Application Note AN-80 BridgeSwitch Family

BridgeSwitch FAULT Communication Interface

Introduction

This application note describes the software implementation guide for the BridgeSwitch™ FAULT status communication interface feature. It includes the overview of the BridgeSwitch FAULT status communication interface, the state machine that captures and processes received status updates, the reference code and its data structures, and the demonstration of the software by displaying status updates through UART terminal as well as an example fault protection implementation in an inverter board.

BridgeSwitch FAULT Status Communication Interface

A BridgeSwitch device can communicate status updates, including internal and system level faults to the system MCU through its open drain FAULT output. It uses a 7-bit word pattern followed by an odd parity bit to report a status update.

The following sections describe the FAULT bus specifications in detail.

Hardware Configuration

To communicate all detected status updates to the system micro-controller, all FAULT pins connects to a single-wire bus pulled up to the system supply voltage. Figure 1 shows the typical interface of three BridgeSwitch devices to the system MCU in a single-wire bus configuration.

The device ID pin connection allows each device to assign itself a unique device ID depending on the device ID pin connection. This device ID allows communicating the physical location of a detected fault condition to the system micro-controller by pulling the FAULT bus low for the respective device ID period t_{ID} at the start of status communication.

Table 1 lists the device ID, resulting device ID time period t_{ID}, and how to program the respective ID through ID pin connection.

Device ID	t_{ID}	ID Pin Connection
1	40 µs	BPL Pin
2	60 µs	Floating
3	80 µs	SG Pin

Table 1 – Device ID Selection through ID Pin.

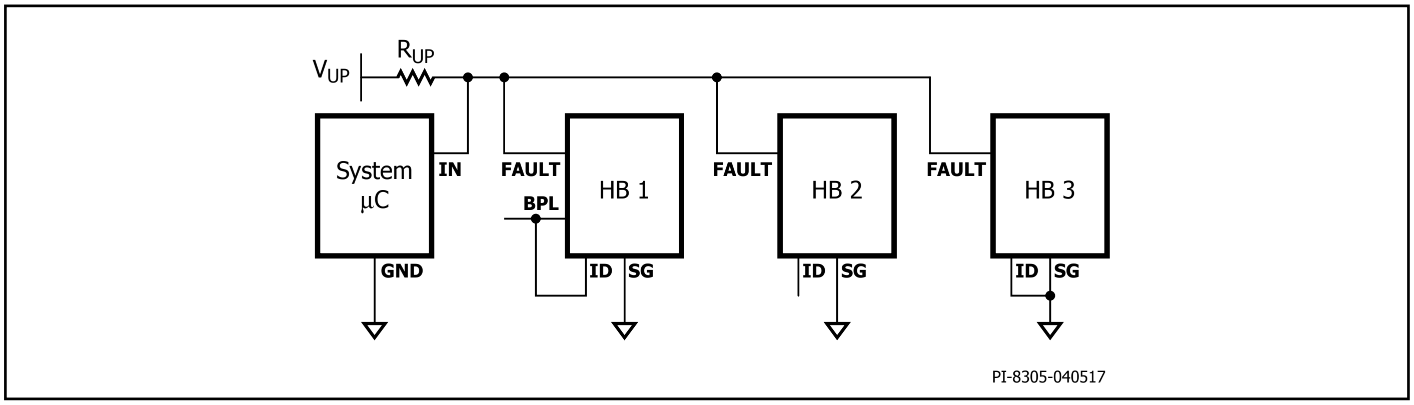


Figure 1. Single Wire Status Communication Bus with Device ID Programming.

FAULT Status Communication Bus Specification

Status Encoding

The 7-bit word followed by a parity bit encodes the FAULT information. Table 2 summarizes encoding of various status updates the device may communicate to the system micro-controller. The status word consists of five blocks with status changes grouped together that cannot occur at the same time. This enables simultaneous reporting of multiple status updates to the system micro-controller without having to take care about fault priorities and a fault-reporting queue.

The last row (7-bit word "000 00 0 0") encodes Device Ready status and is used to communicate a successful power-up sequence to the system, communicated when a certain fault is cleared and sent it to acknowledge a status request the system MCU in case no fault is present.

Communication on the FAULT bus initiates for one of the following reasons:

- Ready for mission mode communication after a successful power-up.
- A FAULT status register update communication initiated by one of the devices.
- A current status communication following a query of the system micro-controller.

FAULT	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6
HV bus OV	0	0	1				
HV bus UV 100%	0	1	0				
HV bus UV 85%	0	1	1				
HV bus UV 70%	1	0	0				
HV bus UV 55%	1	0	1				
System thermal fault	1	1	0				
LS Driver not ready[1]	1	1	1				
LS FET thermal warning				0	1		
LS FET thermal shutdown				1	0		
HS Driver not ready[2]				1	1		
LS FET over-current						1	
HS FET over-current							1
Device Ready (no faults)	0	0	0	0	0	0	0

Notes:
1. Includes XL pin open/short-circuit fault, IPH pin to XL pin short circuit, and trim bit corruption.
2. Includes HS-to-LS communication loss, V_{BPH} or internal 5 V rail out of range, and XH pin open/short-circuit fault.

Table 2. Status Word Encoding.

Figure 2 depicts the FAULT interface communication flowchart for all of those three cases.

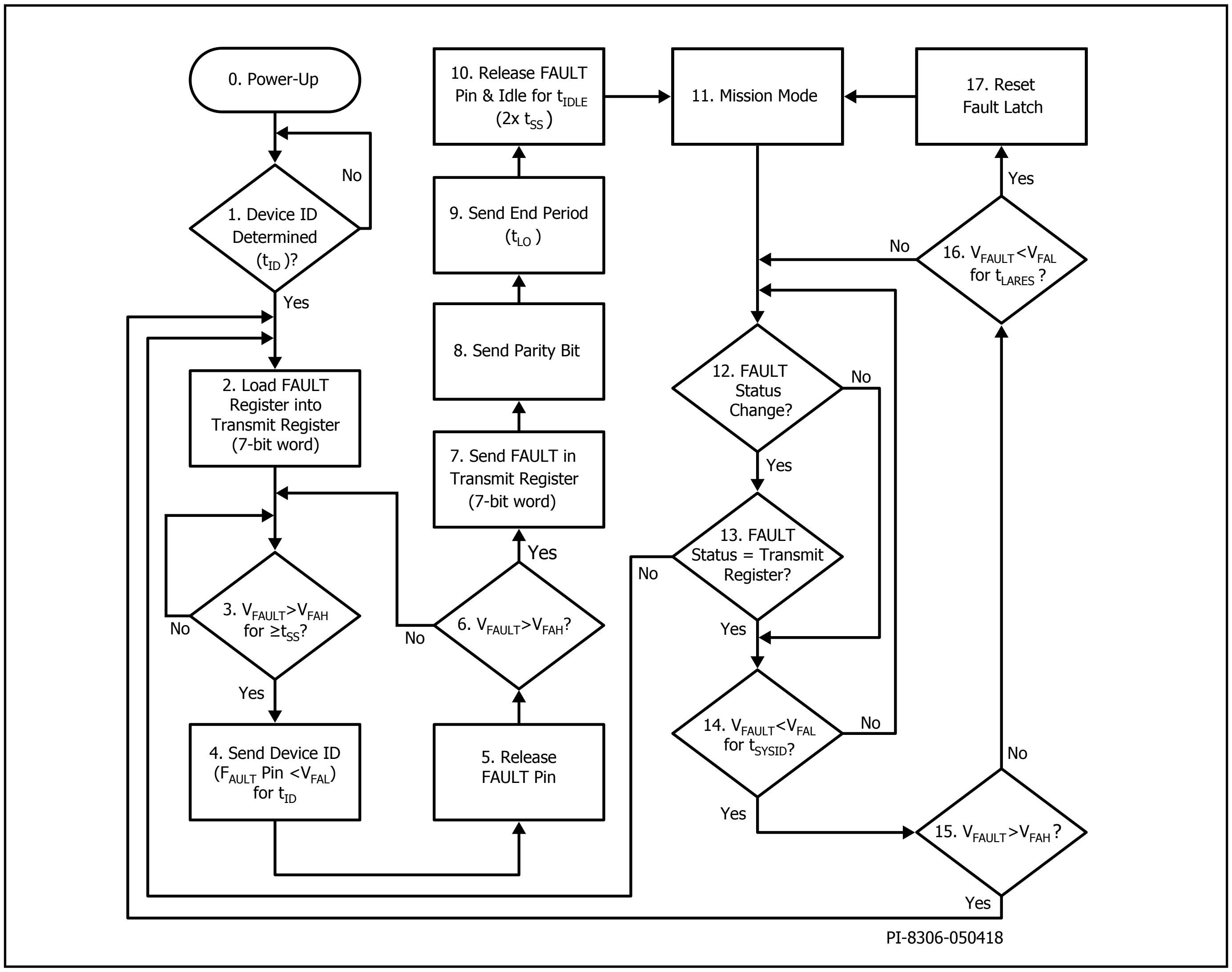

0. Power-Up
1. Device ID Determined (t_{ID})?
No
Yes
2. Load FAULT Register into Transmit Register (7-bit word)
3. $V_{FAULT}>V_{FAH}$ for $\geq t_{SS}$?
No
Yes
4. Send Device ID (F_{AULT} Pin $<V_{FAL}$) for t_{ID}
5. Release FAULT Pin
6. $V_{FAULT}>V_{FAH}$?
No
Yes
7. Send FAULT in Transmit Register (7-bit word)
8. Send Parity Bit
9. Send End Period (t_{LO})
10. Release FAULT Pin & Idle for t_{IDLE} (2x t_{SS})
11. Mission Mode
12. FAULT Status Change?
No
Yes
13. FAULT Status = Transmit Register?
No
Yes
14. $V_{FAULT}<V_{FAL}$ for t_{SYSID}?
No
Yes
15. $V_{FAULT}>V_{FAH}$?
No
Yes
16. $V_{FAULT}<V_{FAL}$ for t_{LARES}?
No
Yes
17. Reset Fault Latch
PI-8306-050418

Figure 2. Status Communication Flowchart.

