应用指南AN-80
BridgeSwitch产品系列

BridgeSwitch故障状态通信接口

简介

本应用指南介绍BridgeSwitch™故障状态通信接口特性的软件实现指南。下面将分别讲解BridgeSwitch状态通信接口、捕获并处理所接收的状态更新的状态调节器、参考代码及其数据结构、通过UART端子显示状态更新的软件演示、以及在逆变板中的示例故障保护实现。

BridgeSwitch故障状态通信接口

BridgeSwitch器件可以就状态更新进行通信，包括通过其开漏故障输出将内部及系统级故障传递至系统微控制器(MCU)。它采用7位字模式加一个奇校验位来报告状态更新。

下面将详细介绍故障总线的规格规范。

硬件配置

为了将所有检测到的状态更新传递至系统微控制器，所有故障引脚必须连接到已拉升到系统供电电压的单线总线。图1所示为单线总线配置中三个BridgeSwitch器件与系统微控制器的典型接口。

器件ID引脚连接允许每个器件根据器件ID引脚连接方式为自身分配唯一的器件ID。在状态通信开始时，根据各自的器件ID时间t_{ID}将故障总线拉低，就可以将检测到的故障情况的物理位置传递到系统微控制器。

表1列出了器件ID、得出的器件ID时间t_{ID}以及如何通过ID引脚连接来设定各自的ID。

器件ID	t_{ID}	ID引脚连接
1	40 µs	BPL引脚
2	60 µs	悬空
3	80 µs	SG引脚

表1. 通过ID引脚选择器件ID

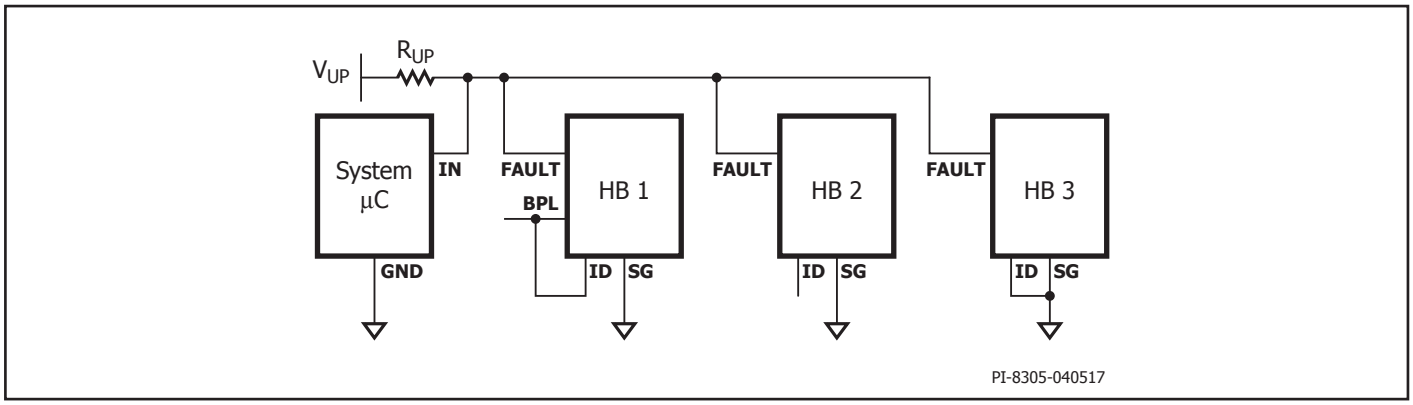

图1. 采用器件ID编程的单线状态通信总线

故障状态通信总线规范

状态编码

7位字加一个奇校验位可对故障信息进行编码。表2汇总了可能传递给系统微控制器的各种状态更新的编码。状态字包括五个含状态变化信息的块，这五个块组合在一起并且不可能同时发生。因此，这种编码可以同时向系统微控制器报告多个状态更新，无需考虑故障优先级和故障报告队列。

最后一行（7位字“000 00 0 0”）编码为“器件就绪”状态，用来向系统传递上电序列成功信息。当某个故障清除后，向系统微控制器传递此编码，表示已无故障存在。

故障总线通信可基于下面其中一个原因触发：

- 成功上电后任务模式通信已就绪。
- 故障状态寄存器更新通信由其中一个器件触发。
- 当前状态通信跟随着系统微控制器查询。

故障	位0	位1	位2	位3	位4	位5	位6
上管总线过压	0	0	1				
上管总线欠压100%	0	1	0				
上管总线欠压85%	0	1	1				
上管总线欠压70%	1	0	0				
上管总线欠压55%	1	0	1				
系统热故障	1	1	0				
下管驱动器未就绪[1]	1	1	1				
下管FET热告警				0	0		
下管FET热关断				1	0		
上管驱动器未就绪[2]				1	1		
下管FET过流						1	
上管FET过流							1
器件就绪（无故障）	0	0	0	0	0	0	0

注释：

1. 包括XL引脚开路/短路故障、IPH引脚到XL引脚短路以及微调位错乱。
2. 包括上管-下管通信损耗、V_{BPH}或内部5 V供电超出范围以及XH引脚开路/短路故障。

表2. 状态字编码

图2所示为所有这三种情况的故障接口通信。

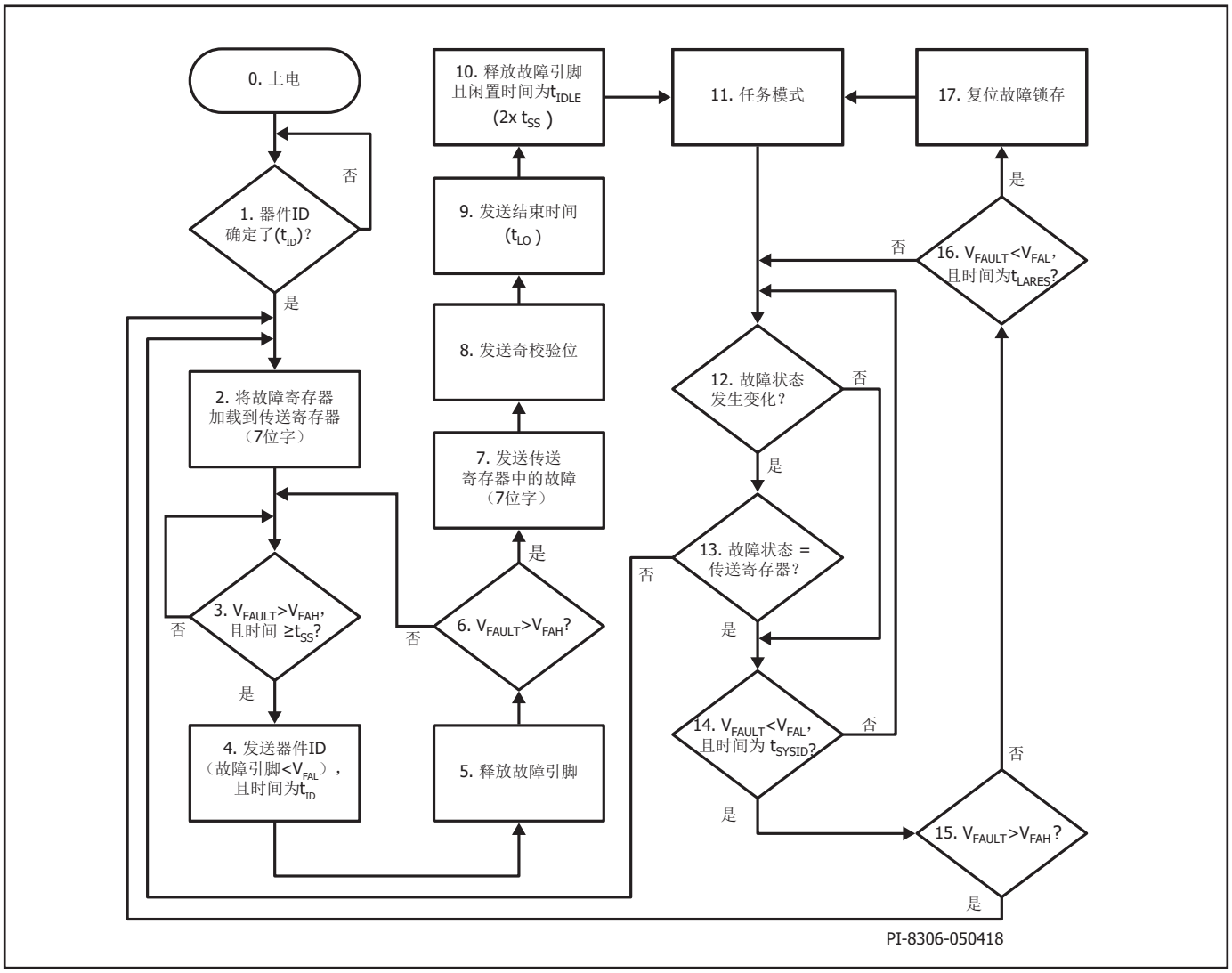

图2. 状态通信流程图

状态最新信息通信总是从通信设备所触发的总线判优开始的。如果总线持续“安静”至少80 µs，则会拉低故障引脚，以此传送各自的器件ID时间t_{ID}。器件赢得总线判优后，将会发送后跟奇校验位的当前故障寄存器（7位字）和传送结束信号，如通信流程图所示（见图2）。

