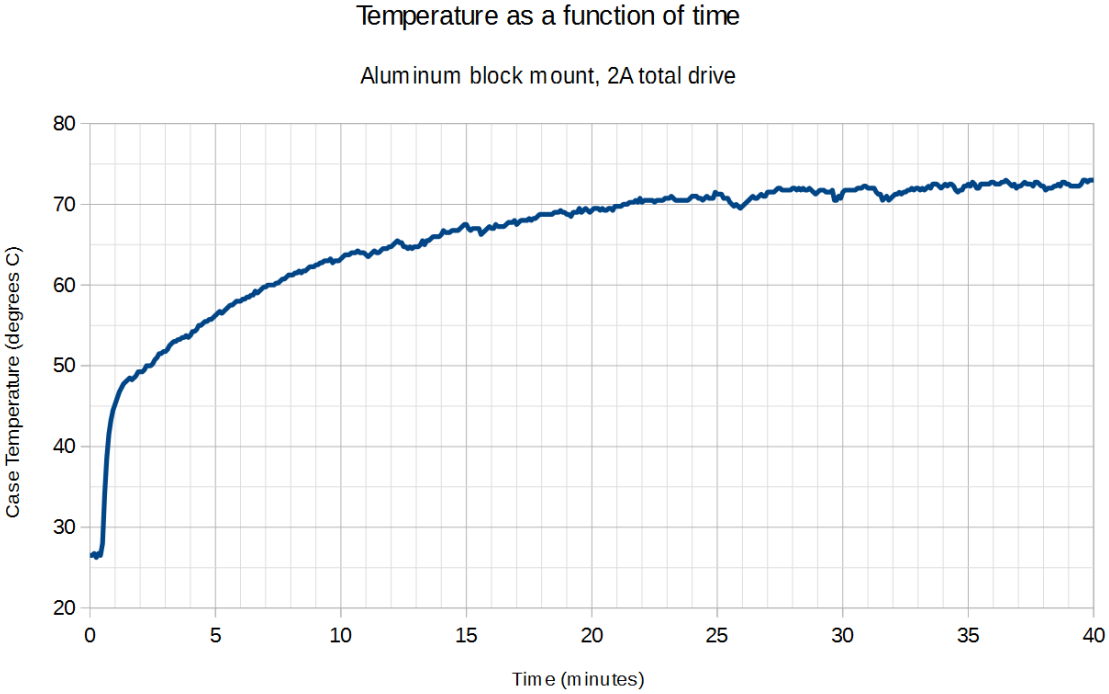


Illustration 4: Using a thermal mass as heat sink

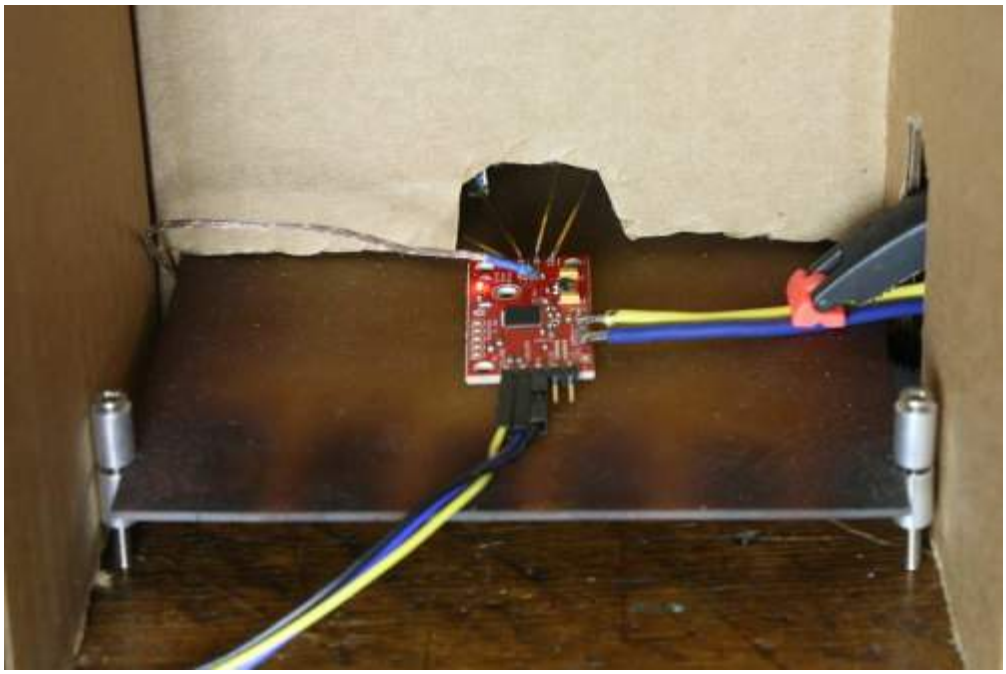


Illustration 5: Uncooled test

Illustration 6: TO220 heat sink

Illustration 7: Forced air cooled test

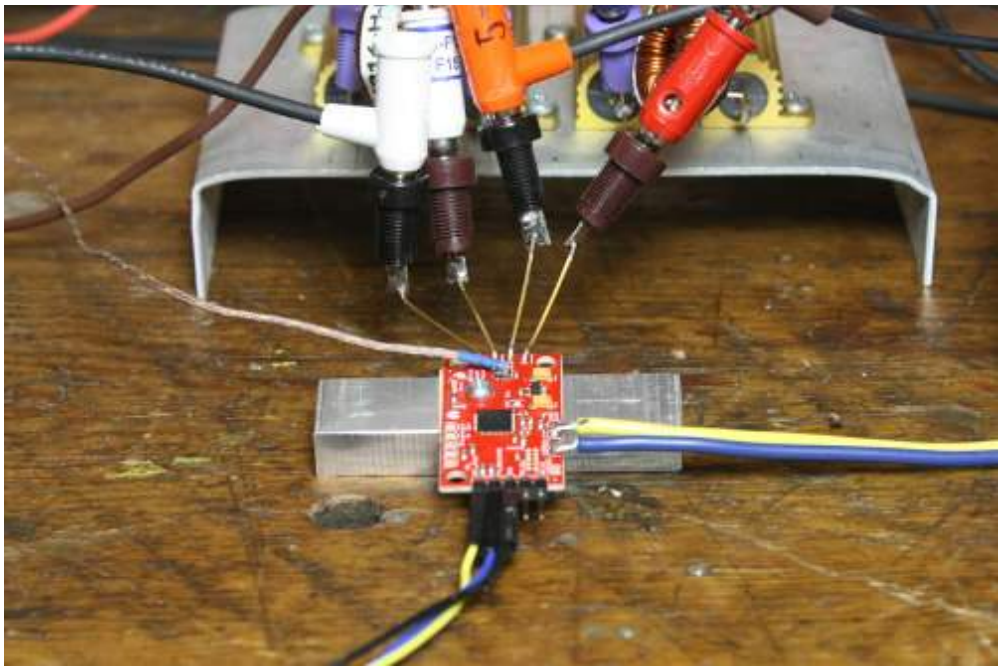


Illustration 8: Thermal mass test

Communication Interfaces

The user port can operate as I2C, SPI, or TTL UART.

User port I2C

In I2C mode, the SCMD emulates the standards for bus hardware and timing, but is a little different when it comes to actual getting data from the device. The PSoC is not configured to directly read from its memory with the serial peripheral. Reading from the device requires two bus operations, one to set the address and prepare the data, and one to get the data back.

Write operation

Write operations are straightforward. To write a byte of data, send the write command with address (Address with cleared read bit), address to write, then data to write.

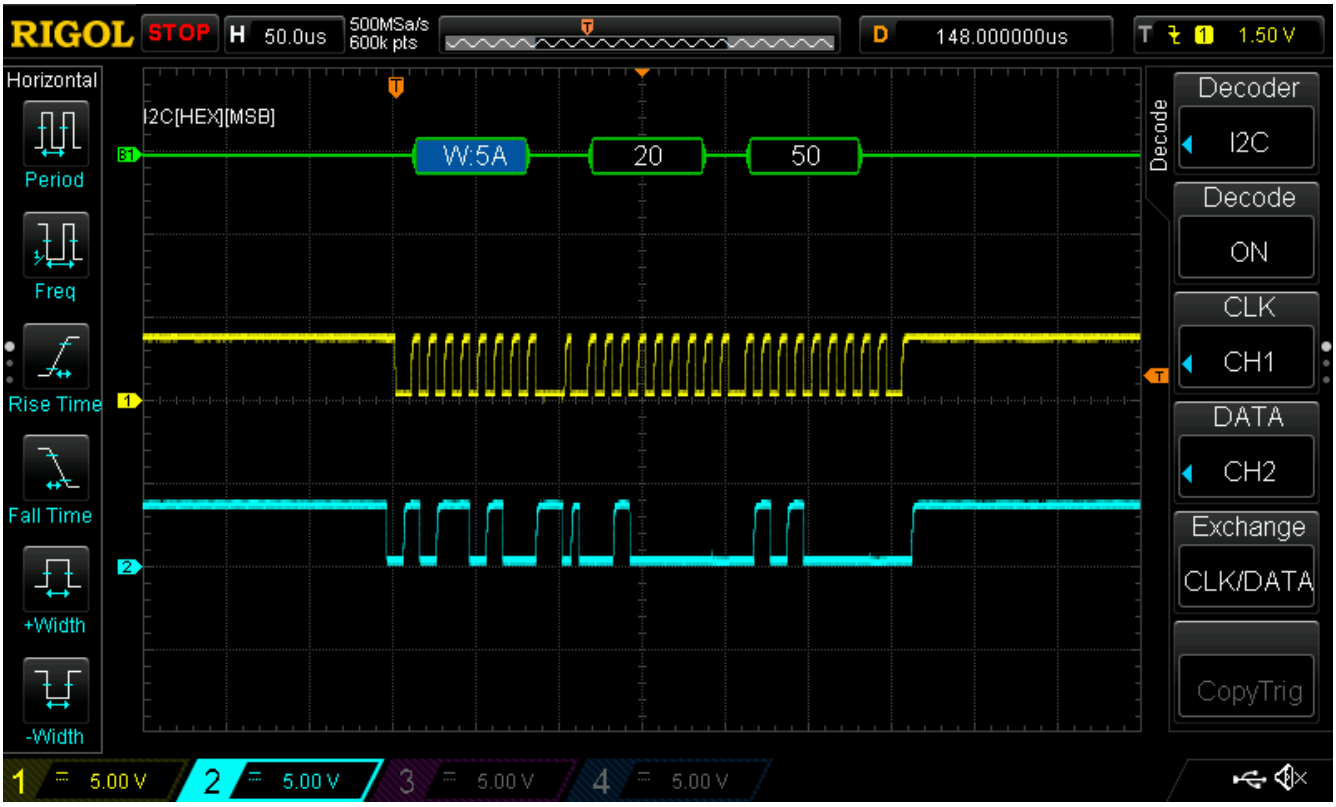


Illustration 9: A write operation on the user port by I2C

Example Arduino Routine:

```
//Write a byte
Wire.beginTransmission(settings.I2CAddress);
Wire.write(offset);
Wire.write(dataToWrite);
Wire.endTransmission();
```

Read operation

To read from the SCMD, send two commands. The first is a write command with address byte, but no data. The next is read command with read bit set, and 1 byte requested. The returned data will be come from the data location specified in the previous write command.