A status update communication always starts with bus arbitration initiated by the communicating device. It transmits the respective device ID Time Period t_{ID} by pulling down the FAULT pin if the bus has been quiet for at least 80 µs. After the device has won bus arbitration the current fault register (7-bit word) followed by the odd parity bit and the transmission end signal is sent as depicted in the communication flowchart (Figure 2).

Bit Stream Timing

Figure 3 depicts the bit stream timing diagram BridgeSwitch uses for a status update communication. The two logic states are encoded with two different voltage signal high-time periods at the FAULT pin followed by a low-time period t_{LO} (typically 10 µs). A logic "1" is encoded with a period t_{BIT1} (typically 40 µs) and a logic "0" is encoded with a period t_{BIT0} (typically 10 µs). Table 3 lists the timing table of the FAULT status communication.

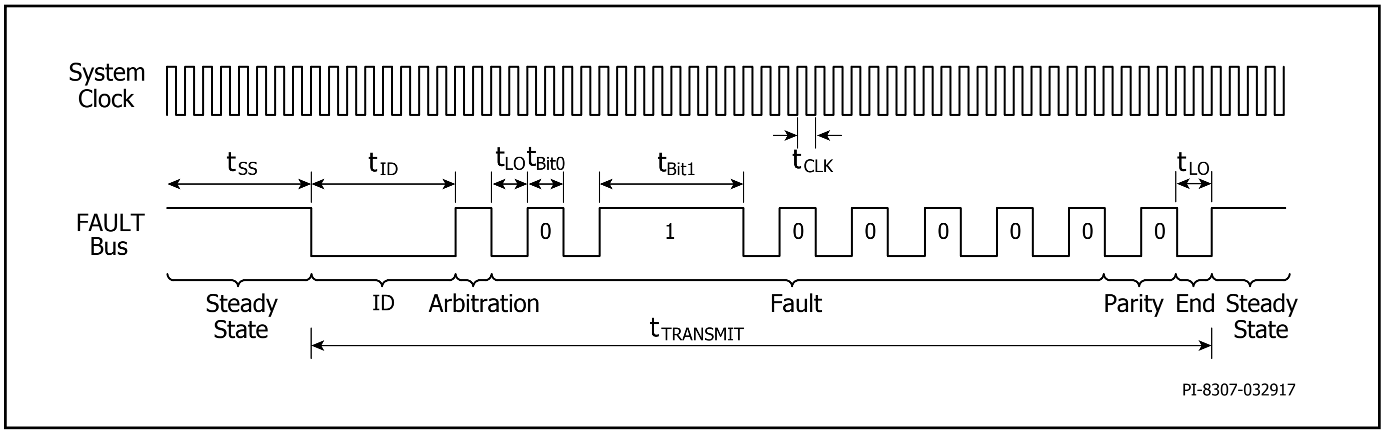


Figure 3. Status Communication Bit Stream.

Symbol	Description	Logic State	Period (Typical)
t_{ID}	Device ID	0	Refer to Table 1
t_{LO}	Low-time	0	10 µs
t_{Bit0}	Logic 0	1	10 µs
t_{Bit1}	Logic 1	1	40 µs

Table 3. Bits Stream Timing Table.

After each completed transmission the device will idle for t_{IDLE} (typically 2 × t_{SS} = 160 µs) before starting a new communication. This enables other devices on the bus to communicate a possible occurred status change or to respond to a status inquiry sent by the system micro-controller.

The device communicates each detected status update only once. It also reports a status change for all system level faults to the system MCU. This includes DC bus undervoltage and overvoltage conditions and external temperature monitor faults. It also reports all status level changes for device internal faults with the exception of the LS power FREDFET thermal shutdown.

The BridgeSwitch device also monitors the FAULT bus for possible commands sent by the system micro-controller once it is in mission mode. This could be a status update query (see step 15 in Figure 2) by the micro-controller through its pulling the bus low for a period of t_{SYSID} (typically 160 µs). Or it could be a command to reset the device status register including over-temperature shutdown latch and to enter the power-up sequence mode (see step 17 in Figure 2) by pulling the FAULT bus low for a period of t_{LARES} (2x t_{SYSID} = typically 320 µs). A power-up sequence is recommended after the MCU has sent a latch reset command. Table 4 summarizes available system micro-controller commands.

Bus Pulldown Period	Command
t_{SYSID}	Status Query
$t_{LARES(2xTSYSID)}$	Status register including over-temperature latch reset and power-up sequence mode

Table 4. System MCU Commands.

Software Implementation

This section describes the implementation of a state machine that captures and handles status updates from each BridgeSwitch device based on the status communication specification described in the previous section.

The presented example uses an interrupt based implementation. The user has to decide interrupt priority based on their specific application requirements, such as motor control algorithm or type of micro-controller.

System MCU Peripherals

To demonstrate the FAULT bus communication interface implementation, the reference code was developed using Cypress PSoC Creator IDE version 4.1 and tested on the Cypress PSoC 4 MCU (CY8CKIT-042 PSoC Pioneer Kit). The MCU board provides an onboard programmer and debugger interfaced through USB connector to communicate to PC. The state machine was demonstrated by printing received status updates on a UART console depicted in the fault detection example section.

The FAULT status communication bus connects to a single bi-directional MCU pin in an open Drain drive mode. The pin is attached to a timer which captures both rising and falling edges of the signal. The handling of the FAULT signal is interrupt based using two 16-bit timer/counter blocks *Bit_counter_timer* and *ID_counter_timer* with a 12 MHz clock. The *Bit_counter_timer* captures signals from rising-to-falling edge while the *ID_counter_timer* captures signals from falling-to-rising edge. These two timers capture the count value and generate an interrupt when falling edge and rising edge of the fault signal has been received respectively. The fault state machine routine handles each received interrupt.

Software Description

The software implementation begins by initializing *fault_bus_state* variable to idle state *(STEADY_STATE)*. The *fault_bus_state* variable captures the state of the FAULT signal once an interrupt is received. The initialization function *init_fault_bus_interrupt()* initializes the timer/counter capture port and enables the capture interrupt for both the rising and falling edges. Whenever an interrupt service routine (ISR) is triggered, *fault_detect()* function will be called. This function is the fault state machine routine that captures and processes received faults. The high-level view of the main SW flow is depicted in Figure 5.

The fault state machine routine basically handles the ISR events and updates *fault_bus_state* variable based on the current state in the fault processing. The fault machine states are *STEADY_STATE, ID_DET, ARBITRATION, T_LO and BIT_DETECT*. The detailed SW flow diagram of the fault state machine is depicted in Figure 6. Whenever a complete packet of fault status has been received with no parity error it calls the fault processing function *fault_process()*, else re-synchronization takes place where the *fault_bus_state* is reset to *STEADY_STATE*.

The fault processing function decodes the received FAULT status update and invokes required action. For example, shutting down the inverter after receiving an over-current fault or reducing inverter output power after receiving a thermal warning status update. The FAULT status is stored in *fault* variable, which updates every time a new status update has been received. The user should provide the necessary action or decision of what the MCU should do depending on the FAULT status that was received and the application requirement. Table 5 lists the exemplary actions the system micro-controller could take after receiving a STATUS update. The *fault_process()* function SW flow diagram is depicted in Figure 7.

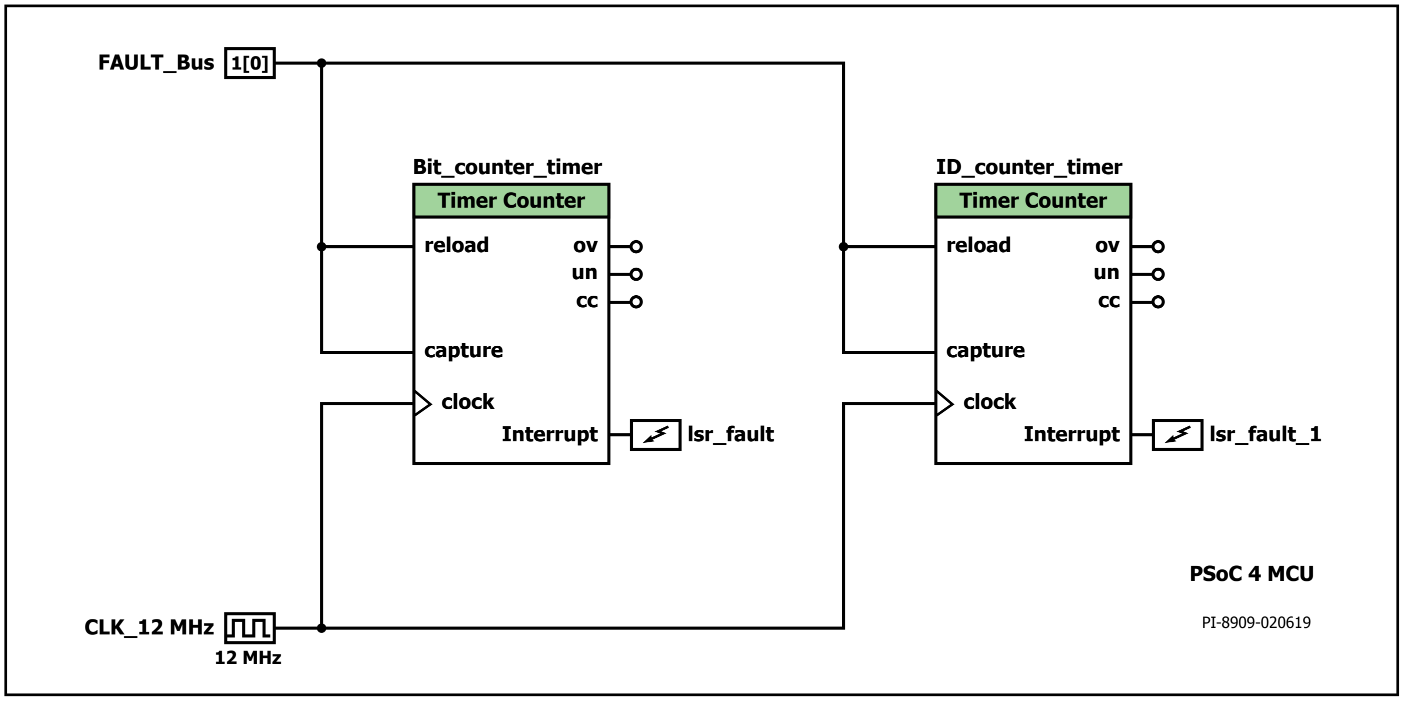


Figure 4. System MCU Peripherals for Fault Signal Handling Example with PSoC 4 MCU.