位流定时

图3所示为BridgeSwitch用于状态更新通信的位流定时图。两个逻辑状态按照故障引脚两个不同的电压信号长时后跟一个短时t_{LO}（典型值10 µs）进行编码。逻辑“1”按照时间t_{BIT1}（典型值40 µs）进行编码，逻辑“0”按照时间t_{BIT0}（典型值10 µs）进行编码。表3列出了故障状态通信的定时表。

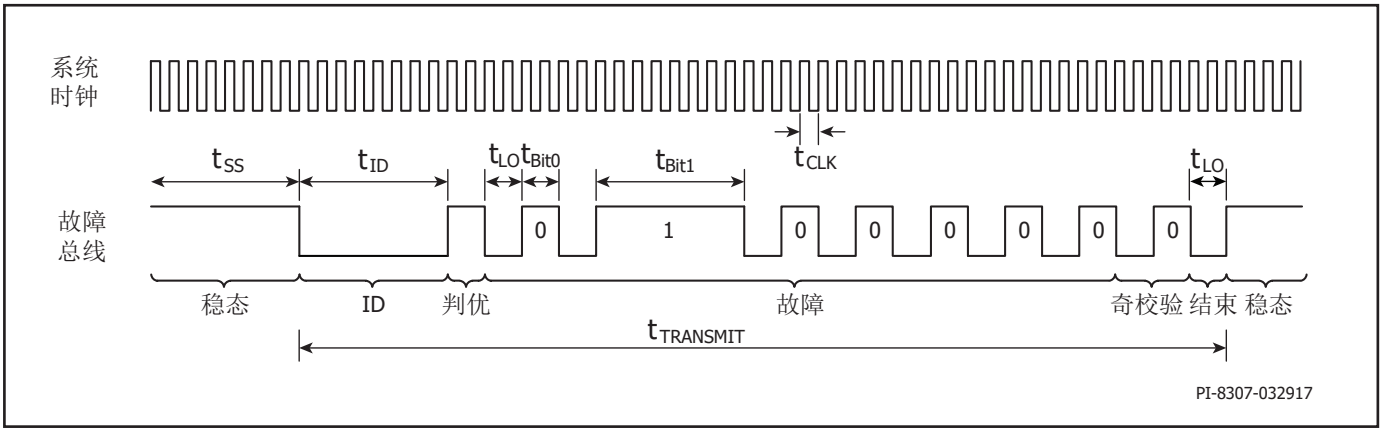

图3. 状态通信位流

符号	说明	逻辑状态	时间（典型值）
t_{ID}	器件ID	0	参考表1
t_{LO}	短时	0	10 µs
t_{Bit0}	逻辑0	1	10 µs
t_{Bit1}	逻辑1	1	40 µs

表3. 位流定时表

每次传送完成后，器件将会闲置一定的时间t_{IDLE}（典型值$2 \times t_{SS}$ = 160 µs），然后再开始新的通信。这样可使总线上的其他器件传递可能发生的状态变化或者对系统微控制器提出的状态查询做出响应。

器件传递每个检测到的状态更新的次数只有一次。它还会向系统微控制器报告所有系统级故障的状态变化。这些变化包括直流总线欠压及过压情况以及外部温度监测故障。此外，它还会报告器件内部故障的所有状态变化，但下管功率FREDFET热关断除外。

BridgeSwitch器件还可以在进入任务模式后，根据系统微控制器所发送的可能命令来监测故障总线。该指令可能是微控制器通过在t_{SYSID}（典型值160 µs）内拉低总线所产生的状态更新查询（参见图2中的步骤15）。该命令也可能是复位器件状态寄存器（包括过温关断锁存），并通过在t_{LARES}（$2 \times t_{SYSID}$ = 典型值320 µs）内拉低故障总线进入上电序列模式（参见图2中步骤17）。在微控制器发送锁存复位命令后会建议应用上电序列。表4汇总了可用的系统微控制器命令。

总线下拉时间	命令
t_{SYSID}	状态查询
$t_{LARES(2xTSYSID)}$	状态寄存器，包括过温锁存复位和上电序列模式

表4. 系统微控制器命令

软件实现

这一部分介绍状态调节器的实现方法，该状态调节器可以根据上一部分所述的状态通信规范捕获并处理来自每个BridgeSwitch器件的状态更新。

下文中的示例采用基于中断的实现方法。用户必须根据自己特定的应用要求（例如，电机控制算法或微控制器类型）决定中断优先级。

系统微控制器外设

为了演示故障总线通信接口的实现方法，我们使用Cypress PSoC Creator IDE 4.1版开发了参考代码，并基于Cypress PSoC 4 MCU（CY8CKIT-042 PSoC Pioneer Kit）进行了测试。MCU板提供了一个板载编程器和调试器，通过USB连接器接口与PC进行通信。我们通过在UART控制台上打印接收到的状态更新来演示状态调节器的工作方式，具体请参见“故障检测示例”部分。

故障状态通信总线连接到一个处于开漏驱动模式的双向微控制器引脚。该引脚与可同时捕获信号上升沿和下降沿的定时器相连。故障信号的处理基于中断方法，并且采用两个16位定时器/计数器块*Bit_counter_timer*和*ID_counter_timer*以及一个12 MHz时钟。*Bit_counter_timer*可捕获来自上升-下降沿的信号，而*ID_counter_timer*可捕获来自下降-上升沿的信号。这两个定时器将在已分别收到故障信号的下降沿和上升沿后捕获计数值并生成中断。故障状态调节器例程会处理每个接收到的中断。

软件说明

软件实现开始时先将*fault_bus_state*变量初始化为闲置状态*(STEADY_STATE)*。*fault_bus_state*变量可在收到中断后捕获故障信号的状态。初始化函数*init_fault_bus_interrupt()*可初始化定时器/计数器的捕获端口，并启用上升沿和下降沿的捕获中断。只要触发中断服务例程(ISR)，就会调用*fault_detect()*函数。该函数是可捕获和处理接收到的故障的故障状态调节器例程。主要软件流程的高级视图如图5所示。

故障状态调节器例程基本上处理ISR事件，并在故障处理过程中根据当前状态更新*fault_bus_state*变量。故障调节器的状态包括*STEADY_STATE*、*ID_DET*、*ARBITRATION*、*T_LO*和*BIT_DETECT*。故障状态调节器的详细软件流程图如图6所示。只要接收到无奇校验错误的完整故障状态数据包，它就会调用故障处理函数*fault_process()*，如果*fault_bus_state*复位到*STEADY_STATE*，则会进行重新同步。

故障处理函数可解码接收到的故障状态更新并调用所需的操作。例如，在接收到过流故障后关断逆变器，或者在接收到热警告状态更新后减小逆变器的输出功率。故障状态存储在*fault*变量中，该变量会在每次接收到新的状态更新时进行相应更新。用户应该提供必要的操作，或者应该根据接收到的故障状态和应用要求决定微控制器应当采取何种操作。表5列出了系统微控制器在接收到状态更新后可以采取的典型操作。*fault_process()*函数软件流程图如图7所示。

