Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

THIS SPEC IS OBSOLETE

Spec No: 001-75813

Spec Title: H BRIDGE BASED MOTOR DRIVE PROTECTION USING PSOC(R)3 - AN75813

Sunset Owner: K Sanjeev Kumar (KUK)

Replaced by: None

AN75813

H Bridge Based Motor Drive Protection Using PSoC® 3

Author: Sudip Mondal
Associated Project: Yes
Associated Part Family: CY8C38xx
Software Version: PSoC® Creator™ 2.0
Related Application Notes:
"For a complete list of the application notes, click here."

If you have a question, or need help with this application note, contact the author at smon@cypress.com.

AN75813 demonstrates the use of a PSoC 3 for brushed DC motor drive protection and diagnostics. The PSoC 3 protection system is optimized for the widely used H bridge, but it can easily be adapted to other DC motors. The implementation emphasizes the use of digital logic present on the PSoC 3 to free the CPU for more involved tasks such as motor control. This application note specifically addresses motor drive protection and diagnostics and does not discuss motor control.

Contents

Introduction

DC motors are widely used in many automotive applications such as HVAC, power seats, wipers, and power windows. It is important to protect the motor drive system, because failure to do so may result in damaged components and, in some cases, hazardous conditions. Motor drive systems consist of switching elements and drivers. The switching elements (e.g. FETs, relays, IGBTs) are typically arranged in the form of an H bridge. The drivers turn the switching elements on and off according to the commands from the controller. Drivers are required, because microcontrollers typically cannot handle the voltage and current levels required to operate the switches. Fault detection, the protection mechanism, or both are often built into the switches or the drivers to signal an abnormal state such as over current or over voltage to the controlling microprocessor. These "smart" switches and drivers use different methods to report faults ranging from simple digital pins to inter-device communication channels such as SPI or I^2C. When they detect a fault, they either disable the motor drive or leave that task to the controlling microprocessor.

In this application note, we discuss an alternate approach to fault detection and protection. Instead of using pre-packaged motor drivers with built-in protection systems, we use basic components such as FETs and gate drivers. This method results in a larger number of available test signals and hence better determination of fault conditions. Combined with the configurability of PSoC 3 devices, this method also leads to a scalable and programmable system that can be easily shared across multiple bridges. This eliminates the need to build a fault detection system for each newly added motor. We briefly describe the limitations of smart drivers and alternate methods to drive gates. Then we discuss faults and ways to use the PSoC 3 as a fault detection and protection system.

Smart Drivers and their limitations

Drivers are necessary to interface a microcontroller with a switching device as most microcontrollers cannot handle the voltage and current levels required to control switches. Drivers with built-in, smart protection mechanisms are generally more expensive than their less smart counterparts. Smart drivers are attractive for two reasons:

- They provide the level translation required for high-side gate driving.
- It is not easy to find microcontrollers with sufficient programmable analog and digital capabilities to implement a central fault detection and protection system.

However, smart drivers have limited configurability and a fixed set of fault detection abilities. The system designer has limited control over the functional definition of a fault and may have to resort to different devices for different motors. Moreover, many of the drivers can only report a fault when the motor is running. This reduces their effectiveness for infrequently driven motors such as an HVAC flap controller motor or a seat positioning motor.

The PSoC 3 is an excellent solution in these cases. Configurable precise analog and digital blocks inside the device handle all fault detection and protection. This removes the necessity to use expensive smart drivers and switches. The system described in this application note is capable of:

- Easy and dynamic configuration of the definition of fault parameters such as over current threshold, blanking time and so on without the use of additional components
- Flexible fault reporting ranging from the presence of a fault to a detailed status of the motor driving bridge
- Continuous monitoring of inactive motors to reduce the risk of failures and hazards

Because the PSoC 3 cannot handle the voltage and current levels required for a high-side gate drive, we need alternative methods to do so. In the next section, we demonstrate two ways to implement level translation for a high-side gate drive without the use of specialized drivers.

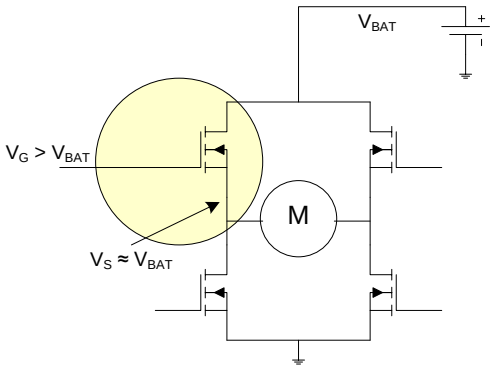
Alternate Methods to Drive H Bridges

It is difficult to drive the gates of high-side N-type FETs, because the gate voltage must be higher than the available battery voltage. Figure 1 illustrates the problem.

When the highlighted switch is turned on, its source voltage V_S is close to the battery voltage V_{BAT}. Therefore, its gate voltage must sufficiently exceed the sum of the source voltage and the gate-to-source threshold. The FETs used in motor drives are designed to handle substantial current (commonly two to four amps) and often have large threshold voltages. Therefore, we need a method to generate a voltage higher than the available battery voltage. Two common methods to generate the higher voltage are:

- Use a capacitive charge pump to drive the gate voltage to almost double the battery voltage
- Use of a boost converter

Figure 1. Gate Voltage Requirement for High Side Switch



Capacitive Pump

Figure 2 illustrates capacitive pumping of the voltage with an example application. It requires a PWM on the high side FET HS 1 while the low side FET LS 2 is kept ON. Low side FETs are relatively easier to drive and are not discussed in this application note. The other two switches remain inactive during the operation. We use an FET M1 with a small threshold voltage to enable direct operation by microcontroller pins.

In Figure 2.a, the microcontroller pin is high (HIGH SIDE CONTROL = 1). As the gate of M1 goes high, it is turned ON which leads to current flow through the resistors R1 and R2. The resistors R1 and R2 are appropriately sized so that the gate voltage (V_G) is near 0, and the FET HS 1 is OFF. As a result, the source voltage of HS1 (V_S) is close to ground. The capacitor (C) is charged to a voltage (V_{BAT}) less one diode drop. The dashed line traces the path of the current.

When the microcontroller pin is low (HIGH SIDE CONTROL = 0). M1 turns OFF. This makes the voltage $V_G = V_C \approx V_{BAT}$ (the gate current of FET HS 1 is negligible). The gate to source voltage exceeds the threshold voltage of the FET HS 1, and it starts to turn ON.

Figure 2. Capacitive Pump Based High-Side Gate Drive

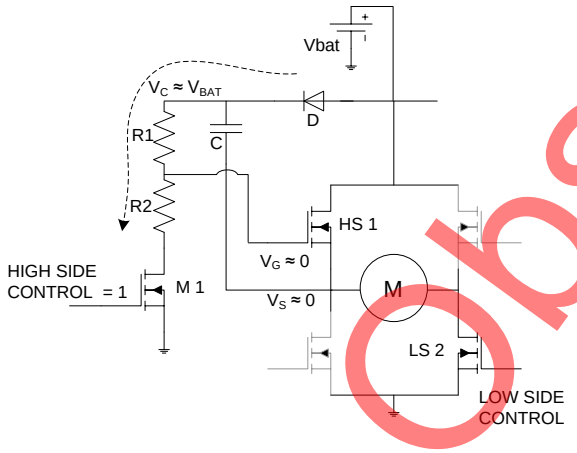


Figure 2.a

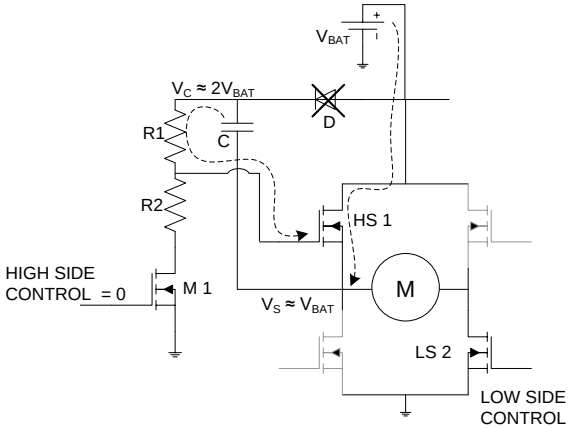


Figure 2.b

As HS 1 turns ON, its source voltage rises. Since the voltage across the capacitor cannot change instantaneously, V_C also increases. In turn, V_G increases since the capacitor supplies the gate of HS 1. The Diode (D) becomes reverse biased and no current flows through it. This process continues until HS 1 is fully turned ON. At this point, V_C is almost equal to twice the battery voltage (V_{BAT}), as illustrated in Figure 2.b. The cycle repeats as the controller pulls the HIGH SIDE CONTROL high again. In this manner, switch HS 1 can be cycled ON and OFF.

Extra drivers can be added in front of the FETs to supply the required gate current in cases where the size of the main FETs and the desired turn ON/turn OFF time require it. Because the capacitor (C) supplies the gate current, it must be correctly sized.

Boost Converter

Figure 3 illustrates the use of a boost converter. A boost converter accepts the battery voltage (V_{BAT}) as its input and produces an output voltage (V_{BOOST}) higher than V_{BAT}. This voltage is then used to drive the high-side switch.

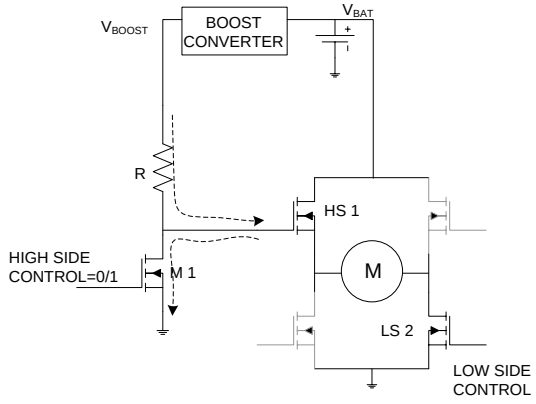
When HIGH SIDE CONTROL is high (1), the FET M1 is ON. This condition pulls the gate of HS 1 to ground and turns it off. When HIGH SIDE CONTROL is low, the voltage V_{BOOST} raises the gate of HS 1 and turns it ON.

The requirement to use additional components to monitor and control the boost means that the boost converter method is used infrequently. However, this method has two advantages over the capacitive pump method:

- The boost converter does not require replication for each high side switch. This is an advantage in applications which drive several motors such as an HVAC.
- The capacitive method can be used only when the high side switch needs a PWM. The capacitor C must be periodically charged so that it can supply the gate.

The configurable analog and digital peripherals in PSoC 3 allow us to build the boost control circuit inside the device. This eliminates the need for external control components

Figure 3. Boost Converter Based High-Side Gate Drive



and makes the boost converter method particularly easy to implement.

Nevertheless, the fault detection and protection methods are not limited to any particular implementation of high side gate drive, and we demonstrate the functionality of the system with both the capacitive pump and the boost converter. We start with a discussion of the common faults that occur on an H bridge based motor drive.

Common Faults in H Bridge Based Motor Drives

H bridges are commonly used in motor drives as well as many other applications. As such, the faults discussed in this section are not limited to motor drives. The major faults found with motor drives involve the switching elements (e.g. MOSFETs, IGBTs, and so on). Each switching element has its Safe Operating Area (SOA), and faults lead to operations outside the SOA. Operations outside the SOA reduce the life of the switches significantly and may lead to hazardous conditions.

The SOA is estimated based on the junction temperature of the switch. The junction temperature depends on the ambient temperature, the power dissipation across the switch, and the thermal resistance of the junction. Device datasheets generally present a graphical representation of the SOA and mandate that the operation be restricted to the SOA. Figure 4 shows a typical example of the SOA for a MOSFET switch, as presented in the datasheet.

The figure shows that the SOA is governed by the drain to source voltage, the drain current and the duration of operation. Generally, the voltage tolerances of the switches are an order of magnitude higher than the voltages applied in practice. Also, when the switch is conducting, the voltage across the switch is close to zero. The major parameter affecting the safe operation of the switch is the current through the switch.

Current is generally measured using a low side or a high side shunt resistor (sometimes both), which converts the current to a voltage signal. This voltage can then be compared against a threshold for detection of over or under current, or measured otherwise (e.g. using and ADC). With the availability of continuously configurable analog and digital hardware in PSoC 3, it becomes easy to design a configurable current monitoring and limiting system, which ensures that the switching elements operate in their SOA.

It should be noted that the SOA is specified for a particular ambient temperature (25 ºC in the figure). For different ambient temperatures, the allowable period of operations become different (they decrease with increasing ambient temperature). Therefore, it is useful to be able to change the allowable ON-time with changing temperature. Most smart drivers set this time using an external component such as a resistor or a capacitor and therefore lack the ability to change it according to temperature.

Different conditions may lead to excessive current through the switches. One of the reasons may be excessive load on the motor (a jammed motor for example). Motor resistance limits the maximum amount of current from a large load and is generally specified as the blocked rotor current for a given voltage by the motor manufacturer. Under certain circumstances, much larger currents may flow. If a motor terminal is shorted to battery or ground and a corresponding switch is turned on, hazardous shorts may be created. Similar circumstances may arise due to a shorted motor. Figure 5 shows a few examples, where the highlighted switch is turned on for a pre-specified time as part of normal operation (e.g. motor drive, active braking and so on).

Such conditions, if prolonged beyond safe limits, not only lead to operations outside the SOA, but also to dangerous conditions. It is necessary to detect and act upon such conditions immediately; it is also advantageous to be able to prevent such occurrences by periodically monitoring the H bridges when the motor is not in operation.

An open load does not lead to operation outside the SOA of the switches, as no current flows through the switches. Therefore, this fault is less critical to the electrical safety of the system. However, it leads to a functional fault (i.e. the motor can no longer perform the task it is assigned) and may be critical (e.g. in a wiper motor or door lock motor). When we account for functional safety, this is also outside the safe operating area in an extended sense. Therefore, the load also must be monitored periodically to guarantee

Figure 4. Example of the Safe Operating Area (SOA) of a MOSFET Switch

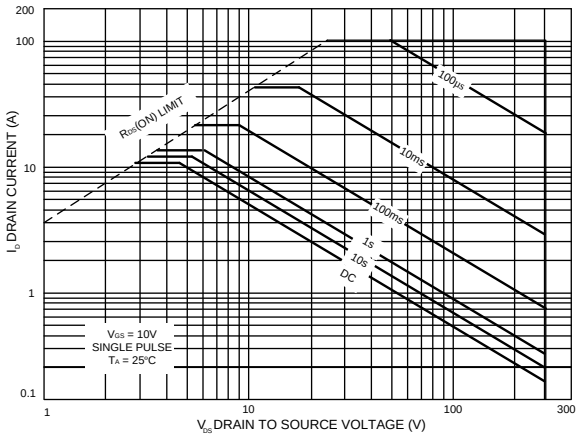
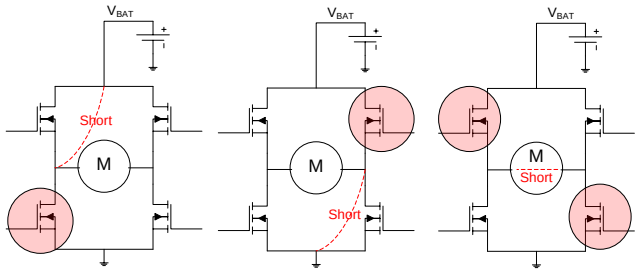


Figure 5. Hazardous Short Conditions on an H Bridge



functionality.

In this application note, we consider the following faults:

- Over Current
- Shorts to Battery
- Shorts to Ground
- Load Open

Because a shorted motor can be treated as a simultaneous short battery and ground, it is covered by this fault list. In the next section, we discuss the effects of the faults on electrical parameters of generic H bridge based motor drives. These parameters indicate operation outside SOA and are important for a fault detection and protection system.

Operation of H Bridge Based Motor Drives Outside the Safe Operating Area

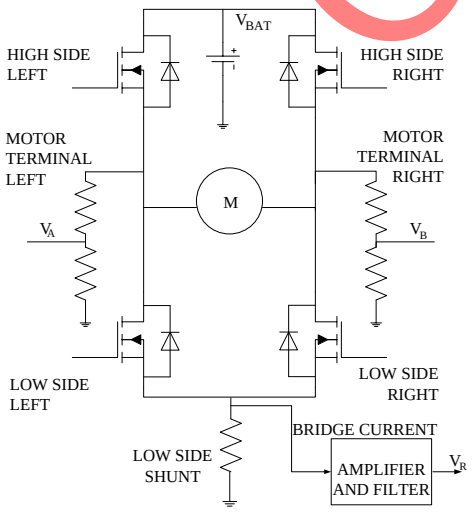
Figure 6 illustrates a generic H bridge based motor drive. The two main operation modes during which the motor spins are:

- The High Side Left and the Low Side Right switches are ON; the other two switches are OFF.
- The High Side Right and the Low Side Left switches are ON; the other two switches are OFF.

In most cases, one of the switches is not continuously ON. It is pulse width modulated for better control of the amount of current through the motor, and subsequently its speed.

The low side shunt is used to measure the instantaneous current through the motor when one high side and one low side switches are conducting. The voltage V_R is indicative of the bridge current. If one switch is suddenly turned OFF, the current through the motor diverts through one of the diodes across the switches until it drops to zero. This may or may not be registered by the low side shunt since the current may not be flowing through it, or it may reverse its direction.

Figure 6. Generic H Bridge Based Motor Drive with Associated Electrical Parameters



The fault detection system monitors the Motor Terminal Left and Motor Terminal Right nodes. The state of the bridge can be inferred from the voltages V_A and V_B. Resistor dividers are generally used to reduce high voltages close to V_{BAT} (12 to 48 V) to a range that can be read by the electronic control and protection system (for example a microcontroller) which generally operates at a lower voltage (3 to 5 V).

Under normal operating conditions (a spinning motor), V_R is limited to a finite range. The minimum current through an unloaded motor sets the lower bound. The system designer sets the upper bound as deemed safe for a maximum allowable load. A measurement of V_R less than the lower bound indicates an open load ($V_{R\ (LOAD\ OPEN)}$). A measurement larger than the upper bound indicates over current ($V_{R\ (OVER\ CURRENT)}$).

Similarly, the signals V_A and V_B also have a normal operating range. For example, with the High Side Left FET turned ON, V_A is pulled up close to V_{BAT}. The minimum value of V_A is determined by the maximum drop across the FET, which in turn is determined by the maximum allowable current. The maximum value of V_A is actually V_{BAT} (assuming zero drop across the FET). These two voltages can be termed as $V_{A\ (NO\ LOAD)}$ and $V_{A\ (FULL\ LOAD)}$. With the Low Side right FET turned ON, V_B has a maximum value of $V_{B\ (FULL\ LOAD)}$ and a minimum value of $V_{B\ (NO\ LOAD)}$.

The definition of a faulty or hazardous condition is a state where these signals are outside normal operating ranges. Table 1 lists the various possibilities, with High Side left switch turned ON (corresponding Low Side left switch turned OFF) and Low Side right switch turned ON (corresponding High Side right switch turned OFF).

The configuration shown in Table 1 can detect the MOTOR TERMINAL LEFT shorted to ground and MOTOR TERMINAL RIGHT shorted to V_{BAT} faults. The complimentary configuration (High Side Right ON and Low Side Left ON) can detect MOTOR TERMINAL LEFT shorted to V_{BAT} and MOTOR TERMINAL RIGHT shorted to ground.

Table 1. Monitor Signal States for Normal Operations and Fault Conditions

Case	High Side Left	High Side Right	Low Side Left	Low Side Right	VA	VB	VR	Inference
1	1	0	0	1	$V_A \geq V_{A\ (FULL\ LOAD)}$	$V_B \leq V_{B\ (FULL\ LOAD)}$	$V_{R\ (LOAD\ OPEN)} \leq V_R \leq V_{R\ (OVER\ CURRENT)}$	Normal Operation/ (V_A shorted to battery and V_B shorted to ground cannot be detected)
2	1	0	0	1	$V_A \leq V_{A\ (FULL\ LOAD)}$	$V_B \geq V_{B\ (FULL\ LOAD)}$	$V_R > V_{R\ (OVER\ CURRENT)}$	Over current condition
3	1	0	0	1	$V_A << V_{A\ (FULL\ LOAD)}$	$V_B \leq V_{B\ (FULL\ LOAD)}$	X	V_A possibly shorted to ground
4	1	0	0	1	$V_A \geq V_{A\ (NO\ LOAD)}$	$V_B >> V_{B\ (FULL\ LOAD)}$	X	V_B possibly shorted to battery
5	1	0	0	1	$V_A << V_{A\ (FULL\ LOAD)}$	$V_B >> V_{B\ (FULL\ LOAD)}$	$V_R > V_{R\ (OVER\ CURRENT)}$	The load is shorted
6	1	0	0	1	$V_A \geq V_{A\ (NO\ LOAD)}$	$V_B \leq V_{B\ (NO\ LOAD)}$	$V_R < V_{R\ (LOAD\ OPEN)}$	The load is open
Similar conclusions can be drawn about the configuration High Side Left=0, High Side Right=1, Low Side Left=1 and Low Side Right=0								

Henceforth, for ease of reading, we often abbreviate the names of the switches by the initials (High Side Left = HSL and so on). A similar table can be prepared for complimentary operation (HSL = 0, LSL =1, HSR =1 and LSR = 0).

Over Current Condition: If the voltages V_A and V_B are within normal operating range, then excess V_R can be due to over current conditions due to load malfunction, switch malfunction, or excessive load. Case 2 in Table 1illustrates an example.

MOTOR TERMINAL LEFT Shorted to Ground: In this case, the voltage at V_A is much lower than its lower limit of normal operation; in fact it is very close to ground. Thus, if HSL is ON and V_A is much lower than the threshold, then V_A can be assumed to be shorted to ground. Case 3 in Table 1 illustrates an example.

MOTOR TERMINAL RIGHT Shorted to Battery: In this case, the voltage at V_B is not close to zero, in spite of LSR being turned on. This indicates a short to battery. In such cases, heavy current flows through the sense resistor, a condition which provides additional identification. Case 4 in Table 1 illustrates an example.

Load is Shorted: A shorted load has its two terminals connected through a wire or a very low resistance. Both V_A and V_B are close to the mid-range voltage, and heavy current flows through the sense resistor showing large V_R. Case 5 in Table 1 illustrates an example

Load is Open: In this case, the voltages V_A, V_B and V_R are all outside normal operating ranges. Case 6 in Table 1 illustrates example.

The battery voltage may drop due to excessive current during a short condition. Because voltages V_A and V_B are proportional to the battery voltage, their values might be inappropriately measured in these cases. The car battery is essentially a rather large capacitor, and therefore changes in battery voltage are slow. Therefore, if the fault detection is quick enough, we can avoid incorrect reading of the voltages. Table 1 demonstrates that the identification of the faults requires comparison of the three monitor signals against a large number of thresholds. Completing so many comparisons in a limited amount of time requires a large number of comparators. To use an Analog to Digital Converter, voltage comparisons and corresponding corrective actions during faults must be implemented in firmware. This may result in non-deterministic timing of operations due to uncertainties in firmware execution, especially if it contains large number of interrupts. A hardware solution is preferable.