Status Query and Latch Reset Command

The software implementation of status query and latch reset commands supports the following example use cases:

Status query command:
MCU may send a status query every time it wants to restart the inverter (i.e. send PWM signals) after the inverter was off for some time. For example after a reported line overvoltage or an over-current fault. The main purpose for the status query is to check if all devices are ready or if the MCU has to initiate a power-up sequence. Figure 8 shows the flowchart for the status query example implementation.

The status query routine checks the status of each BridgeSwitch device to determine which of the following conditions applies:

A. Each device responds with a status READY, (no faults are present): The MCU invokes a RESTART command to restart the inverter and sends PWM signals to control inputs of BridgeSwitch.
B. One or more devices respond with a High-side driver not ready fault: The MCU invokes a START UP command, which initiates a power-up sequence to charge the High-side driver supply voltages (V_{BPH} with respect to HB pin) to its nominal value. After completion of the power-up sequence, the MCU follows up with another status query command. If all devices respond with READY, the MCU invokes a RESTART command to restart the inverter. If any of the devices respond with a status other than READY, the inverter will remain in shutdown mode.

The status query command is handled by the *status_query()* function that pulls the FAULT bus low for t_{SYSID} = 160 μs (refer to Table 4). Each device follows this command and will send successively their status. After a status query is sent by the system micro-controller, the detected fault status will be processed by the *process_status_query_command()* function (placed inside the fault bus state machine function), which will store each detected device status and process them in the *status_query_action()* function. The *status_query_action()* function checks the received faults status and provides actions following the conditions as described in Table 5.

Latch reset command:
MCU could send a latch reset command some time after one (or all) of the devices had reported an over-temperature fault (and latched off). Figure 9 shows the flow chart for the latch reset command example. A power-up sequence is recommended after a latch reset command. This ensures that the bypass high-side voltage is at the nominal level before switching resumes.

The latch reset command is handled by the *latch_reset()* function that pulls the FAULT bus low for t_{LARES} = 320 μs (refer to Table 4) to reset each device status. A power-up sequence is invoked after a latch reset command is sent by the system micro-controller. Note that the fault detect function must be disabled whenever a latch reset command is sent.

Fault	Status Word	Software Action/Decision	Note
High-Voltage Bus OV	001 xxx x	Shutdown	Typically only one device monitors HV bus, MCU shuts down entire inverter.
High-Voltage Bus UV 100%	010 xxx x	None	MCU could increase motor output to nominal power – if previous status update was UV85, UV70, or UV50.
High-Voltage Bus UV 85%	011 xxx x	Warning	MCU could decrease motor output power (speed/torque) to reduce inverter load.
High-voltage Bus UV 70%	100 xxx x	Warning	MCU could decrease motor output power (speed/torque) to reduce inverter load.
High-Voltage Bus UV 55%	101 xxx x	Warning	MCU could decrease motor output power (speed/torque) to reduce inverter load.
System Thermal Fault	110 xxx x	Warning/Shutdown	Depends which external component temperature is monitored.
LS Driver Not Ready	111 xxx x	Shutdown	MCU could attempt an inverter restart after some time to check if fault has cleared.
LS FET Thermal Warning	xxx 010 x	Warning	MCU could decrease motor output power (speed/torque) to reduce inverter load or to limit PCB temperature.
LS FET Thermal Shutdown	xxx 10x x	Shutdown	Latching shutdown may occur only at one device, MCU should shut down entire inverter, MCU could make an inverter restart attempt after a cooling off period.
LS FET Over-Current	xxx xx1 x	Shutdown	Device automatically turn-off respective FREDFET to protect motor from a stall or over-load condition. MCU shuts down entire inverter.
HS Driver Not Ready	xxx 11x x	Shutdown	MCU could attempt an inverter restart after some time (a few seconds) to check if fault has cleared.
HS FET Over-Current	xxx xxx 1	Shutdown	Device automatically turn-off respective FREDFET to protect motor from a stall or over-load condition. MCU shuts down entire inverter.
Device Ready (no faults)	000 000 0	None	MCU could restart inverter after previous status update was HV Bus OV or LS/HS FET over-current. It could also increase motor power to nominal if previous status update was Thermal Warning.

Table 5. Exemplary Actions Taken by the Micro-controller After Receiving STATUS updates.

Software Flowchart

Below flowchart illustrates a high level view of the software for the FAULT signal handling.

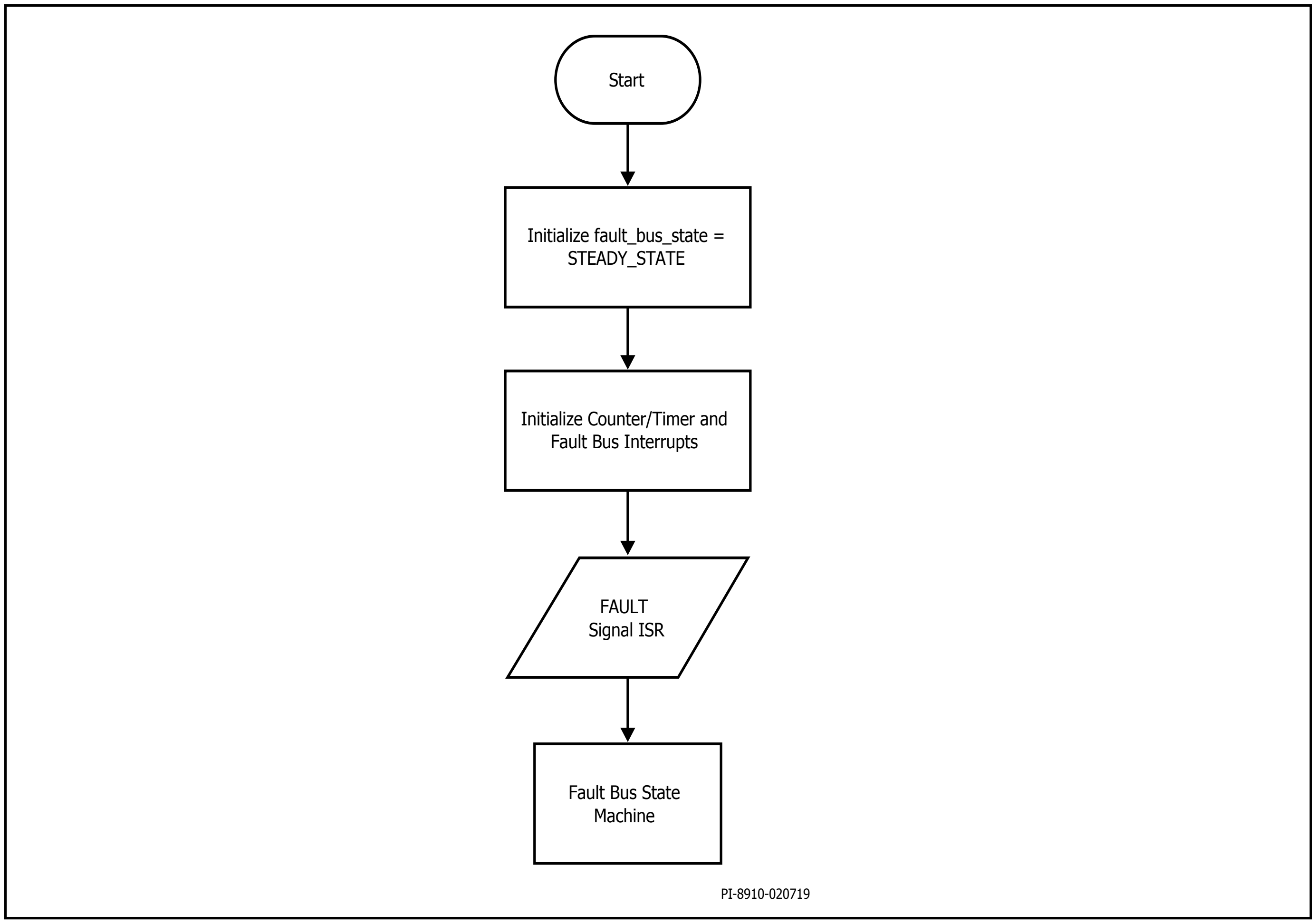

Start
Initialize fault_bus_state = STEADY_STATE
Initialize Counter/Timer and Fault Bus Interrupts
FAULT Signal ISR
Fault Bus State Machine
PI-8910-020719

Figure 5. High-Level View of the Fault Bus Implementation in Software.

Fault Bus State Machine

Figure 6 illustrates the SW flowchart of the fault bus state machine.