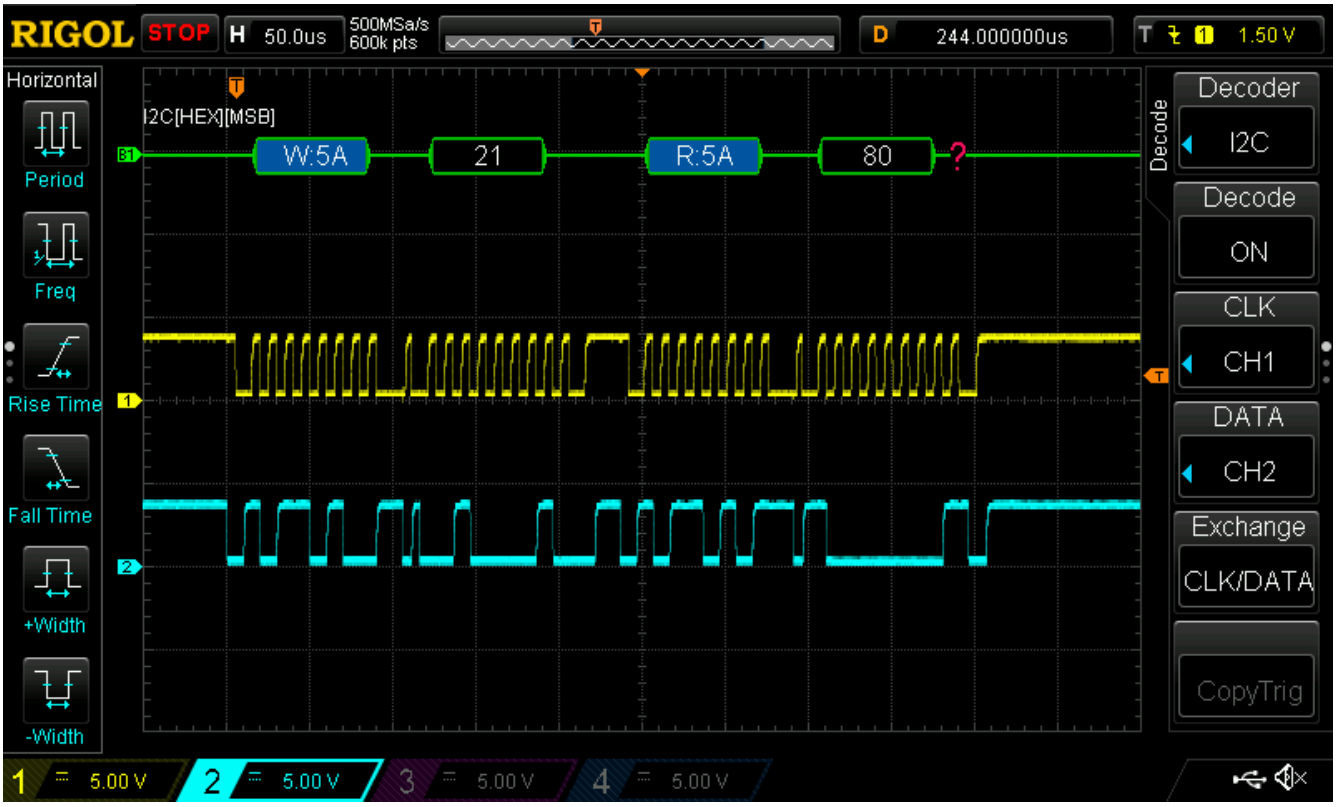


Illustration 10: A read operation on the user port by I2C. Offset 21 is retrieved from I2C address 0x5a, and the data is 0x80

Example Arduino Routine:

```
//Read a byte
Wire.beginTransmission(settings.I2CAddress);
Wire.write(offset);
Wire.endTransmission();
Wire.requestFrom(settings.I2CAddress, numBytes);
while ( Wire.available() ) // slave may send less than requested
{
     result = Wire.read(); // receive a byte as a proper uint8_t
}
```

User port SPI

Like SPI mode, the SCMD emulates the standards for bus hardware and timing, but is a little different when it comes to actual getting data from the device.

SPI mode can operate up to a 1 MHz bit rate, but requires 20 us delay between operations. The bus operates in SPI mode 0, with a low-idling clock and data being sampled on the rising edge.

Illustration 11 shows an example transfer at the max allowed speed, 1 MHz. It can be seen that MOSI data (trace 2) is asserted for the low period before the clock line is raised, and location 0x16 is being accessed for read. The MISO data (trace 3) shows data that was left over from a previous transfer (it will be discarded), but serves to illustrate that the SCMD sets bit values in response to a falling clock. The propagation delay can be seen well at the horizontal marker.

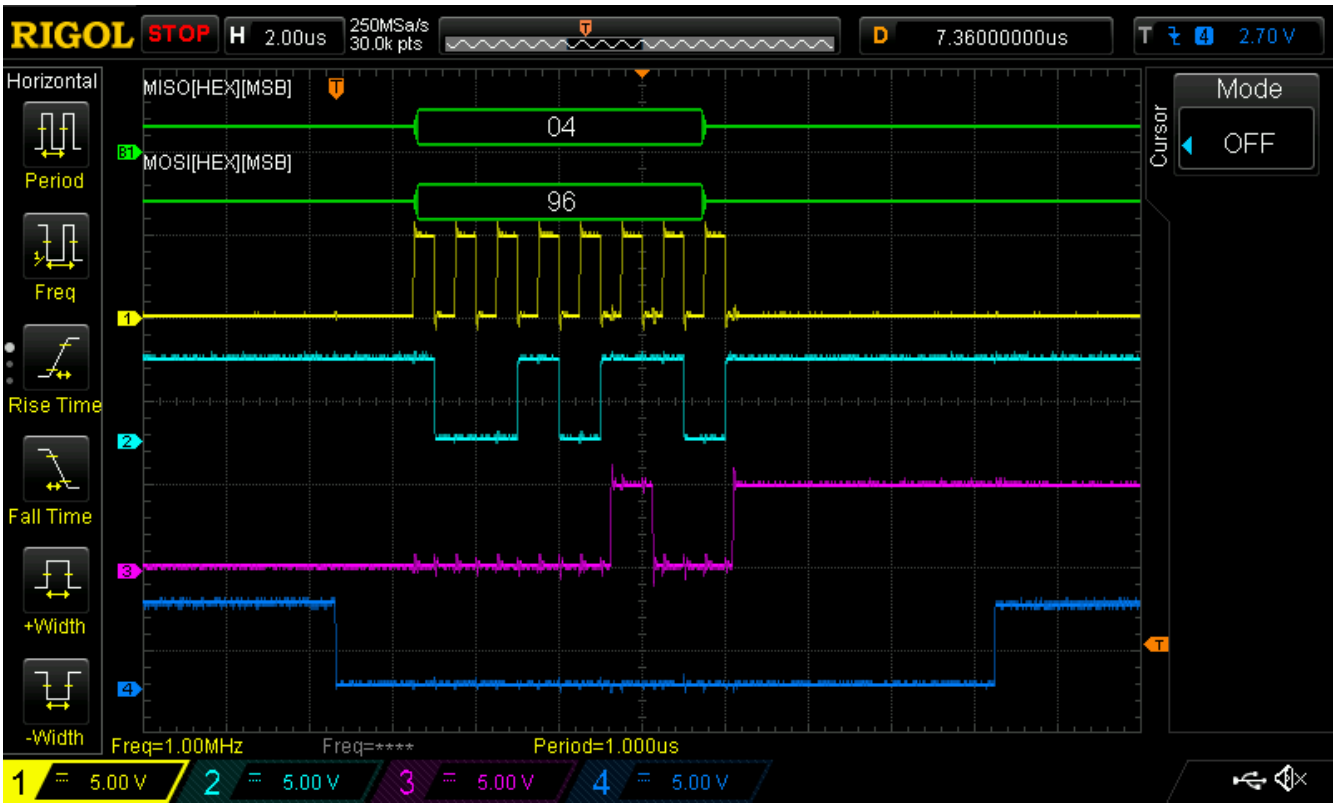


Illustration 11: An example SPI transfer showing clock and data alignment

Write operation

Writing to the bus operates as would be expected for SPI, with memory address and mode combined into the first byte, and a following byte of data. After the transfer is initiated by the master, at least 20us should be given before attempting further access.

(Note: the Arduino library adds 50us delay by for-loop counting)

Illustration 12 shows a complete write operation. The master requests to write address 0x20 with data 0xA0. The returned data is discarded.

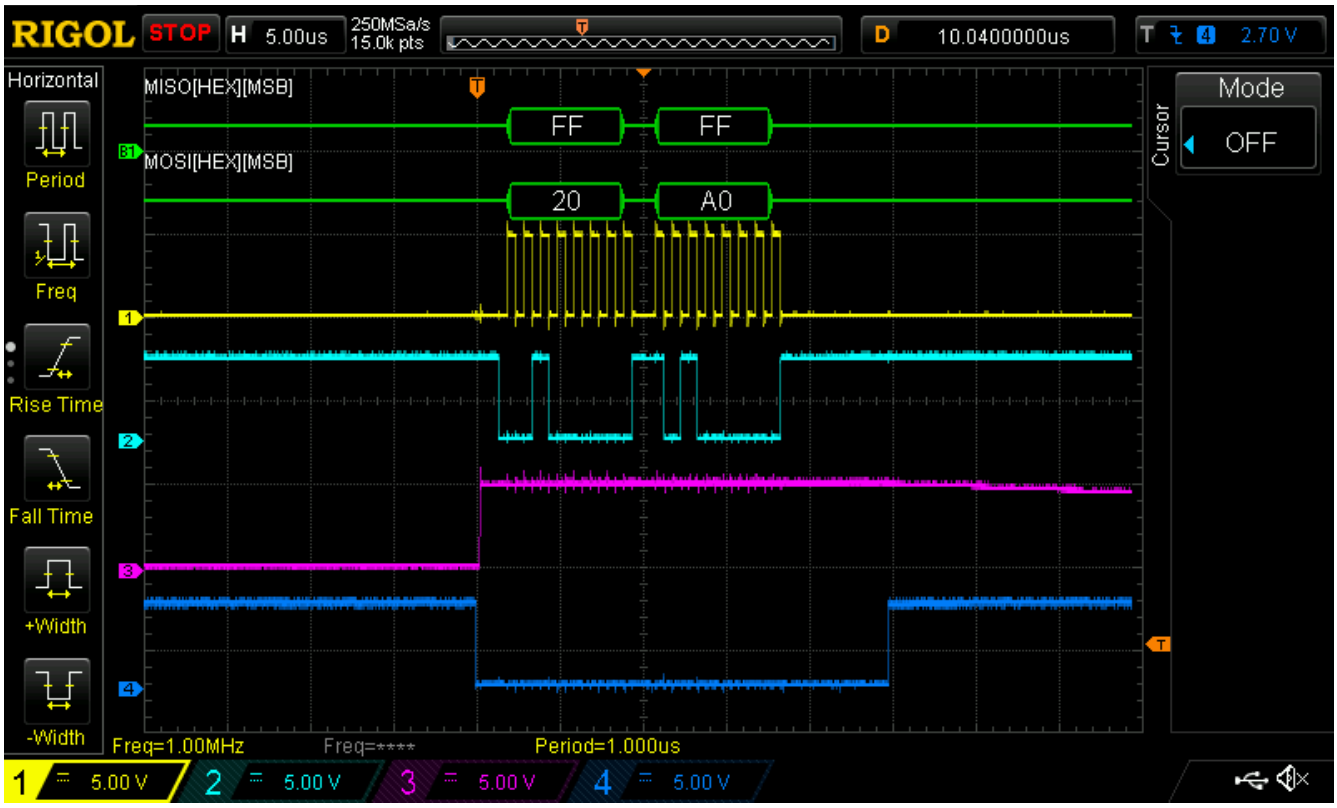


Illustration 12: A complete SPI write operation

Read operation

Reading requires a read to the target address with the MISO data being discarded, then a second read to any address. The MISO data from the second read will be the data at the first location. Afterward, the master should wait 20us before further access.