The first corrective action for all of these fault conditions is to deactivate the bridge and to suspend the current operation; this simplifies the problem. It is often sufficient to detect the existence of a fault and to deactivate the bridge, without immediate identification of the exact fault. This action requires a reduced set of observations to detect the presence of a fault and preferably consumes fewer resources than more comprehensive fault detection methods. In the next section, we describe a method to achieve that.

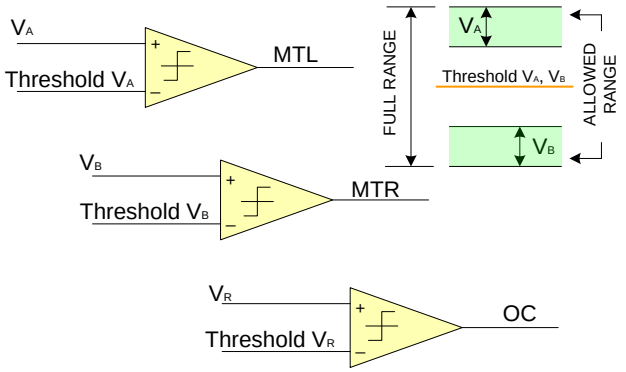
Finite Set of Valid Bridge Operations

The key to quickly determine the presence of a fault is to reduce the number of possible comparisons without sacrificing the detectability of any critical fault. To do this, we group the faults into two sets:

- Faults that affect the H bridge elements, lead to hazardous condition, and so require immediate action including: shorts to battery, shorts to ground and over current condition.
- Faults that affect functionality but do not lead to hazardous condition on the H-bridge, and so do not require immediate action such as the open load fault.

The modified detection requirements allow us to reduce the necessary comparisons to three. With a single threshold each for V_A, V_B, and V_R, we have the simple, three-comparator implementation shown in Figure 7. The typical allowable ranges of V_A and V_B for the condition are in Table 1.

Figure 7. Three Comparator Fault Condition Detection



The MTL, MTR and OC are digital signals representative of the voltages at nodes MOTOR TERMINAL LEFT, MOTOR TERMINAL RIGHT and excessive current through the bridge respectively. Table 4 depicts their interpretations. In the presence of a fault, the deviation of the voltages V_A and V_B from their allowable ranges is large; this allows us to use only a single threshold for both the voltages. A current threshold should be chosen based on the motor and the allowable load. PSoC enables us to configure the thresholds dynamically and cater to different situations. For example, we can modify the current threshold during motor start to prevent false over current detection due to high inrush current.

The motor can either be driven or stationary, as required by the application. Depending on the direction of rotation of the motor, only certain combinations of these three signals are allowable as shown in Table 2. The table shows only the ON state of the switches (i.e. the OFF periods in a PWM application are ignored).

Table 1 illustrates that certain faults are undetectable for each direction of rotation. Also, it is not possible to infer the existence of a load open fault from the digital signals when the motor is running. To circumvent this difficulty, we use a special Diagnostic Sequence (Table 3) to monitor the H bridge periodically when the motor is stationary.

The Diagnostic sequence has the following characteristics:

- A high side switch and a low side switch are not simultaneously ON at any point of the sequence.
- The sequence can successfully detect and identify the presence of one fault. If the bridge suffers from multiple faults then the sequence may not correctly identify any single fault. However, it still detects at least one fault.

The Diagnostic Sequence can both detect and identify a fault. As a high side and a low side switch are never ON at the same time, there should be no current through the motor. This enables us to run this sequence whenever required without driving the motor.

Although not explicitly shown in the table, there is a state with all switches OFF inserted between two consecutive

Table 2. Allowable States of the Digital Signals Based on the Direction of Motion

Signal	Direction 1	Direction 2
HSL	1	0
HSR	0	1
LSL	0	1
LSR	1	0
MTL	1	0
MTR	0	1
OC	0	0

Table 3. Diagnostic Sequence

HSL	HSR	LSL	LSR	Target Fault	Expected state of MTL and MTR
0	0	0	0	Short to V_{BAT}	MTL=0 MTR=0
1	0	0	0	Short to GND	MTL=1 MTR=X
0	0	1	1	Neutralize bridge	MTL=X MTR=X
0	1	0	0	Short to GND	MTL=X MTR =1
0	0	1	1	Neutralize bridge	MTL=X MTR=X
1	0	0	0	Load open	MTL=1 MTR=1

Table 4. Interpretation of Digital Signals

Digital Signal	Set	Indicates
MTL	1	MOTOR TERMINAL LEFT closer to the battery voltage than to ground
MTL	0	MOTOR TERMINAL LEFT closer to ground than to battery voltage
MTR	1	MOTOR TERMINAL RIGHT closer to the battery voltage than to ground
MTR	0	MOTOR TERMINAL RIGHT closer to ground than to battery voltage
OC	1	Current through the bridge higher than allowed
OC	0	Current through the bridge within safe limits

steps. This ensures that an active switch is first turned OFF before activating a different switch. If there are no faults on the H bridge or the motor, both the signals MTL and MTR have the same polarity because the motor resistance is only a few ohms. When a high side switch is turned ON, the motor terminals are close to V_{BAT}, and hence MTL and MTR are both HIGH. When a low side switch is turned ON, they are both LOW. When all switches are turned OFF, the two motor terminals would start going LOW, as the resistor dividers on both sides will pull them down (Figure 6). Therefore, with all switches OFF, both MTL and MTR are LOW. Any deviations from

the behaviors mentioned in the table are indicative of faults. Motor terminals may have large capacitances, leading to slow transitions of MTL and MTR from HIGH to

LOW. The bridge is neutralized actively between high side switch activations to bring both the signals LOW. The sequence can be run at high speeds; in the associated project the total sequence takes 200 µs. The designer can decide how frequently the sequence needs to run based on the application. In the associated project, it is run every 2 seconds on a stationary motor.

Table 2 and Table 3 demonstrate how we can interpret a particular combination of digital signals as a fault condition. Only a few of the combinations correspond to a faultless bridge and motor. This enables us use only the digital signals to encode the state of a bridge and allows us to define the Bridge Status Word (BSW) described in Table 6.

The BSW is a snapshot of the voltage and current conditions of the H bridge converted to a single byte. It should be sampled at proper instants to obtain meaningful information. Switches take finite time to turn ON or OFF, and hence the BSW should be sampled only when switches are in defined states and not in transition. A correctly sampled BSW directly maps to either a normal operating condition or a faulty state using a lookup table; this makes fault identification straightforward. It also allows the system to "log" the bridge activity through a sequence of bytes. The system firmware may periodically store the BSW in a circular buffer, maintaining a history for a predetermined amount of time. Table 5 shows a few BSWs and their interpretation. It is by no means an exhaustive list.

If any fault condition is detected when the motor is running, then the motor is stopped and a diagnostic is performed on the motor. It is also performed periodically on stationary motors as a fault discovery mechanism.

From the above discussion, it is clear that the system needs to monitor the control signals (HSL, HSR, LSL and LSR), as well as the status signals (MTL, MTR and OC). If it detects a fault, the system communicates appropriate status to the master. It is advantageous for the system to generate the control signals, as it can take the protective action itself, rather than relying on the master to do so.

Table 5. Select BSWs and Their Interpretation

BSW	Meaning
0x4C	The motor is running with HSL and LSR turned ON.
0x4D	The motor is running with HSL and LSR turned ON. There is over current.
0x48	The motor is running with HSL and LSR turned ON. There may be a possible short to ground.
0x4E	The motor is running with HSL and LSR turned ON. There may be a possible short to battery.
0x4F	Same as above.
0x80	The motor is stationary. All switches are OFF. Bridge is in expected condition.
0x86	The motor is stationary. All switches are OFF. There is a short to battery.
0xC6	The motor is OFF. HSL is turned ON. Motor is OK
0xC4	The motor is OFF. HSL is turned ON. Motor is open.
0xC0	The motor is OFF. HSL is turned ON. There is a short to ground.

This reduces the load on the master. In the next section, we present a high level architecture of a system which can:

- Communicate with a master and configure itself as requested
- Generate the control signals as per request from a master
- Generate the status signals, and the BSW
- Process the BSW to analyze and report faults

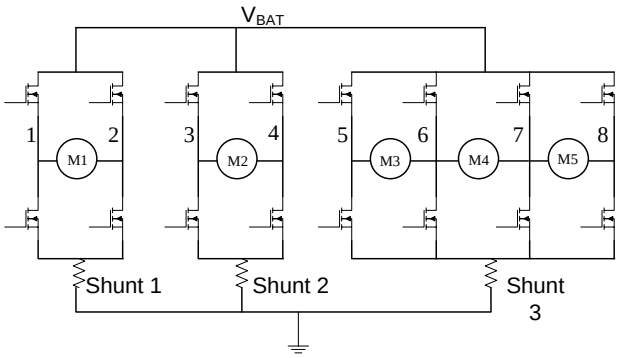
Table 6. The Bridge Status Word

Bit Position	7	6	5	4	3	2	1	0
Bit Name	Mode	HSL	HSR	LSL	LSR	MTL	MTR	OC
Description	0 = The motor is being driven. 1 = The motor is stationary	The gate of the High Side Left transistor, depicts whether it is ON (1) or OFF (0).	The gate of the High Side Right transistor, depicts whether it is ON (1) or OFF (0).	The gate of the Low Side Left transistor, depicts whether it is ON (1) or OFF (0).	The gate of the Low Side Right transistor, depicts whether it is ON (1) or OFF (0).	Depicts whether MOTOR TERMINAL LEFT is pulled to V_{BAT} (1) or to ground (0)	Depicts whether MOTOR TERMINAL RIGHT is pulled to V_{BAT} (1) or to ground (0)	Depicts Over Current condition. 1 = Over Current, 0 = Normal condition

High Level Architecture of Fault Detection and Protection System

The availability of a large number of analog and digital resources in the PSoC 3 allows one to easily design the fault detection and protection system. A good architecture

Figure 8. Example Arrangements of H Bridge and Motors



is scalable, i.e. it allows the easy addition of a new H bridge or a new motor. Also, a good architecture minimizes resource usage to enable the implementation of additional features if required.

The physical connection of the motors and the bridges significantly influences architectural choices. In many applications, H bridges are shared across motors. This is often the case where the motors need not run both simultaneously and independently as in the example shown in Figure 8.

Motors M1 and M2 can run simultaneously and independently with any speed and direction, but no two motors among M3, M4 and M5 can be driven independently at the same time. *Isolated* motors are those where there are no connections with all the switches off between the motors except through the battery or ground. In Figure 8, M1 and M2 are isolated as are M2 and M4. *Connected* motors are all motors which are not isolated. M3, M4 and M5 are connected and belong to a *connected group*.

The fault detection and protection system should work for both isolated and connected motors. Each motor can be uniquely identified by the two half bridges to which it connect. Therefore, it is more useful to allocate resource for each half-bridge rather than each motor, because half-bridges are shared in connected groups. Such an allocation requires a mechanism for independent access of each individual half-bridge, so that the motors can be independently accessed irrespective of the way they connect.

The next major consideration is the ability to share resources across multiple motors. The shared resource paradigm is essentially based on time multiplexing. Figure 9 compares the shared resource paradigm against the "unfolded" or dedicated resource paradigm.

The figure shows the inputs (BSW), the functional unit f1 required for the computation, the sharing mechanism (S and S'), and the outputs. The unfolded mechanism requires more functional units but requires no sharing mechanism.

The choice between the two is influenced by the following factors:

- Number of inputs N
- Resource requirement or cost of the functional unit
- Resource requirement or cost of the sharing mechanism
- Required rate of outputs

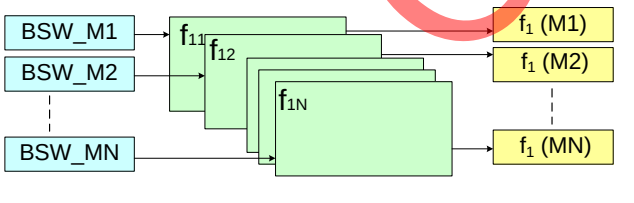
The major determinant of the architecture is the output update rate. If new outputs are required every *T* unit time, and the functional unit f_1 requires f_T unit time to compute the output for every new input, then the maximum number of inputs which a single functional unit can process is:

$$\left\lfloor T / f_T \right\rfloor$$

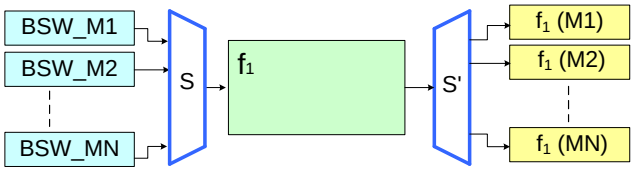
where $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. This calculation neglects any delay introduced by the sharing mechanism. This delay may become significant with a large number of inputs.

The next factor which affects the architecture is the total cost of the system. The cost of the functional units in both architectures scales linearly with the number of inputs. However, the time delay introduced by the sharing mechanism scales logarithmically with the number of inputs. Often, designers select a mixed architecture where a functional unit is allocated for every m inputs out of a total of N inputs.

Figure 9. Dedicated vs. Shared Resource Paradigm



UNFOLDED



SHARED

One can further optimize the architecture with knowledge of connected groups. As motors in a connected group do not operate simultaneously and independently, they may not require dedicated resources.

Calculation of the optimum cost based on a given motor connection and a required output update rate is beyond the scope of this application note. It is important to carefully consider this issue when choosing a particular connection.

We selected the architecture used in this application note with consideration for the reusability of blocks, scalability and modularity of functions. Resources are shared wherever possible. Figure 10 depicts the architecture of our example. It consists of six major blocks:

- Bridge Interface
- Over Current
- Motor Drive Signals generation
- Fault Detect
- Diagnostic
- Firmware

These six blocks, along with the user interface, make up the complete system. The system designer can configure these blocks using APIs.

The Bridge Interface block interfaces with the physical H bridge. It configures the device pins correctly so as to transmit the control signals from the device to the bridge and the status signals from the bridge to the device. The Bridge Interface block consists of multiple half-bridge interface blocks. Each of these half-bridge interface blocks receives the H_BRIDGE DRIVE SIGNALS from the MOTOR DRIVE SIGNALS generation system or control signals from the Diagnostic block. The Fault Detect block observes the control signals incident on the bridge, and the resultant status signal from the bridge. In case of an abnormal condition, an invalid BSW is detected. The Fault

Figure 10. Example Operation of a Given Motor Configuration with the High Level Architecture

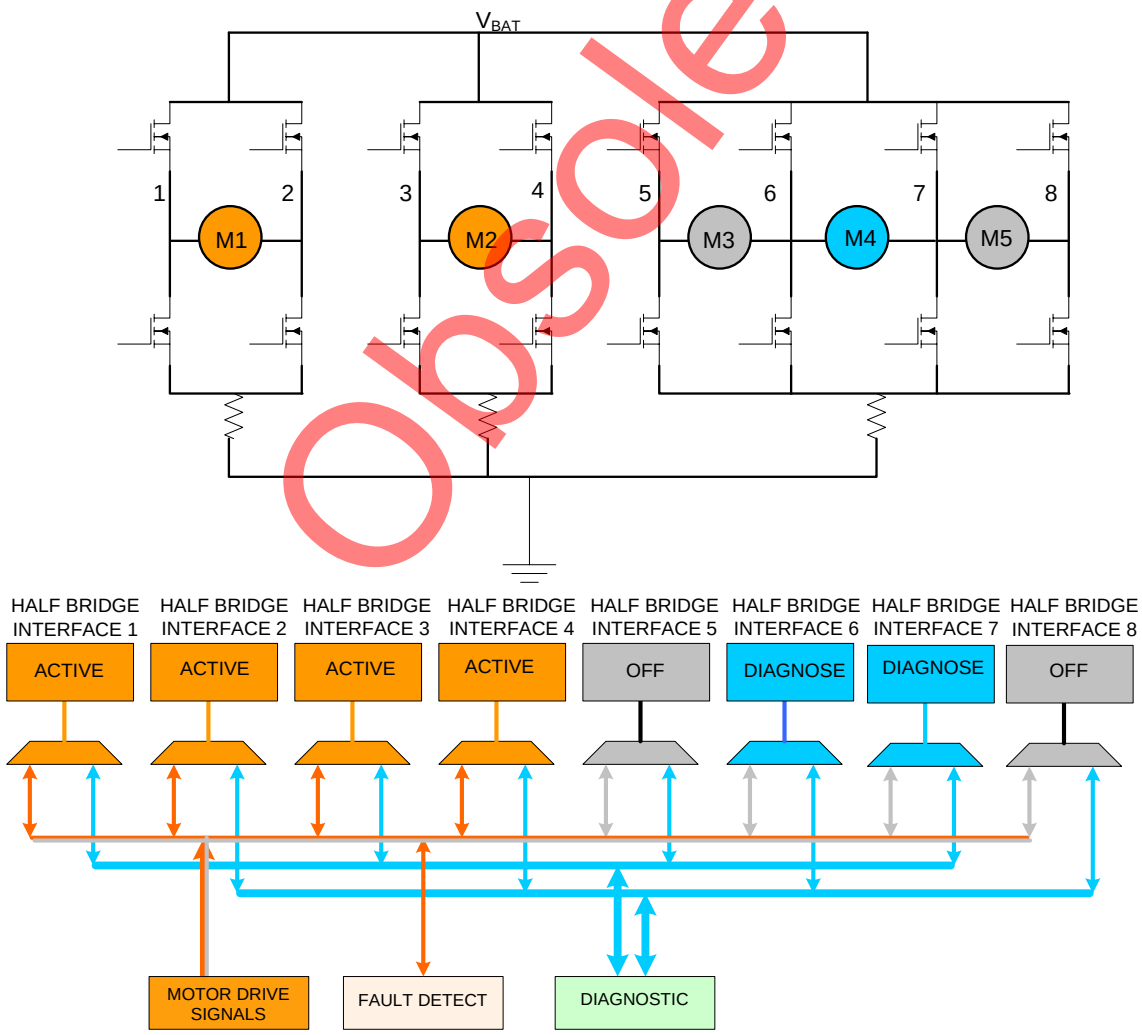


Table 7. Half-Bridge Interface Block Signals and Their Significance

Signal Name	Signal Type	Significance
Enable	Digital Input	The signal that controls the half-bridge drive signals. If this signal is 0, then the PIN_HS and PIN_LS disable the gate drivers, thus deactivating both the high side and the low side switches.
HS_Ctrl	Digital Input	The signal that controls the high side switch through PIN_HS. If the bridge is enabled and HS_Ctrl is high, the switch is turned ON; otherwise the switch is turned OFF.
LS_Ctrl	Digital Input	The signal that controls the low side switch through PIN_LS. If the bridge is enabled and LS_Ctrl is high, the switch is turned ON; otherwise the switch is turned OFF.
PIN_HS	Digital Output	The pin that controls the high side gate driver.
PIN_LS	Digital Output	The pin that controls the low side gate driver.
Bridge	Digital Output	The digital state produced by comparing V_Bridge and Threshold, as seen in Figure 11.
PIN_BRIDGE	Analog Input	An input pin used to connect to the V_Bridge signal.
Threshold	Analog Input	The signal used as the reference for digitizing the V_Bridge signal, as seen in Figure 11.
PIN_ANALOG	Analog Input	An optional pin used to bring an analog signal into the system. In the present application, used monitor the current through the bridge.
Analog Signal	Analog Output	An optional signal which might route into the system. It may be any analog signal deemed necessary to be monitored. In the present application, used for over current detection and hence connected to the low side shunt signal.

Detect block identifies this as a fault, and overrides the control signals to deactivate the bridge. The Fault Detect block can observe and alter the H_BRIDGE DRIVE SIGNALS as depicted in the diagram by the bidirectional lines between the block and the signals.

Figure 10 demonstrates an example of the architecture in operation. The Motor Drive Signals block generates signals for all motors which route to the half-bridge interface blocks. The firmware configures the routing so that each half-bridge interface block receives the correct set of signals. The figure illustrates a case where motors M1 and M2 are running, M3 and M5 are off, and M4 is being diagnosed. The color coding shows the type of signals received by each half-bridge and each motor. The Over Current block is not shown in the figure. In this scenario, it monitors motors M1 and M2.

Implementation of the Fault Detection and Protection System

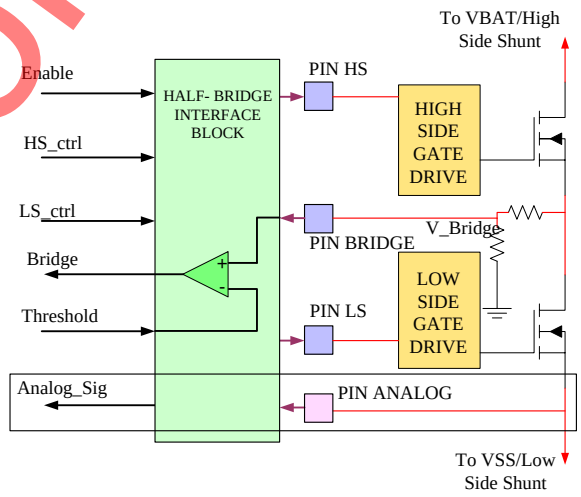
We used PSoC 3 to implement an example fault detection and protection system based on the architecture discussed in the previous section. For an introduction to design with PSoC 3, please see AN54181. We designed the system to drive, monitor and protect two isolated motors.

The Half Bridge Interface Block

Conceptual Realization

This block interfaces the system with the physical H bridge and the motors. Figure 11 illustrates the conceptual realization of the block. Table 7 lists the inputs and outputs of the Half-Bridge Interface block along with their significance.