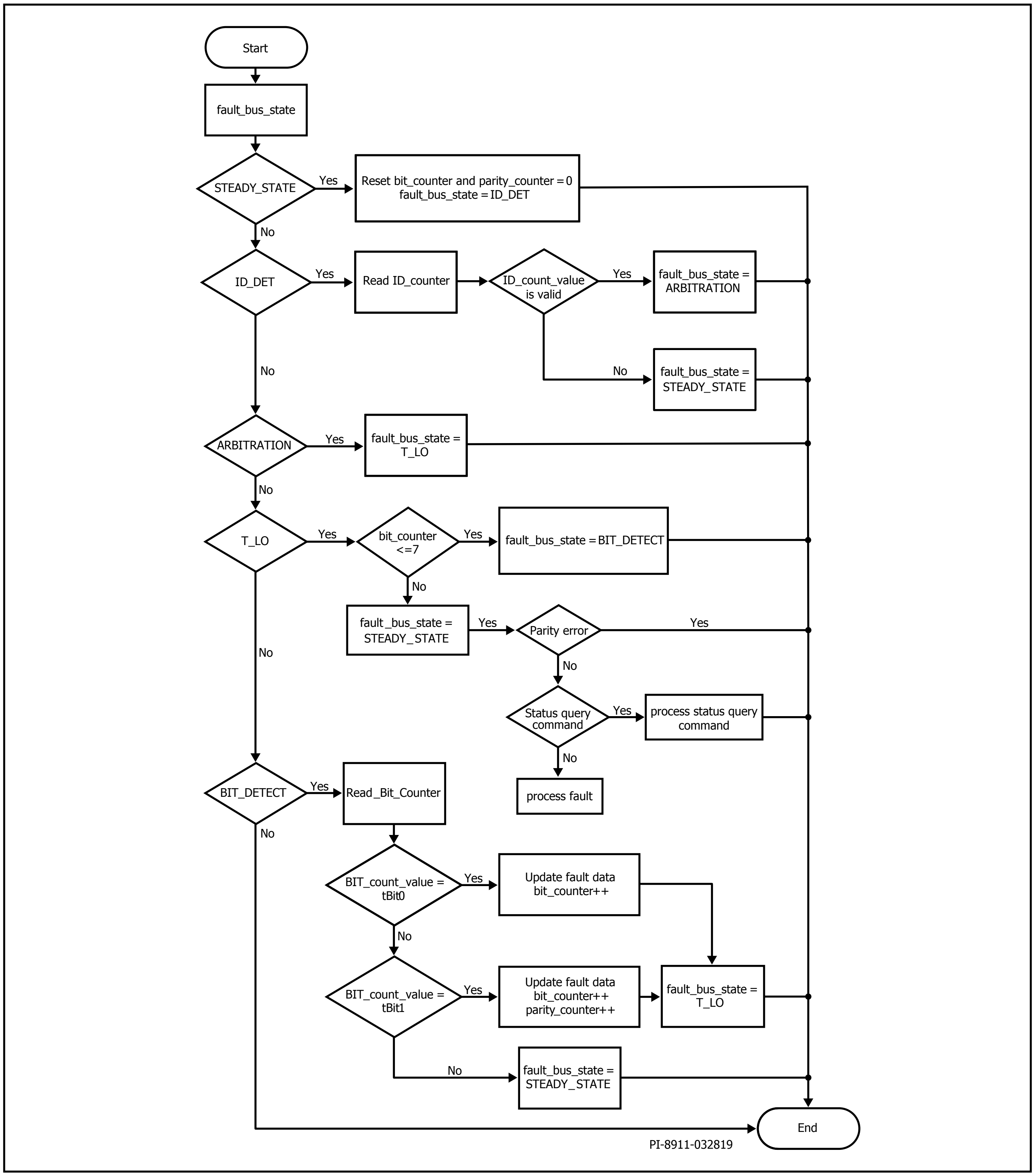


Figure 6. Fault Bus State Machine.

Figure 7 illustrates the SW Flowchart of the Fault Processing Function.

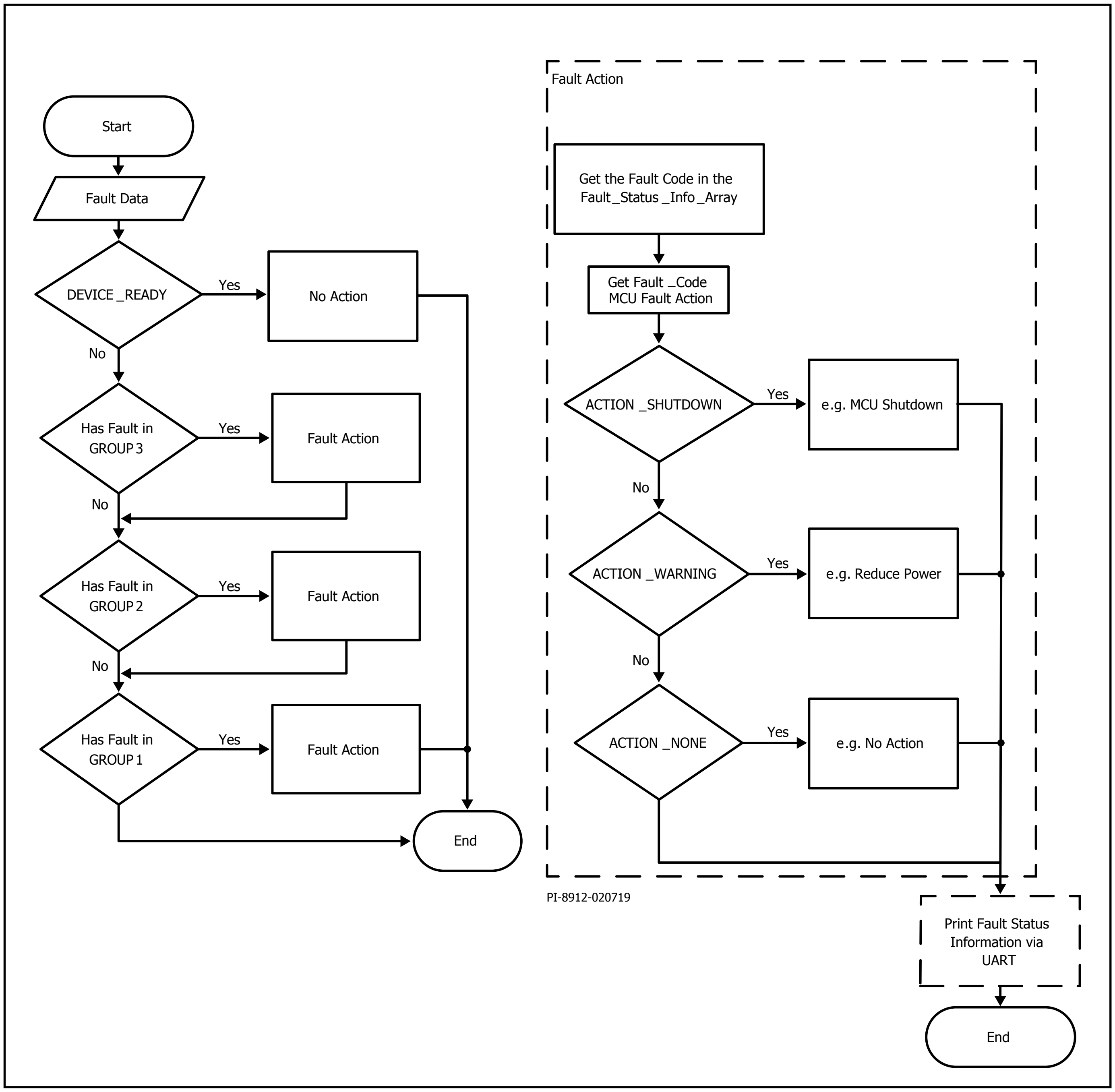


Figure 7. Fault Processing Function.

The fault data received may contain more than one type of status update and the fault processing function should be able to handle each fault type. Fault bits, which cannot occur at the same time, are grouped together to determine the fault type. Table 6 lists the status word groups. A fault from *GROUP1*, *GROUP2*, *Low-side FET over-current* and *High-side FET over-current* can be reported simultaneously within a single status word.

GROUP	FAULT	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6
GROUP1	HV bus OV	0	0	1				
	HV bus UV 100%	0	1	0				
	HV bus UV 85%	0	1	1				
	HV bus UV 70%	1	0	0				
	HV bus UV 55%	1	0	1				
	System thermal fault	1	1	0				
	LS Driver not ready[1]	1	1	1				
GROUP2	LS FET thermal warning				0	1		
	LS FET thermal shutdown				1	0		
	HS Driver not ready[2]				1	1		
LS FET over-current							1	
HS FET over-current								1

Table 6. FAULT Status Groups.

In this exemplary implementation, the fault action function will be called for every reported fault type. The fault action function picks from the *FAULT_STATUS_INFO_ARRAY* a specific fault action corresponding to the fault code, which the MCU then subsequently executes. Refer to Table 5 for a list of possible specific actions. Actions following reported status updates need to be adjusted depending on specific application requirements.

The Fault Detection Example paragraph in this document demonstrates the status update decoding by means of displaying the fault status via UART console. The paragraph also illustrates specific actions the MCU takes in a 3-phase inverter board.

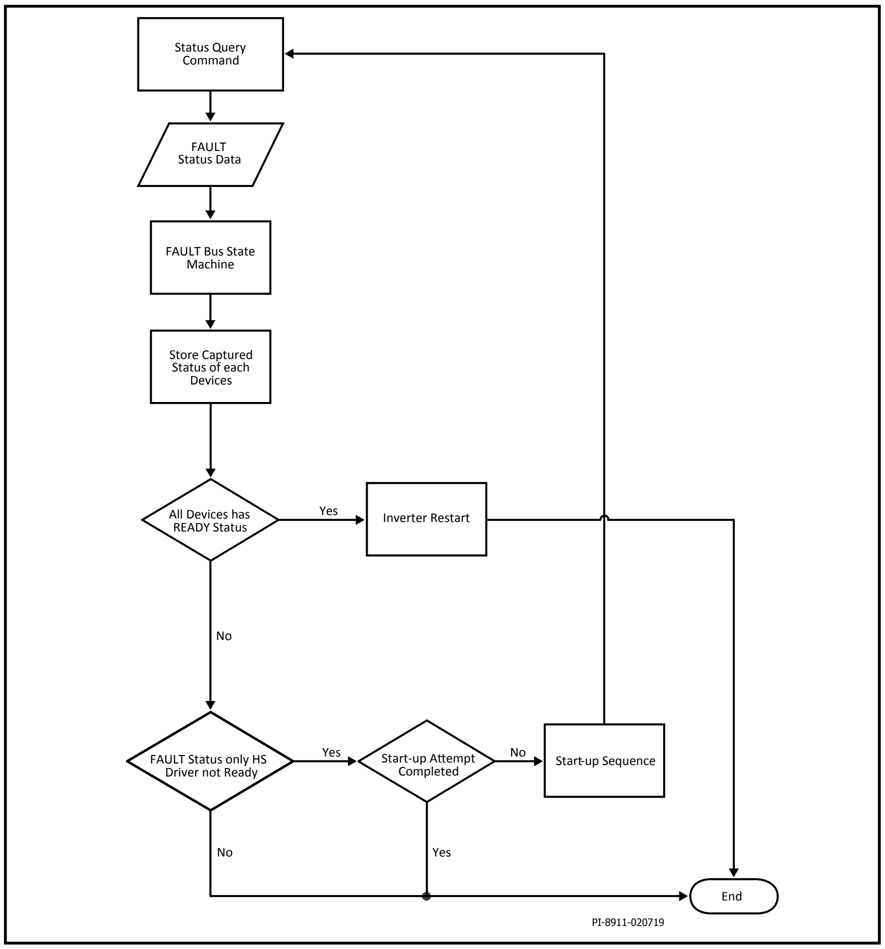

Status Query Command
FAULT Status Data
FAULT Bus State Machine
Store Captured Status of each Devices
All Devices has READY Status
Yes
Inverter Restart
No
FAULT Status only HS Driver not Ready
Yes
Start-up Attempt Completed
No
Start-up Sequence
No
Yes
End
PI-8911-020719

Figure 8. Status Query Command Processing Function.