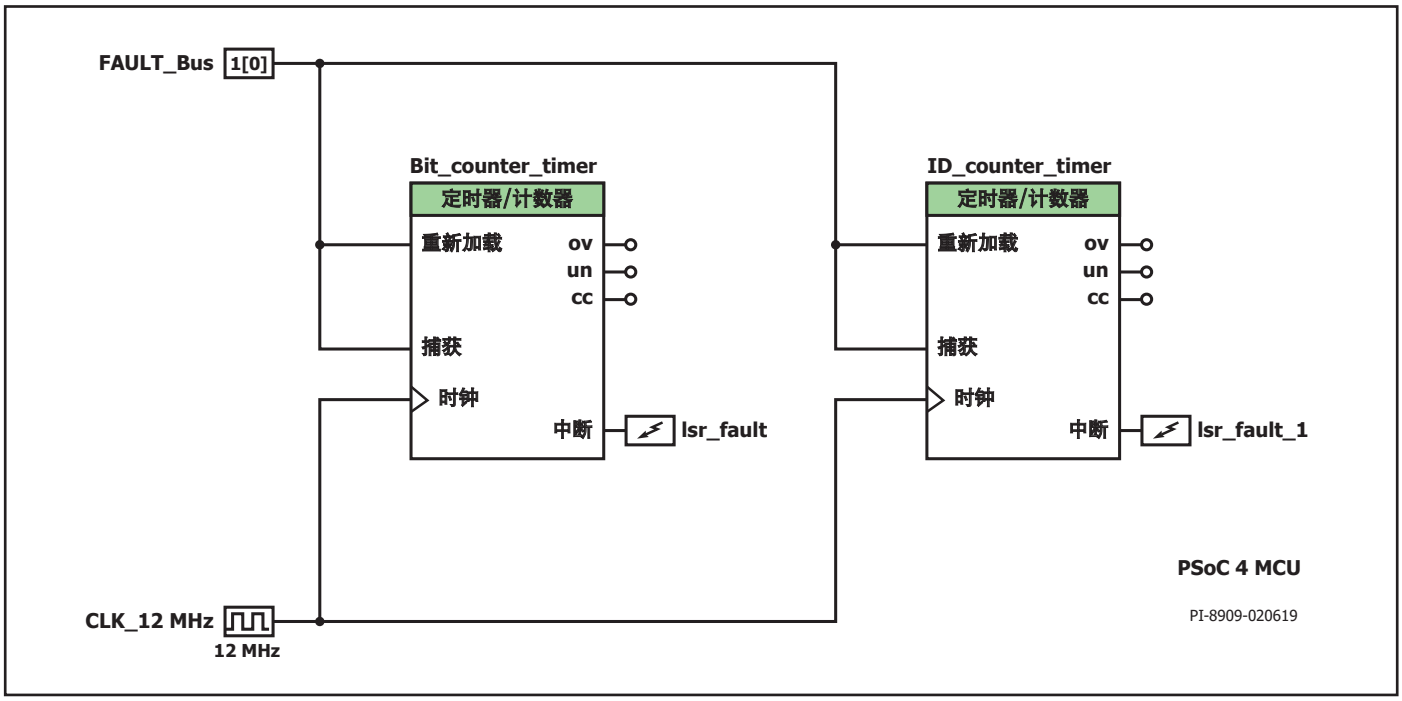

图4. 故障信号处理示例（采用PSoC 4 MCU）的系统微控制器外设

状态查询和锁存复位命令

状态查询和锁存复位命令的软件实现支持下列用例示例：

状态查询命令：

微控制器可在每次逆变器关断一定时间后想要重启动逆变器时发送状态查询（也即，发送PWM信号）。例如，状态查询发生在已报告输入过压或过流故障之后。状态查询的主要目的是检查所有器件是否都已就绪或者微控制器是否必须启动上电序列。图8所示为状态查询示例实施的流程图。

状态查询例程可检查每个BridgeSwitch器件的状态以确定符合下列哪种情况：

A. 每个器件的响应状态为“就绪”（不存在故障）：微控制器调用RESTART（重启动）命令重启动逆变器并发送PWM信号以控制BridgeSwitch的输入端。

B. 一个或多个器件的响应状态为存在上管驱动器未就绪故障：微控制器调用START UP（启动）命令，启动上电序列将上管驱动器供电电压（V_{BPH}，对应于HB引脚）充电至其额定值。上电序列完成后，微控制器将开始另一个状态查询命令。如果所有器件的响应状态均为就绪，微控制器将调用RESTART（重启动）命令重启动逆变器。如果有任何器件的响应状态不是就绪，逆变器将保持在关断模式。

状态查询命令由*status_query()*函数处理，将故障总线拉低的持续时间为t_{SYSID} = 160 µs（参见表4）。每个器件都将遵循此命令，并相继发送各自的状态。系统微控制器发送状态查询后，检测到的故障状态将由*process_status_query_command()*函数（置于故障总线状态调节器函数内）处理，该函数将存储每个检测到的器件状态并在*status_query_action()*函数中进行处理。*status_query_action()*函数将检查接收到的故障状态并根据表5中列出的情况提供相应操作。

锁存复位命令：

微控制器可以在其中一个（或所有）器件已报告过温故障（和锁存关断）之后的某个时间发送锁存复位命令。图9所示为锁存复位命令示例的流程图。在锁存复位命令后会建议应用上电序列。这可以确保旁路上管电压在开关恢复之前处于额定水平。

锁存复位命令由*latch_reset()*函数处理，将故障总线拉低的持续时间为t_{LARES} = 320 µs（参见表4），以复位每个器件状态。在系统微控制器发送锁存复位命令后会调用上电序列。请注意，故障检测函数必须在锁存复位命令发送后立即禁用。

故障	状态字	软件 操作/决定	注释
高压总线过压	001 xxx x	关断	通常，只有一个器件监测高压总线，微控制器可关断整个逆变器。
高压总线欠压100%	010 xxx x	无	微控制器可将电机输出功率增大至额定功率 – 如果之前的状态更新为UV85、UV70或UV50。
高压总线欠压85%	011 xxx x	告警	微控制器可减小电机输出功率（速度/扭矩），以降低逆变器负载。
高压总线欠压70%	100 xxx x	告警	微控制器可减小电机输出功率（速度/扭矩），以降低逆变器负载。
高压总线欠压55%	101 xxx x	告警	微控制器可减小电机输出功率（速度/扭矩），以降低逆变器负载。
系统热故障	110 xxx x	告警/关断	取决于所监测的外围元件温度为何。
下管驱动器未就绪	111 xxx x	关断	微控制器可在一定时间后尝试重启动逆变器，以检查故障是否清除。
下管FET热告警	xxx 010 x	告警	微控制器可减小电机输出功率（速度/扭矩），以降低逆变器负载或限制PCB温度。
下管FET热关断	xxx 10x x	关断	可能只有一个器件发生锁存关断，微控制器应关断整个逆变器，微控制器可在一个冷静期后尝试重启动逆变器。
下管FET过流	xxx xx1 x	关断	器件自动关断各自的FREDFET，防止电机发生停转或过载故障。微控制器关断整个逆变器。
上管驱动器未就绪	xxx 11x x	关断	微控制器可在一定时间后（数秒后）尝试重启动逆变器，以检查故障是否清除。
上管FET过流	xxx xxx 1	关断	器件自动关断各自的FREDFET，防止电机发生停转或过载故障。微控制器关断整个逆变器。
器件就绪（无故障）	000 000 0	无	如果之前的状态更新为高压总线过压或下管/上管FET过流，微控制器将重启动逆变器。微控制器还可以将电机功率增大至额定功率 – 如果之前的状态更新为热告警。