Figure 11. Conceptual Realization of the Half-Bridge Interface Block

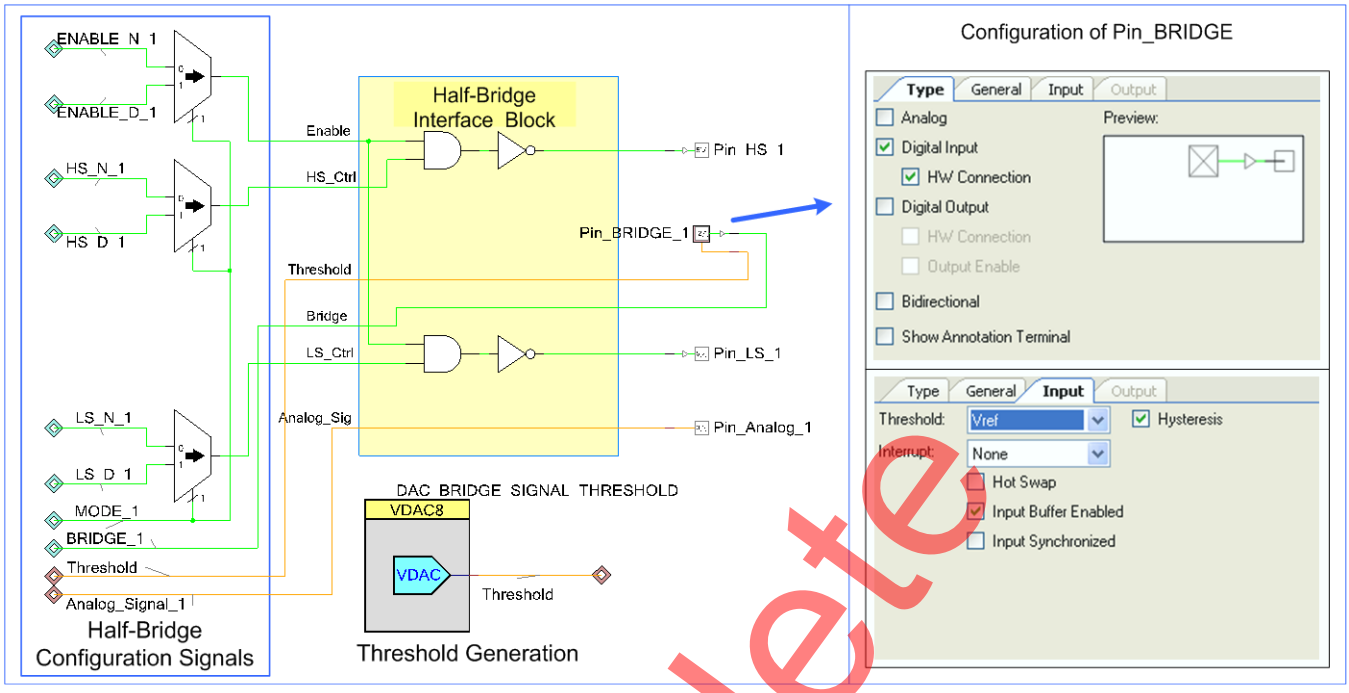


We discussed two ways to drive a high side gate in the section Alternate Methods to Drive H Bridges. The V_Bridge is the resistor divided voltage at the middle of the half-bridge, i.e. at the motor terminal. In this application note, PIN ANALOG interfaces to the low side shunt, but you may choose to connect it to the high side shunt or to any other analog signal deemed necessary.

PSoC 3 Implementation

Figure 12 shows the PSoC3 implementation of the Half-Bridge Interface block and the associated configuration signals. We can drive the digital control signals (Enable, HS_Ctrl, and LS_Ctrl) with either the normal mode signals, marked by _N, or the diagnostic mode signals, marked by _D. The project uses the signal MODE to select the

Figure 12. PSoC 3 Implementation of the Half Bridge Interface Block



correct set of signals that go into the half-bridge. The project uses four instances of the Half-Bridge Interface block.

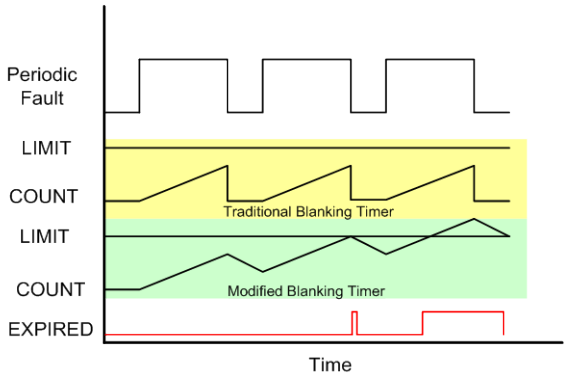
The HS_Ctrl and LS_Ctrl signals are logically AND'd with the Enable signal, so that they are ignored when Enable is OFF. The NOT gates are used before the pins, because the gate drivers are active low (i.e. they turn the switches ON when their inputs are LOW).

If required, you can insert an analog buffer between the Pin_Analog and the Analog signal to prevent loading of the signal. The availability of analog buffers on PSoC 3 makes this easy to implement.

The Serial IO (SIO) pins available in PSoC 3 allow us to implement the Pin_BRIDGE and the comparator shown in Figure 11. The SIO pin is configured as a digital input pin with hardware connection, and its Threshold is chosen to be V_{REF}. This effectively configures the pin as a comparator, with the output as the digital state read by the pin, and the V_{REF} signal as the reference. A digital to analog converter generates this reference (the Threshold signal) to allow easy modification of the threshold. Alternately, it can be driven by another analog signal directly derived from the battery. For details on the SIO implementation, please consult the pin component datasheet and AN60580.

The block does not interact with firmware, and it does not have any APIs. The Over Current, Fault Detect and Diagnostic blocks use signals from this block.

Figure 13. Behavioral Differences Between Traditional and Modified Blanking Timers



Over Current, Fault Detect and Diagnostic Blocks

The Over Current, Fault Detect and Diagnostic blocks are each divided into a few major parts:

- **Fault Signal Detection Block**: Detects the actual fault. The implementation varies according to the type of fault being identified.
- **Blanking Timer**: Verifies that the detected fault is a true fault and not misidentified noise. This is common to all three block types.
- **Fault Latch**: Ensures that once a fault is verified by the Blanking Timer, it is latched and cannot be cleared even if the fault condition goes away. The latch must be cleared in firmware and requires master intervention to ensure that the master acknowledges all faults.

- **Status Register**: Records the fault signals at the time of the occurrence of any fault. The firmware reads the status register to determine the source of the fault.
- **Control Register**: Interfaces to the firmware.
- **Application Programming Interface**

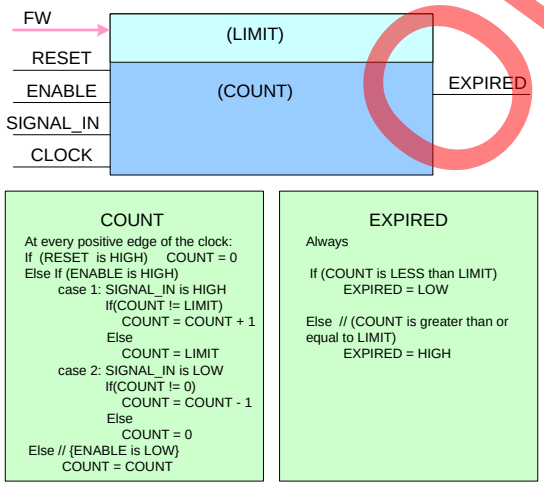
Blanking Timer

The Blanking Timer “blanks” or becomes zero from some non-zero starting value. Certain “events” trigger the timer. Once triggered, it starts to count down. If it “blanks” or reaches zero, then it may trigger an action. However, if it is interrupted (the triggering event does not persist, or otherwise), then it returns to the initial count and maintains the count until triggered again. A Blanking Timer ensures that a decision is made based on the occurrence of a true event, as against temporary disturbance.

In this application note, we modified the behavior of the Blanking Timer to best suit the needs of the application. We maintain the name as the essential purpose is the same. Without loss of generality, we assume that both the modified and the traditional timers start at 0 (instead of a non-zero starting value) and count up to a pre-specified limit.

Figure 14 describes the behavior of the modified timer. The input signals RESET, ENABLE, SIGNAL_IN and CLOCK control the timer. The output signal EXPIRED indicates a fault. The firmware sets the internal parameter LIMIT. This parameter determines the “blanking time”, or the period for which the event needs to persist in order to be acknowledged as a true event. The figure shows how inputs affect the variable COUNT. It also shows how COUNT, in turn, controls the output.

Figure 14. Functional Realization of the Blanking Timer



The major difference between the present realization and a traditional Blanking Timer is the symmetric movement of the internal variable COUNT between its two terminal values (0 and LIMIT). Figure 13 illustrates this point.

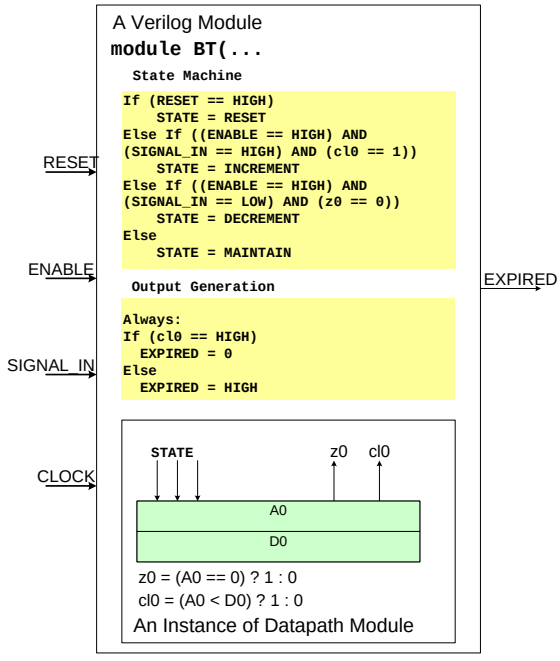
The COUNT variable of the modified timer is not reset to 0 as soon as the Fault signal disappears; rather, it smoothly transitions from its present value to 0. Therefore, the decision about the occurrence of the fault is not based on its continuous persistence over a period of time, but by its “average” persistence. The average persistence is better suited to the present application, as faults, though not continuously present, reduce the reliability of the H bridge and the load as a whole.

For example, imagine a case where a switch undergoes periodic fault conditions but with the period being less than LIMIT. With a traditional blanking timer, the temperature might increase slowly and eventually put the device outside its SOA. However, the modified timer expires if the fault persists more often than not. The disadvantage of the modified approach is its reduced robustness to noise as compared to the traditional realization. If noise has high average persistence then it may be recognized as a true fault.

We implement the blanking timer with datapaths. Datapaths are hardware modules available in the Universal Digital Blocks (UDB) in PSoC 3. Each UDB contains one datapath element. PSoC Creator allows these modules to be instantiated inside a user defined Verilog module, as shown in Figure 15.

The datapath consists of several registers. We discuss only the registers relevant to the implementation, A0 and D0. The Datapath Configuration Tool allows eight possible operations of the datapath. At a given clock cycle, the value of STATE determines the particular operation to be performed. For more detailed information on Verilog and datapath based components, please refer to these

Figure 15. Datapath Based Implementation of the Blanking Timer



trainings (Verilog, Datapath).

The datapath operations used in this application note are:

- RESET: Loads the register A0 with value 0.
- INCREMENT: Increases the value of A0 by 1.
- DECREMENT: Decreases the value of A0 by 1.
- MAINTAIN: Maintains the value of A0.

The implementation uses two outputs from the datapath: z0 and cl0. Their significances are as follows:

- z0 = 1 if A0 is equal to 0; it is zero otherwise.
- cl0 = 1 if A0 is less than D0; it is zero otherwise.

Using these two values and the input signals to the Blanking Timer, a state machine is designed in Verilog which controls the datapath operation at each clock cycle. The A0 register essentially acts as the COUNT variable and the DO register acts as the LIMIT parameter. The D0 register can be written in firmware, allowing us to configure the Blanking Time.

All three fault detection blocks use a blanking timer. The Over Current, Fault Detect, and Diagnostic blocks use their associated APIs to configure their own blanking timers. There is no API particular to the blanking timers.

Over Current Block

The Over Current block monitors the current through the bridge, and takes appropriate actions when the current exceeds the set threshold for a significant amount of time. A low side shunt converts the current into voltage, and hence the over current block actually monitors voltage.

Conceptual Realization

As shown in Figure 16, the Over Current block consists of three parts: the signal conditioning portion, the thresholding portion and the blanking timer.

Depending on the nature of the signal and the requirements, the analog signal from the Half-Bridge Interface block may need amplification and . It may also require differential amplifiers depending on the noise levels. The implementation in this application note uses single ended amplification.

The system compares the processed against a set threshold. The result is a digital signal whose state depends on the analog signal. Figure 17 shows how the Blanking Timer accumulates the digital signal accumulates over a period of time to determine the over current fault.

The system then compares the Current Signal, which is actually the output from the Signal Processing block against the over current threshold (OC threshold) to produce the comparison result (ThC). This output accumulates or integrates until it reaches another pre-set threshold. At this point the Over Current alarm is set off. The accumulation approach has three principle advantages:

Figure 16. Conceptual Realization of the Over Current Block

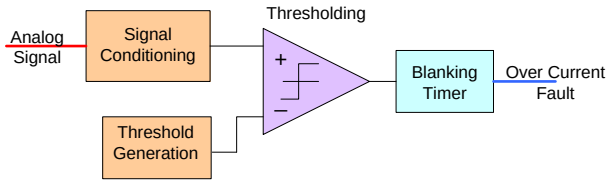
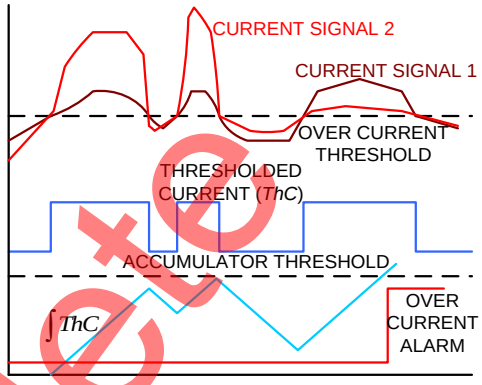


Figure 17.Over Current Alarm Generation from Thresholded Current Output



- The SOA of the switch suggests that it can tolerate a specific amount of over current for a specific amount of time based on the temperature. The accumulation approach therefore leads to better use of the switch rather than instantaneous over current.
- It eliminates effect of spurious noise.
- As the Blanking Timer does the integration in the digital domain and the result compared against a digital threshold (the blanking time), it is easy to configure the system for different H bridges and motors with different over current tolerance without using external components such as capacitors.

A larger over current threshold implies that the switch can tolerate over current for a shorter time the current without violating the SOA. Therefore, the over current threshold and the accumulator threshold are related and should be set appropriately.

As seen in Figure 17, the current signal1 and current signal 2, although indicative of different severity, produces identical effect. This is a disadvantage of the discussed approach. The current signal 2, however, is unlikely to be encountered in a practical application. A carefully chosen over current threshold, accumulator threshold, and slope of accumulation can circumvent such problems.

PSoC 3 Implementation

Ready availability of configurable analog blocks makes the PSoC 3 implementation of the Over Current block straightforward. Figure 18 shows the implementation of two Over Current blocks for two isolated motors. A Programmable Gain Amplifier (PGA) amplifies the voltage

signal which originates from the shunt resistor or from a preconditioning circuit to allow a quick gain change without change of any external components.

A voltage DAC generates the over current threshold. The firmware can modify the DAC value at runtime to change the threshold. Choose the range and value of the DAC based on the current rating of the motor and the value of the shunt resistor. As the threshold changes infrequently, a slow speed setting is sufficient.

The amplified signal goes to the positive input of a comparator, and the generated threshold to the negative input. Configure the comparator with hysteresis enabled and with non-inverting polarity (output is high when positive input is greater than negative input). Each comparator synchronizes to a clock with half the frequency of the blanking timer clock to ensure that the SIGNAL_IN input of the blanking timer is always stable at its clock edges. For more information, see the PGA, DAC and, Comparator datasheets.

The over current signals Oc_Signal_1 and Oc_Signal_2 are input to the Blanking Timer blocks. A high signal for a sufficiently long time triggers an Over Current fault. The two EXPIRED signals from the Blanking Timers are logically OR'd to produce the resultant Over Current fault. The Fault Latch latches the fault. Once latched, only firmware can clear the fault.

When a motor starts, a large amount of inrush current flows through the motor until it reaches a stable speed. An OC Threshold set for normal operating conditions will misidentify the inrush current as an over current condition. Similarly, an OC threshold set for the inrush current may result in switches operating outside their SOA. Heavy motors with large inertia are particularly prone to this problem. A soft start strategy solves this problem. The soft starting strategy works as follows:

- Define the `inrushPeriod` for each motor as the duration that inrush current is most likely to.
- During this time, set the SoftStart_OC_x (x=1,2) to HIGH. This gates the Blanking Timer output from the Fault Latch. Instead, it feeds the output back to the motor drive signal generation unit (Inrush_x) to turn the bridge OFF.
- The over current blanking time is set to the value STARTING_BLANKING_TIME which is typically much less than the normal operating conditions.

It is clear that the switches operate inside the SOA, because the threshold is the same as that for regular operating condition and the blanking time is less than the regular blanking time. The following events lead to motor start:

Figure 18. PSoC 3 Implementation of the Over Current Blocks

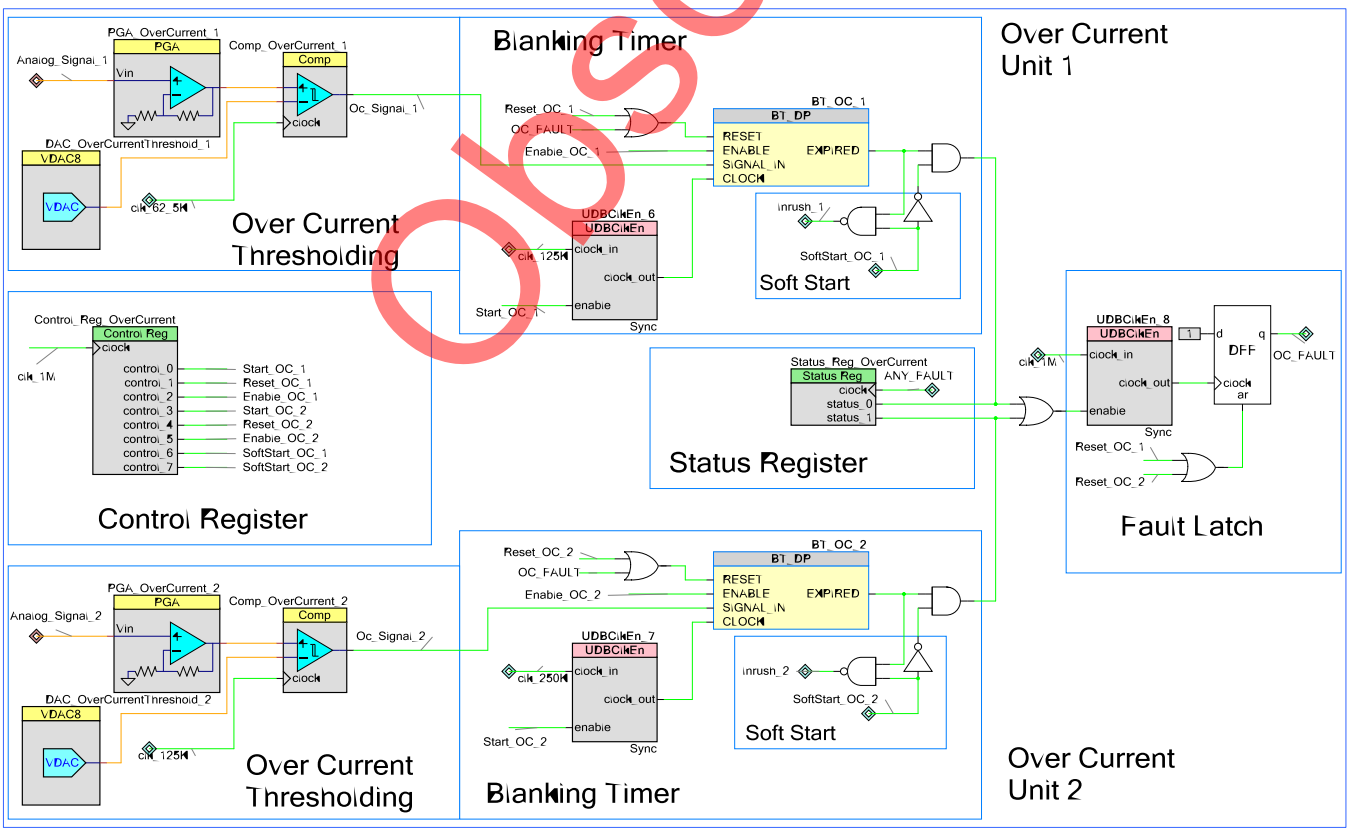


Table 8. API Functions of the Over Current Block

Function	Action
`OverCurrent_x_Start()`	Starts and configures all the analog peripherals used by the OverCurrent _x block and resets the Blanking Timer, Fault Latch and Status Register blocks.
`OverCurrent_x_Stop()`	Stops all the analog components associated with OverCurrent_x block and resets the Blanking Timer, Fault Latch, and Status Register blocks.
`OverCurrent_x_SetThershold (uint8 threshold)`	Sets the Over Current threshold by modifying the DAC value with the supplied parameter `threshold`.
`uint8 OverCurrent_x_GetThreshold()`	Returns the current threshold for OverCurrent_x block.
`OverCurrent_x_SetBlankingTime (uint8 blankingTime)`	Sets the blanking time for OverCurrent_x block by modifying the blanking timer D0 register according to the supplied parameter `blankingTime`.
`uint8 OverCurrent_x_GetBlankingTime()`	Returns the current blanking time.
`OverCurrent_x_Enable()`	Enables the OverCurrent_x block by enabling its blanking timer.
`OverCurrent_x_Disable()`	Disables the blanking timer of the OverCurrent_x block from counting.
`OverCurrent_x_SoftStartEnable()`	Enables the soft start function for OverCurrent_x block by setting the SoftStart_OC_x signal HIGH, and changing the blanking time value to STARTING_BLANKING_TIME_x.
`OverCurrent_x_SoftStartDisable()`	Disables the soft start function for OverCurrent_x block by setting the SoftStart_OC_x signal LOW, and changing the blanking time value to normal operating value.

- The current through the motor increases till it exceeds the OC threshold for the duration STARTING_BLANKING_TIME.
- The Blanking Timer expires, setting the Inrush_x signal to high. This turns the bridge OFF.
- The current through the motor decreases, and the Blanking Timer output goes LOW. This sets Inrush_x low, and the bridge turns ON again. This restarts the cycle.

In this way the motor starts without transgressing the SOA of the switches. After the `inrushPeriod`, the SoftStart_OC_x signal turns OFF, and the blanking time is set to its normal operating value.

A sudden and large increase in speed may also require large current. The user may modify the threshold and blanking time values for such events accordingly.

When the system detects any kind of fault, it sets the ANY_FAULT signal to HIGH. On the occurrence of any faults, the status register logs the Blanking Timer output.

The Over Current control register initializes and configures the Over Current blocks. It controls the passage of the clock to the blanking timers and handles their initialization and configuration as required by the application. It is also used as an interface between the firmware and the actual hardware.

The Over Current block offers a set of APIs listed in for easy configuration of the system. The `OverCurrent_x` prefix denotes that the API is particular to the Over Current unit x (x = 1 or 2).

Fault Detect Block

The Fault Detect block detects any abnormal bridge behavior while the motor is in operation.