(Note: the Arduino library adds 50us delay by for-loop counting)

Illustration 13 shows a complete read operation. The first transfer causes the SCMD to collect the data in the requested register (0x01, but with read bit set so it appears to be 0x81), which is available on the next read operation. The contents of 0x01 (the ID register) are then returned after the necessary delay.

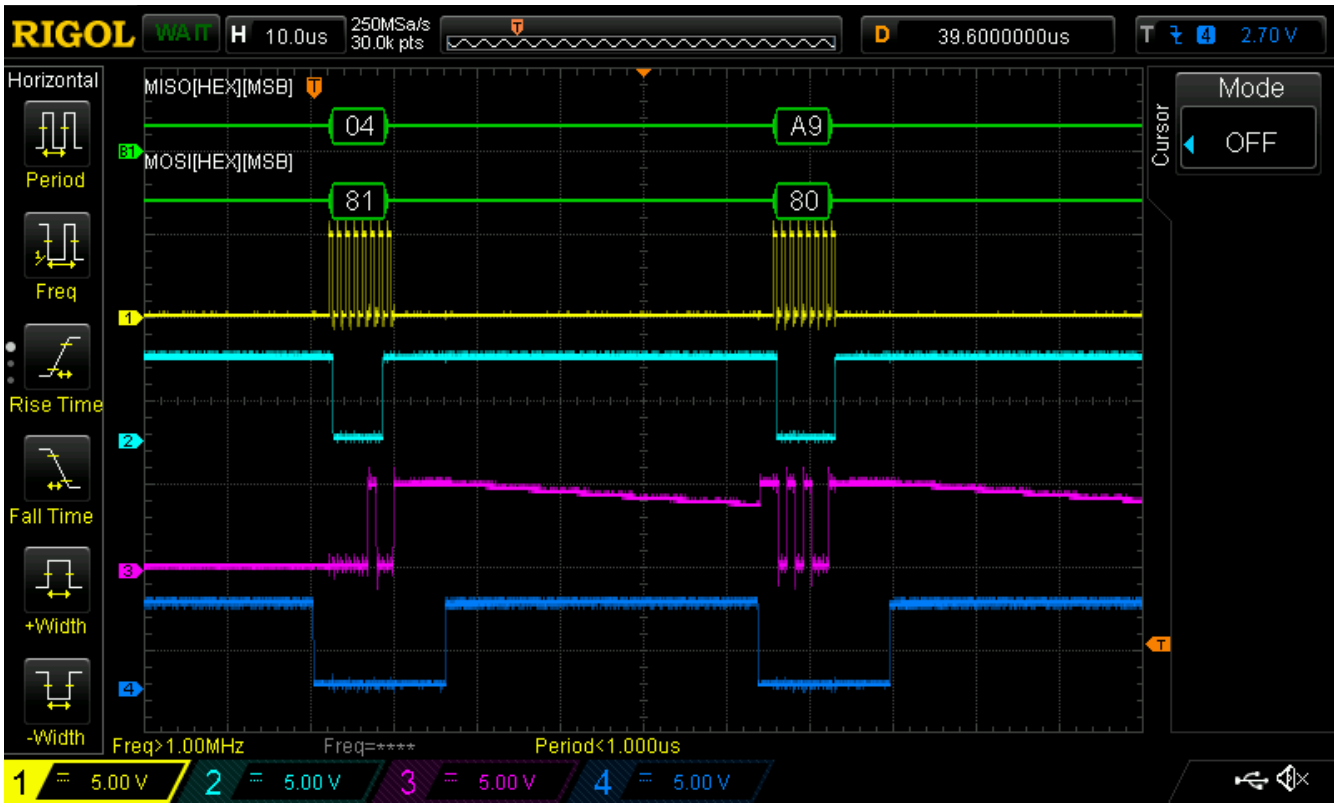


Illustration 13: A complete SPI read operation

User port UART and command set

The command parser is built to accept short strings of ascii data, compacted to reduce data transfer size while being easy to handle by standard UART hardware. The general form is a letter indicating which operation to perform followed by a series of numbers and letters, and finally a carriage return and line feed. As a string, an example command would be “M0F50\r\n”.

Return codes:

Overflow (“**ovf**”): The input buffer is filled. Send delimiter to clear buffer

Invalid Syntax (“**inv**”): The first character in the command is not one of the defined prefixes.

Formatting Error (“**fmt**”): There was something generally wrong with the command, like it had the wrong number of characters or an out of range value was detected.

No Motor (“**nom**”): The motor’s number is past the most downstream slave detected.

Drive motor:
M*ndl*
n = motor number, single or double digits
d = direction, R or F
l = level, 0 to 100

Example:
"M1F34\r\n" – Drive master motor B forward at 34%.
"M2R80\r\n" – Drive slave motor A reverse at 80%.

Invert motor polarity:
M*n*I
n = motor number, single or double digits

Example:
"M1S\r\n" – Invert the polarity of motor 1.

Clear inversion:
M*n*C
n = motor number, single or double digits

Example:
"M1C\r\n" – Set polarity of motor 1 to default.

Write register:
"W*rh*"
r = two digit hex address
h = two digit hex data

Example:
"W20FF\r\n" – Write 0xFF to register 0x20 (MA_DRIVE).

Read register:
R*r*
r = two digit hex address

Example:
"R01\r\n" – Read address 0x01 (ID), bus will display "A9", ID word of 0xA9

Enable/disable outputs:
E

Example:
"E\r\n" – Enable all outputs

Disable outputs:
D

Example:
"D\r\n" – Disable all outputs

Bridge outputs:
B*r*
r = Motor driver number, 0 is master, 1-16 is slave

Example:
"B0\r\n" – Bridge master's outputs
"B2\r\n" – Bridge the 2nd physical slave's outputs (motors 4&5)

Un-bridge outputs:
N*r*
r = Motor driver number, 0 is master, 1-16 is slave

Example:
"N0\r\n" – Un-bridge master's outputs
"N2\r\n" – Un-bridge the 2nd physical slave's outputs (motors 4&5)

Change UART baud:
U*n*
n = baud rate selection

Rates supported:
1 – 2400
2 – 4800
3 – 9600 (Default for config = "0000")
4 – 14400
5 – 19200
6 – 38400
7 – 57600 (Default for config = "1101")
8 – 115200 (Default for config = "1110")

Example:
"U3\r\n" – Set baud rate to 9600
"U8\r\n" – Set baud rate to 115200

Expansion port

The expansion port is designed to allow addition of up to 16 slave SCMDs. It contains an I2C bus, enumeration control IO, and a ground.

The config in and out pins are used to signal to a slave that it's time to accept an address, and to signal to the next slave that it's done, and that the next slave should accept an address. See Slave address enumeration for a full explanation.

The I2C bus can operate at 50, 100, or 400 kHz based on user configuration. The default speed is 100 kHz. Table 3 Shows the length of data on the wire. Turn around time before sending the next packet is always 150 us.

Bitrate	Length of data packet
50kHz	800us
100kHz (default)	420us
400kHz	140us

Table 3: Typical slave packet duration

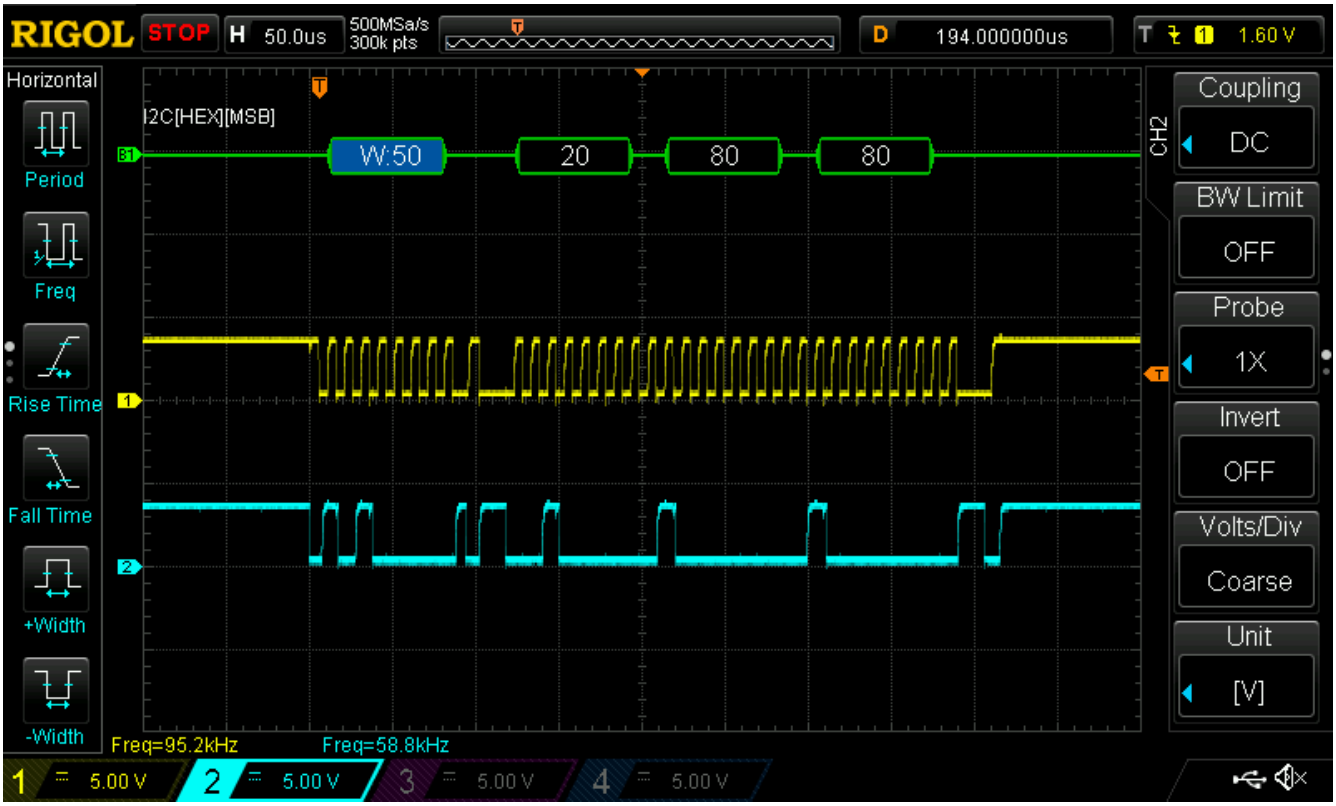


Illustration 14: A single slave packet operating at 100kHz

During regular operation, the master periodically sends motor drive data to the slaves as a two byte payload. Illustration 14 Shows a data transfer to a single slave. It can be seen that the master writes two values of 80 (center) to slave address 0x50, at data address 0x20. The duration the master waits before sending new data is dependent on the UPDATE_RATE register. Table 4 Shows time between slave transmissions.

FORCE_UPDATE can be used to send a packet at any time by writing a self clearing 0x01. When UPDATE_RATE is set to 0x00, this is the only way to update motor drivers, but it can be done during

regular operation too, especially if long update times are used but more instant response is intermittently desired.

UPDATE_RATE	Idle time
0x01	1.5ms
0x0A (default)	10.6ms
0xFF	255ms

Table 4: Typical dead time between slave transmissions

To calculate max update rate for motors, use the following equation. It is important to note that the remote update loop is of low priority in the SCMD and that this is an approximation.:

$$T\_cycle = (number\_of\_slaves * (packet\_time + 150us)) + idle\_time$$

as an example, when 16 slaves are used at 400 khz at a UPDATE_RATE set to 0x01,

$$T\_cycle = (16*(140 + 150) + 1500)us = 6.14ms$$

As seen in Illustration 15, the cycle count is close to the calculated approximation, but the dead time is longer than 1.5ms as expected.