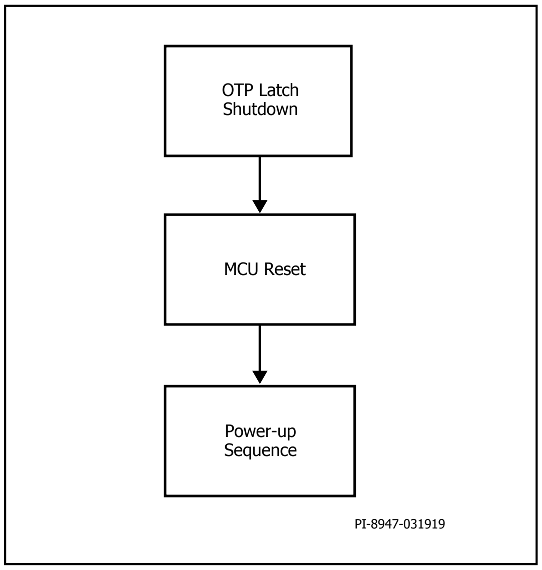


Figure 9. Latch Reset Command Function.

Reference Code Data Structures

Fault State Machine States

Below are the fault bus states which determine the current fault signal state during detection process.

```
STEADY_STATE =0,
ID_DET,
ARBITRATION,
T_LO,
BIT_DETECT,
```

Fault_Status_Info_Array

This FAULT_STATUS_INFO array is a list of specific actions following a decoded fault status update.

```
{HV_BUS_OV, ACTION_SHUTDOWN},
{HV_BUS_UV_100, ACTION_NONE},
{HV_BUS_UV_85, ACTION_WARNING},
{HV_BUS_UV_70, ACTION_WARNING},
{HV_BUS_UV_55, ACTION_WARNING},
{SYSTEM_THERMAL_FAULT, ACTION_SHUTDOWN},
{LS_DRIVER_FAULT, ACTION_SHUTDOWN},
{LS_FET_THERMAL_WARNING, ACTION_WARNING},
{LS_FET_THERMAL_SHUTDOWN, ACTION_SHUTDOWN},
{HS_DRIVER_FAULT, ACTION_SHUTDOWN},
{LS_FET_OVERCURRENT, ACTION_SHUTDOWN},
{HS_FET_OVERCURRENT, ACTION_SHUTDOWN},
```

For example, the entry for overvoltage {HV_BUS_OV, ACTION_SHUTDOWN} indicates that the MCU should shut down the system when this error occurs.

In this implementation, the specific actions following a fault status update are:
ACTION_SHUTDOWN,
ACTION_WARNING,
ACTION_NONE,

Fault Status Codes
Fault states which can occur are:

```
//GROUP1 FAULTS
HV_BUS_OV = 4u,
HV_BUS_UV_100 = 2u,
HV_BUS_UV_85 = 6u,
HV_BUS_UV_70 = 1u,
HV_BUS_UV_55 = 5u,
SYSTEM_THERMAL_FAULT = 3u,
LS_DRIVER_FAULT = 7u,

//GROUP2 FAULTS
LS_FET_THERMAL_WARNING = 16u,
LS_FET_THERMAL_SHUTDOWN = 8u,
HS_DRIVER_FAULT = 24u,

//LS FET OVERCURRENT
LS_FET_OVERCURRENT = 32u,

//HS FET OVERCURRENT
HS_FET_OVERCURRENT = 64u,

//FAULT CLEAR
DEVICE_READY = 128u,
```

FAULT_STRUCT
This structure contains the fault code and device id of the occurring fault.

dev_id
fault

Reference Code

This example code was developed using the PSoC Creator IDE version 4.1 tested on the CY8CKIT-042 PSoC Pioneer Kit device with DER-654 reference design inverter board. Below code presents reference functions associated with the fault signal processing. The reference code presented excludes the code snippet that prints fault status information through UART console (refer to the Notes paragraph for more details). Please refer to the provided code file for the definition of other variables used.

```
/* ===========================================================================
 * THE SOFTWARE INCLUDED IN THIS FILE IS FOR GUIDANCE ONLY.
 * Power Integrations SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR
 * CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM USE OF THIS
 * SOFTWARE.
 * ==========================================================================*/

/****************************************************************************
* Function Name: void fault_detect(void)
*****************************************************************************
*
* Summary:
* This function is the state machine for the fault bus.
*
* Parameters: None
*
* Return: None
****************************************************************************/

void fault_detect(void)
{

 switch(fault_bus_state)
   {
  case STEADY_STATE:  bit_counter = 0;
                      parity_counter = 0;
                      /* change state to ID detect */
                      fault_bus_state = ID_DET;
                      break;

  case ID_DET:        /* change state to ARBITRATION */
                      fault_bus_state = ARBITRATION;
                      /*Read ID_counter_timer capture value */
                      ID_count_value = Read_ID_Counter;

   if((ID_count_value >= ID_40uS_MIN)&&(ID_count_value <= ID_40us_MAX))
                            {
                       //Device 1
                       fault_struct.dev_id = DEVICE_ID_1; }

   else if((ID_count_value >= ID_60uS_MIN)&&(ID_count_value <= ID_60us_MAX))
                            {
                       //Device 2
                       fault_struct.dev_id = DEVICE_ID_2; }
```

```
      else if((ID_count_value >= ID_80uS_MIN)&&(ID_count_value <= ID_80us_MAX))
                                 {
                                   //Device 3
                                   fault_struct.dev_id = DEVICE_ID_3; }


      else {
                                   //Re-synchronize fault detection if
                                   //invalid ID was received
                                  fault_bus_state = STEADY_STATE; }

                            break;

    case ARBITRATION:       /* change state to T_LO */

                            fault_bus_state = T_LO;

                            break;

    case T_LO:         if(bit_counter <= 7)
                                 {
                            /* change state to BIT_DETECT*/
                            fault_bus_state = BIT_DETECT;   }

                            else
                                 {
                            /* change state to STEADY_STATE */
                            fault_bus_state = STEADY_STATE;

                            if(!(parity_counter & 1))
                                  {

                                   //Parity Error
                                  }

                             else

                                   //Process fault
                                  process_fault();
                                      }

                                 }
                                break;
      case BIT_DETECT:  /* Read Bit_counter_timer capture value*/
                        BIT_count_value = Read_Bit_Counter;

          if((BIT_count_value >= T_BIT0_MIN) && (BIT_count_value <= T_BIT0_MAX))
                                      {
                     /* change state to T_LO*/
                     fault_bus_state = T_LO;

                     //update fault status variable
                     fault_struct.fault = fault_struct.fault & ~(1 << bit_counter);
                     bit_counter++;
                                      }
          else if((BIT_count_value >= T_BIT1_MIN)&&(BIT_count_value <= T_BIT1_MAX))
                                      {
                     /* change state to T_LO*/
                     fault_bus_state = T_LO;

                     // update fault status variable
                     fault_struct.fault = fault_struct.fault | (1 << bit_counter);
                     parity_counter++;
                     bit_counter++;
                                       }
          else     {
                    //Re-synchronize fault detection when invalid BIT was received
                    fault_bus_state = STEADY_STATE;
                                       }

                          break;


              default:
                          break;

          }

      }

      /**************************end of function ********************************/
```

1.

```
/******************************************************************************
* Function Name: void process_fault(void)
*******************************************************************************
*
* Summary:
* This function is to process fault after receiving it.
*
* Parameters: None
*
* Return: None
*
******************************************************************************/
void process_fault(void){

    /*If the received fault is DEVICE_READY*/
    if(fault_struct.fault == DEVICE_READY){

    //user own implementation
    }

    else{

    /*Low-side FET Overcurrent*/
    if((fault_struct.fault & BIT5) != 0){
    tfault = (fault_struct.fault & BIT5);
    action_fault(tfault);
    }

    /*High-side FET Overcurrent*/
    if((fault_struct.fault & BIT6) != 0){
    tfault = (fault_struct.fault & BIT6);
    action_fault(tfault);
    }

    /*Group1 Faults*/
    if((fault_struct.fault & GROUP1) != 0){
    tfault = (fault_struct.fault & GROUP1);
    action_fault(tfault);
    }

    /*Group2 Faults*/
    if((fault_struct.fault & GROUP2) != 0){
    tfault = (fault_struct.fault & GROUP2);
    action_fault(tfault);
    }


   }

 }



/******************************end of function******************************/
```

```
/*****************************************************************************
*
* Function Name: void fault_action(unit8)
*****************************************************************************
*
* Summary:
* This function is to command an action after a fault is received
*
* Parameters: masked fault by group
*
* Return: None
*
****************************************************************************/

void action_fault(uint8 tfault){

/*Look the fault code into the fault_status_info_arr array and the
corresponding MCU action*/

   int loop_count = sizeof(fault_status_info_arr)/sizeof(FAULT_STATUS_INFO);
   for (int i=0; i<=loop_count; i++){

        if(tfault != (fault_status_info_arr[i].fault_code))
        continue;

        switch(fault_status_info_arr[i].fault_action){

                  case ACTION_NONE:
                  /* do nothing */
                  break;

                  case ACTION_WARNING:
                  /* user own implementation */
                  break;

                  case ACTION_SHUTDOWN:
                  /* Shutdown MCU */
                  break;

        }


        /**OPTIONAL -print fault information for debugging purposes only**/
        print_fault_info(tfault);

  }

}


/******************************end of function******************************/
```