表5.　微控制器在接收到状态更新后所采取的典型操作

软件流程图

下面的流程图是软件进行故障信号处理的高级视图。

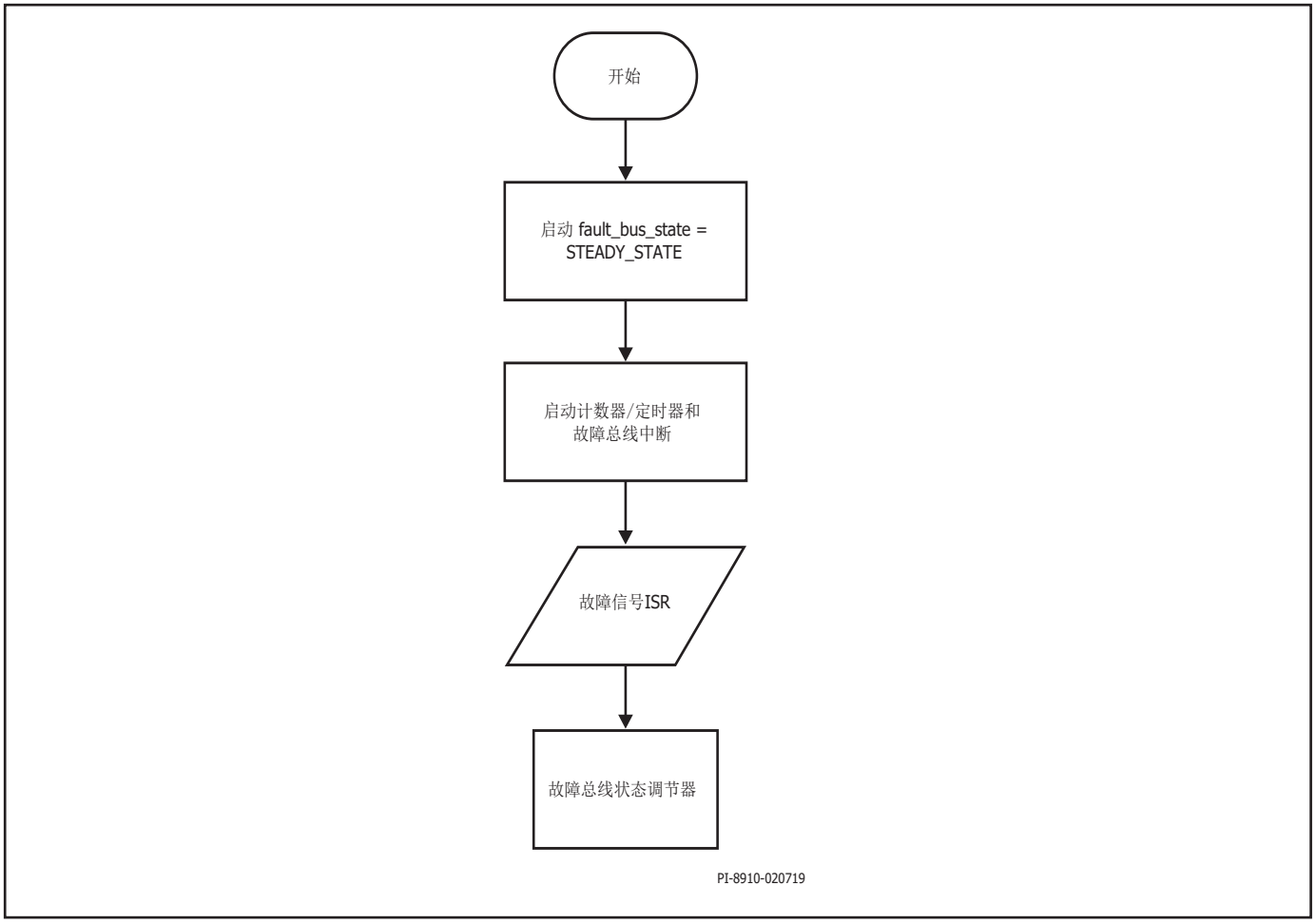

图5. 软件中故障总线实现的高级视图

故障总线状态调节器

图6所示为故障总线状态调节器的软件流程图。

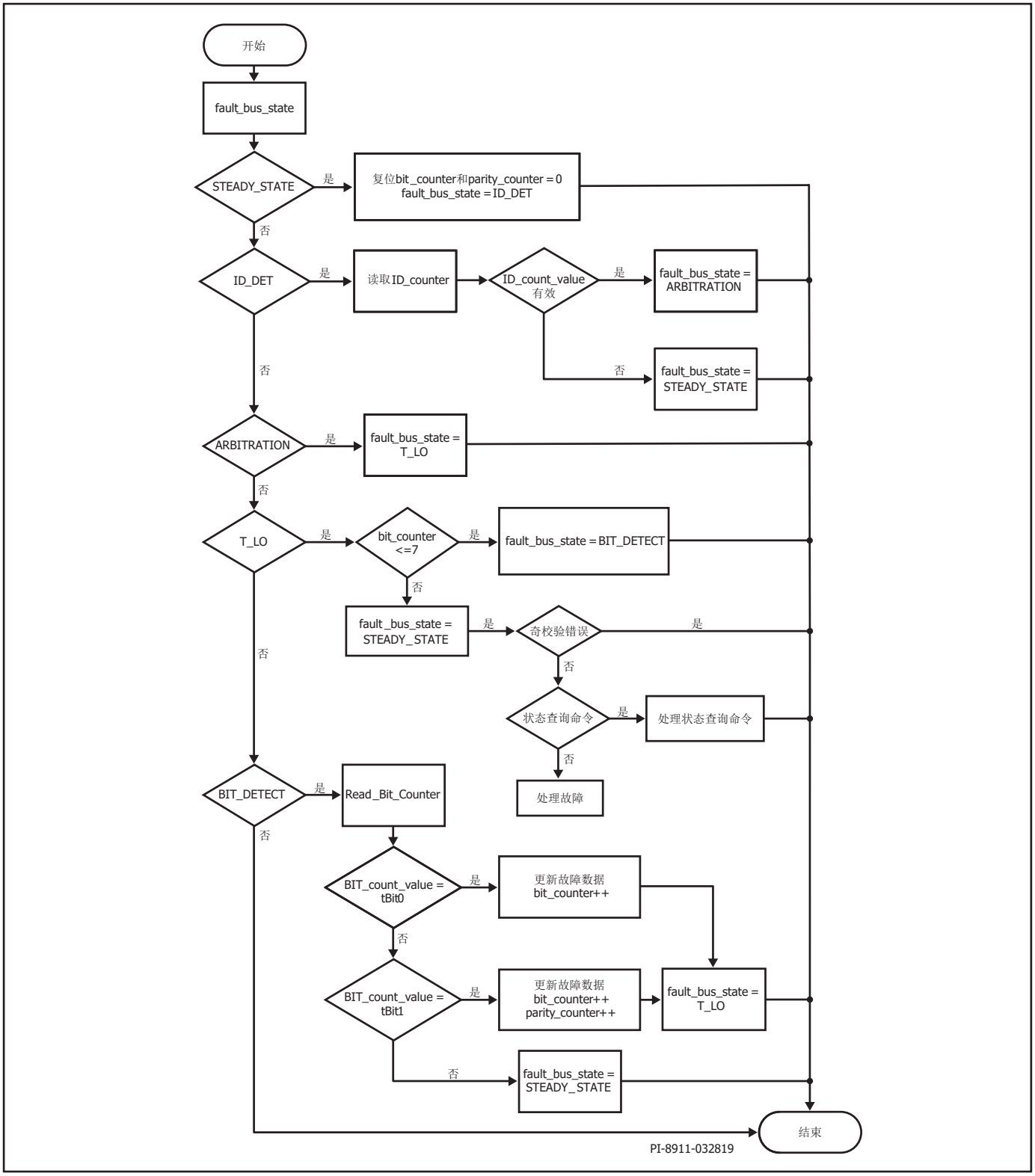

图6. 故障总线状态调节器

图7所示为故障处理函数的软件流程图。

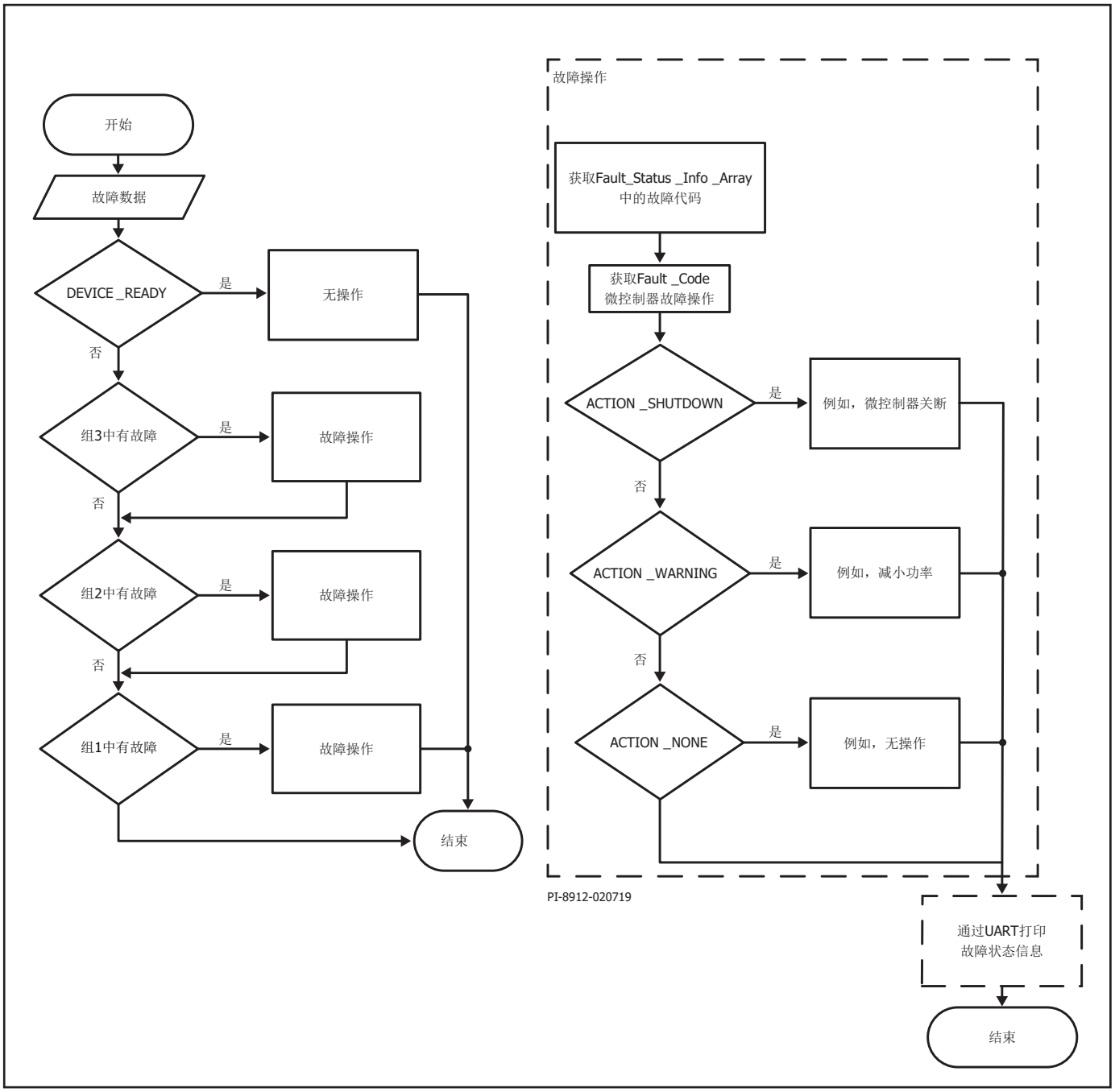

图7. 故障处理函数

接收的故障数据可能包括多个状态更新类型，故障处理函数应能够处理每个故障类型。故障位无法同时出现，组合起来可确定故障类型。表6列出了状态字组。来自*组1*、*组2*、*下管FET过流*和*上管FET过流*的故障可同时在一个状态字内报告。

分组	故障	位0	位1	位2	位3	位4	位5	位6
组1	上管总线过压	0	0	1				
	上管总线欠压100%	0	1	0				
	上管总线欠压85%	0	1	1				
	上管总线欠压70%	1	0	0				
	上管总线欠压55%	1	0	1				
	系统热故障	1	1	0				
	下管驱动器未就绪[1]	1	1	1				
组2	下管FET热告警				0	0		
	下管FET热关断				1	0		