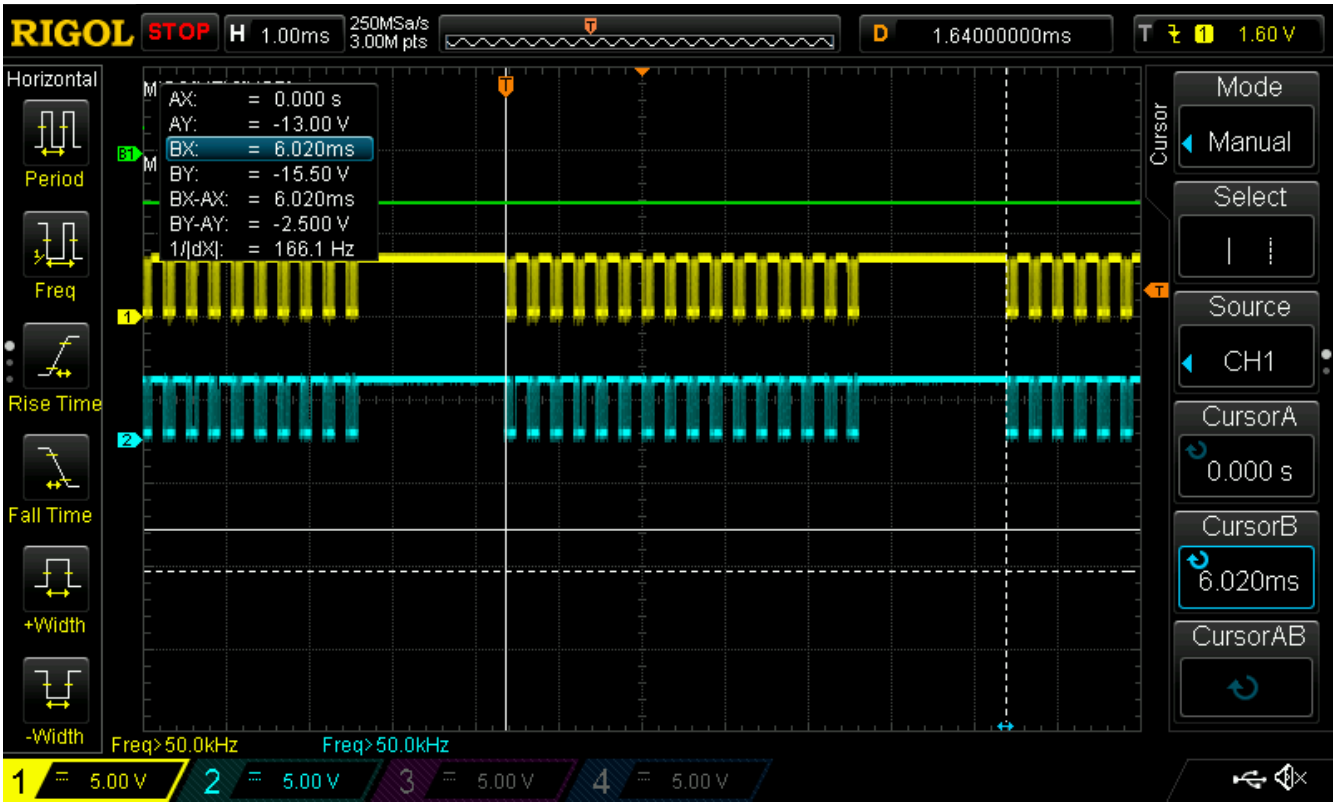


Illustration 15: A complete cycle with 16 slaves attached, 400kHz bus speed, and 1ms dead time setting

Arduino Library Reference

Classes and Structures

There are a few classes used in the library. The main class is called SCMD, which is the object that talks to the motor drivers. There are also a couple structs in use -- SCMDSettings and SCMDDiagnostics. A SCMDSettings object named settings is present within the SCMD class for configuration.

SCMDDiagnostics contains a bunch of 8 bit values of data for use with getDiagnostics and getRemoteDiagnostics. Declared objects are passed as a reference to the diagnostic function and written by the collected data.

```
struct SCMDDiagnostics
{
     public:
     //Attainable metrics from SCMD
     uint8_t numberOfSlaves = 0;
     uint8_t U_I2C_RD_ERR = 0;
     uint8_t U_I2C_WR_ERR = 0;
     uint8_t U_BUF_DUMPED = 0;
     uint8_t E_I2C_RD_ERR = 0;
     uint8_t E_I2C_WR_ERR = 0;
     uint8_t LOOP_TIME = 0;
     uint8_t SLV_POLL_CNT = 0;
     uint8_t MST_E_ERR = 0;
     uint8_t MST_E_STATUS = 0;
     uint8_t FSAFE_FAULTS = 0;
     uint8_t REG_OOR_CNT = 0;
     uint8_t REG_RO_WRITE_CNT = 0;

 };
```

SCMDSettings is an type for the settings member of SCMD. It is declared public to be configured by the user. This isn't the best way to do it but protective methods was considered overly complicated.

```
struct SCMDSettings
{
  public:

  //Main Interface and mode settings
    uint8_t commInterface;
    uint8_t I2CAddress;
    uint8_t chipSelectPin;

};
```

SCMD is used to declare a single chain of motor drivers at specified port and modes in settings. The contained functions are described in a later section.

```
class SCMD
{
  public:
     //settings
     SCMDSettings settings;
     SCMD( void );

     uint8_t begin( void );
     ...   (Other functions...)

     uint16_t i2cFaults; //Location to hold i2c faults for alternate
driver

};
```

Construction

The library is made such that new motor driver objects are constructed without parameters, and are configured later before calling .begin().

Example:

```
SCMD myMotorDriver; //This creates an instance of SCMD which will be
bound to a single master.
```

Settings

The main SCMD class has a public member which is named settings. To configure settings, use the format,

```
myMotorDriver.settings.I2CAddress = (...);
```

then call .begin() to apply.

settings contains the following members:

`uint8_t commInterface` -- Set equal to I2C_MODE or SPI_MODE

`uint8_t I2CAddress` -- Set to address that master is configured to in case of I2C usage

`uint8_t chipSelectPin` -- Set to chip select pin used on Arduino in case of SPI

Functions

`uint8_t` **`begin`**`( void );`

Call after providing settings to start the wire library, apply the settings, and get the ID word (return value should be 0xA9). Don't progress unless this returns 0xA9!

`bool` **`ready`**`( void );`

This function checks to see if the SCMD is done booting and is ready to receive commands. Use this after .begin(), and don't progress to your main program until this returns true.

`bool` **`busy`**`( void );`

This function checks to see if the SCMD busy with an operation. Wait for busy to be clear before sending each configuration commands (not needed for motor drive levels).

`void` **`enable`**`( void );`

Call after .begin(); to allow PWM signals into the H-bridges. If any outputs are connected as bridged, configure the driver to be bridged before calling .enable();. This prevents the bridges from shorting out each other before configuration.

`void` **`disable`**`( void );`

Call to remove drive from the H-bridges. All outputs will go low.

`void` **`reset`**`( void );`

This resets the I2C hardware for Teensy 3 devices using the alternate library, and nothing otherwise.

`void` **`setDrive`**`( uint8_t channel, uint8_t direction, uint8_t level );`

This sets an output to drive at a level and direction.

`channel:` Motor number, 0 through 33.

`direction:` 1 or 0 for forward or backwards.

`level:` 0 to 255 for drive strength.

`void` **`inversionMode`**`( uint8_t motorNum, uint8_t polarity );`

This switches the perceived direction of a particular motor.

`motorNum`: Motor number, 0 through 33.

`polarity`: 0 for normal and 1 for inverted direction.

`void` **`bridgingMode`**`( uint8_t driverNum, uint8_t bridged );`

This connects any board's outputs together controlling both from what was the 'A' position.

`driverNum`: Number of connected SCMD, 0 (master) to 16.

`bridged`: 0 for normal and 1 for bridged.

`void` **`getDiagnostics`**`( SCMDDiagnostics &diagObjectReference );`

Returns a diagnostic report from the master.

`&diagObjectReference`: Pass a local SCMDDiagnostics object that will be written into.

`void` **`getRemoteDiagnostics`**`( uint8_t address, SCMDDiagnostics &diagObjectReference );`

This returns a diagnostic report from a slave.

`address`: address of intended slave. This starts at 0x50 for the first slave and goes up from there.

`&diagObjectReference`: Pass a local SCMDDiagnostics object that will be written into.

`void` **`resetDiagnosticCounts`**`( void );`

Clears the diagnostic counts of a master.

`void` **`resetRemoteDiagnosticCounts`**`( uint8_t address );`

Clears the diagnostic counts of a slave.

`address`: address of intended slave. This starts at 0x50 for the first slave and goes up from there.

`uint8_t` **`readRegister`**`(uint8_t offset);`

Returns the contents of a memory location of the master.

> `offset`: Memory address to read.

`void` **`writeRegister`**`(uint8_t offset, uint8_t dataToWrite);`

Writes data to a memory location of the master.

> `offset`: Memory address to write.
>
> `dataToWrite`: Data to write to that address.

`uint8_t` **`readRemoteRegister`**`(uint8_t address, uint8_t offset);`

Returns the contents of a memory location of a slave.

> `address`: address of intended slave. This starts at 0x50 for the first slave and goes up from there.
>
> `offset`: Memory address to read.

`void` **`writeRemoteRegister`**`(uint8_t address, uint8_t offset, uint8_t dataToWrite);`

Writes data to a memory location of a slave.

> `address`: address of intended slave. This starts at 0x50 for the first slave and goes up from there.
>
> `offset`: Memory address to write.
>
> `dataToWrite`: Data to write to that address.

Control Registers

The user facing memory is organized into the following categories:

- Diagnostics and Debug -- Reports counts and time for this driver.
- Local configuration -- Registers that set the drive parameters and clocking of this driver, which do not transfer to slaves when written.
- Motor drive levels -- drive 0 to 255 for -100% to 100% drive. There is one address per possible motor. The first two effect this driver's A and B channel, while the rest are periodically transferred to the appropriate slave.
- Remote configuration -- Packed configuration bits for inversion and bridging in the master that are decoded and sent to the appropriate slaves when written.
- System configuration -- Fail safe, status, enable, and expansion bus control registers.
- Remote access -- Use to perform remote reads and writes from connected slaves.