The code below presents the reference functions associated with the status query and latch reset command example implementation described in this document. The call to status query and latch reset commands should be handled separately in the actual implementation depending on each user use cases. Please refer to the provided code file for the definition of the variables used.

```
/*****************************************************************************
***
* Function Name: void status_query(void)
*****************************************************************************
***
*
* Summary:
* This function is to command a status query
*
* Parameters: None
*
* Return: None
*
*****************************************************************************
**/

void status_query(void){


    /*Clear FAULT Bus ISRs*/
    FAULT_Bus_ClearInterrupt();

    /*Pull down the FAULT Bus for 160 uS*/
    FAULT_Bus_Write(0);
    CyDelayUs(160);

    FAULT_Bus_Write(1);

    /*Enable FAULT_Bus ISRs*/
    init_fault_bus_interrupt();

    /*Set status query flag*/
    status_query_state = TRUE;

}
```

```
/**********************************************************************
***
* Function Name: void process_status_query_command(void)
**********************************************************************
***
*
* Summary:
* This function is to process the status query command
*
* Parameters: None
*
* Return: None
*
**********************************************************************
**/

void process_status_query_command(){

    //store each devices fault status
    device_fault_arr[fault_struct.dev_id] = fault_struct.fault;

    //increment device_counter
    device_counter++;

    if(device_counter == DEVICE_COUNT){

    //status_query_action
    status_query_action();

    //reset status query state
    status_query_state = FALSE;

    //reset device counter
    device_counter = 0;

        }

}
```

```
/****************************************************************************
***
* Function Name: void status_query_action(void)
****************************************************************************
***
*
* Summary:
* This function is to process the captured fault status from a status query
* command
* Parameters: None
*
* Return: None
*
****************************************************************************
**/
void status_query_action(void){

    //Function that checks if all devices are READY
    if (device_ready_check()){

    /*All devices are READY, Inverter restart function should be placed here
    *
    */ }

    //Function that checks for only HS driver not ready fault
    else if(hs_driver_not_ready_check()){

    //Command a startup sequence after the first status query command
    if(startup_flag == FALSE){

    /*Startup sequence function should be placed here
    *
    */

    /*Check the status if HS not ready fault/s is/are cleared*/
    status_query();

    //Assert startup_flag after start up sequence
    startup_flag = TRUE;
     }
    else{

    //HS driver not ready fault still exists

    //De-assert startup_flag
    startup_flag = FALSE;

        }
      }
    else{

    //Other faults are present
    startup_flag = FALSE;
       }
      }
```

```
/**************************************************************************
***
* Function Name: boolean device_ready_check(void)
**************************************************************************
***
*
* Summary:
* This function is to check if all devices are ready
*
* Parameters: None
*
* Return: boolean
*
**************************************************************************
**/

boolean device_ready_check(void){

  uint8 tfault_status =0;

  //Check if all devices are READY
  for(uint8 i=0; i<sizeof(device_fault_arr); i++){

   tfault_status  |= device_fault_arr[i];

  }

  //If all devices are READY
  if(tfault_status == DEVICE_READY){

    //return TRUE
    return TRUE;

    }else{

    //return FALSE
    return FALSE;
    }


}
```

```
/***************************************************************************
***
* Function Name: boolean hs_driver_not_ready_check(void)
****************************************************************************
***
*
* Summary:
* This function is to check if all devices are READY
*
* Parameters: None
*
* Return: boolean
*
****************************************************************************
**/

boolean hs_driver_not_ready_check(void){

 //Default hs_driver_fault_flag
 hs_fault_flag = FALSE;

 for(uint8 i=0; i<sizeof(device_fault_arr); i++){

    if((device_fault_arr[i] == DEVICE_READY) || (device_fault_arr[i] ==
HS_DRIVER_NOT_READY_FAULT)){

        if(device_fault_arr[i] == HS_DRIVER_NOT_READY_FAULT){

        //Assert hs_driver_fault flag
        hs_fault_flag = TRUE;

        continue;
        }

    }else{

        //Other fault/s is/are present
        return FALSE;
        }

    }

    return hs_fault_flag;

}
```

```
/***************************************************************************
***
* Function Name: void latch_reset(void)
****************************************************************************
***
*
* Summary:
* This function is to command latch reset
*
* Parameters: None
*
* Return: None
*
****************************************************************************
**/
void latch_reset(void){

    /*Disable FAULT Bus ISRs*/
    FAULT_Bus_ClearInterrupt();

    /*Pull down the FAULT Bus for 320 uS*/
    FAULT_Bus_Write(0);
    CyDelayUs(320);

    FAULT_Bus_Write(1);
    }
/***************************************************************************
***
* Function Name: void mcu_latch_reset(void)
****************************************************************************
***
*
* Summary:
* This function is to command latch_reset followed by a power up sequence
*
* Parameters: None
*
* Return: None
*
****************************************************************************
**/
void mcu_latch_reset(void){

    //latch reset command
    latch_reset();

    /*Power up sequence function should be placed here
    *
    *
    */

    //Enable FAULT Bus ISRs
    init_fault_bus_interrupt();

   }
```

Fault Detection Example

The presented FAULT detection examples and decisions made by the micro-controller follow the exemplary actions listed in Table 5 using the reference code detailed above.

The UART terminal displays the fault status information to illustrate the fault state machine. The displayed information has a format of *[device ID, fault, action]*. For example, UART message of *W, STS, and S* represents the status update from Device W (device 1, 2 and 3 are designated U, V and W respectively), fault status is System Thermal Shutdown (STS), and the MCU action is Shutdown (S).

Figure 10 depicts a reported system thermal fault and the exemplary action of shutting down the inverter (see UART terminal output in Figure 11).

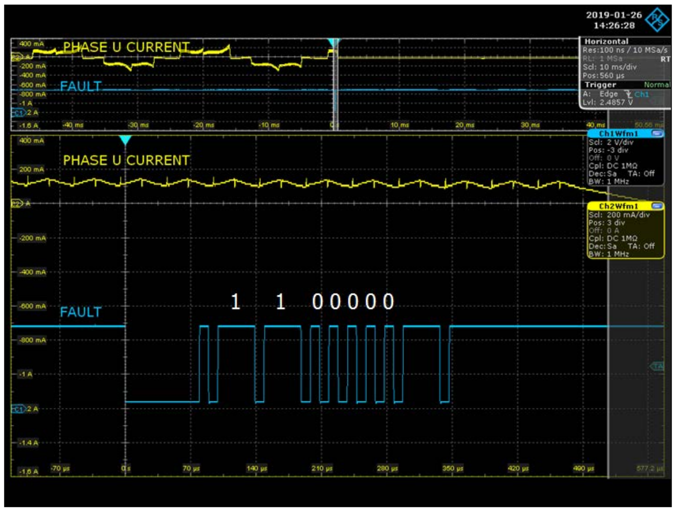
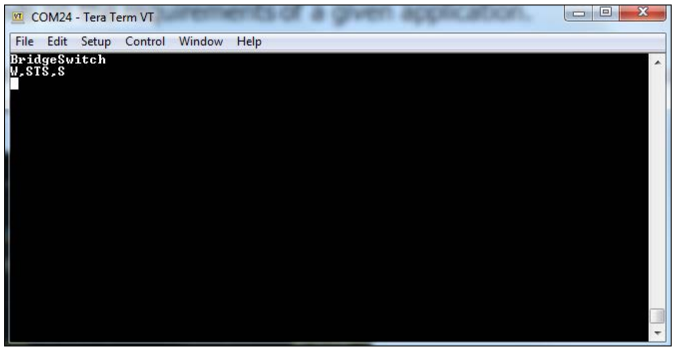


Figure 10. Example Inverter Shutdown after System Temperature Status FAULT.



```
BridgeSwitch
W,STS,S
```

Figure 11. UART Terminal Output after Receiving a System Thermal FAULT.

Figure 12 depicts a reported low-side over-current fault and the exemplary action of shutting down the inverter (see UART terminal output in Figure 13).

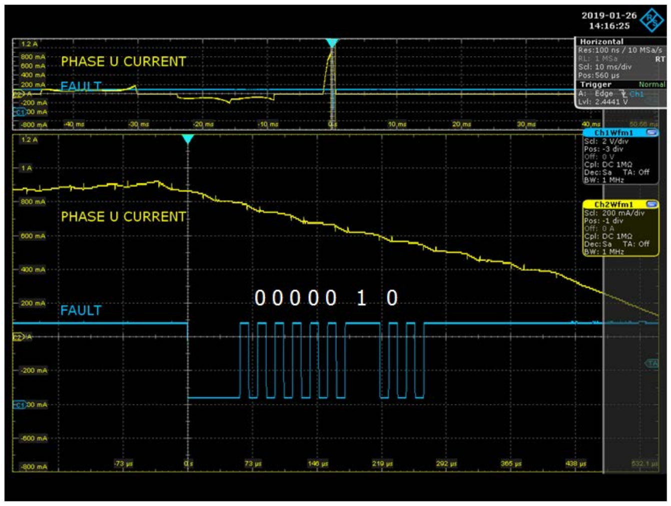
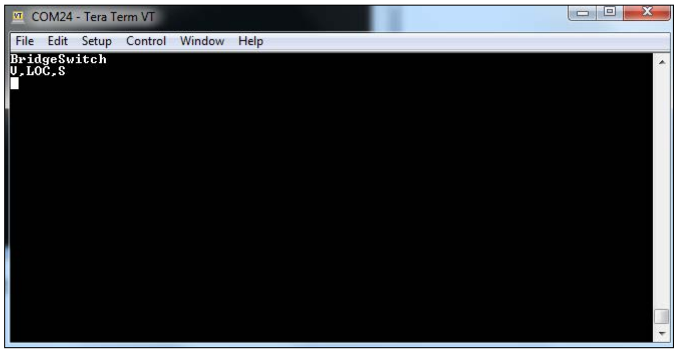


Figure 12. Example of Inverter Shutdown after Receiving a Low-side Over-Current FAULT.



```
BridgeSwitch
U,LOC,S
```

Figure 13. UART Terminal Output after Receiving a Low-Side Over-Current FAULT.