	上管驱动器未就绪[2]				1	1		
下管FET过流							1	
上管FET过流								1

表6. 故障状态分组

在此实施示例中，每个已报告故障类型都会调用故障操作函数。故障操作函数从*FAULT_STATUS_INFO_ARRAY*选取与故障代码对应的特定故障操作，随后由微控制器执行。请参见表5了解可能的特定操作。针对报告的状态更新所采取的操作，需要根据特定的应用要求进行调整。

本文档中的“故障检测示例”部分演示了通过UART控制台显示故障状态的状态更新解码方式。此外还介绍了微控制器在三相逆变板中所采取的特定操作。

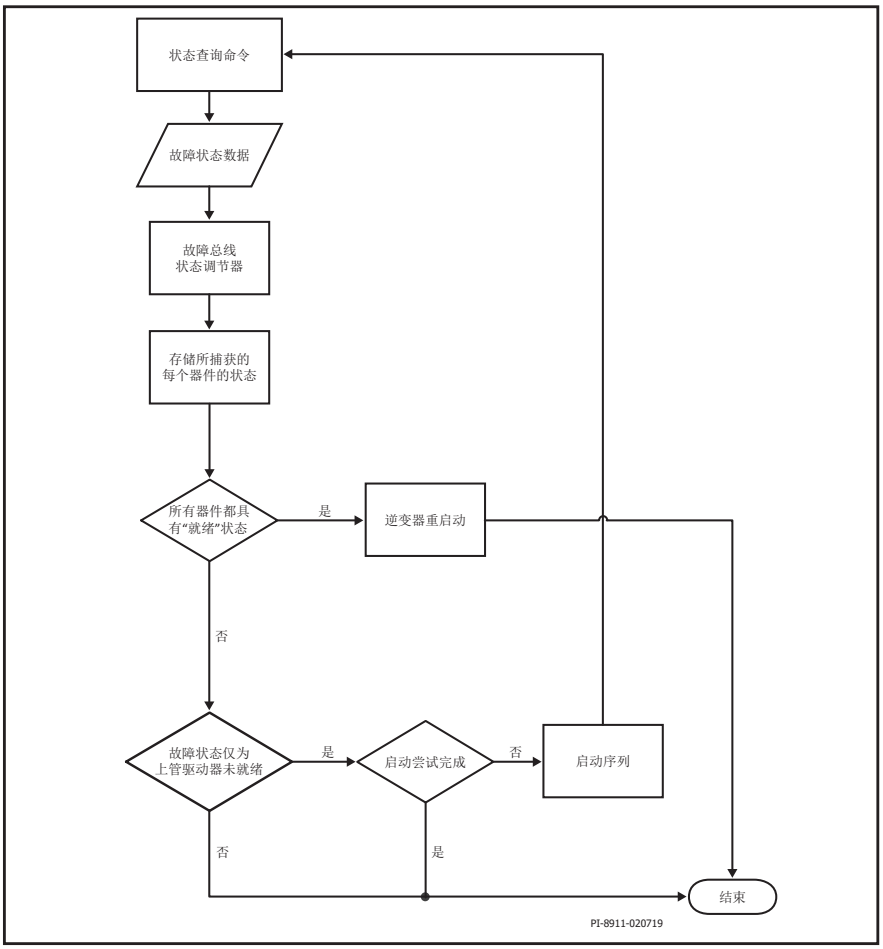

图8. 状态查询命令处理函数

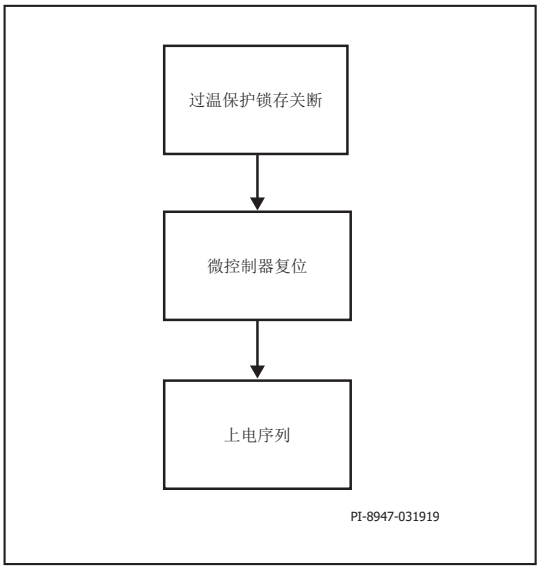

图9. 锁存复位命令函数

参考代码数据结构

故障状态调节器的状态
下面将介绍可在检测过程中确定当前故障信号的故障总线状态。

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

Fault_Status_Info_Array
此FAULT_STATUS_INFO数组是跟随已解码的故障状态更新的特定操作列表。

```
{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},
```