Register map

(Page 1 of 3)

	Address	Name	Read/ Write	Default	User Lock	Function / Notes
	0x00	FID	R	0x05		Firmware version.
	0x01	ID	R	0xA9		Bit pattern 10101001 chosen for ease of scope viewing and measurement while not being too regular
	0x02	SLAVE_ADDR	R	UNDEF		Address if this is a slave
	0x03	CONFIG_BITS	R	UNDEF		Populated from jumpers on underside of board
Diagnostics and Debug	0x04	U_I2C_RD_ERR	RW	0x00		Measures RD_ERR bits on USER port at time of status clear, caps at 0xFF. Write 0 to reset.
	0x05	U_I2C_WR_ERR	RW	0x00		Measures WR_ERR bits on USER port at time of status clear, caps at 0xFF. Write 0 to reset
	0x06	U_BUF_DUMPED	RW	0x00		Counts of numbers of buffers that were dumped because they had too much data.
	0x07	E_I2C_RD_ERR	RW	0x00		Counts slave mode I2C bus read errors
	0x08	E_I2C_WR_ERR	RW	0x00		Counts slave mode I2C bus write errors
	0x09	LOOP_TIME	RW	0x00		Reports time in 100uS of main loop, peak value (LSB is worth 100uS). Write 0x00 to clear.
	0x0A	SLV_POLL_CNT	R	UNDEF		Reports number of polls looking for slave devices
	0x0B	SLV_TOP_ADDR	R	UNDEF		Highest slave address (top slave is active on this address)
	0x0C	MST_E_ERR	RW	0x00		Reports number of expansion I2C errors counted by a master
	0x0D	MST_E_STATUS	RW	0x00		Status of master controller board, expansion port I2C (write status)
	0x0E	FSAFE_FAULTS	RW	0x00		Reports failsafe condiditons hit. Write 0 to reset
	0x0F	REG_OOR_CNT	RW	0x00		Counts number of register acces attempts outside the specified range
	0x10	REG_RO_WRITE_CNT	RW	0x00		Counts number of attempts to write locked registers (user and master)
	0x11	GEN_TEST_WORD	R	0x00		Writing causes data to be applied to REM_DATA_RD. 0x01: Read config bit pins

	Address	Name	Read/ Write	Default	User Lock	Function / Notes
Local Configuration (Writing causes Local changes only)	0x12	MOTOR_A_INVERT	RW	0x00	Yes	Invert direction of motor A, or both for bridged output
	0x13	MOTOR_B_INVERT	RW	0x00	Yes	Invert direction of motor B
	0x14	BRIDGE	RW	0x00	Yes	Bridge output channels
	0x15	LOCAL_MASTER_LOCK	RW	0x00	Yes	Unlocked when equal to 0x9B (allows writes to read only registers)
	0x16	LOCAL_USER_LOCK	RW	0x5C		Unlocked when equal to 0x5C
	0x17	MST_E_IN_FN	RW	0x00		Set what happens when the master's config in goes high. B1-0: Restart operation 0x0 do nothing 0x1 reboot 0x2 re-enumerate B2: User port behavior 0 – do nothing 1 – reinit user port B3: expansion port behavior 0 – do nothing 1 – reinit expansion port
	0x18	U_PORT_CLKDIV_U	R	UNDEF		This is the divison of the internal 24MHz clock used by the user port. Writing to U_BUS_UART sets this register. For SPI and I2C modes this must be faster than the bit rate, but how is it handled?
	0x19	U_PORT_CLKDIV_L	R	UNDEF		
	0x1A	U_PORT_CLKDIV_CTRL	R	UNDEF		Used to initiate the user port clock
	0x1B	E_PORT_CLKDIV_U	R	UNDEF		This is the divison of the internal 24MHz clock used by the expansion port. Writing to E_BUS_SPEED set this register, either accuratly for master side or faster than bus speed by a margin for the slave side.
	0x1C	E_PORT_CLKDIV_L	R	UNDEF		
	0x1D	E_PORT_CLKDIV_CTRL	R	UNDEF		Used to initiate the expansion port clock
Local Configuration (Writing causes Local changes only)	0x1E	U_BUS_UART_BAUD	R	0x07		Current baud rate for UART. Only accessable through uart command structure. (Even locked, writing has no effect) 0x00: 2400, 0x01: 4800, 0x02: 9600, 0x03: 14400, 0x04: 19200, 0x05: 38400, 0x06: 57600, 0x07: 115200
	0x1F	FSAFE_CTRL	RW	0x11	Yes	Use to configure what happens when failsafe occurs. B0: output behavior 0 – maintain last motor drive levels 1 – Center output drive levels for 0 drive B2-1: Restart operation(Do not use on slaves) 0x0 do nothing 0x1 reboot 0x2 re-enumerate B3: User port behavior 0 – do nothing 1 – reinit user port B4: expansion port behavior 0 – do nothing 1 – reinit expansion port
Motor Drive Levels	0x20	MA_DRIVE	RW	0x80		Drive motor 0 (master A channel)
	0x21	MB_DRIVE	RW	0x80		Drive motor 1 (master B channel)
	0x22	S1A_DRIVE	RW	0x80		Drive motor 2 (slave 1, A channel)
	0x23	S1B_DRIVE	RW	0x80		Drive motor 3 (slave 2, B channel)
	0x24	S2A_DRIVE	RW	0x80		Drive motor 4 (slave 3, A channel)
	0x25	S2B_DRIVE	RW	0x80		Drive motor 5 (slave 4, B channel)
	0x26	S3A_DRIVE	RW	0x80		(...)
	0x27	S3B_DRIVE	RW	0x80		(...)
	(...)	(...)	RW	0x80		(...)
	0x40	S16A_DRIVE	RW	0x80		Drive motor 0 (slave 16, A channel)
	0x41	S16B_DRIVE	RW	0x80		Drive motor 1 (slave 16, B channel)

	Address	Name	Read/ Write	Default	User Lock	Function / Notes
Remote Configuration (Written data is translated to the slaves)	0x50	INV_2_9	RW	0x00	Yes	Invert slave motors LSB: Motor 2 (also S2A) through MSB: Motor 9 (also S5B). Set 1 to invert motor
	0x51	INV_10_17	RW	0x00	Yes	Invert motors 10 though 17
	0x52	INV_18_25	RW	0x00	Yes	Invert motors 18 through 25
	0x53	INV_26_33	RW	0x00	Yes	Invert motors 26 through 33
	0x54	BRIDGE_SLV_L	RW	0x00	Yes	Bridge outputs, referenced by slave in chain. LSB: first slave, MSB: eighth slave. Set 1 to bridge slave
	0x55	BRIDGE_SLV_H	RW	0x00	Yes	Bridge outputs, referenced by slave in chain. LSB: ninth slave, MSB: sixteenth slave. Set 1 to bridge slave
	(...)					
	0x6F	PAGE_SELECT				Select memory page for multi-page applications (Not implemented)
System Configuration (Most are passed directly to the slaves)	0x70	DRIVER_ENABLE	RW	0x00	Yes	Enables or disables all drivers (1 to enable)
	0x71	UPDATE_RATE	RW	0x0A	Yes	Time in ms. All motors are written every UPDATE_RATE. Set 0 to not update (use FORCE_UPDATE) Writes local and sends to slaves.
	0x72	FORCE_UPDATE	RW	0x00		Set to 1 to force motor updates once. This auto resets to 0.
	0x73	E_BUS_SPEED	RW	0x01	Yes	Change the expansion bus bitrate. 0x00: 50kHz 0x01: 100kHz 0x02: 400kHz
	0x74	MASTER_LOCK	RW	0x00	Yes	Unlocked when equal to 0x9B (allows writes to read only registers) Writes local lock and sends to slaves.
	0x75	USER_LOCK	RW	0x5C		Unlocked when equal to 0x5C. Writes local lock and sends to slaves.
	0x76	FSAFE_TIME	RW	0x00	Yes	Set time between I2C messages before failsafe, Tms = 10 * value, or 0 for no failsafe (max 2.55 seconds)
	0x77	STATUS_1	R	0x00		Read back basic program status B0: 1 = Enumeration Complete B1: 1 = Device busy
	0x78	CONTROL_1	RW	0x00	Yes	Low level system control B0: Restart program B1: Re-enumerate slaves (self clears)
Remote access window	0x79	REM_ADDR	RW	0x00		Slave I2C address
	0x7A	REM_OFFSET	RW	0x00		Offset to access (data address)
	0x7B	REM_DATA_WR	RW	0x00		Data staged for write operation
	0x7C	REM_DATA_RD	R	0x00		Data returned from read operation
	0x7D	REM_WRITE	RW	0x00		Write REM_DATA_WR to REM_OFFSET (perform operation)
	0x7E	REM_READ	RW	0x00		Read REM_OFFSET into REM_DATA_RD (perform operation)

Descriptions

0x00: FID

Reports firmware version. This corresponds with the numbering within the gitub repository.

https://github.com/sparkfun/Serial_Controlled_Motor_Driver/tree/RevisionRepaired/Firmware/Releases

0x01: ID

Reports hard-coded ID byte of 0xA9

0x02: SLAVE_ADDR

This register reflects the driver's address as a slave on the expansion bus only. Writing to it changes the address of the slave. This register is used by the master during slave enumeration.

0x03: CONFIG_BITS

Reflects jumper position on bottom of board, and controls how the firmware behaves on a fundemental level. This value should not be changed during operation.

0x04: U_I2C_RD_ERR

This reports number of times the PSoC's USER PORT serial device had a read error while in I2C slave mode.

Write 0x00 to clear.

0x05: U_I2C_WR_ERR

This reports number of times the PSoC's USER PORT serial device had a write error while in I2C master mode.

Write 0x00 to clear.

0x06: U_BUF_DUMPED

This reports the number of times the USER PORT's RX buffer abandoned with data still in it, while in I2C or SPI mode. This can happen if data comes in too fast or if unformed garbage data is received.

Write 0x00 to clear.

0x07: E_I2C_RD_ERR

This reports number of times the PSoC's EXPANSION PORT serial device had a read error while in I2C slave mode.

Write 0x00 to clear.

0x08: E_I2C_WR_ERR

This reports number of times the PSoC's EXPANSION PORT serial device had a write error while in I2C slave mode.

Write 0x00 to clear.

0x09: LOOP_TIME

This register shows a measurement of internal processing loop time. It contains a peak value since last clear. The range is from 0 to 25.5ms, with a LSB worth 100 us.

Write 0x00 to clear.

0x0A: SLV_POLL_CNT

This register is used by the master to count the number of times it has polled for a slave. After enumeration is complete, this will read the max poll limit plus 1. (201)

0x0B: SLV_TOP_ADDR

This register reflects the address of the most down-stream slave, and is determined during slave enumeration. The master uses this value to determine how many slaves to broadcast to, and it is unused in the slaves. By comparing it with the #defined START_SLAVE_ADDR, the number of attached slaves can be determined by the user. Writing to this register will effect the depth of updates that occur, and should never need to be done.

If this register reads a value lower than the #defined START_SLAVE_ADDR, no slaves were detected.

0x0C: MST_E_ERR

Counts number of times a master had a general timeout on communication with the attached slaves.

Write 0x00 to clear.

0x0D: MST_E_STATUS

Reports the last error status response from writing a double value to the slaves.

0x0E: FSAFE_FAULTS

The failsafe ISR hit count. This is incremented every time the watchdog is not satisfied and the ISR is allowed to run.

Write 0x00 to clear.

0x0F: REG_OOR_CNT

This counts the number of times anybody tried to write outside of the range of available registers. This should always remain at 0.

Write 0x00 to clear.

0x10: REG_RO_WRITE_CNT

This reports the number of times anybody tried to write to a locked register. This is very useful to determine if the proper keys are being applied. If, after a program is run, this contains a non-zero value, the program is trying to write locked registers and is being denied.

Write 0x00 to reset.

0x11: GEN_TEST_WORD

This causes the current state of the config jumpers to be written to the REM_DATA_RD, for manufacturing test (but I suppose they could be used as general inputs if necessary.

Write any value to transfer the config jumper state once.

0x12: MOTOR_A_INVERT

0x13: MOTOR_B_INVERT

Motor inverts for the A and B channels. These are used on all boards, master and slave, to control the directions.

Writing 0x01 to either register causes the forward direction of that channel to be reversed

Writing 0x00 returns to the driver to default mode.

0x14: BRIDGE

This causes the channel B source mux to match channel A source in order to allow bridging.

Writing 0x01 to cause channel bridging.

Writing 0x00 to unbridge channels.

0x15: LOCAL_MASTER_LOCK

This register allows all other registers to be writable, for unknown effect.