Figure 14 depicts a reported high-voltage bus UV85 fault with a warning status. In this exemplary implementation, the MCU did not take a specific action but only displayed a Warning status. See UART terminal in Figure 15.

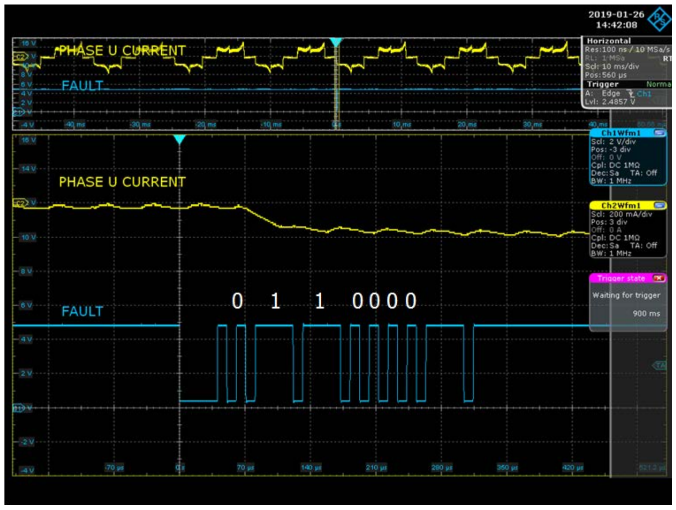
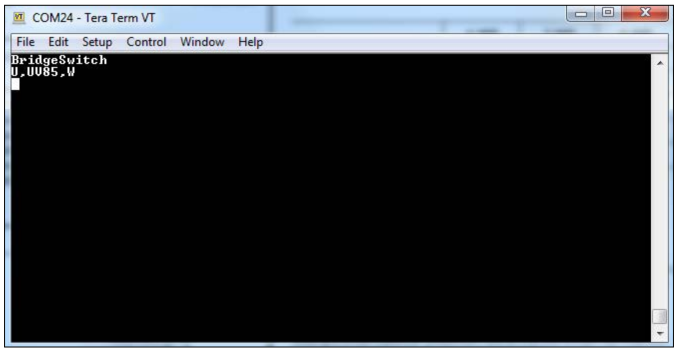


Figure 14. Warning Status after Receiving a High-Voltage Bus UV85.



```
BridgeSwitch
U,UV85,W
```

Figure 15. UART Terminal Output after Receiving a High-Voltage Bus UV85.

Figure 16 depicts a reported high-voltage bus overvoltage and the exemplary action of shutting down the inverter (see UART terminal output in Figure 17).

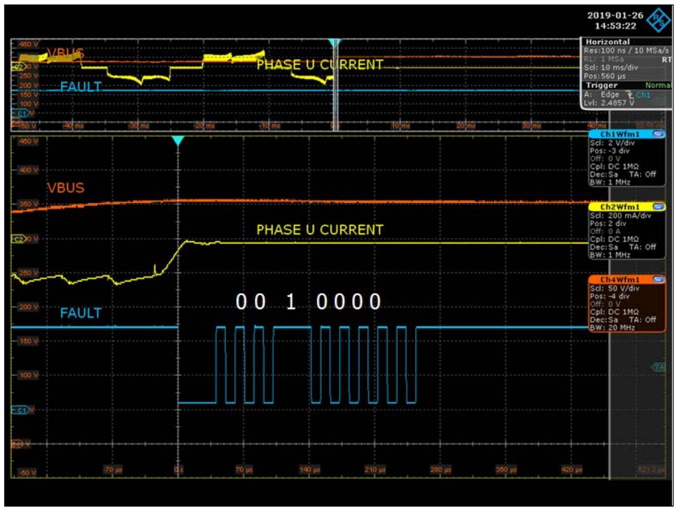
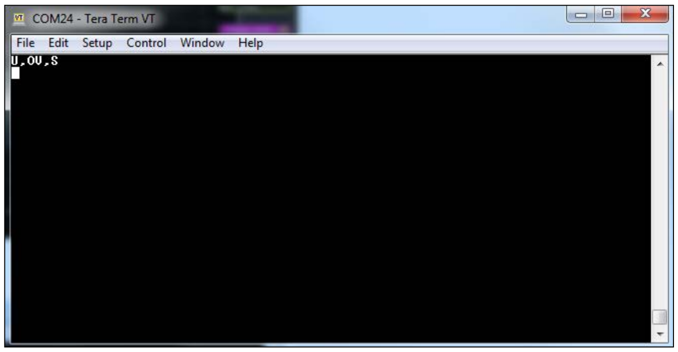


Figure 16. Example of Inverter Shutdown after Receiving a High-Voltage Bus Overvoltage.



```
U,OV,S
```

Figure 17. UART Terminal Output after Receiving a High-Voltage Bus Overvoltage.

MCU Commands Example

Figure 18 depicts an inverter restart after a status query command following a shutdown caused by a line overvoltage fault condition.

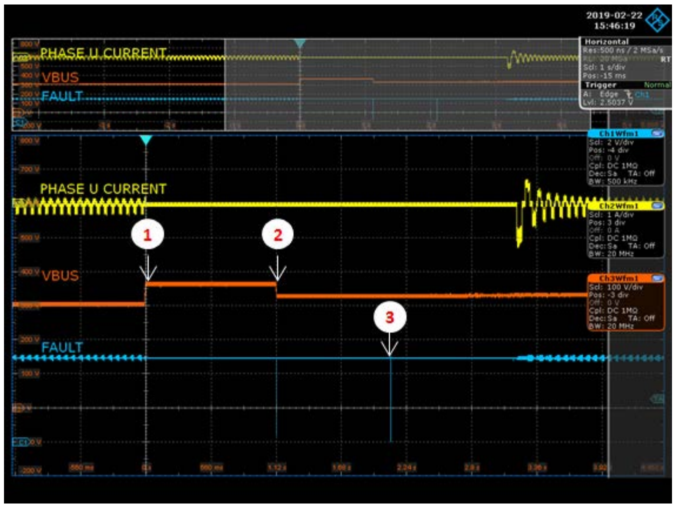


Figure 18. Status Query Command After a Line Overvoltage Condition.

(1) A line overvoltage occurs and the inverter is shutdown with the OV status depicted by Figure 19. (2) Overvoltage condition has cleared and at (3) the system micro-controller sends a status query command to check devices status. Figure 20 shows the status query command and the respective status report from all three devices. All devices reported READY and the system micro-controller restarts the inverter.

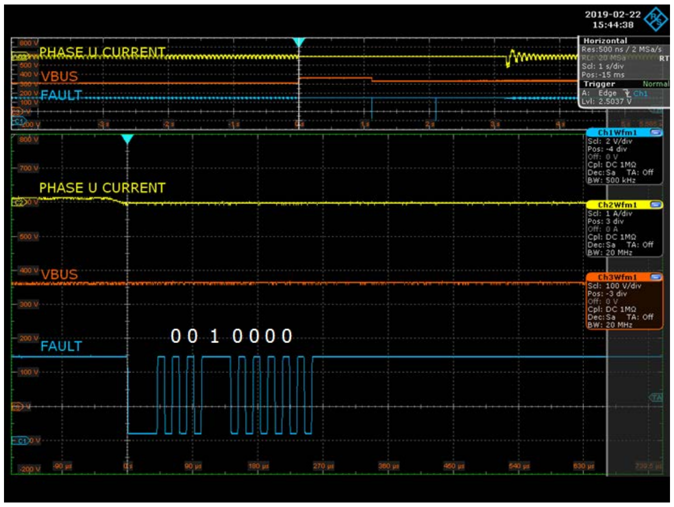


Figure 19. Inverter Shutdown After a Line Overvoltage Condition.

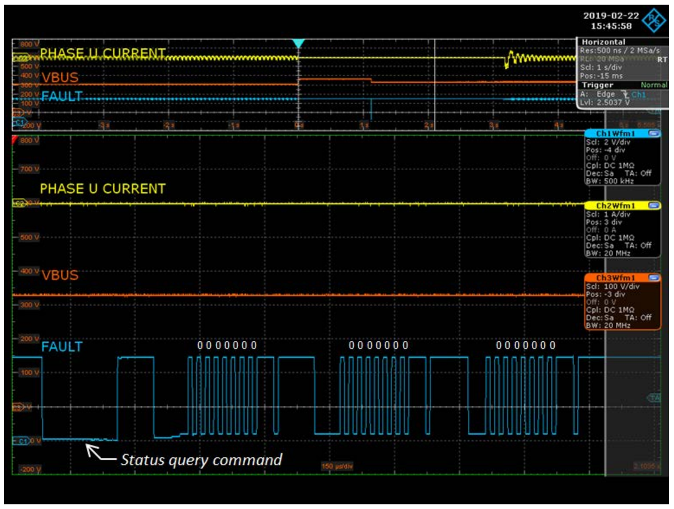


Figure 20. Status Query Command After a Line Overvoltage Condition (All Devices Reported READY).

Figure 21 depicts an inverter restart after a status query (2) command following a shutdown caused by high-side driver not ready fault (1).

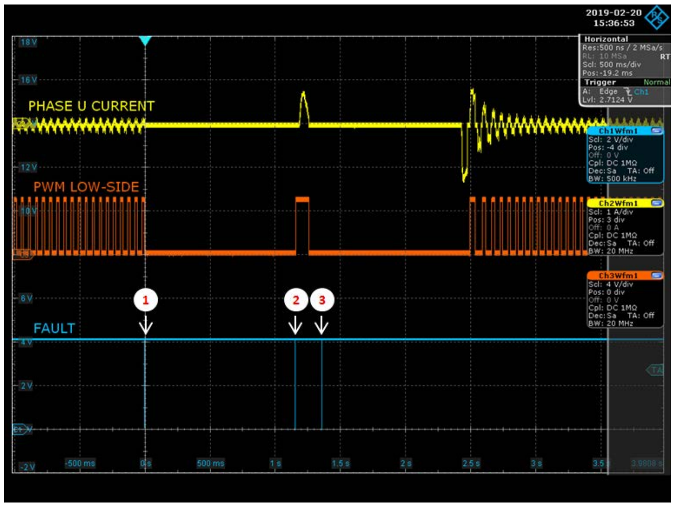


Figure 21. Status Query Command After a High-Side Driver not Ready Fault.

