應用說明 AN-80
BridgeSwitch 產品系列

BridgeSwitch 故障通訊介面

簡介

本應用說明介紹了 BridgeSwitch™ 故障狀態通訊介面功能的軟體實施指南。其中概述了 BridgeSwitch 故障狀態通訊介面、擷取和處理接收狀態更新的狀態機、參考代碼及其資料結構，透過 UART 終端顯示狀態更新來演示軟體，同時給出了在變頻器電路板中實施故障保護的範例。

BridgeSwitch 故障狀態通訊介面

BridgeSwitch 裝置可透過其開汲極故障輸出將狀態更新 (包括內部和系統級故障) 傳達給系統 MCU。它使用 7 位元字詞模式後跟奇校驗位元來報告狀態更新。

以下部分詳細介紹了故障匯流排規格。

硬體配置

要將所有偵測到的狀態更新均傳送到系統微控制器，所有故障接腳都連線到單線匯流排，此匯流排上拉至系統供應電壓。圖 1 顯示了單線匯流排配置中三個 BridgeSwitch 裝置與系統 MCU 通訊的典型介面。

裝置 ID 接腳連線允許每個裝置根據裝置 ID 接腳連線為自己分配唯一的裝置 ID。該裝置 ID 允許透過在狀態通訊開始時將故障匯流排拉低相應的裝置 ID 週期 t_{ID}，將偵測到的故障狀況的物理位置傳達給系統微控制器。

表1列出了裝置 ID、得到的裝置 ID 時間段 t_{ID}，以及如何透過 ID 接腳連線對相應 ID 進行程式化。

裝置 ID	t_{ID}	ID 接腳連線
1	40 µs	BPL 接腳
2	60 µs	浮接
3	80 µs	SG 接腳

表 1 – 透過 ID 接腳選擇裝置 ID。

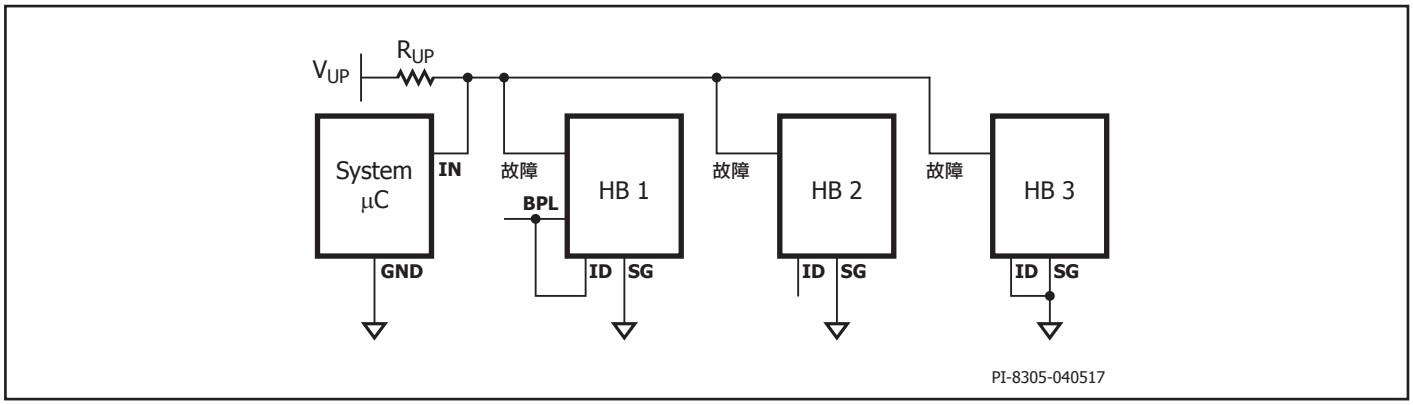

圖 1.　採用裝置 ID 程式化的單線狀態通訊匯流排。

故障狀態通訊匯流排規格

狀態編碼

透過 7 位元字詞後跟校驗位元對故障資訊進行編碼。表 2 總結了裝置可以傳達給系統微控制器的各種狀態更新的編碼。狀態字詞由五個區塊組成，其中不能同時發生的狀態變更分為一組。這樣就可以同時向系統微控制器報告多個狀態更新，而無需關注故障優先級和故障報告佇列。

最後一列 (7 位元字詞 “000 00 0 0”) 對裝置就緒狀態進行編碼，用於在清除某個故障時向系統傳送成功的開機序列，並在無任何故障時將其發送給系統 MCU 以確認狀態請求。