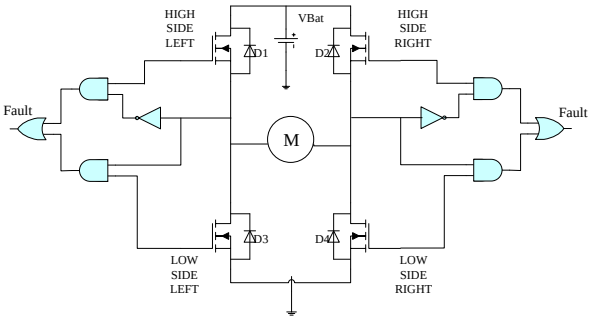
Conceptual Realization

Table 9 lists the BSWs for particular faults. For each half-bridge, we can define the fault as a logical operation on the high-side control signal, the low-side control signal, and the state of the bridge. This enables us to easily map the BSW to a specific fault condition. Figure 19 shows the logical definition for both half-bridges.

Table 9. Abnormal Conditions and Associated BSW for Fault Conditions with Active Motors

Signal 1	Signal 2	BSW(binary)	Meaning
HSL = 1	MTL = 0	01x0x0xx	High side left FET is on, but the motor left terminal is close to ground.
LSL = 1	MTL = 1	00x1x1xx	Low side left FET is on, but the motor left terminal is close to V_{BAT}
HSR = 1	MTR = 0	0x1x0x0x	High side right FET is on, but the motor right terminal is close to ground.
LSR = 1	MTR = 1	0x0x1x1x	Low side right FET is on, but the motor right terminal is close to V_{BAT}

Figure 19. Logical Definition of Bridge Fault



The logical definition in Figure 19 automatically protects against turning on the high side and the low side switch of the same half-bridge simultaneously. The availability of digital logic gates on PSoC 3 makes it particularly easy to implement the above fault detection circuit.

PSoC 3 Implementation

The heart of the Fault Detect block is the FD component. We use Verilog to implement the FD component. Figure 20 describes it behavior.

The BT CONTROL portion generates the RESET and ENABLE signals for the Blanking Timer block. The system resets the Blanking Timer every time there is a transition on any of the bridge drive signals. This ensures that the system does not incorrectly detect any intermediate condition during the transition of a switch as a fault. Also, it disables the Blanking Timer unless at least one of the switches on the H bridge is ON. The Fault Logic block is the Verilog implementation of the fault definitions in Figure 19.

Figure 20. The FD Component

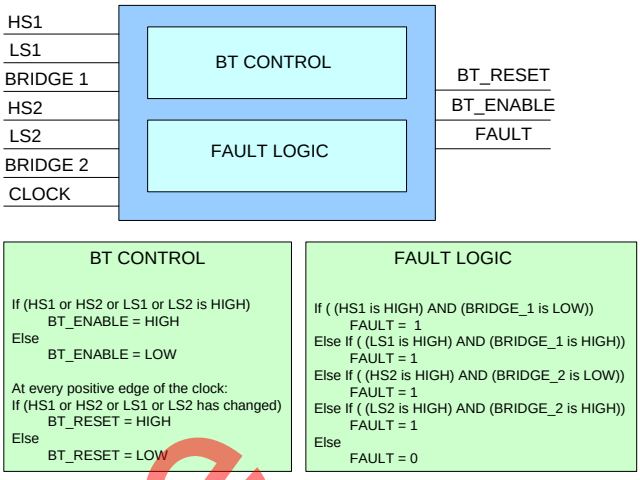


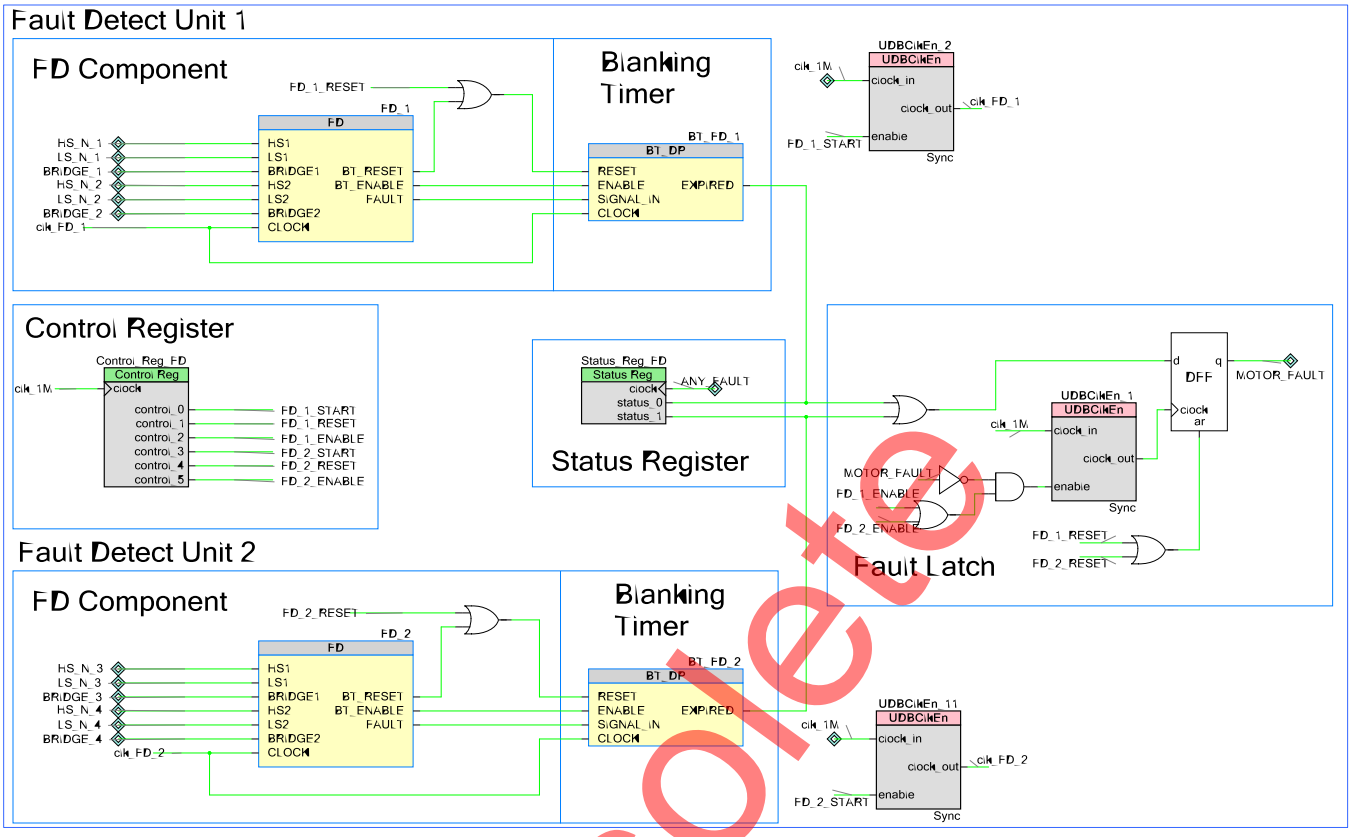
Figure 20 shows the complete implementation of two Fault Detect blocks. During soft start, the bridge is continuously in transition; therefore, the blanking timers are held in RESET during this period. The EXPIRED signals from the Blanking Timers are logically OR'd to produce the fault signal to indicate that at least one of the H bridges sustained a fault. The Fault Latch block then latches the signal. The ANY_FAULT signal logs the status of the two EXPIRED signals into the Status Register. The status is used later to analyze the source of the fault.

The Fault Detect Control Register configures the system initially and between system stalls due to any fault. The Firmware block uses this as an interface to the Fault Detect system. Table 10 lists the API functions of the Fault

Table 10. API Functions for the Fault Detect Block

Function	Action
`FaultDetect_x_Start()`	Starts the Fault Detect system by allowing the clock to FD_x component and the Blanking Timer BT_FD_x. Resets the Blanking Timer internal count to 0. Clears the Status Register and the Fault Latch flip-flop of any pre-existing fault status.
`FaultDetect_x_Stop()`	Gates the clock to the FD_x component and the Blanking Timer BT_FD_x. The Blanking Timer internal count is reset to 0. Clears the Status Register and the Fault Latch flip-flop of any pre-existing fault status.
`FaultDetect_x_Enable()`	Enables the Fault Latch functionality.
`FaultDetect_x_Disable()`	Disables the Fault Latch functionality.
`FaultDetect_x_AssertReset()`	Resets the Blanking Timer BT_FD_x, the Status Register and the Fault Latch flip flop. These are held in reset until released.
`FaultDetect_x_ReleaseReset()`	Releases the Blanking Timer BT_FD_x, the Status Register and the Fault Latch from reset condition.
`FaultDetect_x_SetBlankingTime (uint8 blankingTime)`	Configures the blanking timer BT_FD_x so that the blanking time is equal to the provided parameter `blankingTime`. The Blanking Timer and the Fault Latch are also reset.
`uint8 FaultDetect_x_GetBlankingTime()`	Returns the current blanking time of BT_FD_x.

Figure 21. PSoC3 implementation of the Fault Detection System



Detect_x (x= 1,2) block.

Diagnostic Block

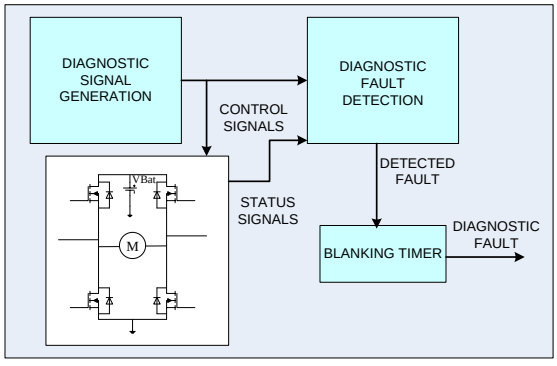
The Diagnostic block:

- Periodically monitors loads which are inactive.
- Diagnoses a bridge reported by the Over Current or the Fault Detect blocks.

Conceptual Realization

The Diagnostic block manages the sequence described in Table 3. The Diagnostic sequence excites the H bridge, and the resulting status signals combine with the sequence to form the BSW. The BSW maps to a normal condition or a particular fault. The Diagnostic system as shown in Figure 22 consists of two parts: signal generation and BSW analysis. As usual, it uses the Blanking Timer to distinguish true faults from noise.

Figure 22.Conceptual Realization of the Diagnostic Block



The Diagnostic Signal Generation component generates the control signals for two half-bridges: Half-bridge 1 and Half-bridge 2 as well as a few configuration signals for the Diagnostic Fault Detection component. For ease of discussion, we assume that Half-bridge 1 is the left half-bridge. Table 11 shows the inputs and outputs of the Diagnostic Signal Generation component. Figure 23 illustrates the corresponding timing diagram as well as the clock-cycles, with 0 being the first cycle after ENABLE is set high.

When a load is inactive, all the switches on the H bridge are OFF. The resistor dividers on the Motor Left Terminal (MLT) and Motor Right Terminal (MRT) passively pull the half-bridges down to ground. Therefore, with all switches off, MTL and MTR should both be LOW. The short-to-battery fault checks these two signals before turning ON any switches. The LV_B1 signal indicates the Diagnostic Fault Detection block to sample the MTL signal. If it is HIGH, a short-to-battery fault is reported. Similarly, the LV_B2 checks for the MTR signal. If any one

Table 11. Input and Output Signals of the Diagnostic Signal Generation Block

Signal Name	Signal Type	Signal Description
MAIN_CLOCK	INPUT	This is the input clock to the signal generation system. All output signal changes are synchronized to this clock
RESET	INPUT	If this signal is HIGH, then all the output signals are set logic low.
ABORT	INPUT	This signal is used for aborting an active sequence. It sets DONE to HIGH, and all the other outputs to LOW.
HS1	OUTPUT	This is the signal to turn ON the High Side switch for Half-Bridge 1, as described in Table 3.
HS2	OUTPUT	This is the signal to turn ON the High Side switch for Half-Bridge 2, as described in Table 3.
LS1	OUTPUT	This is the signal to turn ON one of the Low Side switch for Half-Bridge 1, as described in Table 3.
LS2	OUTPUT	This is the signal to turn ON the Low Side switch for Half-Bridge 2, as described in Table 3.
LV_B1	OUTPUT	When this signal is HIGH, the Diagnostic Fault Detection component checks Half-Bridge 1 for possible shorts-to-battery. The fault should be evaluated only when this signal is HIGH.
LV_B2	OUTPUT	When this signal is HIGH, the Diagnostic Fault Detection component checks Half-Bridge 2 for possible shorts-to-battery. The fault should be evaluated only when this signal is HIGH.
LG_B1	OUTPUT	When this signal is HIGH, the Diagnostic Fault Detection component checks Half-Bridge 1 for possible shorts-to-ground. The fault should be evaluated only when this signal is HIGH.
LG_B2	OUTPUT	When this signal is HIGH, the Diagnostic Fault Detection component checks Half-Bridge 2 for possible shorts-to-ground. The fault should be evaluated only when this signal is HIGH.
LATCH_LO	OUTPUT	When this signal is HIGH, the Diagnostic Fault Detection component checks for possible open load. The fault should be evaluated only when this signal is HIGH.
DONE	OUTPUT	This indicates that the Diagnostic Sequence is over, that is two half-bridges and its associated load has been tested.

of the half-bridges is shorted to battery, both MTL and MTR go HIGH. With this configuration, it is not possible to determine exactly which half-bridge is shorted to battery. Nevertheless, the system checks both sides, because, in the case of an open load, only one side may be HIGH while the other side stays close to ground.

If the system detects no shorts-to-battery, it checks for shorts-to-ground. The Diagnostic Signal Generator turns on a high side switch and checks whether the half-bridge is pulled up accordingly. The system interprets the half-bridge at low as a short-to-ground fault. The LG_B1 signal signals the Fault Detection block to check for polarity of MTL. Similarly, it uses LG_B2 to check MTR. There is a delay of one clock cycle between the turning ON of a high side switch (for example, HS1) and the corresponding fault-checking signal (LG_B1). This ensures that the switch has turned ON and settled, and the system does not interpret a transition condition as a fault. Unlike the shorts-to-battery test, the shorts-to-ground test turns on a switch to check for a fault. Therefore, if a fault exists, this leads to heavy current through the switch for the entire duration of the test. For example, if Half-bridge 1 is shorted to ground, then excessive current flows through HS1 for the entire duration it is on (i.e. during clock cycles 6, 7 and 8 in Figure 23). Increasing the clock frequency reduces the duration.

If the system detects no shorts to either battery or ground, then it checks for an open load. It turns ON HS1 which pulls Half-bridge 1 to battery. It allows sufficient time for the Half-bridge 2 to be pulled up through the motor. The system then turns ON the LATCH_LO signal which signals the Diagnostic Fault detection unit to check whether both the motor terminals (MTL and MTR) are HIGH. It interprets both not HIGH as a load open fault. When the sequence is over, the system sets the DONE signal HIGH and the sequence stops until restarted. Once started by setting ENABLE high, the entire sequence completes in 26 clock cycles (0 through 25). In the example project, we use a clock frequency of 125 KHz which results in a sequence time of 208 µs.

All signals mentioned in Table 11 are inputs to the Diagnostic block. In addition, the block uses the MTL and MTR signals to determine the existence of a fault. The block also configures the Blanking Timer which verifies that the faults reported by the Diagnostic block are persistent. Whenever any of the control signals changes, the block detects the transition and resets the internal count of the Blanking Timer to 0. Table 12 lists the inputs and outputs of the Diagnostic block.

Figure 23. Timing Diagram of the Diagnostic Signals

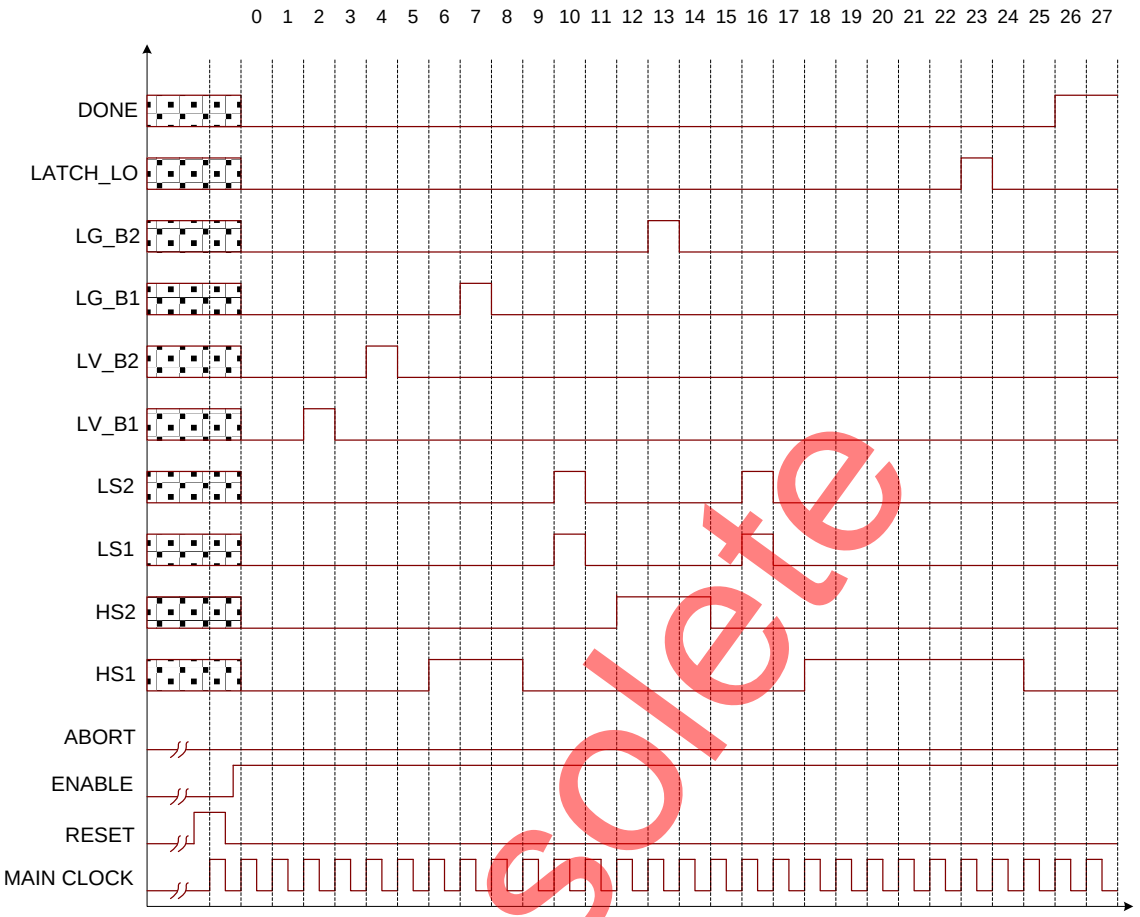
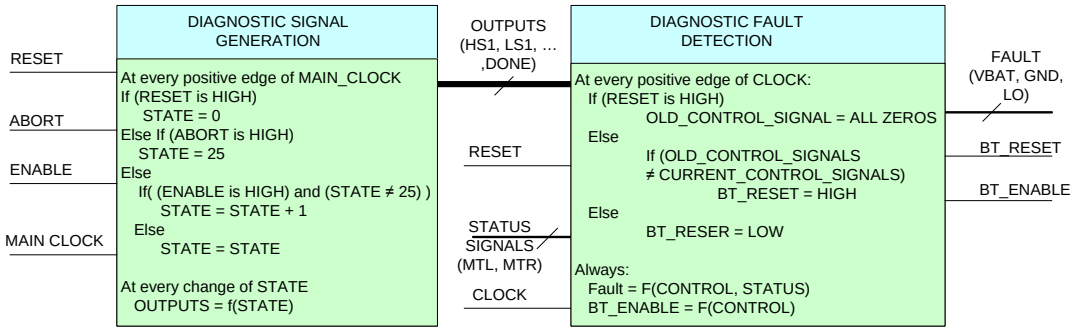


Table 12. Inputs and Outputs of the Diagnostic Fault Detection block

Signal Name	Type	Signal Description
CLOCK	INPUT	This is the input clock to the fault detection system. It controls the timing of the BT_RESET signal
RESET	INPUT	The RESET signal is used for clearing the internal logic used for control signal transition detection.
B1	INPUT	This signal is same as MTL (we assume that Half-bridge 1 is the left half-bridge)
B2	INPUT	This signal is same as MTR (we assume that Half-bridge 2 is the right half-bridge)
HS1, HS2, LS1, LS2, LV_B1, LV_B2, LG_B1, LG_B2, LATCH_LO	INPUT	Described in Table 11. The HS1, HS2, LS1, LS2 make up the BSW along with B1 and B2. The LV_B1, LV_B2, LG_B1, LG_B2 and LATCH_LO indicate when to sample the BSW for a particular fault.
VBAT	OUTPUT	This indicates that a short to battery condition has been detected.
GND	OUTPUT	This indicates that a short to ground condition has been detected.
LO	OUTPUT	This indicates that an open load condition has been detected.
BT_RESET	OUTPUT	This signal goes high for one clock cycle whenever any of the control signals change state.
BT_ENABLE	OUTPUT	This signal is high, whenever a BSW is ready to be sampled for a fault. This enables the Blanking Timer. This is kept low at other times to ensure that spurious signals from transitioning BSWs are not treated as faults.

Figure 24. Implementation of the Diagnostic Signal Generation and Fault Detection Components



PSoC 3 Implementation

Figure 24 shows the implementation of the Diagnostic Signal Generation and Fault Detection components. We use Verilog to implement both the components. The bulk of the Diagnostic Signal generation component is a state machine that counts the clock-cycles and sets the outputs based on the clock-cycle. It consists mainly of sequential logic. The outputs are computed as a function of the current state.

The Diagnostic Fault generation component consists mostly of combinatorial logic. Sequential logic is used for transition detection of the control signals. The combinatorial logic extracts the faults information from the control and status signals. The two components, the Blanking Timer, the Control and Status Registers and the Fault Latch are put together to implement the complete Diagnostic System, as shown in Figure 25.

The Fault Latch, the Status Register and the Control Register have purposes similar to those used in the Over Current and the Fault Detect blocks. The ABORT signal is asserted on any occurrence of a Diagnostic fault to ensure that the sequence does not continue to run. This prevents the system from entering further hazardous conditions. The interrupt service routine DIAGNOSTIC_DONE sets a flag to indicate that one Diagnostic Sequence is complete, and the Diagnostic system is available to diagnose another inactive motor.

Table 13 lists the API functions to control and configure the system offered by the Diagnostic System.