In this example, the reported status update is only a High-side driver not ready fault. The MCU will command a start-up routine (i.e. applying a logic high to low-side PWM inputs INL for a periods of 100 ms). At (3) the MCU sends another status query command to check if all devices are READY. In this example, all faults are cleared and all devices are READY. The MCU initiates an inverter restart and sends PWM signals to the BridgeSwitch control inputs INL and /INH.

Figure 22 shows the corresponding high-side driver not ready fault status and Figure 23 shows the status query command and the respective status report from all three devices after a start-up sequence attempt. All devices reported READY and the system micro-controller restarts the inverter.

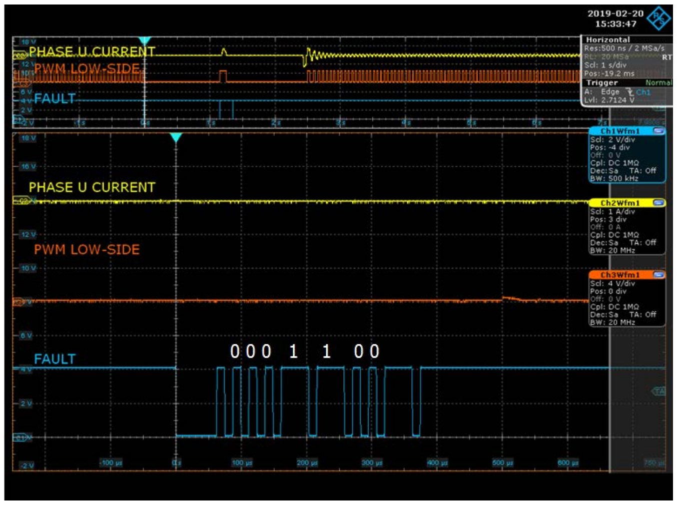


Figure 22. Inverter Shutdown After a High-Side Driver not Ready Fault.

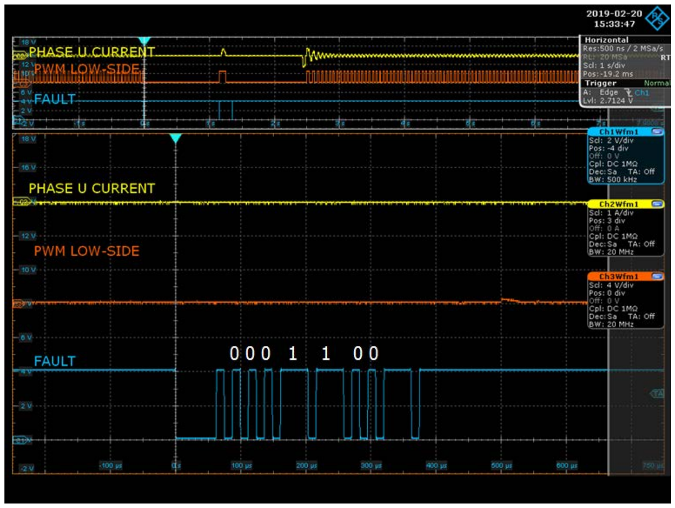


Figure 23 . Status Query Command After a Startup Sequence.

Figure 24 depicts a latch reset command following a latching shutdown caused by an over-temperature fault. The MCU applies a full power-up sequence after sending the latch reset command.

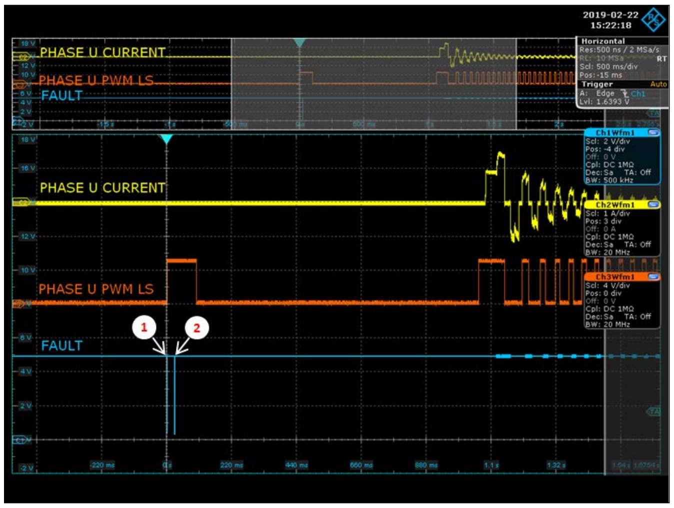


Figure 24. Latch Reset Command and a Power-Up Sequence After a Latching Over-Temperature Protection.

Figure 25 shows the latch reset command (1) and the default status reports of High-side not ready (note that fault detection is disabled when invoking latch reset command). The MCU applies start-up (power-up) sequence after sending a latch reset command. Figure 26 depicts all devices reporting a READY (2) and then the inverter is started.

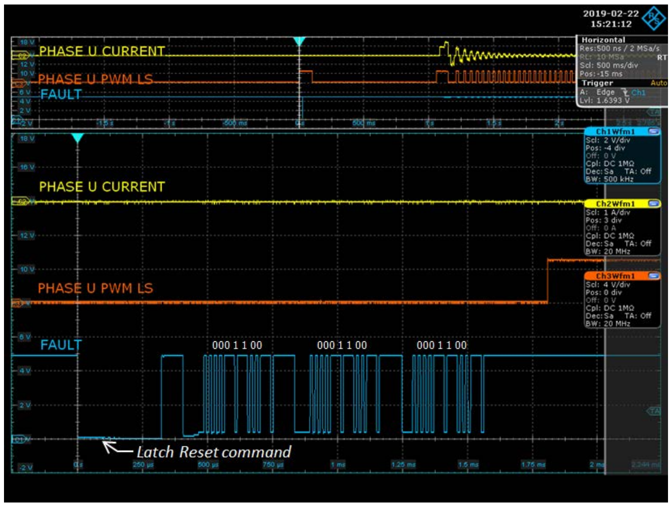


Figure 25. Latch Reset Command and a Power-Up Sequence After a Latch OTP.

Figure 26 depicts successful completion of a power-up sequence with all devices reporting a READY status.

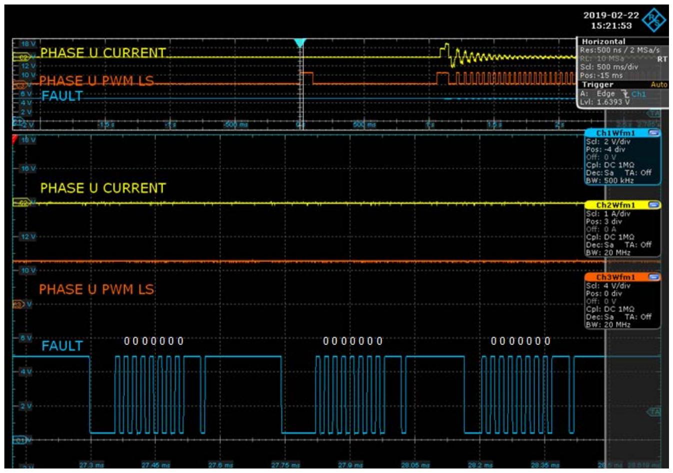


Figure 26. Device Status After Start-Up Sequence.

Example Code Library

An example code library is available for download from the BridgeSwitch product page (www.power.com) using the link below:

https://motor-driver.power.com/products/bridgeswitch-family/bridgeswitch/

Notes

This application note describes displaying of fault information through UART console for debugging purposes. The display execution should be implemented in a polled manner to limit the load on the MCU. Minimizing the amount of information displayed also reduces the load.

Revision	Notes	Date
A	Initial release.	04/19
B	Corrected typos in Table 2 and Table 6.	09/20

Power Integrations Worldwide Sales Support Locations

World Headquarters
5245 Hellyer Avenue
San Jose, CA 95138, USA
Main: +1-408-414-9200
Customer Service:
Worldwide: +1-65-635-64480
Americas: +1-408-414-9621
e-mail: usasales@power.com

China (Shanghai)
Rm 2410, Charity Plaza, No. 88
North Caoxi Road
Shanghai, PRC 200030
Phone: +86-21-6354-6323
e-mail: chinasales@power.com

China (Shenzhen)
17/F, Hivac Building, No. 2, Keji Nan
8th Road, Nanshan District,
Shenzhen, China, 518057
Phone: +86-755-8672-8689
e-mail: chinasales@power.com

Germany (AC-DC/LED Sales)
Einsteinring 24
85609 Dornach/Aschheim
Germany
Tel: +49-89-5527-39100
e-mail: eurosales@power.com

Germany (Gate Driver Sales)
HellwegForum 1
59469 Ense
Germany
Tel: +49-2938-64-39990
e-mail: igbt-driver.sales@power.com

India
#1, 14th Main Road
Vasanthanagar
Bangalore-560052 India
Phone: +91-80-4113-8020
e-mail: indiasales@power.com

Italy
Via Milanese 20, 3rd. Fl.
20099 Sesto San Giovanni (MI) Italy
Phone: +39-024-550-8701
e-mail: eurosales@power.com

Japan
Yusen Shin-Yokohama 1-chome Bldg.
1-7-9, Shin-Yokohama, Kohoku-ku
Yokohama-shi,
Kanagawa 222-0033 Japan
Phone: +81-45-471-1021
e-mail: japansales@power.com

Korea
RM 602, 6FL
Korea City Air Terminal B/D, 159-6
Samsung-Dong, Kangnam-Gu,
Seoul, 135-728, Korea
Phone: +82-2-2016-6610
e-mail: koreasales@power.com

Singapore
51 Newton Road
#19-01/05 Goldhill Plaza
Singapore, 308900
Phone: +65-6358-2160
e-mail: singaporesales@power.com

Taiwan
5F, No. 318, Nei Hu Rd., Sec. 1
Nei Hu Dist.
Taipei 11493, Taiwan R.O.C.
Phone: +886-2-2659-4570
e-mail: taiwansales@power.com

UK
Building 5, Suite 21
The Westbrook Centre
Milton Road
Cambridge
CB4 1YG
Phone: +44 (0) 7823-557484
e-mail: eurosales@power.com