例如，过压条目{HV_BUS_OV, ACTION_SHUTDOWN}表示微控制器应在出现此错误时关断系统。

在此实现过程中，跟随故障状态的特定操作为：
ACTION_SHUTDOWN,
ACTION_WARNING,
ACTION_NONE,

故障状态代码
可出现的故障状态有：

```
//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
此结构包括所产生故障的故障代码和器件ID。

dev_id
fault

参考代码

此示例代码使用PSoC Creator IDE 4.1版开发而成，并基于CY8CKIT-042 PSoC Pioneer Kit器件及DER-654参考设计逆变板进行了测试。下面的代码演示了与故障信号处理相关的参考函数。此参考代码不包括通过UART控制台打印故障状态信息的代码片段（更多详情，请参见“注释”部分）。请参考所提供的代码文件了解所用的其他变量的定义。

```
/* ========================================================================
 * 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*****************************/
```

下面的代码演示了与本文档所介绍的状态查询和锁存复位命令示例实现相关的参考函数。对状态查询和锁存复位命令的调用应根据每个用户的用例在实际实现中单独处理。请参考所提供的代码文件了解所用变量的定义。

```
/*******************************************************************************
***
* 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();

   }
```

故障检测示例

所演示的故障检测示例和微控制器做出的决定将利用上文详述的参考代码执行表5中的典型操作。

UART终端可显示故障状态信息，用以阐明故障状态调节器的工作过程。所显示的信息采用*[device ID, fault, action]*格式。例如，UART消息*W, STS, and S*表示，状态更新来自器件W（器件1、2和3分别以U、V和W表示），故障状态是系统热关断(STS)，微控制器的操作是关断(S)。

图10所示为报告的系统热故障和关断逆变器的典型操作（参见图11中的UART终端输出）。

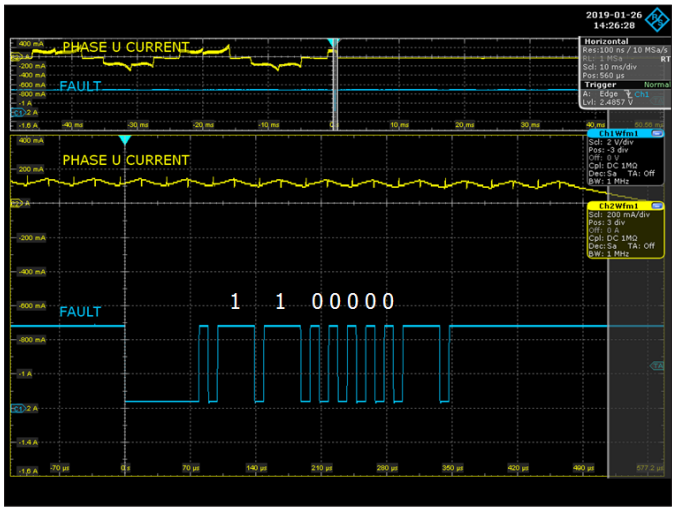

图10. 发生系统温度状态故障后的逆变器关断示例

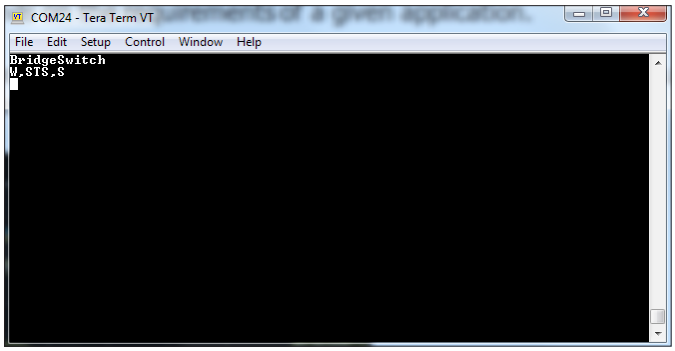

图11. 接收到系统热故障后的UART终端输出

图12所示为报告的下管过流故障和关断逆变器的典型操作（参见图13中的UART终端输出）。

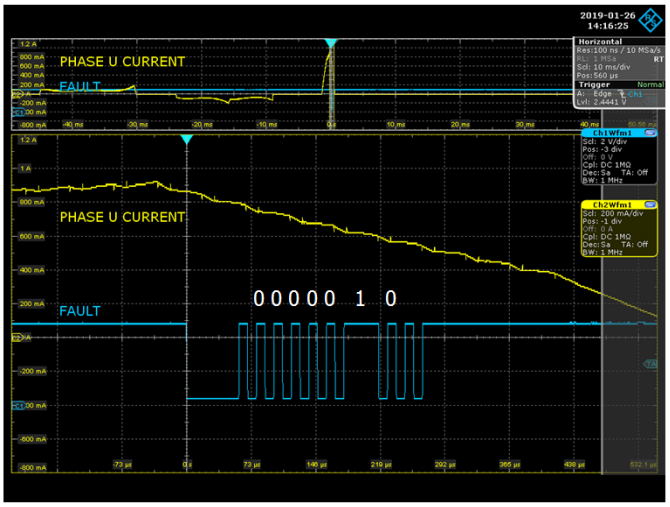

图12. 接收到下管过流故障后的逆变器关断示例

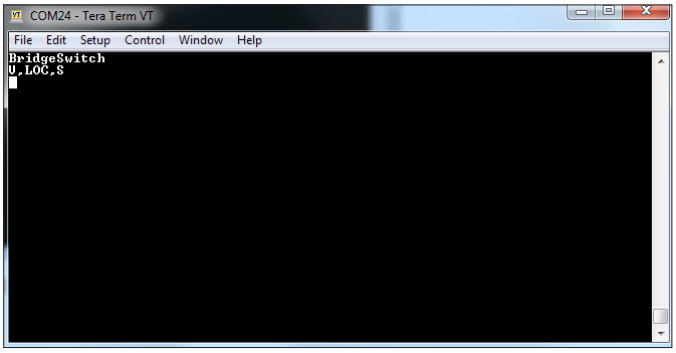

图13. 接收到下管过流故障后的UART终端输出

图14所示为报告的含有告警状态的高压总线UV85故障。在此实施示例中，微控制器未执行具体的操作，但只显示了告警状态。请参见图15中的UART终端。

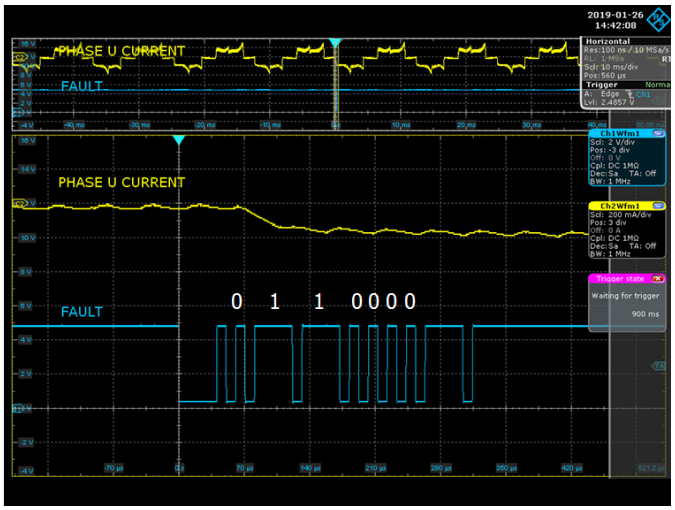

图14. 接收到高压总线UV85后的告警状态

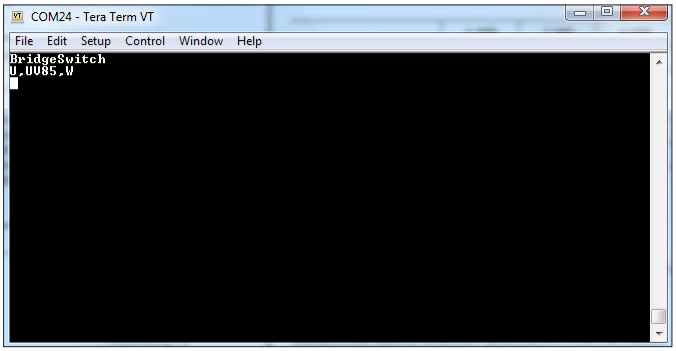

图15. 接收到高压总线UV85后的UART终端输出

图16所示为报告的高压总线过压和关断逆变器的典型操作（参见图17中的UART终端输出）。

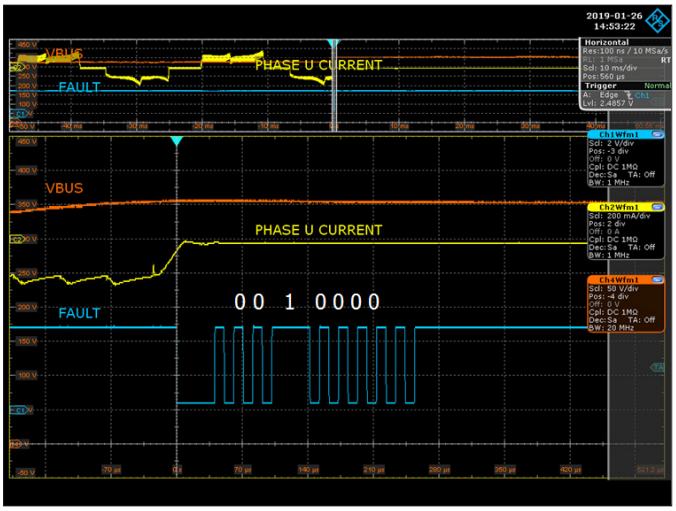