Write 0x9B to allow free register use.

Write other values to lock read-only registers.

0x16: LOCAL_USER_LOCK

This register allows the configuration registers to be locked from inadvertent access. Anything that can change modes is protected by the user lock, and is unlocked by default. When locked, writing garbage data will invoke garbage motor movements but will not cause the configuration to be lost when correct data is returned. Any register with "USER LOCK" as "YES" can become read-only with this register. (Note: it is possible to accidentally write the user lock key to this register with garbage data.)

Write 0x5C to prevent writes to configuration registers

Write other values to allow writes.

0x17: MST_E_IN_FN

This sets the behavior of the master SCMD's config_in line, on the expansion bus. The user can attach a wire to config_in, program this register, and send a pulse to cause the following behaviors:

0x00: Config_in has no effect

0x01: Full restart – The SCMD will reboot.

0x02: Re-enumerate slaves and resend 'slave configuration' – The enumeration state machines will be restarted. When complete, the master's local copy of intended slave behavior will be vended to the slaves. This may be useful in a situation where slaves are dropping and reappearing on the bus, and a full reconfiguration is not desired.

0x18: U_PORT_CLKDIV_U

0x19: U_PORT_CLKDIV_L

These registers report the clock dividers applied to the serial component's input clock. The CLKDIV_U register contains the upper 8 bits of a 16 bit word, while the CLKDIV_L contains the lower.

Writing has no effect.

0x1A: U_PORT_CLKDIV_CTRL

This register controls the associated clock divider registers. Write any value to cause the values in the U and L registers to be combined and applied to the clock component. The associated serial device will be reinitialized.

0x1B: E_PORT_CLKDIV_U

0x1C: E_PORT_CLKDIV_L

These registers report the clock dividers applied to the serial component's input clock. The CLKDIV_U register contains the upper 8 bits of a 16 bit word, while the CLKDIV_L contains the lower.

Writing has no effect.

0x1D: E_PORT_CLKDIV_CTRL

This register controls the associated clock divider registers. Write any value to cause the values in the U and L registers to be combined and applied to the clock component. The associated serial device will be reinitialized.

0x1E: U_BUS_UART_BAUD

This register reports the current UART baud rate, as indicated by a number. This is used internally by the master, and is read when UART reinitialization occur. Writing has no immediate effect but may cause the baud rate to change during other operations. The UART "U" command should be used to change baud, and not be done here.

Modes:

0x00: 2400
0x01: 4800
0x02: 9600
0x03: 14400
0x04: 19200
0x05: 38400
0x06: 57600
0x07: 115200

0x1F: PAGE_SELECT

This register is reserved for future PSoC based components that may need to access multiple pages of register data. It currently has no effect.

0x20: MA_DRIVE

0x21: MB_DRIVE

The output driver PWMs are directly controlled by these registers.

Writing a value from 0 to 255 sets the duty cycle of the associated channels on any board, master or slave. Note: Writing a value of 0x80 sets the duty cycle to 50%, which will appear as off at the outputs. Write values up and down from there for forward and backwards drive.

0x22: S1A_DRIVE through 0x41: S16B_DRIVE

These registers are used by the master to hold the slave drive information. Setting these registers sets the next value the master will send to the slave's MA_DRIVE and MB_DRIVE registers. See UPDATE_RATE and FORCE_UPDATE to control how often this information is pushed out to the slaves.

This register is unused in the slaves.

0x50: INV_2_9

0x51: INV_10_17

0x52: INV_18_25

0x53: INV_26_33

These registers set the inversion mode for all slave motors, as a bit pattern.

Writing data to this register causes the data to be instantly transferred to the slaves. Setting a single bit in the register causes that motor number to be inverted.

When setting or clearing a single motor's inversion state, a read-modify-write operation should be done to maintain the existing configuration of other motors.

0x54: BRIDGE_SLV_L

0x55: BRIDGE_SLV_H

These registers set the bridging mode for all slave SCMDs, as a bit pattern.

Writing data to this register causes the data to be instantly transferred to the slaves. Setting a single bit in the register causes that slave number to be bridged.

When setting or clearing a single driver's bridging state, a read-modify-write operation should be done to maintain the existing configuration of other motors.

0x70: DRIVER_ENABLE

This register controls the driver enable line and can effectively remove all energy from the outputs.

Write 0x01: Enable this driver

Write 0x00: Disable this driver.

Writing data to this register as a master causes the master to send the configuration to the slaves. As a slave, writing data effects output drive only.

0x71: UPDATE_RATE

This register is used to determine how often to send data out to the slaves. The LSB is worth 1 ms, so the update rate can be set between 1 and 255 ms.

Write 0x00: Do not automatically update the slaves.

Write other values: Update at a rate of 1ms to 255 ms.

0x72: FORCE_UPDATE

When the UPDATE_RATE is set to 0x00, writing a non-zero value to this register causes the contents of the XXX_DRIVE registers to be send out to the slaves at the next available time (sent by enumeration state machines). This register is automatically cleared.

Write 0x01: Send the current drive values to the attached slaves

0x73: E_BUS_SPEED

Writing changes the expansion bus bit rate and reinitialize the bus for the master, and has no effect for the slaves.
Available speeds:
0x00: 50kHz
0x01: 100kHz
0x02: 400kHz

0x74: MASTER_LOCK

This is a global lock location for the LOCAL_MASTER_LOCK.

Writing a value to this register causes the master to apply that value to it's own LOCAL_MASTER_LOCK, and transmit the value to all attached slave LOCAL_MASTER_LOCK registers.

0x75: USER_LOCK

This is a global lock location for the LOCAL_USER_LOCK.

Writing a value to this register causes the master to apply that value to it's own LOCAL_USER_LOCK, and transmit the value to all attached slave LOCAL_USER_LOCK registers.

0x76: FSAFE_TIME

This register sets the watchdog timeout time, from 10 ms to 2.55 seconds. The LSB is worth 10 ms.

Write 0x00: Do not use the watchdog.

Write other values: Set the watchdog timeout time to 10ms * value.

0x77: STATUS_1

This register uses bits to show status. Currently, only b0 is used.

Bit 0: Enumeration complete. Reads 1 if the master state machine has reached the fully enumerated state.

Bit 1: Busy. The busy bit is set when certain operations are requested, and cleared when all blocking operations are done. It should be observed when making changes to related operations.

If writing commands while the SCMD is busy, previously requested commands can be lost.

The bit is set as a logical OR for any changes to the following registers:

INV_2_9, INV_10_17, INV_18_25, INV_26_33, BRIDGE_SLV_L, BRIDGE_SLV_H, DRIVER_ENABLE, MASTER_LOCK, USER_LOCK, FSAFE_TIME, REM_WRITE, REM_READ, FORCE_UPDATE

0x78: CONTROL_1

This register can be used to issue low level orders to the system.

Bit 0: Reset the processor now. The slave chain will also be reset, and all configuration is erased.

Bit 1: Initiate re-enumeration of the slave chain. Afterward, slave configuration data is delivered to the slaves. Current slave drive levels are lost.

0x79: REM_ADDR

Write the address of the slave to communicate with.

0x7A: REM_OFFSET

Write the register number to access of the slave.

0x7B: REM_DATA_WR

Write the data that will be written to the slave REM_OFFSET register, at the REM_ADDR.

0x7C: REM_DATA_RD

Read data from the slave REM_OFFSET register, at the REM_ADDR will be contained here.

0x7D: REM_WRITE

Write any value to initiate a remote write operation. The data in REM_DATA_WR will be written to the slave at REM_ADDR, at address REM_OFFSET. This register will be set to 0x00 automatically

0x7E: REM_READ

Write any value to initiate a remote read operation. The data in the slave at REM_ADDR, REM_OFFSET memory location will be copied to the REM_DATA_RD location, and this register will be automatically cleared.

Wait 5 ms before reading the REM_DATA_RD register to allow the expansion bus to fetch the data.

Program Architecture

The firmware is written to emulate a I2C or SPI peripheral. The register map is the core of the operation, and is surrounded by function calls that operate on the data. This allows the memory controller to report changed status of memory, and to record misuse of the memory.

During initialization, the SCMD:

- Clears and prepares the user space memory, setting default values and lock states
- Reads the config jumpers
- Configures default hardware, ISR and timer operation
- Initializes the PWM generators
- Calculates serial clock rates based on config jumpers
- Initializes its serial peripheral blocks based on config jumpers

After the SCMD is initialized by the boot and initialization calls, it runs a loop that takes the following actions:

- Calls appropriated serial reception parser (or leaves it up to the hardware ISRs to call)
- Steps the master or slave enumeration state machine
- Deals with changed register data
- Checks for, and deals with, changes to hardware reset functions

The config jumpers change the general behavior and role of the expansion port based on slave or master mode. Both roles function the same regarding memory access and H-Bridge driver operation, but the master has more registers that are checked for changed values, and transmits data to the slaves if necessary.

Illustration 16 shows how the master behaves. During master operation the user port directly controls the memory, and the master enumeration state machine is active. This state machine chooses how and when to deal with changing data and serves to look for slaves on the expansion bus.

Illustration 17 Shows how a slave behaves. The expansion bus then deals with the memory directly, and the slave enumeration state machine is active. It is a matching state machine to the master and waits to be discovered, then does things based on the data it has available.

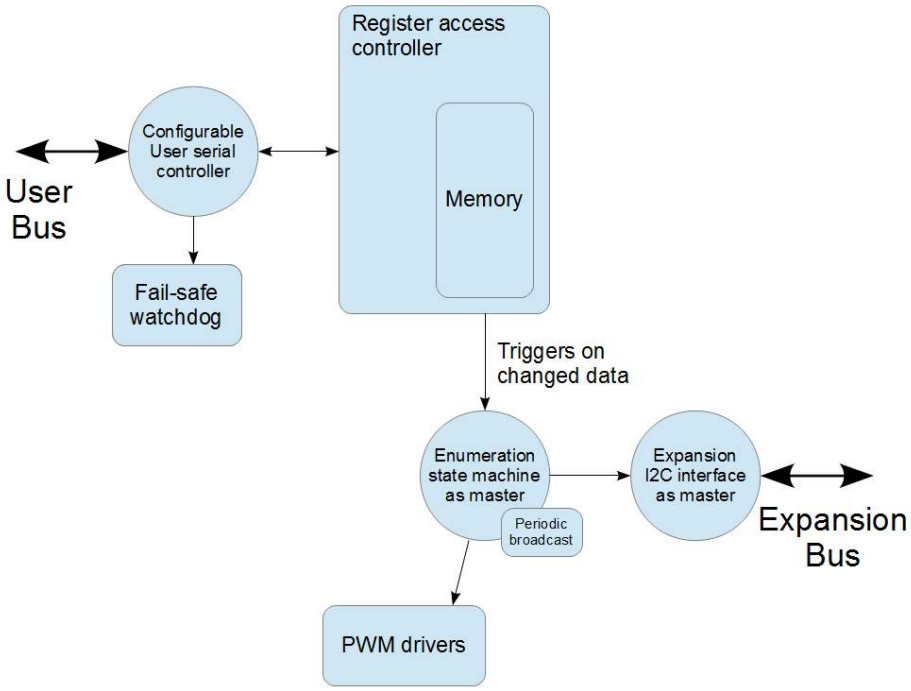


Illustration 16: Topology of a SCMD as a master (or single)

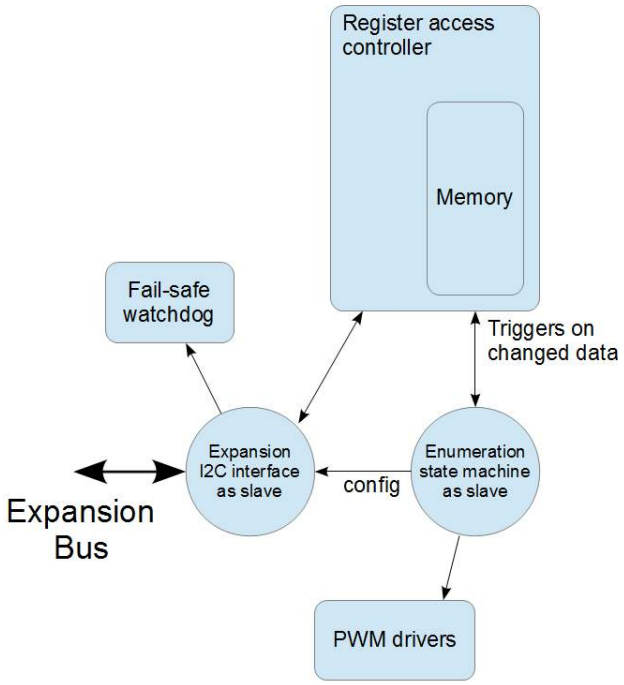


Illustration 17: Topology of a SCMD as a slave

Slave address enumeration

When the master is explores the expansion bus, it assigns addresses to the attached slaves. This happens after a fresh power up, or by master config_in when MST_E_IN_FN is set to 0x02 for bus re-enumeration.