Figure 25. Complete Diagnostic System

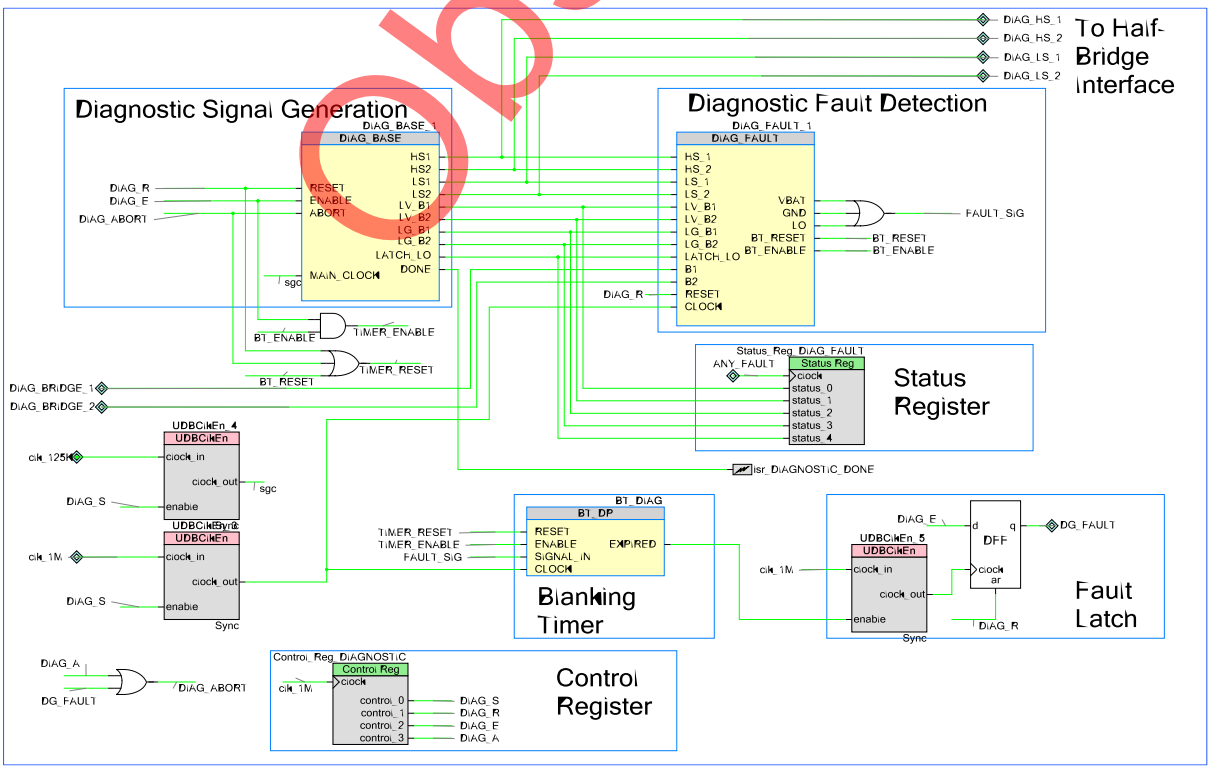


Table 13. API Functions of the Diagnostic Block

Function	Action
`Diagnostic_Start()`	Resets and starts the signal generation, the fault detection components and the Fault Latch. It also clears the Status Register. However, the Diagnostic Signal Generation component is not yet enabled.
`Diagnostic_Stop()`	Resets and stops the Diagnostic Signal Generation, the Diagnostic Fault Detection components and the Fault Latch. The status register is cleared.
`Diagnostic_Enable()`	Enables the signal generation block by setting its ENABLE signal high. The signal generation starts, if the block has been previously started by calling `Diagnostic_Start()`.
`Diagnostic_Disable()`	Disables the signal generation by setting its ENABLE signal low.
`Diagnostic_Abort()`	Aborts the sequence, sets DONE to HIGH, and clears the Status Register.
`Diagnostic_SetBlankingTime (uint8 blankingTime)`	Resets the signal generation, fault detection and Blanking Timer blocks, sets the D0 register of the Blanking Timer according to the value `blankingTime`, and clears the Status Register.
`uint8 Diagnostic_GetBlankingTime()`	Returns the current value of the diagnostic blanking time.
`Diagnostic_AssertReset()`	Holds the signal generation, fault detection, Blanking Timer, and Fault Latch in reset. Clears the Status Register.
`Diagnostic_ReleaseReset()`	Removes the reset condition from the signal generation, fault detection, Blanking Timer and the Fault Latch. Clears the Status Register.
`Diagnostic_ClearInterrupt()`	Clears any pending interrupt caused by the DONE signal.

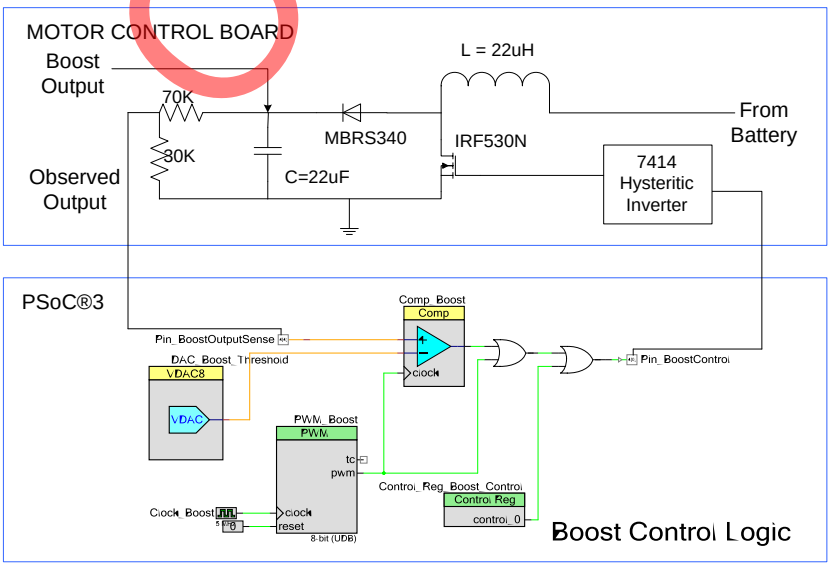
We have discussed the three main blocks (Over Current, Fault Detect, and Diagnostic) shown in Figure 6. The remaining blocks are simpler so we show their implementation directly without a conceptual realization.

Motor Drive Signals Block

The Motor Drive Signals block generates the PWM signals for the H bridge switches. It includes one Boost converter which allows PSoC 3 to drive one H bridge without special drivers. A capacitive pump drives the other H bridge. Figure 26 shows the Boost converter implementation.

A resistor divider circuit produces the Observed Output, which is proportional to the Boost output voltage. The comparator Comp_Boost compares the Observed Output to a reference set by a DAC. Whenever the boost output voltage is less than expected, the Boost FET is activated. A fixed period and duty cycle square wave (produced by the PWM Boost block) cycles the FET. An extra hysteretic inverter on the Motor Control Board prevents spurious

Figure 26. PSoC 3 Implementation of the Boost Converter



switching of the FET due to noise. Depending on the circumstances, the Control Register allows or gates the PWM signal from reaching the Motor Control Board.

The Boost converter block provides two APIs to control the converter:

- `Boost_Start()`
  Starts and configures the DAC, the comparator and the PW then writes a 0 to the Control Register to allow the PWM signal to reach the FET controlling pin (Pin 4_0 in this case).
- `Boost_Stop()`
  Stops the DAC, the comparator and the PWM. The Control Register is written with the value 1, which sets the FET controlling pin high. This effectively disables the Boost converter.

The output of the boost converter is then used as described in the section Alternate Methods to Drive H Bridges. More details about the Motor Control Board are available in Appendix 1.

Figure 27 shows the implementation of the Motor Drive Signals block. Two PWM blocks generate the drive signals for the two motors. Motor 1 uses a capacitive pump based high side gate drive that requires its high side switches to be pulse width modulated. Motor 2 uses a Boost converter based gate drive that can modulate both high side and low side switches.

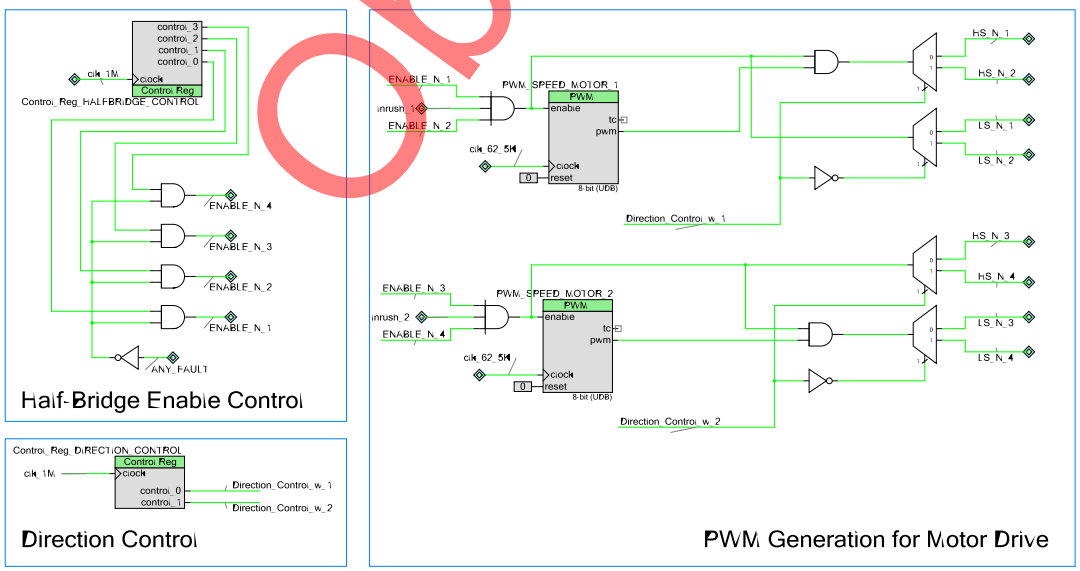
We choose to modulate the low side switches for this motor. The Control_Reg_HALFBRIDGE_CONTROL is a control register that enables the particular half-bridges based on the motor that we want to turn ON. In case of any fault (indicated by the ANY_FAULT signal), the ENABLE_N signals go LOW, disabling all half bridges.

Both the PWMs are 8 bit and implemented in UDB. The PWMs are disabled unless the motors they drive need to be ON. This leads to less switching in the UDBs and consequently less power consumption. Once the ENABLE signal of an active PWM turns OFF, the PWM completes the period before becoming inactive. We further gate the signals from the PWM with the half-bridge enable signals so that the control signals turn OFF as soon as the half-bridge enable signals go LOW, without waiting for the period to get over.

The drive signals also depend upon the Inrush_x signals. For example, if the Inrush_1 signal goes LOW, then the signals HS_N_1, HS_N_2, LS_N_1 and LS_N_2 signals are turned OFF. This is required for a soft start as described in the Over Current Block section.

Control_Reg_DIRECTION_CONTROL sets the direction. The two outputs from the control register control the select lines of the de-multiplexers. The de-multiplexer selects the switches that need to be ON for each half-bridge.

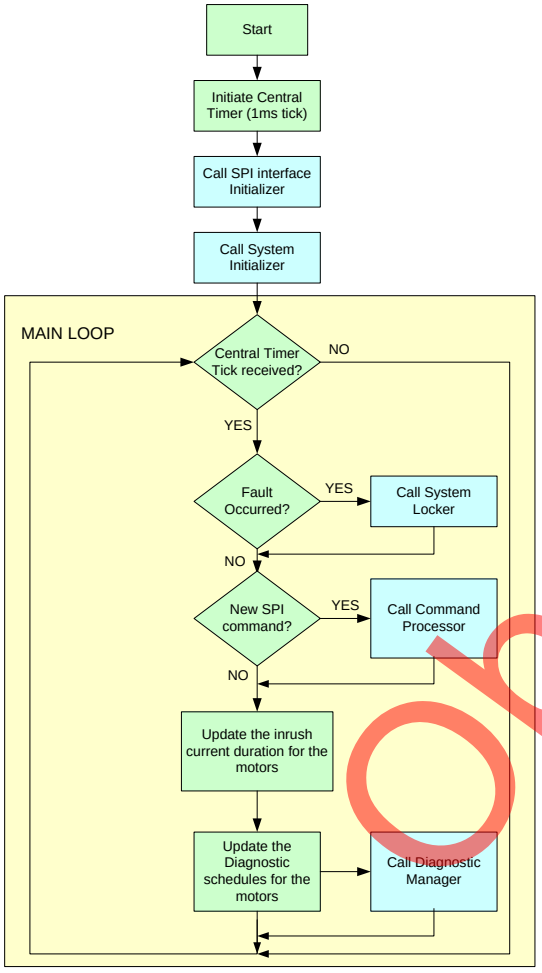
Figure 27. PSoC 3 Implementation for the Motor Drive Signals Generation Block



Firmware Block

The preceding discussion illustrates that our example implements the virtually the entire fault detection and protection system in hardware. What remains is for the Firmware block to configure the blocks as requested by the master and to periodically monitor the status of the system. The Firmware block also maintains the status of the motors and runs diagnostic on them whenever possible. Figure 28 illustrates the top level firmware flow.

Figure 28.Top Level Firmware Flow



The firmware performs three tasks before entering the main loop:

- Starts the Central Timer
- Sets up the interface with the host through the SPI Interface Initializer
- Initializes all hardware with default values through the System Initializer

The main loop runs every 1 ms on the tick of the Central timer. Please note that the fault detection and protection latencies are not related to this period; they are handled in hardware and hence their responses are in all practical sense immediate. This allows the firmware designer greater latitude, as it removes the tight time constraints for responses to faults. The designer may reduce or increase the central timer frequency per application requirements.

The main loop can only run every 1 ms, if the loop time is less than 1 ms. The LCD routines associated with the user interface may take as long as 50-60 ms to execute. In such cases, the main loop does not run on every timer tick. The LCD routines are not an essential part of the system and can be disabled in firmware. These routines are for demonstration purposes. A change in system configuration through the SPI or a fault condition uses these routines to visually convey the status. All the information displayed on the LCD is otherwise available to the master through the SPI interface. The intended mode of operation in a practical application is without the LCD. If the LCD is disabled, the main loop time is always less than 1 ms for any bus clock frequency greater than or equal to 8 MHz.

When the system detects a fault, it triggers an interrupt which sets the fault flag and stores the fault information for the main loop to consider later. The main loop inspects this flag, and calls the System Locker if a fault is indicated. A locked system does not accept most commands, unless the master releases it through a system release command.

Similarly, another flag lets the main loop know of new SPI transaction. The main loop calls the Command Processor to process any new SPI commands. As mentioned, most commands are not processed if the system is locked by the System Locker.

The main loop keeps track of the `inrushPeriod` of each motor (as explained in Over Current Block) with 1 ms resolution. Based on this, the main loop handles the switch from a soft start to normal configuration of the Over Current block.

The main loop also maintains a schedule for running the diagnostic sequence on each motor. There are two conditions for a motor to be able to run a diagnostic sequence:

- There must be a pending request for a diagnostic on the particular motor.
- The motor must be inactive for a pre-specified time.

The request for diagnostic may come from the master, or the main loop may itself put in an automatic request if the motor has been inactive but has not been diagnosed for some duration. The minimum duration between two successive automatic requests for a particular motor is called the `interDiagnosticInterval` of the motor. The value of the `interDiagnosticInterval` is set to 2 seconds in the example project.

As the diagnostic sequence can only work with an inactive bridge, both the half-bridges must be pulled to ground

before the sequence can start. If the motor is rotating due to inertia even after it is turned OFF, there might be back-emf from the motor which will prevent the bridges from settling to ground. Therefore, a minimum off-period of the motors is required before a diagnostic can run on the motor. The minimum off time is set to 1 second in the example project.

The main loop keeps track of the inactive period and the diagnostic request for each motor. In other words, it maintains the diagnostic schedule for each motor. It calls the Diagnostic Manager with the schedule. The Diagnostic Manager inspects the schedule and decides whether to run diagnostic on any motor. It handles the connection of the motor to the Diagnostic Block, and calls the Diagnostic APIs to run the diagnostic on the motor.

In summary, the main loop maintains the system with the help of the Central Timer, the SPI Interface Initializer, the System Initializer, the System Locker, the Command Processor, and the Diagnostic Manager.

System Initializer

The main task of the System Initializer is to initialize the half-bridges, the motor driver, over current, fault detection and diagnostic hardware to default states. It configures the Motor Drive Signal Generation block to keep all half-bridges disabled, and it sets their initial mode to Diagnostic (Figure 12). It loads default direction settings into the Control_Reg_DIRECTION_CONTROL.

A DAC generates the threshold signal for the half-bridge interface blocks. The System Initializer starts the DAC and loads the default threshold into it. Next, it starts the boost converter through the `Boost_Start()` API.

The System Initializer configures the Over Current, Fault Detect, and Diagnostic blocks with default values. It then starts and enables all of the blocks with the exception of the Diagnostic block. It does not start the Diagnostic block as it is not in use until a diagnostic request is set for at least one motor.

Finally, the System Initializer initializes a set of status variables to maintain the following information:

- The number of active motors (initialized to 0)
- The active motors' inrush period (initialized to 0)
- The active motors' off period (initialized to 0)
- Time elapsed since the motors last underwent Diagnostic (initialized to 0)
- Status of the motors' diagnostic request (initialized to 0)
- Whether Diagnostic is running (initialized to FALSE, as Diagnostic is not started)
- The motor currently being diagnosed (initialized to one more than the number of motors in the system, i.e. a non-existing motor, to indicate that none of the motors in the system are being diagnosed at the moment)
- The `systemLocked` variable (initialized to FALSE to indicate that no fault has been detected and hence the system is not locked)

Once initialized, the system enters the main loop. The main loop examines the fault status of the system and proceeds accordingly. An interrupt, triggered by any fault, updates the fault status as illustrated in Figure 29.

Three kinds of faults generated by the Over Current (OC_FAULT), Fault Detect (MOTOR_FAULT) and Diagnostic (DG_FAULT) blocks are logically OR'd to generate the ANY_FAULT signal. This fault triggers the Fault interrupt (isr_FAULT). The fault interrupt sets the `systemLockRequest` variable to TRUE and sets the `systemLocked` variable to FALSE to indicate the logging of a request to lock but that the request is not yet serviced. It also reads the Status Register associated with each of the fault generation blocks and stores their values.

The main loop calls the System Locker if it finds `systemLockRequest` and `systemLocked` to be TRUE and FALSE respectively.

System Locker

Like the System Initializer, the System Locker brings the system to a deactivated state. It disables all half-bridges and sets their mode to Diagnostic. It then stops the Over Current, Fault Detect and Diagnostic blocks as well as the PWMs in the motor driver blocks. Unlike the System Initializer, the System Locker does not load default parameter values into the blocks, as the host may have configured them in the course of operation with values appropriate for the application. The System Locker ascertains the source of the fault by checking the variables written by the fault interrupt. It then updates the SPI interface appropriately.

Similar to the System Initializer, the System Locker resets the status variables with one important exception. If the source of the fault is the Over Current or the Fault Detect block, then the faulty motor must undergo a diagnostic. A request for a diagnostic sequence is set for the faulty motor, and any pre-existing request for any other motor is ignored and deleted. In this way, the faulty motor has priority above other motors. Nevertheless, it would still have to stay OFF for the minimum amount of time specified before the sequence can start.

A variable keeps count of consecutive calls to the System Locker. Two calls to the System Locker are consecutive if

Figure 29. Fault Interrupt

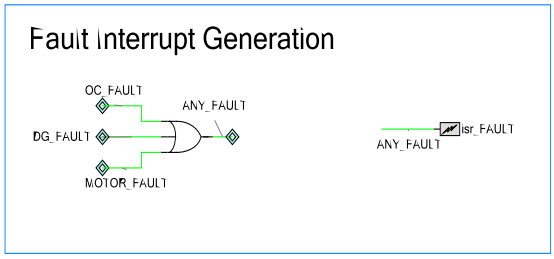


Table 14. Motor Commands and the Associated Actions

Motor Command	Actions Performed
Start/Stop	For each motor: The associated half-bridges are determined. If the motor is under diagnostic, and it needs to be started then diagnostic is aborted. A variable is set to 0 indicating that Diagnostic cannot be run on the motor. If the motor needs to be started, then the `inrushState` variable for the motor is set to TRUE, indicating that it will exhibit inrush current for some time. The Over Current block in charge of that motor is configured for soft start. A variable is set to indicate that the particular motor is active. The direction is set as requested. Then the half bridges are enabled or disabled as per the start/stop request.
Set Direction	For each motor: 1. A change in direction is not allowed on an active motor. For all active motors, the direction is kept unchanged, regardless of the requested direction. 2. For all inactive motors, the direction is set as requested.
Change Speed	For each motor that needs speed change: 1. It is determined whether speed needs to be increased or decreased. 2. If increase is requested, and motor is already not at maximum speed, the new duty-cycle is calculated by adding a predetermined value to the current duty cycle. If this duty cycle is less than the maximum allowed duty cycle, then the current duty cycle is updated with the new duty cycle; otherwise the current duty cycle is updated with the maximum allowed duty cycle. 3. If reduction in speed is required, and motor is already not at minimum speed, the new duty-cycle is calculated by subtracting a predetermined value from the current duty cycle. If this duty cycle is greater than the minimum allowed duty cycle, then the current duty cycle is updated with the new duty cycle; otherwise the current duty cycle is updated with the minimum allowed duty cycle.

the system has not been released from lock condition between the two calls. The main loop uses this variable to know the number of times a faulty motor's diagnostic request is set. If a faulty motor has already been diagnosed, but the master has not released the system, then the diagnostic system is turned off till further intervention by the master. This ensures that we do not unnecessarily run a diagnostic on a bridge known to be faulty.

After exiting the System Locker, the main loop checks if a new command is available and calls the Command Processor to process the new command.

Command Processor

The Command Processor identifies and executes all the SPI commands from the master. We divide the commands c into two main categories based on their end result:

- **Motor Commands**: Used to change the state (ON/OFF), direction, and speed of motors.
- **Configuration Commands**: Used to alter the system configuration but not affect the motors such as to change over current thresholds and blanking times, send a diagnostic request, or to release the system from a lock condition.

Changing the direction of a spinning motor can result in heavy current. This is more likely with a heavy motor running at high speed. Therefore, the example project does not allow change in direction of an active motor.

The configuration commands call the respective APIs of the Over Current, Fault Detect or Diagnostic blocks. They check if the requested parameter value is within the allowed range. If the value is outside range then it is replaced by the closest allowable value.

The Diagnostic Request command sets the diagnostic request variable for the particular motor to true.

A noteworthy configuration command is the System Release command, which brings out the system from locked condition. If the correct code is provided with the command, then the System Release command is executed. The System Release command restarts the Motor Drive PWMs, and clears all existing fault status and associated variables. It configures the Over Current, Fault Detect and Diagnostic blocks similar to the System Initializer.

If the system is locked then all commands, except the Diagnostic Request and the System Release are ignored. Also, though not specifically mentioned, all commands update the SPI interface with the latest results from the commands' execution. This ensures that the master receives updated values on reading a status.