图16. 接收到高压总线过压后的逆变器关断示例

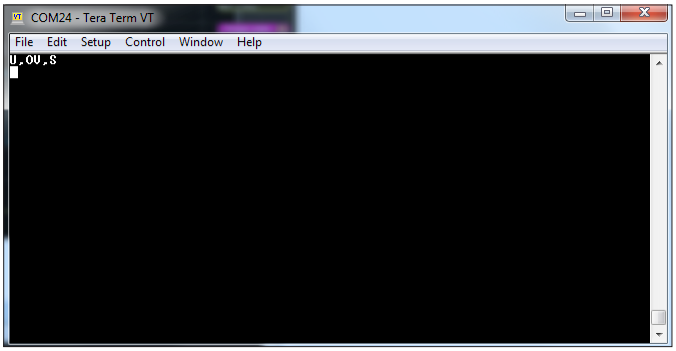

图17. 接收到高压总线过压后的UART终端输出

微控制器命令示例

图18所示为因输入过压故障导致关断而发出状态查询命令后的逆变器重启动。

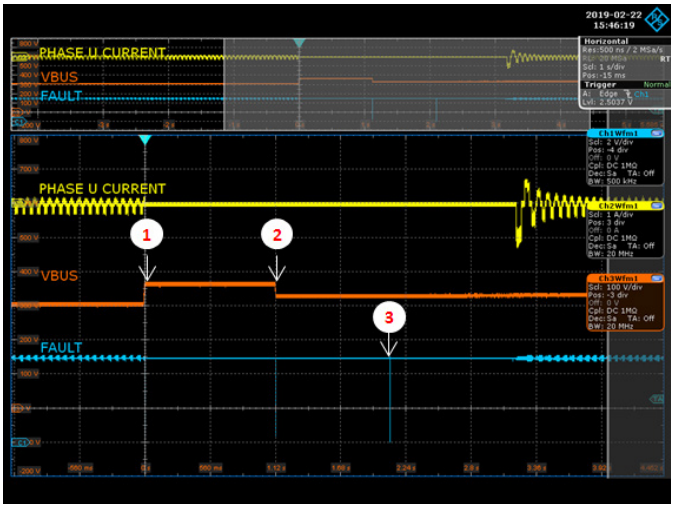

图18. 输入过压后的状态查询命令

(1) 发生输入过压，逆变器关断，过压状态如图19所示。(2) 过压故障已清除，并且(3)系统微控制器发送状态查询命令检查器件状态。图20所示为所有三个器件的状态查询命令和各自的状态报告。所有器件已报告为“就绪”后，系统微控制器重启动逆变器。

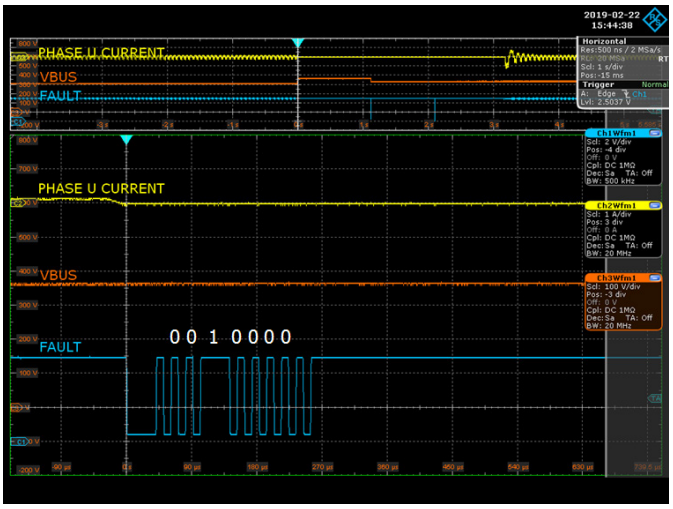

图19. 输入过压后的逆变器关断

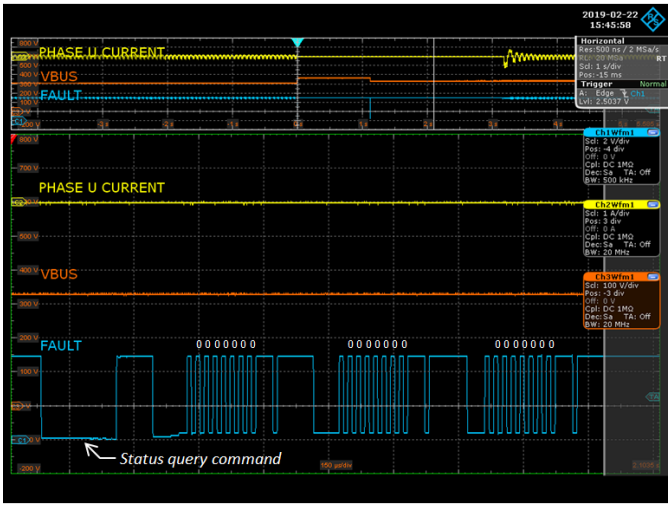

图20. 输入过压后的状态查询命令（所有器件已报告为“就绪”）

图21所示为因上管驱动器未就绪故障(1)导致关断而发出状态查询命令(2)后的逆变器重启动。

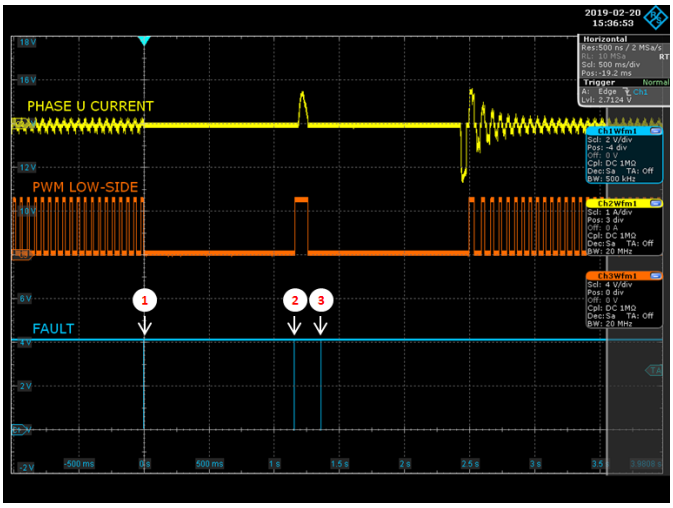

图21. 发生上管驱动器未就绪故障后的状态查询命令

在此示例中，报告的状态更新只有上管驱动器未就绪故障。微控制器将发出启动例程命令（也即，应用一个逻辑上管-下管PWM输入INL，并持续100 ms）。在(3)，微控制器将发送另一个状态查询命令以检查所有器件是否均“就绪”。在此示例中，所有故障都已清除，所有器件均“就绪”。微控制器启动逆变器重启动，并向BridgeSwitch控制输入端INL和/INH发送PWM信号。

图22所示为相对应的上管驱动器未就绪故障状态，图23所示为在尝试启动序列后所有三个器件的状态查询命令和各自的状态报告。所有器件已报告为“就绪”后，系统微控制器重启动逆变器。

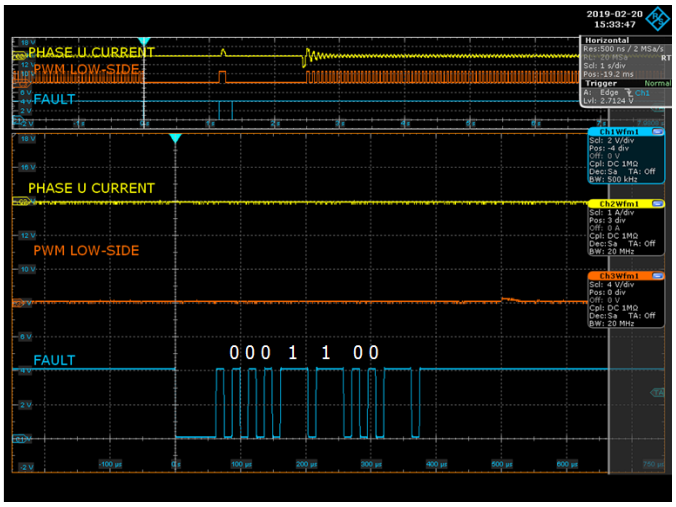

图22. 发生上管驱动器未就绪故障后的逆变器关断

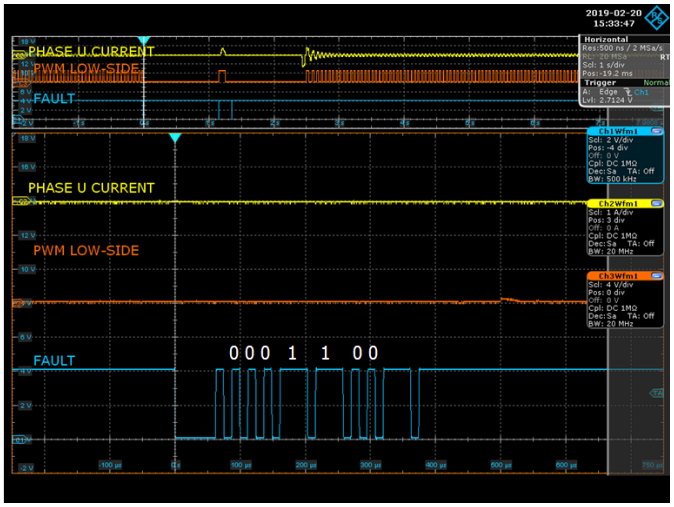

图23. 启动序列后的状态查询命令

图24所示为因过温故障导致锁存关断而发出的锁存复位命令。微控制器在发送锁存复位命令后会应用完整的上电序列。

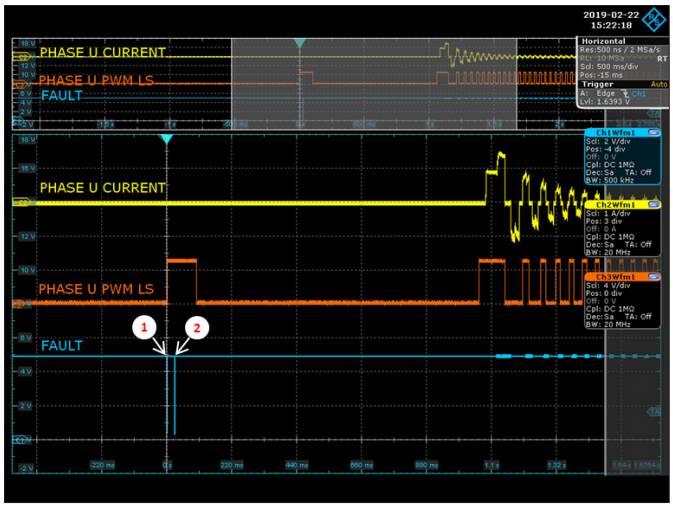

图24. 锁存过温保护后的锁存复位命令和上电序列

图25所示为锁存复位命令(1)和上管未就绪的默认状态报告（请注意，调用锁存复位命令时将禁用默认检测）。微控制器在发送锁存复位命令后会应用启动（上电）序列。在图26中，所有器件均报告为“就绪”(2)，然后启动逆变器。

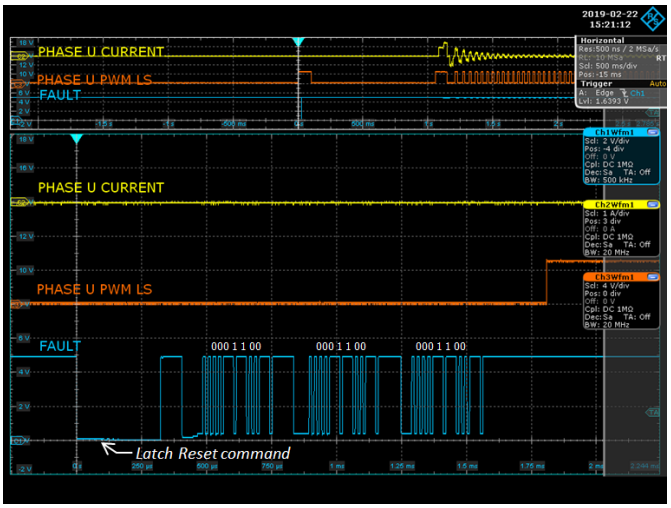

图25. 锁存过温保护后的锁存复位命令和上电序列

图26所示为已成功完成上电序列，并且所有器件均报告为“就绪”状态。

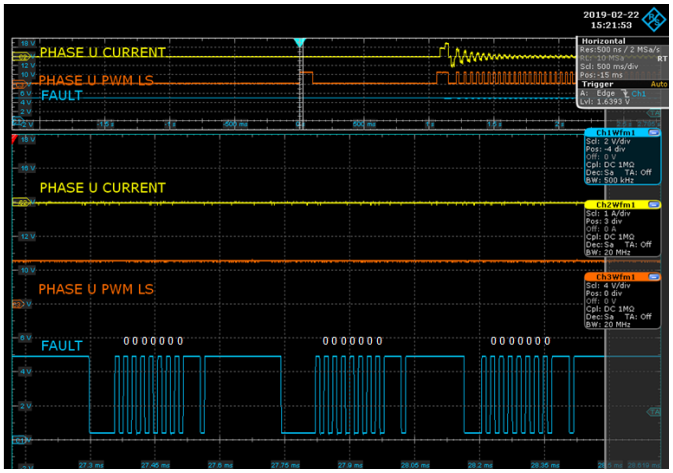

图26. 启动序列后的器件状态

示例代码库

示例代码库可从BridgeSwitch产品页面(www.power.com)下载，链接如下：

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

注释

本应用指南介绍了故障信息通过UART控制台的显示方式，用于调试目的。显示的执行应以轮询方式实现，以限制微控制器的负载。尽量减少所显示信息的数量也可以降低负载。

修订版本	注释	日期
A	初始版本。	04/19

有关最新产品信息，请访问：www.power.com

Patent Information