All slaves boot with the address 0x10 programmed into their I2C interfaces. 0x10 is reserved as an address that should never be accessed, it can have multiple devices with the same address at boot.

After a time delay, the master asserts the config_out line. Upon detection by the first slave, that slave switches address to 0x4A, the general poll address.

The master then polls 0x4A by looking for a valid ID word. When found, the master assigns an address to the slave at 0x4A. This causes that slave to assert their config_out line, readying the next slave. The master then assigns the next available address and the cycle continues.

The operation is complete when the master fails 200 times to get an ID word, or if the programmed max slave address is reached. The SLV_TOP_ADDR is programmed with the address of the last slave in the chain as a way of counting how many slaves are present.

Illustration 18 shows actions taken by the master and slaves during enumeration. Arrows indicate direction of information flow within the system.

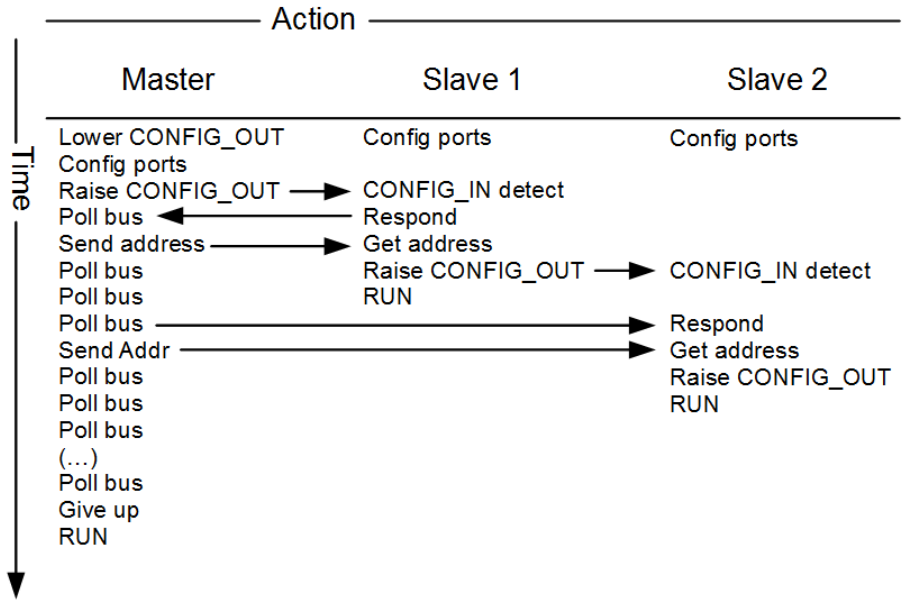


Illustration 18: Actions and communication of the expansion bus

State diagrams

The master and slave state machines work together to accomplish I2C handshaking. These two diagrams implement the handshaking outlined in Illustration 18.

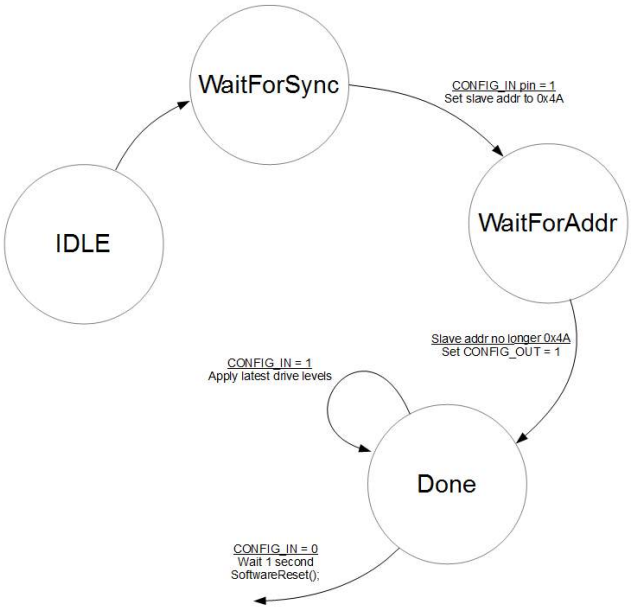


Illustration 19: The slave enumeration state machine

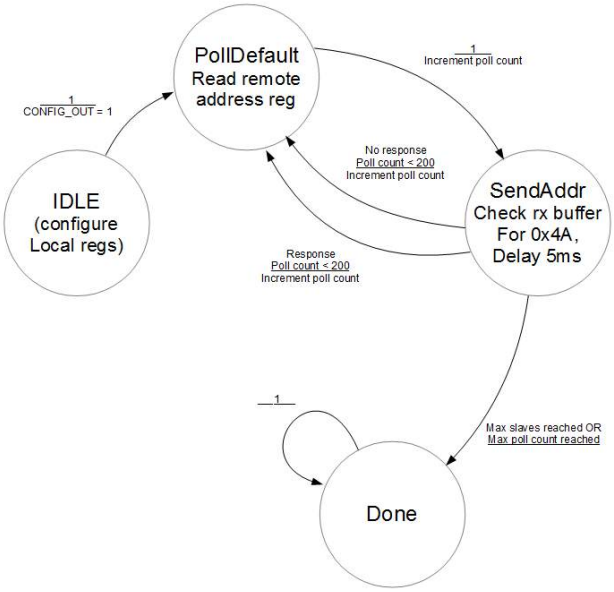


Illustration 20: The master enumeration state machine

Boot timing

When the SCMD first receives power, the slaves get ready to accept a config in request as soon as possible, and the master waits 1 second before beginning to look for slaves. If the slave powers up more than a second before the master it will not be enumerated.

The behavior can be emulated by writing to the CONTROL_1 register's FULL_RESET_BIT.

Illustration 21 shows the timing of a master and 2 slaves power up and enumeration. The user should take care to not attempt to command the motor driver until it has reached completion of boot, while read access will be valid earlier.

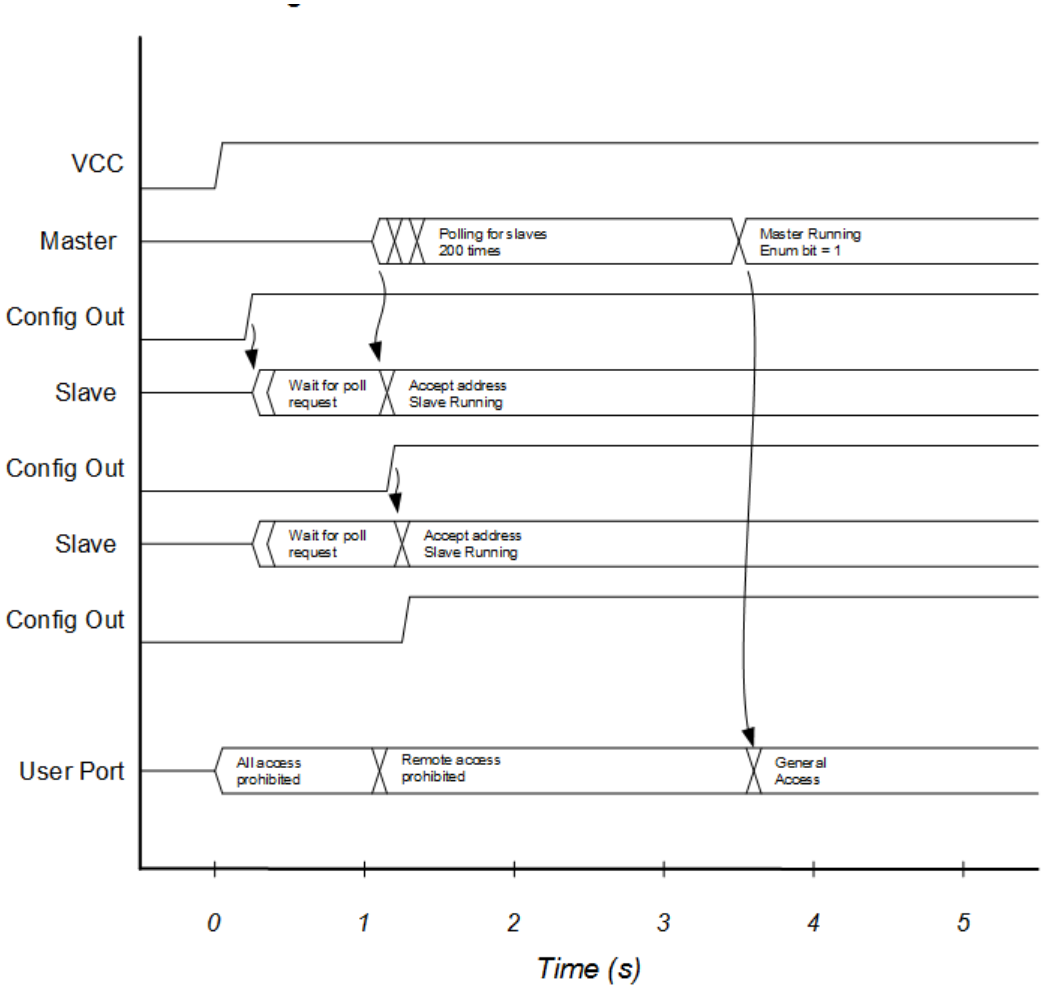


Illustration 21: Power on and full reset enumeration behavior.

re-enumeration timing

CONTROL_1 register's RE_ENUMERATE_BIT causes the master to bring the config out line low to reset the slaves. Then it waits 1 second, resets its state machine, and enumerates the slaves.

After completion, the master vends the global configuration to the slaves, which should return the slaves to their pre-re-enumeration state.

Illustration 22 shows a pseudo-timing diagram of a master re-enumerating 2 slaves. During this process, the user should treat SCMD as if it was freshly booting, and not issue commands right away.