Central Timer

The Command Processor, System Initializer and System Locker set a group of variables to indicate the start of certain periods (e.g. inrush current, motor-off and so on). The Central Timer updates these variables. It updates certain flags and variables every 1 ms to enable the acquisition of the following information:

- Whether or not the main loop can be run.
- If any motor is outside its inrush current state and its Over Current system can be switched to normal configuration from a soft start configuration.

- The elapsed time since a motor has undergone a diagnostic sequence.
- The elapsed time since a motor has gone from the active to inactive state.

The system uses a hardware timer and interrupt to implement the Central Timer as shown in Figure 30. It uses a 16-bit, fixed function Timer block. A 1 MHz clock drives the Timer with its period set to 1000. It reaches its terminal count once every 1 ms and triggers the *TimerTick* interrupt which:

- Sets a flag to let the main loop know that 1 ms has elapsed since last iteration of the main loop.
- When the motor is in inrush current state, increments by 1 the `timeSpentInInrushState` variable. When the variable value equals the `inrushPeriod` of the motor, we can safely assume that the motor is operating with normal current levels and switch its Over Current unit from Soft Start to Normal mode.
- When the motor is inactive, increments by 1 the `motorInactivePeriod` variable. When the value of the variable value equals the minimum off-period required for diagnostic, it adds the motor to the list of motors allowed to be diagnosed. Also, if the motor is inactive, then it increments by 1 the `interDiagnosticInterval` variable. If this interval exceeds the default Diagnostic Interval of the system, then an automatic request for a diagnostic sequence of the motor is set. The Diagnostic Manager only diagnoses a motor if it is on the list of allowable motors, and there is a request for a diagnostic on the motor.

Diagnostic Manager

The Diagnostic Manager diagnoses the motors. It checks if a Diagnostic sequence is already in process, and then it starts a diagnostic on an appropriate motor. When a diagnostic sequence completes, it sets the `interDiagnosticInterval` variable of that motor to 0 to indicate that the motor has just completed a diagnostic.

Figure 31. Routing for a Diagnostic System

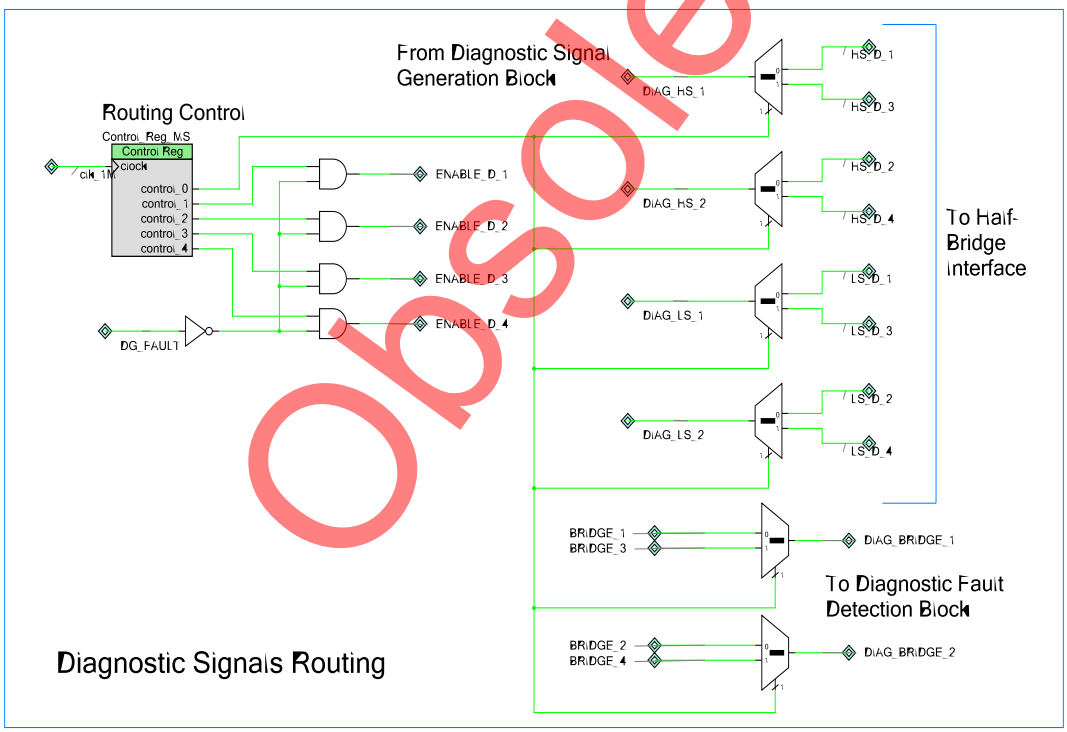
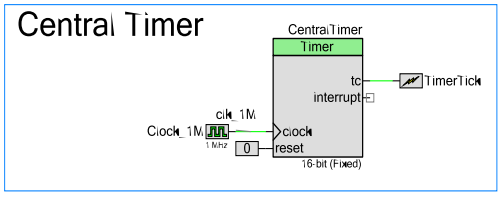


Figure 30. Implementation of the Central Timer



It calls the APIs provided by the Diagnostic block to handle the request. The system generally consists of multiple motors and a single diagnostic system. Therefore, the Diagnostic Manager also needs to set up the routing between the Diagnostic block and the intended motor. In Figure 31 we show the hardware setup for routing.

The motor selection register Control_Reg_MS determines the routing. The LSB of the register acts as the select line of the multiplexers and de-multiplexers which delegate the

signals to and from the Diagnostic block. The Diagnostic Signal Generation block generates the DIAG_HS_x and DIAG_LS_x signals (Figure 26). They are routed to the correct Half-Bridge interface block based on the LSB. The LSB also determines which half-bridge signals should be used as the MTL and MTR (in Figure 26, DIAG_BRIDGE_1 and DIAG_BRIDGE_2).

The Control_Reg_MS further controls the ENABLE_D signals of all the half-bridge interface blocks. The Diagnostic Manager writes the appropriate value to the control register to turn on the appropriate ENABLE_D signals HIGH.

To summarize the actions of the Firmware block:

- The main loop runs every 1 ms.
- The System Initializer initializes the system with default values.
- When a fault occurs, the System Locker locks the system.
- The Command Processor handles commands from the master to run the motors and to configure the entire system.
- The Central Timer maintains several variables representing time between different events.
- The Diagnostic Manager runs diagnostic on a particular motor.

All changes to the motor and the system are communicated to the master by updating the SPI interface. The LCD, if enabled, also conveys this information. When a fault occurs, a digital signal is turned ON as an indicator to the master. This ensures that the master knows about the occurrence of a fault immediately, as opposed to later through the SPI interface. In the example project, we also connect an LED to this signal for a visual detection of a fault condition.

User Interface

The User Interface block allows a user and/or a host to configure and communicate with the system. An LCD and an LED enable visual communication, whereas a SPI interface provides a link for a SPI master. The visual interface is not necessary for the operation of the system; we include it for demonstration purposes only.

SPI Communication

The host device controls the H bridge protection and diagnostic through a SPI interface. The interface is used to:

- Configure and verify parameters relevant to the protection and diagnostic system
- Control the states of the motors
- Start a diagnostic on a particular motor
- Read the Fault Status
- Release the system after it locks up due to a fault

The above tasks can be partitioned into write and read commands. A *write* command intends a change in the behavior of the system. A *read* command observes an existing configuration without changing it. After a write command, the host may use a read command to verify that the correct execution of the write command.

The current implementation uses a register map for commands and their responses. The register map has two sections: WRITE and READ. The WRITE section accepts write commands from the host, while the READ section stores the existing configuration for the host to read. Figure 32 illustrates the register map. After executing a write command, the system appropriately updates the read portion to reflect that change.

Each byte in the register map contains specific

Figure 32. Description of SPI Register Map for a System with M Motors

WRITE	State of All Motors	BYTE 0
	Direction of All Motors	BYTE 1
	Speed Change Request for All Motors	BYTE 2
	Speed Increment/Decrement	BYTE 3
	Over Current Threshold for Unit 1	BYTE 4
	⋮	⋮
	Over Current Threshold for Unit M	BYTE M+3
	Over Current Blanking Time for Unit 1	BYTE M+4
	⋮	⋮
	Over Current Blanking Time for Unit M	BYTE 2M+3
	Fault Detect Blanking Time for Motor 1	BYTE 2M+4
	⋮	⋮
	Fault Detect Blanking Time for Motor M	BYTE 3M+3
	Diagnostic Blanking Time	BYTE 3M+4
	RESERVED	BYTE 3M+5
		BYTE 3M+6
	Request for Diagnostic	BYTE 3M+7
	Release System	BYTE 3M+8
		⋮
		BYTE 127
READ		BYTE 128
	Same as BYTE 0 through BYTE 3M+4	⋮
		BYTE 3M+132
	Bridge Fault Status	BYTE 3M+133
	Over Current Fault Status	BYTE 3M+134
		⋮
	Verification of Communication	BYTE 0xDE

information. Figure 32 shows how the system interprets each byte meant for M motors. The system should have at least one motor (i.e. M must be ≥1).

The first two bytes in the register map (BYTE 0 and BYTE 1) contain information about the state of the motors (whether they are ON or OFF) and the direction of rotation (clockwise/counter clockwise). The third byte (BYTE 2) implies whether the speed for a motor needs to change, and the fourth byte (BYTE 3) implies how the speed needs to change (increase or decrease). The next M bytes (BYTE 4 through BYTE M+3) are the Over Current thresholds for the M Over Current Detection Units. As previously discussed, all systems with M motors need not have M over current detection units. In such cases, the user may modify the register map accordingly. Similarly, the following M bytes (BYTE M+4 through 2M+3) contain the Over Current Blanking Time information for each Over Current unit.

The next M bytes (BYTE 2M+4 through BYTE 3M+3) store the Fault Detection Blanking Time information, while BYTE 3M+4 stores the Diagnostic Blanking Time. The Diagnostic Blanking time is a system parameter and applies to all motors. The next two bytes are reserved, while BYTE 3M+7 holds the Diagnostic request for a particular motor. When the system locks up, most of the write commands are disabled. The host needs to write the correct byte (the System Release Code) into the BYTE 3M+8 to release the system from lock. This is the last significant byte in the WRITE portion of the register map.

The READ portion of the register map stores the same information in the WRITE portion. However, it is not a copy of the WRITE portion. The system updates the information in the READ portion only after the system has been configured according to the commands in the WRITE portion. The system also reports the H bridge fault information and the Over Current fault information in BYTE 3M+133 and BYTE 3M+134 respectively. BYTE 252 (or 0XDE) can hold one of two possible bytes; (0xAA) and its complement 0x55. These two bytes are used to verify that:

- The communication link between the master and the system is intact
- The main loop is running

When the master reads this location, the main loop toggles the value at this location. On two successive reads of this location (in two different transactions), the master should receive 0xAA (0x55) and 0x55 (0xAA).

In this application note, we manage two motors. Therefore, the register map used in this application note can be derived from Figure 32 using M=2. Figure 34 illustrates the SPI transaction. For each byte sent by the master a corresponding byte is received. The first received byte is not dependent on the first transmitted byte of the current command and can be treated as garbage; in fact, it is dependent on the last sent byte of the previous command. The first byte received after a system reset is always 0x55. All other subsequent received bytes are strictly dependent on the bytes transmitted by the master.

Table 15. SPI Register Map for a Two Motor System

Byte Number	Significance
0x00	State of all motors (ON/OFF)
0x01	Direction of all motors (CW/CCW)
0x02	Speed Change Request for all motors (Change/Do not Change)
0x03	Change of speed (positive/negative)
0x04	Over Current Threshold for Unit 1
0x05	Over Current Threshold for Unit 2
0x06	Over Current Blanking Time for Unit 1
0x07	Over Current Blanking Time for Unit 2
0x08	Fault Detect Blanking Time for Motor 1
0x09	Fault Detect Blanking Time for Motor 2
0x0A	Diagnostic Blanking Time
0x0D	Motor ID with Diagnostic Request
0x0E	Release code required after System Lock
0x8B	Bridge Fault Status
0x8C	Over Current Fault Status
0xDE	Communication Confirmation Byte

Table 15 shows the WRITE portion of this register map.

Figure 34 illustrates the SPI transaction. For each byte sent by the master a corresponding byte is received. The first received byte is not dependent on the first transmitted byte of the current command and can be treated as garbage; in fact, it is dependent on the last sent byte of the previous command. The first byte received after a system reset is always 0x55. All other subsequent received bytes are strictly dependent on the bytes transmitted by the master.

The figure shows that the received byte is actually the information at location pointed to by the preceding sent byte. Therefore, if the master wants to receive the information at location L in the n+1-th received byte, the n-th byte it should transmit must be L. For example, in order to read the Over Current Threshold for Over Current detection Unit 1, the master should send 0x84 followed by

Figure 33. Illustration of SPI Transaction

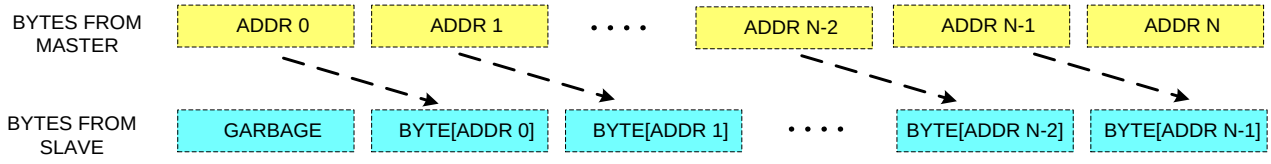


Table 16. Write Commands and Their Parameters

Write Command	# of Parameters	Description of Parameter
0x00	1	Intended motor state (ON/OFF)
0x01	1	Intended motor direction (CW/CC).
0x02	2	Intended speed change (Increase/Decrease)
0x03	N/A	Invalid Command
0x04 - 0x0A	1	The respective parameters (e.g. OC Threshold) intended to be set.
0x0D	1	Motor requiring a diagnostic
0x0E	1	Release code

any other byte. The two bytes it would receive would be garbage, and the Over Current Threshold respectively.

The above description applies to all commands, whether read or write. If the first received byte is less than 128, it is a write command. It is interpreted as follows:

- The first byte is the address of the location to start writing. Let the first received byte be X.
- The subsequent bytes are parameters for the command. They are written into consecutive locations of the register map, starting from location X.

For example if the command transmitted from the master is 0x04 0xAB, then the byte 0xAB will be written into the location 0x04.

The received bytes are meaningful only for read commands, as the actual system status can be inferred from those bytes. Received bytes for write commands bear little significance.

Each write command has at least one parameter following the command. Some commands require more than one parameter. To avoid errors, the master must ensure that it sends the correct number of parameters with each write command. Table 16 shows the number and description of parameters associated with each write command.

The first byte in the register map controls the motor states. Figure 34 shows how the information is encoded into the bytes. The bit positions identify the motors, while the bit value (1/0) determines the state (ON/OFF). For example if BYTE 0 contains 0x03 it means that both motors 1 and 2 should be ON.

Figure 36. Description of MOTOR SPEED CHANGE REQUEST and TYPE Registers

NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	MOTOR 2	MOTOR 1
NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	1=YES 0=NO	1=YES 0=NO

BYTE 2: MOTOR SPEED CHANGE REQUEST

NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	MOTOR 2	MOTOR 1
NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	1= INC 0=DEC	1=INC 0=DEC

BYTE 3: MOTOR SPEED CHANGE TYPE (INC = INCREASE, DEC = DECREASE)

Figure 34. Description of the MOTOR STATE Register

NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	MOTOR 2	MOTOR 1
NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	1 = ON 0 = OFF	1 = ON 0 = OFF

BYTE 0: MOTOR STATE

Similarly, the second byte in the register map stores the direction information. This is shown in Figure 35.

Figure 35. Description of the MOTOR DIRECTION Register

NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	MOTOR 2	MOTOR 1
NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	1=CW 0=CCW	1=CW 0=CCW

BYTE 1: MOTOR DIRECTION

The next two bytes, used for changing the speed of the motors, follow similar encoding scheme. This is shown in Figure 36.

For example, if the speed of Motor 1 must increase and the speed of Motor 2 decrease, the master should transmit the command 0x02 0x03 0x01. If any bit in the MOTOR SPEED CHANGE REQUEST register is 0, then the system ignores the corresponding bit in the MOTOR SPEED CHANGE TYPE register.

Our example application uses two motors, so we do not use the other bits in the register. They can be used when more motors are added to the system.

No special encoding is required for BYTE 3 through BYTE A. The value of the particular parameter is directly written into the byte.

The Diagnostic request byte follows the same encoding as the Motor State byte. The bit positions identify the motors, and the bit value determine the request (1 = Diagnostic Request, 0= No Diagnostic Request). At any instant, only motor can have a Diagnostic request, so no more than one bit can be 1 at a time.

Implementation of the SPI Interface

This project uses the SPI Slave component available in PSoC Creator and some additional resources to implement the SPI interface as shown in Figure 37. The implementation consists of three parts:

- **Communication:** It consists of the SPI Slave component.
- **Data Transfer:** This manages the transfer of data between the register map and the SPI Slave Component.
- **Book Keeping:** This signals the firmware block about a recently completed SPI transaction. It also reconfigures the Data Transfer system for the next SPI transaction.

The SPI Slave is a Creator component and synthesized in the UDB. Detailed information about the block and ways to configure it are in the component datasheet in PSoC Creator. It operates in mode 0 with a bit rate of 4 Mbps and a MSB first manner. The Receive and Transmit Buffer sizes are both set to 4. The component is also configured to generate an interrupt signal at the end of each byte complete. This signal triggers the Data Transfer block to execute the required transfers.

The Data Transfer portion is responsible for two major tasks:

- Transfer the transmitted byte from the SPI slave RX register (receive register) into a temporary location called `scratchpad` in the main memory.
- Transfer the correct data from the register map into the SPI slave TX register (transmit register), so that the behavior described in Figure 33 can be guaranteed.

These transfers execute regardless of the type of transaction (READ or WRITE).

The slave must process the current byte from the master to load the correct byte into the TX register before the start of the next byte. The minimum processing delay determines the inter-byte time. The Direct Memory Access (DMA) functionality available in PSoC 3 allows us to minimize the inter-byte time. The complete data transfer is carried out without any CPU involvement. This allows SPI transaction without interrupting the CPU.

We use two DMA channels for the implementation. Each channel consists of one or more Transaction Descriptors (TDs). Each TD is characterized by:

- Source address
- Destination address
- Number of Bytes
- Next TD

For detailed information on the DMA components and how to configure them, we recommend that you consult the DMA component datasheet available in PSoC Creator and the application note AN52705.

Figure 38 shows how the DMA channels and their TDs can seamlessly carry out the data transfer between the SPI slave component and the main memory.

Figure 37. PSoC Creator Implementation of the SPI Interface

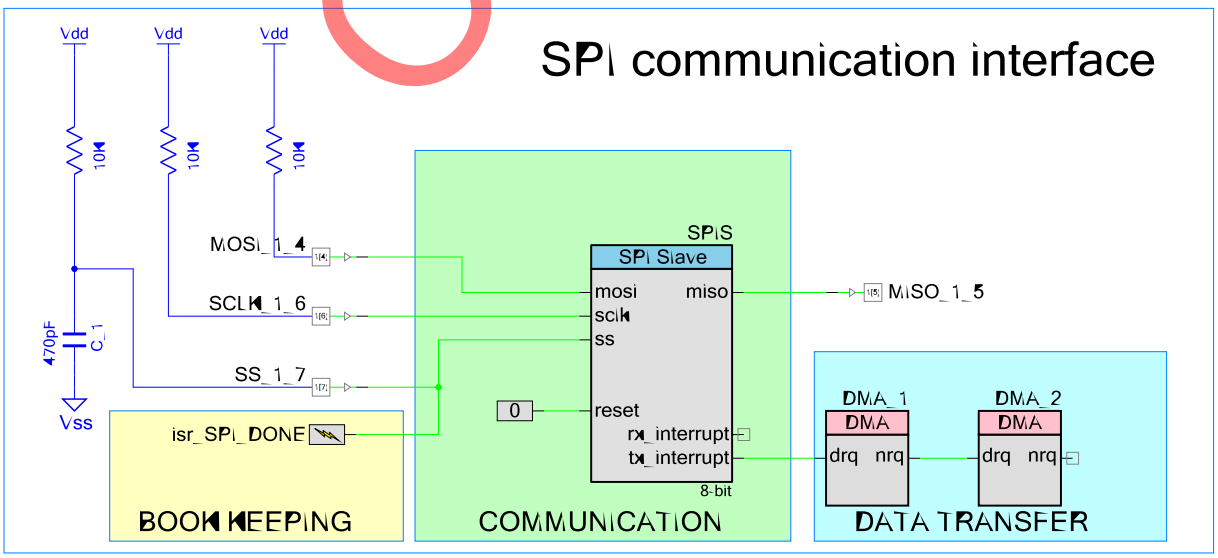
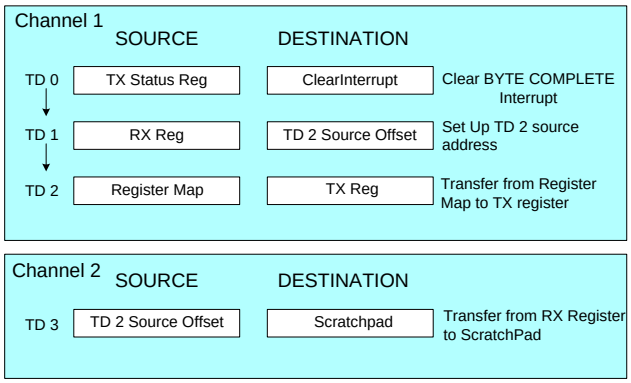


Figure 38. Configuration of DMA Channels and Transaction Descriptors



Channel 1 consists of 3 transaction descriptions which work in sequence:

- **Step 1**: When one byte transmission is complete, the BYTE COMPLETE bit in the TX status register turns HIGH which triggers the Channel 1 to start its first TD (TD 0). The TX status register is "sticky"; thus the BYTE COMPLETE bit stays HIGH, disabling the detection of subsequent interrupts. The TD 0 reads the register to clear this bit. It writes the content of the register to the variable `clearInterrupt`. TD 0 triggers TD 1 as its next TD.
- **Step 2**: The byte from the master is available in the RX Register. TD 1 moves this byte from the RX Register into the offset portion of the source address register of TD 2. The TD 2 source address register base is pre-filled with the base address of the register map. Therefore, the source of the TD 2 now becomes the register map location pointed to by the received byte. For example if the received byte is 0x05, then TD 2's source is now the data at location 0x05 (i.e. BYTE 05). With TD 2 thus configured, it is triggered by TD 1.