The products and applications illustrated herein (including transformer construction and circuits external to the products) may be covered by one or more U.S. and foreign patents, or potentially by pending U.S. and foreign patent applications assigned to Power Integrations. A complete list of Power Integrations patents may be found at www.power.com. Power Integrations grants its customers a license under certain patent rights as set forth at www.power.com/ip.htm.

Life Support Policy

POWER INTEGRATIONS PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF POWER INTEGRATIONS. As used herein:

1. A Life support device or system is one which, (i) is intended for surgical implant into the body, or (ii) supports or sustains life, and (iii) whose failure to perform, when properly used in accordance with instructions for use, can be reasonably expected to result in significant injury or death to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

Power Integrations全球销售支持网络

全球总部
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

中国（上海）
徐汇区漕溪北路88号圣爱广场
1601-1603室
上海|中国，200030
电话：+86-21-6354-6323
电子邮箱：chinasales@power.com

中国（深圳）
南山区科技南八路二号豪威科技大厦
17层
深圳|中国，518057
电话：+86-755-8672-8689
电子邮箱：chinasales@power.com

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

德国（门极驱动器销售）
HellwegForum 1
59469 Ense
Germany
Tel: +49-2938-64-39990
e-mail: igbt-driver.sales@power.com

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

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

日本
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

韩国
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

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

中国台湾
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

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