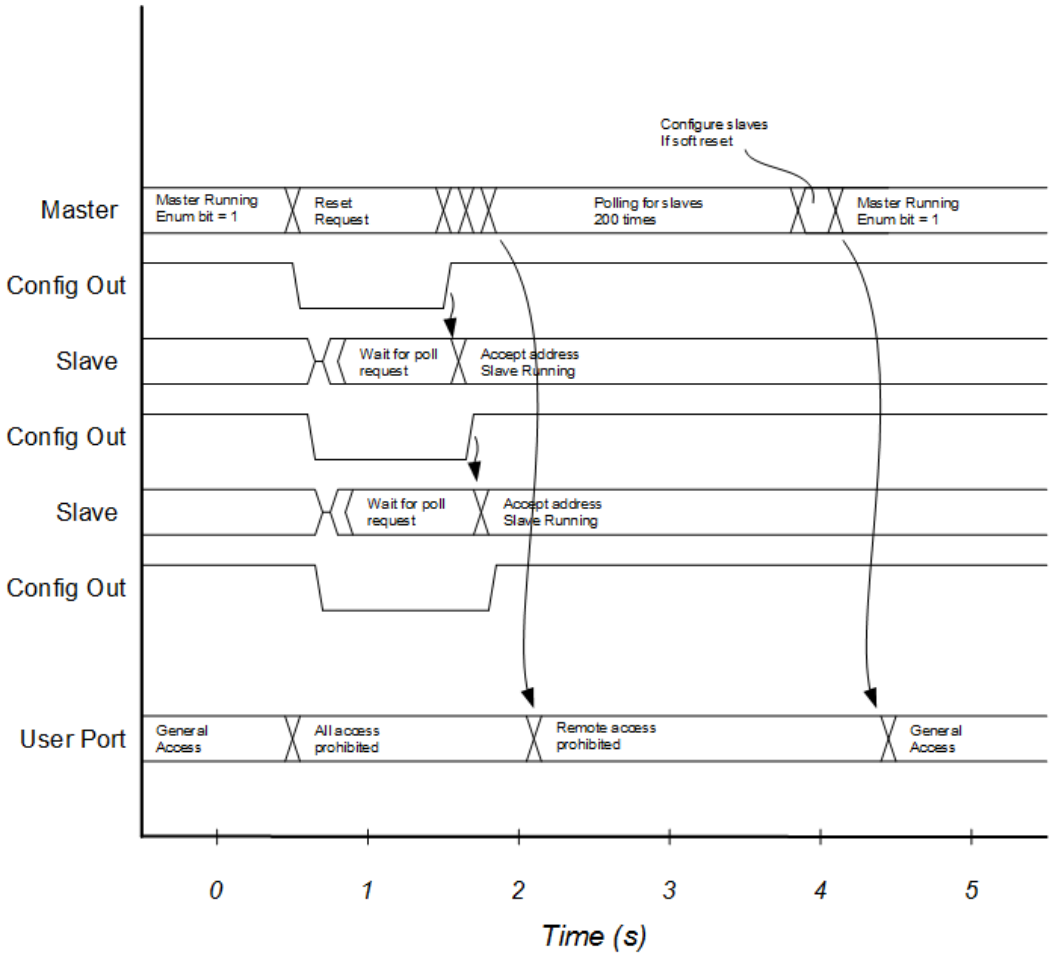
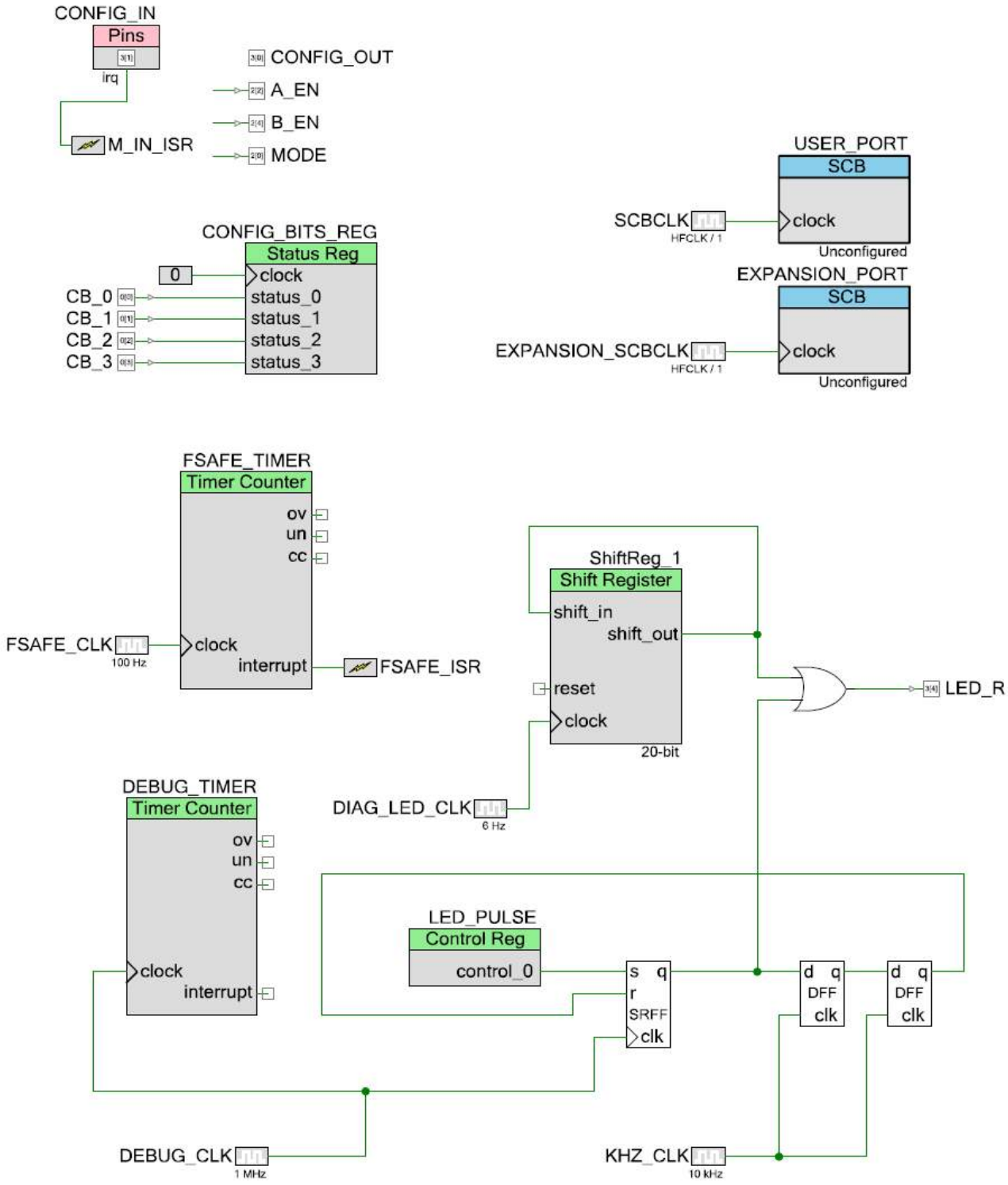
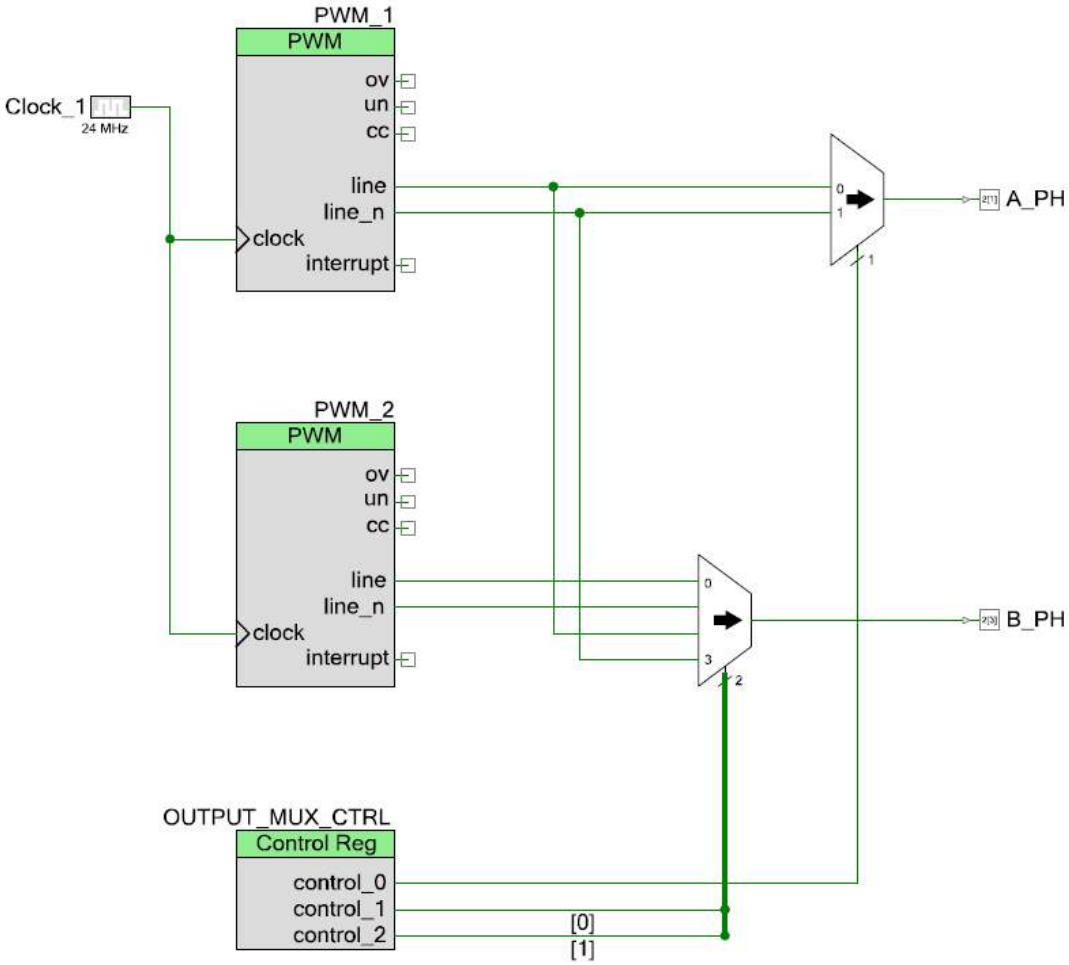


Illustration 22: Re-enumeration behavior.

System Schematic




PWM_1
PWM
ov
un
cc
line
line_n
clock
interrupt
Clock_1
24 MHz
A_PH
PWM_2
PWM
ov
un
cc
line
line_n
clock
interrupt
B_PH
OUTPUT_MUX_CTRL
Control Reg
control_0
control_1
control_2
[0]
[1]

Source Descriptions

registerHandlers.c

registerHandlers.h

Register handlers check for new changes to registers and take action accordingly. Each "active" register is checked, does its tasks, then resets the indicator.

devRegisters.c

devRegisters.h

The memory space used for the registers is defined here, along with a table of control bits for new data checking and lock control. Memory access happens through a set of functions that operate on the table, but prevent unauthorized access and report invalid counts.

serial.c

serial.h

Functions to configure and operate the serial. The serial dividers are calculated by one function, then initialized with another. Contains function to read and write slave data by address. Also contains PSoC configuration structures and application.

customSerialInterrupts.c

customSerialInterrupts.h

Contains routines for dealing with incoming data on each of the bus types. This includes parsing the UART data and taking appropriate actions. The SPI ISR redefines the generated API ISR at the hardware level but the others are called from within the stock ISRs.

SCMD_config.h

Register common names. These are named as in this document with a prefix of SCMD_. Some #define masks and defaults are also defined here. This file should be consistent to the one used by the Arduino library.

slaveEnumeration.c

slaveEnumeration.h

Master and slave state machines for enumeration. Contains code to self-reboot and reset.

main.c

Interrupts and initialization. The main processing loop is in this file.

charMath.c

charMath.h

Light weight ascii conversion.

diagLEDs.c

diagLEDs.h

Control functions for the LED digital logic.

Document History

11-10-2016: Document created.