  The source address of a TD is a 16-bit value. The least significant byte is the offset, and the most significant byte is the base. In this project, the define `REGISTER_MAP_STARTING_ADDRESS` contains the base address.
- **Step 3**: TD 2 transfers the byte from the register map to the TX register of the SPI slave component. The master receives the byte during the next byte transfer. Once TD 2 completes, it sets the termination signal which starts Channel 2.

Channel 2 has only one TD, which transfers the received byte into the `scratchpad`. We recall that the received byte has been transferred into the offset portion of the TD 2 source address register in Step 2. The `scratchpad` is an array in the main memory where received bytes are collected for processing. Only a write command requires the received bytes to be processed; a read command is handled by the DMA channels.

Before any new transaction starts, the destination of TD 3 is set to the first location in the `scratchpad` (`scratchpad[0]`). After every byte transfer in a transaction, the destination address is incremented so that the bytes do not get over-written.

Data transfers without CPU intervention achieve very small inter-byte times. A minimum 8 MHz master clock frequency is needed to support a 4 Mbps SPI interface. The corresponding inter-byte time is 5 µs. For a typical master clock frequency of 48 MHz, the inter-byte time is less than 1 µs.

The master pulls the slave-select line high when a transaction is complete. This event triggers the SPI_DONE interrupt service routine to perform the necessary book keeping activities which consist of:

- Set a flag to indicate availability of new data from master
- Set a flag to indicate that the new data has not yet been processed
- Reinitialize the TD 3 destination address to point to `scratchpad[0]`

Remember that the main loop runs every 1 ms during which the system processes all received write command. Therefore, allow a minimum time of 1 ms between two consecutive SPI transactions.

The system does not count the received bytes to verify correctness of a transfer. The master must follow Table 16 while sending write commands, or there may be lost commands and erroneous behavior.

External 10K resistors pull up the Slave Select (SS), SCLK and MOSI lines to the supply voltage (V_{DD}). There is a capacitor on the SS line to prevent a spike on the line due to electromagnetic interference from the motor. The resistor and capacitor values can be changed to support different bit rates. The resistors may be eliminated if the master has its own pull up resistors.

A part of the firmware called the SPI Interface Handler sets up the initial Register Map, populates it with default values, and sets up the DMA transactions. It also constructs the commands and parameters from the received bytes. The SPI Interface Handler consists of two parts: the SPI Interface Initializer and the SPI Command Constructor.

The SPI Interface Initializer performs the following tasks:

- Initializes the READ and WRITE portions of the Register Map with default values
- Sets up the DMA channels and TDs for the data transfer and enables them

The SPI Command Constructor extracts the command information from the received bytes in the `Scratchpad`. The SPI_DONE interrupt (Figure 37) sets a flag to indicate completion of a new SPI transaction. Accordingly, the

Command Constructor reads the `scratchpad`. It interprets the first byte (`scratchpad[0]`) as the command and calculates the required number of parameters based on Table 16. It reads these parameters from successive locations of the `scratchpad` (`scratchpad[1]`, `scratchpad[2]`,...). Finally, it updates the write portion of the Register Map with the parameters for later use by the Command Processor.

LCD Interface

As already mentioned, we use the LCD interface purely for demonstration purposes in this project. The information visible on the LCD is also available through the SPI interface. You can disable the LCD functionality in firmware by setting the define `LCD_ENABLED` to 0 in the file *configuration.h*. The project uses standard LCD module available on CY8CKIT-001/CY8CKIT-030 along with the PSoC Creator Character LCD component. The LCD has two rows, each of which can display 16 ASCII characters as shown in Figure 39.

Figure 39. Character LCD Component

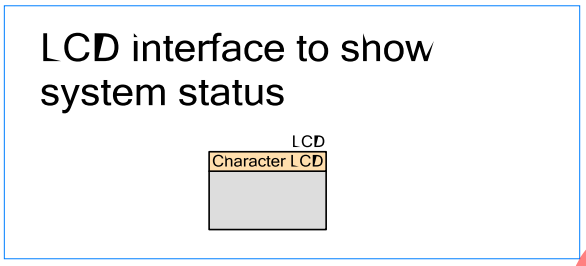


Table 17 shows a list of all the information displayed during the course of the application. The Demonstration section shows these in more details.

Table 17. Description of LCD display for different system status

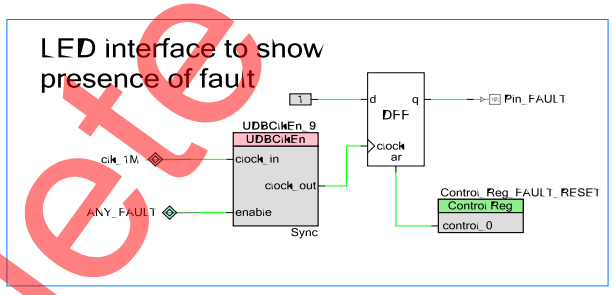
System Status	LCD Display
Startup	When the System is powered on correctly, displays the introductory startup message.
Motor Start/Stop	Displays the Motors IDs, their states (ON/OFF), their direction settings, and the speed settings.
Motor Direction Set	Same as Motor Start/Stop.
Motor Speed Change	Same as Motor Start/Stop.
Over Current/Bridge Fault	On detection of an over current or bridge fault, displays the Motor IDs, associated H bridge status (faulty/ok) and over current status (faulty/ok).
Diagnostic Fault	Appears only if the Diagnostic block detects a fault and then it displays the Motor ID, bridge status, over current status and the fault detected.
Change Parameter	Displays the Motor ID (if applicable), the parameter name, and the new parameter value
System Locked	Appears only when the system is in locked condition following a fault and the master sends a motor or configuration command.
System Released	Displays status when the master sends the correct system release command and code, to release the system following a lock condition.

LED Interface

The LED provides a visual indication of a faulty state. A digital signal controls the state of the LED. The same signal should connect to the master to communicate a fault condition. In the application project, the signal only connects to the LED. You may connect the signal to the master as needed.

Figure 40 shows the PSoC 3 implementation of the LED interface. The flip-flop is set on the occurrence of any fault which turns Pin_FAULT to HIGH. Pin_FAULT connects to the LED which turns ON. When the master sends the system release command, Control_Reg_FAULT_RESET resets the flip-flop to 0.

Figure 40.Implementation of the LED Interface



Demonstration with CY8CKIT-001 PSoC Kit with PSoC 3 Processor Module, and Motor Control Board

In this section, we demonstrate use of the application, using the CY8CKIT-001 development kit and the Motor Control Board. The Motor Control Board houses the FETs required to drive the motors, the associated gate drive electronics, the boost converter, and a few other components. Refer to the Appendix for details about the Motor Control Board as well as a corresponding setup with the CY8CKIT-030 kit.

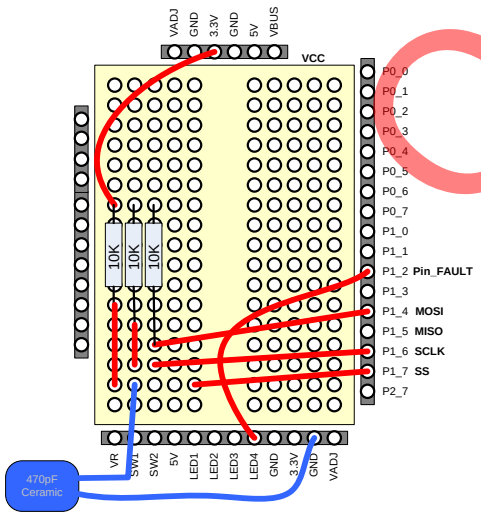
Demonstration Setup

External equipment used for the demonstration includes:

- Power Supply (0-15 V, 0-3 A)
- Dayton Motor, 12 V DC, 3.8 A, 21.4 W, 2350 rpm (Motor 1)
- Buehler Motor (6 V-24 V DC), 0.5 A, 6.3 W, 3000 rpm (Motor 2)
- Aardvark USB to I^{2}C/SPI host adapter
- Computer with USB port
- Cables and wires for connection

The CY8CKIT-001 comes with an embedded whiteboard prototype area for easy development. We use this prototype area for the SPI bus setup and the connection to the LED. Figure 41 shows the whiteboard connections.

Figure 41. CY8CKIT-001 Prototype Area Connection



We chose two different motors with large difference in output power to demonstrate the easy configurability of the system under a wide variety of loads. The computer acts as the master. The Aardvark USB to I^{2}C/SPI adapter connects to the USB terminal on the computer and to the SPI pins on the PSoC 3. We use the Aardvark I^{2}C/SPI ControlCenter software to communicate with the USB to I^{2}C/SPI adapter. You can download the software at http://www.totalphase.com/products/control_center/

Plug the Motor Control board into the Port A of the CY8CKIT-001 development kit. The settings for kit and board are as follows:

- CY8CKIT-001
  - $V_{DD\ SELECT}$ set to 3.3 V
  - J6 and J7 set to V_{DD}
  - J8 set to V_{REG}
  - All VDDIO jumpers (J2, J3, J4, and J5) set to V_{DD}
  - J12 (LCD Power) set to ON
- Motor Control Board
  - J2 and J4 are in place (connected)
  - J3 and J6 are in place (connected)
  - J5 is set to Vin1

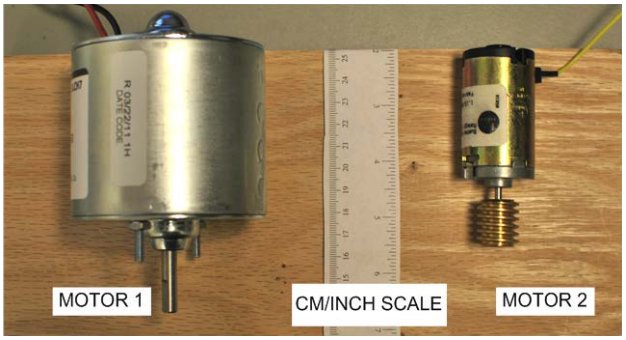
Figure 44.

The grounds of the Motor Control Board and the kit must connect externally as they do not connected through Port A. Use the following sequence to connect and power up the board and kit. Verify that power is removed from both the CY8CKIT-001 dev kit and the motor control board.

1. Plug the Motor Control Board into Port A of the development kit. Connect the grounds externally.
2. Power up the kit.
3. Program the PSoC 3 device on the processor module connected to the kit with the associated project.
4. Power up the Motor Control Board.

If the motor control board is powered before the device is programmed, it may result in heavy current being drawn from the power supply. The motors used for demonstration purposes are shown in Figure 42 with an inch scale added for perspective.

Figure 42. Motors Used for Demonstration



Demonstration of Operation

For the demonstration, we show how to start and stop the motors, change their speed, and change a few configuration parameters. Then, we show the response of the system to faults.

We use the Aardvark Control Center software to send all the commands to the system. Configure the adapter for a SPI+GPIO configuration. Figure 43 shows the Control Center software user interface.

Figure 43. Setup of the Control Center GUI

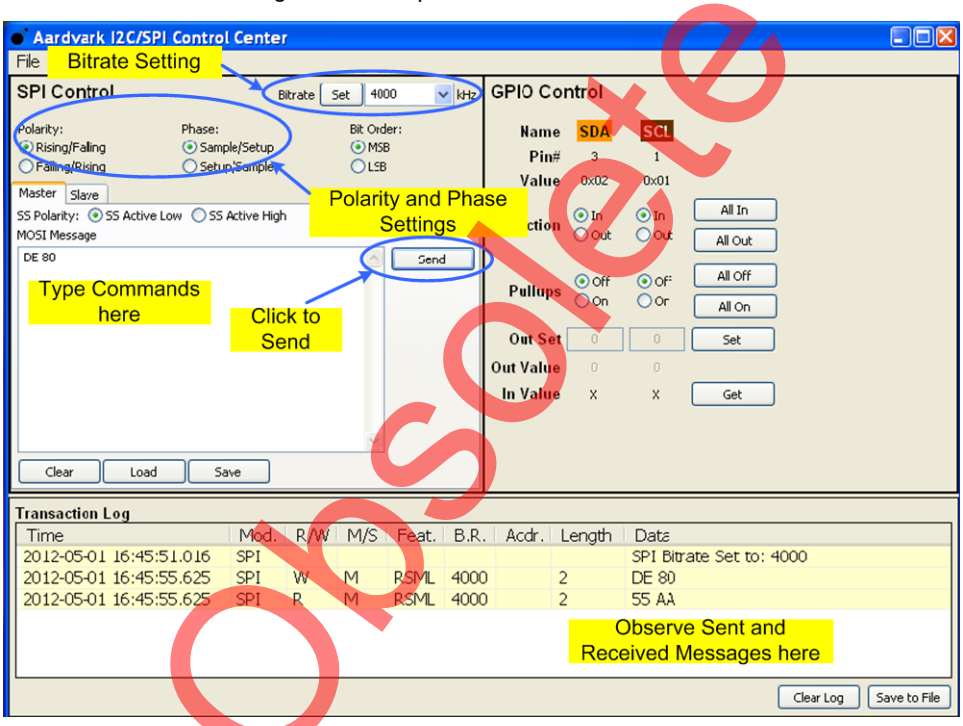
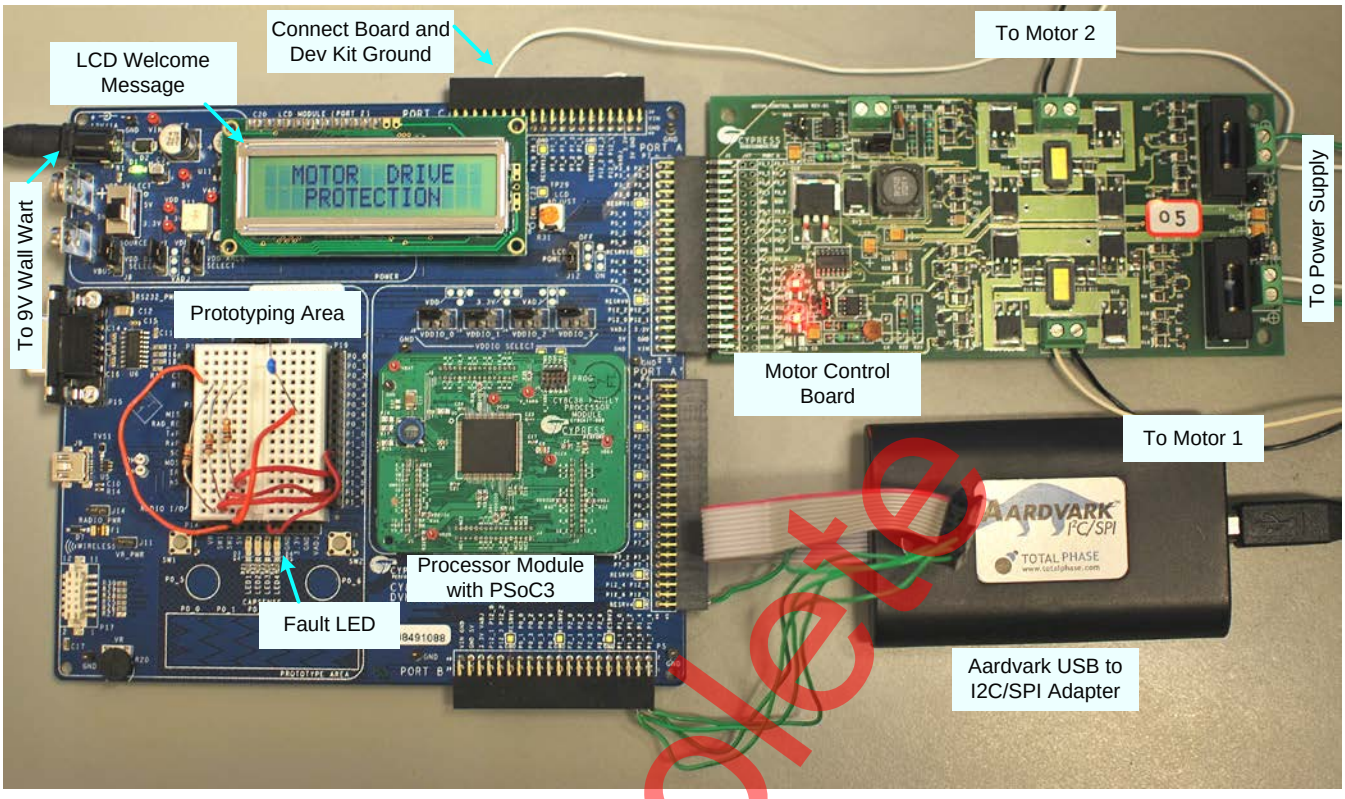
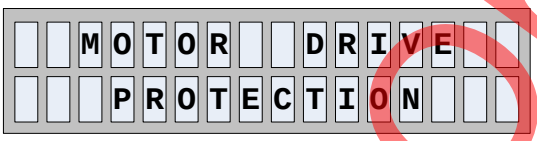


Figure 44. Complete Setup of CY8CKIT-001 for Project



After setting up the SPI Interface, reset the system to prevent the effect of any spurious signals on the SPI bus during setup. After reset, the LCD displays the status shown in Figure 45.

Figure 45.System Startup Message



At the start of operation, the first four bytes of the SPI Register map are:

- BYTE 0 = 00: All motors are off.
- BYTE 1 = 00: Both motors' direction set to CW.
- BYTE 2 = 00: No motors need speed change.
- BYTE 3 = 00. Default Speed change, if needed, is negative.

Next, use the Aardvark Control Center Software to send the command 01 02 to set the direction of Motor 1 to CW and Motor 2 to CC.

- Transmitted Bytes 01 02: Set the direction of Motor 1 to CW and Motor 2 to CC.
- Received Bytes 55 00: The SPI TX register is loaded with the value 55 at the time of setup. This is always the first byte received for any command after system reset The 00 was the initial value in the BYTE 1 of the Register Map, which indicates that the default direction for both motors is clockwise.

After this command, the first four bytes of the Register Map are:

- BYTE 0 = 00: All motors are off.
- BYTE 1 = 02: Motor 1 CW, Motor 2 CC.
- BYTE 2 = 00: No motors need speed change.
- BYTE 3 = 00. Default Speed change, if needed, is negative.

Next, we send a command to turn ON Motor 1.

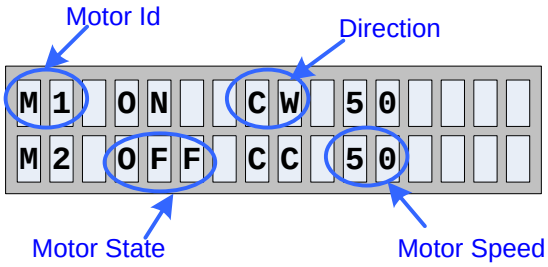
- Transmitted Bytes 00 01: Make Motor 1 state ON and Motor 2 state OFF.
- Received Bytes 00 00: The first 00 is the value pointed to by the last transmitted byte of the previous command (see SPI Communication, Page 34) i.e. BYTE 2 (last command sent was 01 02). The second 00 comes from the byte pointed to by 00 i.e. BYTE 0.

After this command, the first four bytes of the Register Map are:

- BYTE 0 = 01: Motor 1 ON, Motor 2 OFF.
- BYTE 1 = 02: Motor 1 CW, Motor 2 CC.
- BYTE 2 = 00: No motors need speed change.
- BYTE 3 = 00. Default Speed change, if needed, is negative.

Motor 1 turns on. The LCD displays the status as shown in Figure 46.

Figure 46. Motor Status Direction and Speed



The Motor Speed parameter shows the ON times of the PWM that drives the motors. The default ON time is set to 50 clock cycles (with the period being 100 clock cycles). With each speed change request, the ON time increases or decreases by 5 clock cycles, depending on the type of request.

Next, we send a speed setting command to increase the duty cycle of Motor 1 and reduce the duty cycle of Motor 2. The command is 02 03 01.

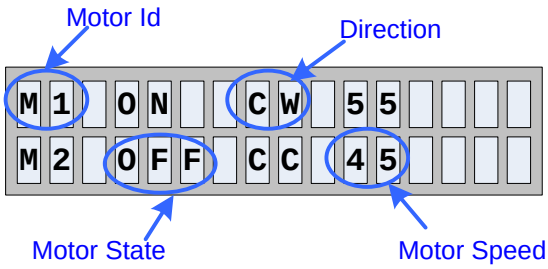
- Transmitted Bytes 02 03 01: Make Motor 1 state ON and Motor 2 state OFF.
- Received Bytes 02 00 00: The 02 comes from the byte pointed to by the last transmitted byte of the previous command i.e. BYTE 1. The first 00 (second received byte) comes from the byte pointed to by the first transmitted byte (02), i.e. BYTE 2, and the second 00 (third received byte) comes from the byte pointed to by the second transmitted byte of the current command, i.e. BYTE 3.

After this command the first three bytes of the Register Map are:

- BYTE 0 = 01: Motor 1 ON, Motor 2 OFF.
- BYTE 1 = 02: Motor 1 CW, Motor 2 CC.
- BYTE 2 = 03: Both motors need speed change.
- BYTE 3 = 01. Motor 1 needs speed increase and Motor 2 needs speed decrease.

The LCD displays the status as shown in Figure 47.

Figure 47. LCD Display After Speed Change Command



In the next few commands, we no longer explain the significance of the received bytes unless necessary. Refer to Figure 32 for more information.

We demonstrate the change of parameter through SPI and change the OverCurrent Threshold for Motor. First, We read the default value.

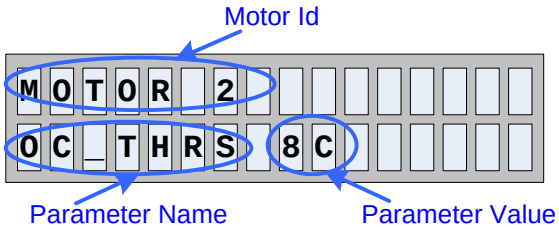
- Transmitted Bytes 84 80: Read location 84. The second byte is a dummy used to extract the BYTE 132 (0x84 = 132).
- Received Bytes 02 82: 02 comes from last transmitted command and is ignored. 0x82 is the Over Current threshold for Motor 1, i.e. 130.

Next, we send a command to modify the threshold to 140.

- Transmitted Bytes 04 8C: Write 140 (0x8C) into location 04 (`OC_Threshold` of Motor 1).
- Received Bytes 01 82: 01 comes from last byte of previous transmitted command (80) and is ignored. 0x82 was the value in BYTE 4, the default Over Current threshold for Motor 1, as verified by previous read command.

Figure 48 shows the LCD display resulting from the change of the OC_Threshold for Motor 1.

Figure 48. LCD Display After Change of OC_Threshold for Motor 1



The same display format applies to all other parameters applicable to the motors such as Over Current blanking times (OC_BLNK) and Fault Detect blanking times (FD_BLNK). The Diagnostic blanking time is not particular to any motor. When we change the Diagnostic blanking

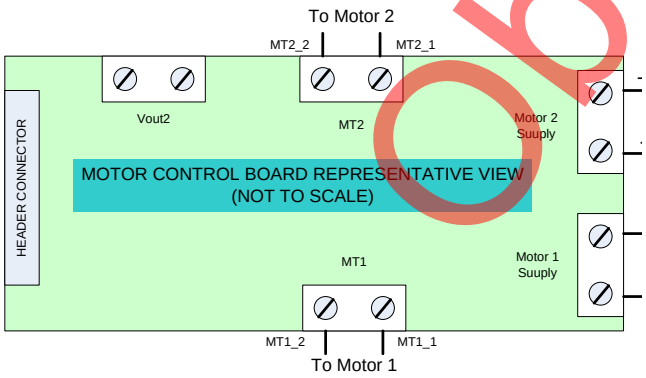
time, the LCD replaces the Motor ID with the text DIAGNOSTIC and the parameter name BLNK_TM. The parameter values are as set by the user.

Next, we demonstrate the Fault Detection. Figure 49 and Table 18 explain the faults and how we create the fault conditions. We explain the respective observations: the LCD messages, the LED and the Register Map contents.

Table 18. Demonstration of Faults and Methods to Create the Fault Conditions

Fault	Description	Method to Create Fault Condition
1	Motor 1 Over Current	Method 1: Run Motor 1, and hold the shaft with one hand to mechanically load. Method 2: Run Motor 1 in CW direction with 50% ON time, and then use a piece of wire to firmly short MT1_1 to battery. Retain the short for 3 seconds.
2	Motor 1 Shorted to Battery	Run Motor 1 in CW direction. Short MT1_2 to battery with a wire connected to battery. Retain the short for 3 seconds.
3	Motor 2 Shorted to ground	Run Motor 2 in CW direction. Short MT2_1 to ground with a wire connected to ground. Retain the short for 3 seconds.
4	Motor 2 is Open	Keep Motor 2 stopped. Motor 1 may be stopped or running. Disconnect Motor 2.

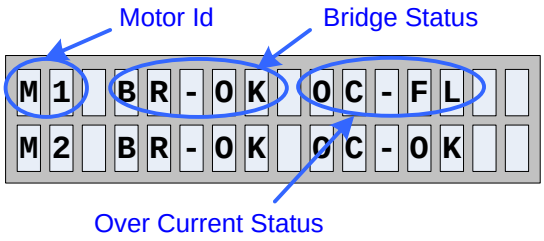
Figure 49. Representative View of Motor Control Board



Fault 1a) Motor 1 Over Current - Method 1

In this exercise, we use excessive mechanical load to create an over current condition. To do this, we hold the shaft with our hands. Motor 1 stops. If Motor 2 were running, it would also stop. The Fault LED turns ON. The LCD display is as shown in Figure 50. When the Over Current fault occurs, a diagnostic sequence runs automatically. However, the system detects no faults as there are no faults on the bridge. Therefore, the Diagnostic block does not report a fault. The fault status is available in the Register Map at BYTE 3M+133 and BYTE 3M+134. Figure 51 shows how to interpret fault status bytes.

Figure 50. LCD Display for Over Current Fault on Motor 1



The fault status is available in the Register Map at BYTE 3M+133 and BYTE 3M+134. Figure 51 shows how to interpret fault status bytes.

Figure 51. Interpretation of Fault Status Bytes

NOT USED	MOTOR 2	MOTOR 1	4	3	2	1	0
NOT USED	1 = FL 0 = OK	1 = FL 0 = OK	LOAD OPEN	SHORT TO GND		SHORT TO BAT	

BYTE 3M+133: BRIDGE FAULT

NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	MOTOR 2	MOTOR 1
NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	NOT USED	1 = FL 0 = OK	1 = FL 0 = OK

BYTE 3M+134: OVER CURRENT FAULT

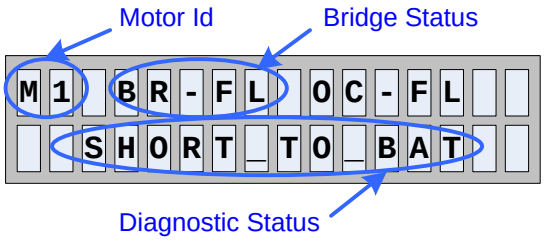
With two motors, the bytes are BYTE 139 and BYTE 140. We transmit the command: 0x8B 0x8C 0x80 to read these. As usual, the last byte is a dummy byte. The received bytes are 0x00 0x00 0x01. The first byte is ignored. The second and third bytes indicate that there are no bridge faults, no faults detected by Diagnostic, and there is Over Current Fault on Motor 1.

Fault 1b) Motor 1 Over Current – Method 2

In this exercise, we use a wire to firmly short MT1_1 to the battery. This is equivalent to suddenly creating a 100% duty cycle from a 50% duty cycle on Motor 1, and it results in an over current condition. The same fault may not have occurred if the motor was running close to 100% duty cycle prior to the short.

The LCD initially displays the same status as shown in Figure 50. But if we hold the wire in place to maintain the short, the Diagnostic routine detects it. After the Diagnostic sequence runs (around 2 seconds from the occurrence of the fault), the display changes to as shown in Figure 52.

Figure 52. LCD Display Reporting Diagnostic Fault



It shows that the Diagnostic module detected a fault on the H bridge that drives Motor 1. It indicates that the possible cause is a Short to Battery. The Over Current status shows that an Over Current fault was detected before Diagnostic began.

The Fault Status is read similarly as before by transmitting 0x8B 0x8C 0x80. The resulting received bytes are 0x00 0x21 0x01. The first byte is ignored. The second received byte indicates that there is a Fault on the H bridge for Motor 1, and it is a short to battery. The third received byte indicates that an Over Current fault has been detected on Motor 1.

Fault 2 Motor 1 Shorted to Battery

Motor 1 is run in CW direction, which sets MT1_2 LOW, and a PWM is set on MT1_1. To create the fault, short MT1_2 to battery. All motors stop, and the LCD display is as shown in Figure 53.a. After the Diagnostic runs, the display changes to Figure 53.b.

Figure 53. LCD Display for Motor 1 Shorted to Battery

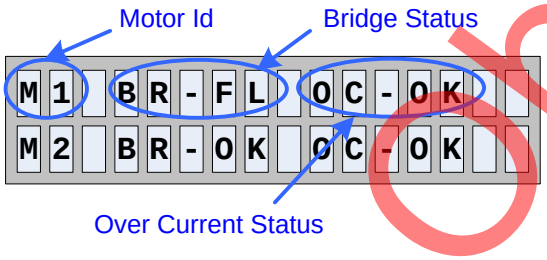


Figure 53.a

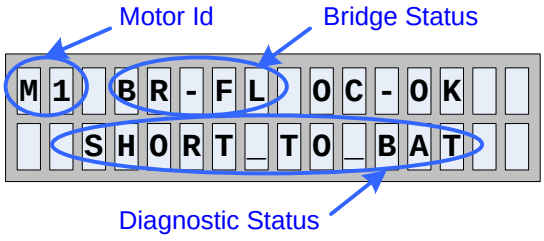
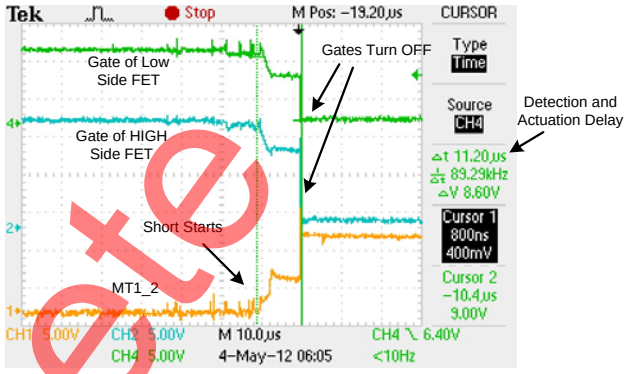


Figure 53.b

The fault status is read by transmitting 0x8B 0x8C 0x80. The resulting received bytes are 0x00 0x21 0x00.

Figure 54 shows a scope shot of the fault condition. The figure shows the active gates of the two FETs and the voltage at terminal MT1_2. Under normal circumstances, the voltage should be close to ground. As soon as it is shorted, the voltage starts to increase. The detection of the fault and the resulting actuation (gate turn OFF) occur within 11.2 µs from the occurrence of the fault. This is with a Fault Detection blanking time of 10 µs. For faster detection and actuation times, reduce the blanking time.

Figure 54. Scope Shot of Fault Detection and Actuation Delay



Fault 3 Motor 2 Shorted to Ground

In this exercise, Motor 2 starts with direction set to CW. Then use a wire connected to ground to short the terminal MT2_1 to ground. Retain the short for 3 seconds. All motors stop immediately. The resulting LCD displays before and after Diagnostic are shown in Figure 55.a and Figure 55.b respectively.

Figure 55. LCD Display for Motor 2 Shorted to Ground

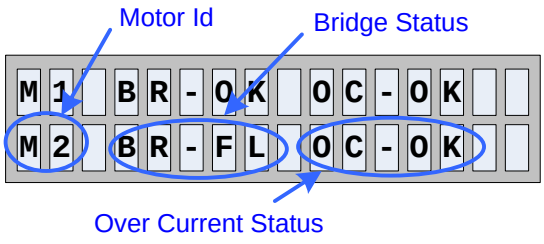


Figure 55.a

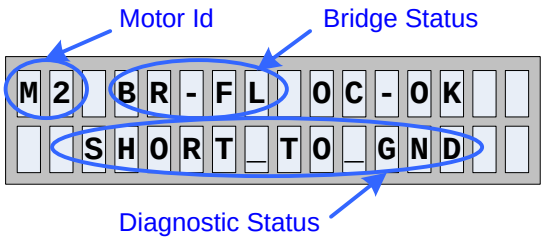


Figure 55.b

The resulting status from the Register Map is extracted by transmitting bytes 0x8B 0x8C 0x80. The received bytes are 0x00 0x44 0x00. The significant part of the received byte indicates that there are no over current faults and that M2 is shorted to ground. During faults, over current conditions may be created. So, in some cases, the Over Current status may be set to Faulty in case of a short to battery or Short to Ground faults.

Fault 4 Motor 2 is Open

Disconnect Motor 2 from the Motor Control Board. Motor 1 may be active or stopped. Within 2 seconds, all motors stop. The LCD display is as shown in Figure 56. The corresponding fault status read from the Register Map is 0x00 0x50 0x00 to indicate that there is a load open fault on Motor 2.

Figure 56. LCD Display for a Load Open Fault on Motor 2

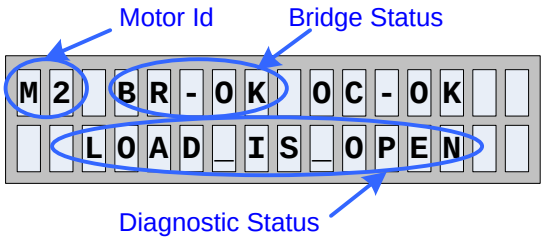


Figure 57. LCD Display for System Lock and System Release

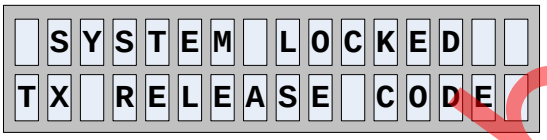


Figure 57.a

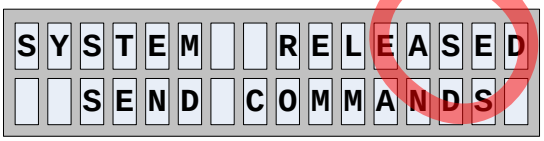


Figure 57.b

Table 19. Faults Detectable by the System

Fault Motor 1 or 2	Condition
Shorted to battery	Detectable when Motor 1 or 2 is either active or stopped.
Shorted to ground	Detectable when Motor 1 or 2 is either active or stopped.
Over Current	Detectable when Motor 1 or 2 is either active or stopped.
Open	Detectable only if the motor is stopped.

If any command other than a Diagnostic request or a system release command is transmitted following a fault condition, the LCD displays that the system is locked (Figure 57.a). On sending the correct release command with the correct release code (0x0E 0xCC for the associated project), the LCD displays that the system has been released (Figure 57.b). Table 19 shows the list of faults that the system can detect.

Summary

In this application note, we introduced an alternate method to protect an H bridge based motor drive that does not require smart drivers. We illustrated methods to use simple components to drive high side. We discussed the different faults that may occur in a motor drive, leading to operations outside the safe operating area of the switches. We proposed several methods to detect and prevent such faults, particularly the use of the Diagnostic sequence to periodically examine the H bridge and the load when not in operation. We also presented a scalable architecture to implement these methods as well as demonstrated a full implementation of the system using PSoC 3. The example exercise used a motor control board and two brushed DC motors with significantly different electrical and mechanical characteristics to fully demonstrate the ease of configurability of the system for different motors.

Related Application Notes

AN54181 - PSoC® 3 - Getting started with a PSoC® 3 design project

AN60580 - SIO Tips and Tricks in PSoC® 3 / PSoC 5

AN52705 - PSoC® 3 and PSoC® 5 - Getting Started with DMA

Appendix

The Appendix provides information about:

- Demonstration Setup for CY8CKIT-030
- Motor Control Board

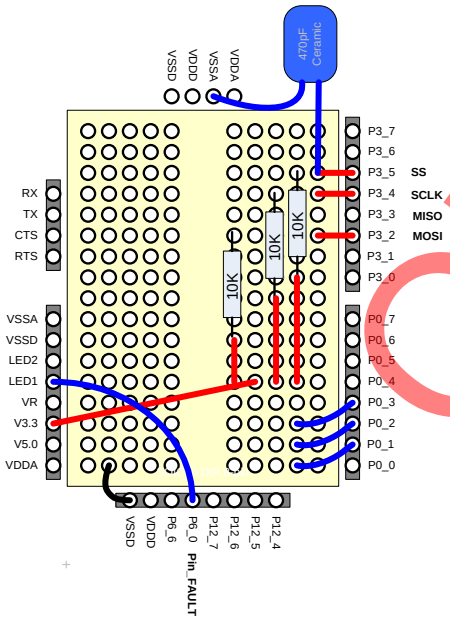
Setup Details for CY8CKIT-030

You can substitute the CY8CKIT-030 kit for the CY8CKIT-001 kit in the examples described in this. The corresponding project is AN75813 CY8CKIT-030. The Motor Control Board must plug into Port E of the kit. The kit and Motor Control Board settings are as follows:

- CY8CKIT-030
  - Jumper J10 (VDDD SEL) set to 3.3 V
  - J11 (VDDA SEL) set to 3.3 V
- Motor Control Board
  - J2 and J4 are in place (connected)
  - J3 and J6 are in place (connected)
  - J5 is set to Vin1

Set up the prototyping area as shown in Figure 58.

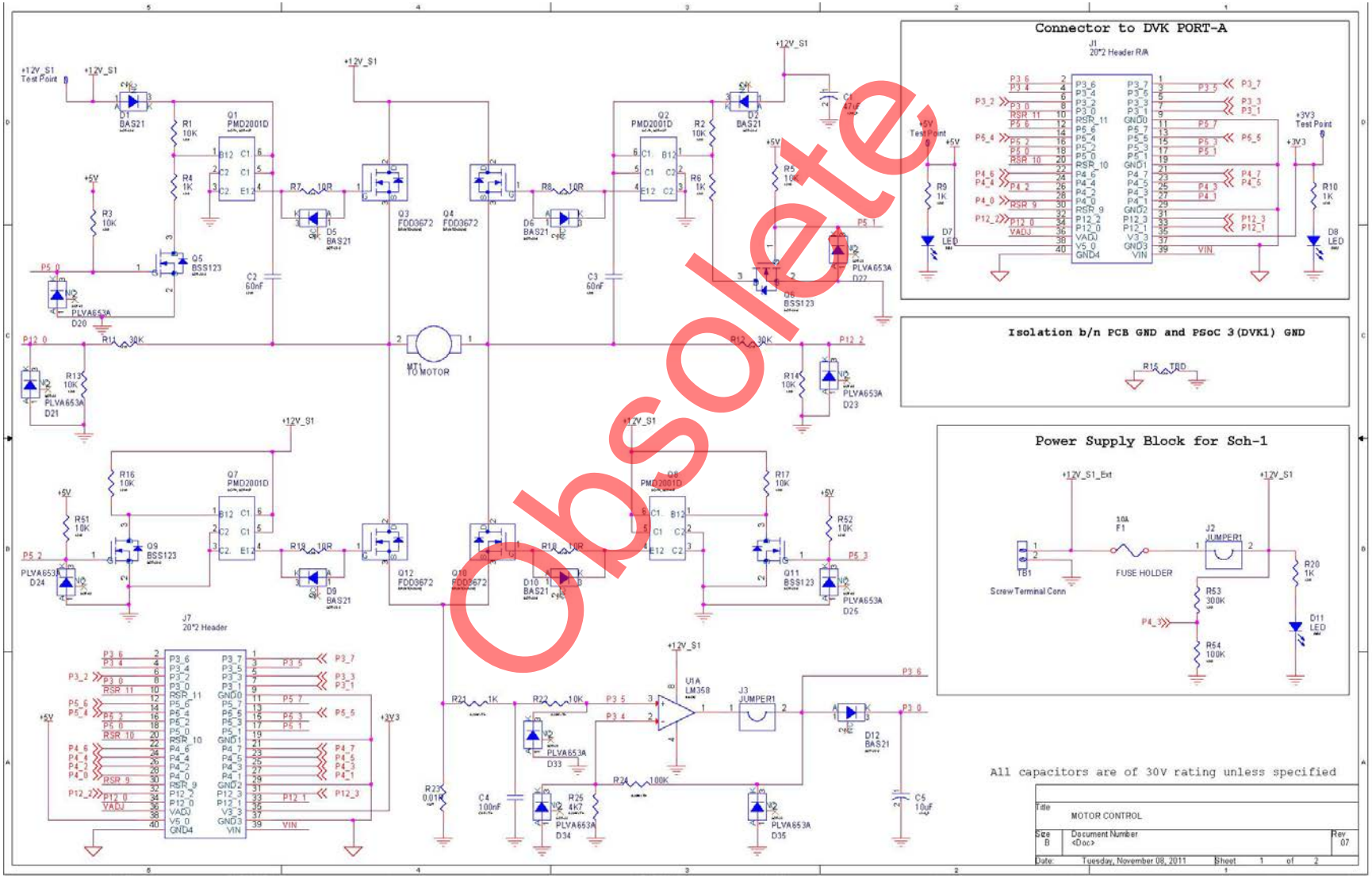
Figure 58. Prototyping Area Setup for CY8CKIT-030

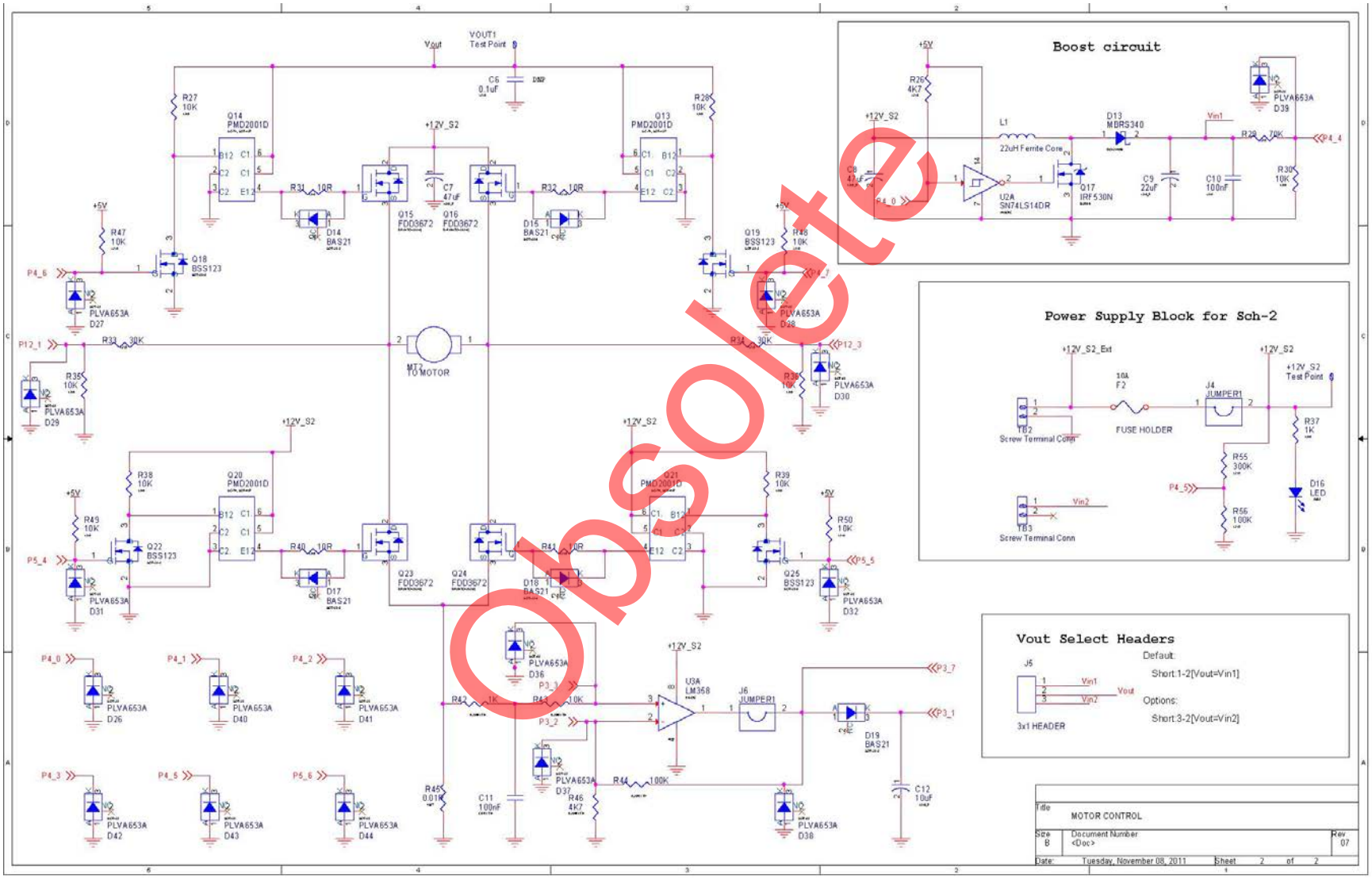


The remaining procedure to set up the demonstration is identical to what has been already presented in the Demonstration section.

Motor Control Board

For reference, the schematic of the Motor Control Board is attached.




Boost circuit
Power Supply Block for Sch-2
Vout Select Headers
MOTOR CONTROL
Obsolete

Document History

Document Title: H Bridge Based Motor Drive Protection Using PSoC® 3 - AN75813

Document Number: 001-75813

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3702878	SMON/VEDTMP2	08/06/2012	New application note.
*A	4719658	KUK	04/10/2015	Obsolete document.

Worldwide Sales and Design Support

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

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com