故障匯流排上的通訊由以下原因之一啟動：

- 成功開機後，就可以進行任務模式通訊。
- 故障狀態暫存器更新通訊由其中一個裝置啟動。
- 查詢系統微控制器後的當前狀態通訊。

故障	位元 0	位元 1	位元 2	位元 3	位元 4	位元 5	位元 6
HV 匯流排 OV	0	0	1				
HV 匯流排 UV 100%	0	1	0				
HV 匯流排 UV 85%	0	1	1				
HV 匯流排 UV 70%	1	0	0				
HV 匯流排 UV 55%	1	0	1				
系統過熱故障	1	1	0				
S 驅動器未就緒[1]	1	1	1				
LS FET 過熱警告				0	0		
LS FET 過溫關閉				1	0		
HS 驅動器未就緒[2]				1	1		
LS FET 過流						1	
HS FET 過流							1
裝置就緒 (無故障)	0	0	0	0	0	0	0

附註：

1. 包含 XL 接腳開路/短路故障、IPH 接腳至 XL 接腳短路，以及調整位元信號錯亂。
2. 包含 HS 至 LS 通訊中斷 V_{BPH} 或內部 5 V 電壓導軌超出範圍，以及 XH 接腳開路/短路故障。

表 2.　狀態字詞編碼。

圖 2 描繪了所有這三種情況下的故障介面通訊流程圖。

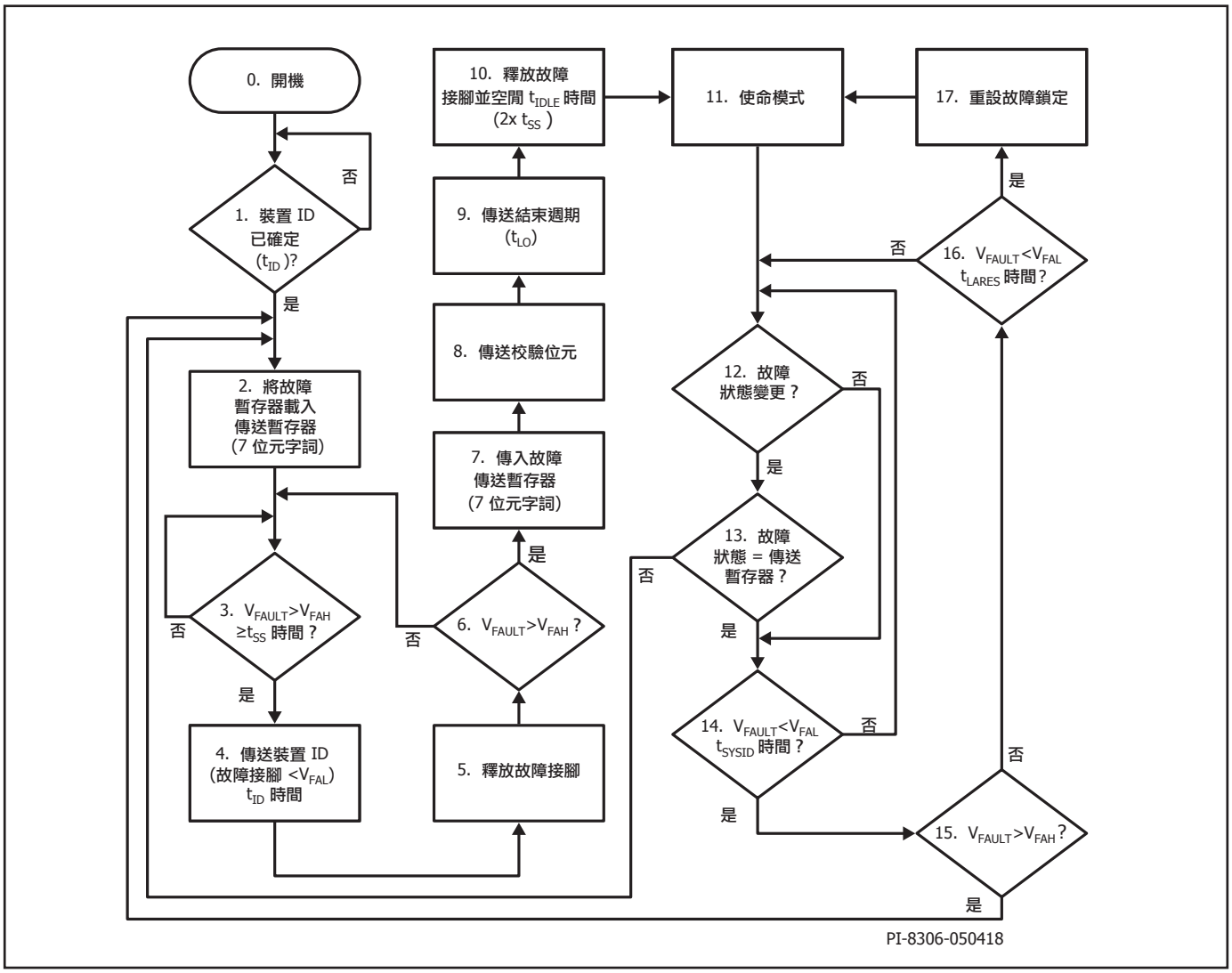

圖 2.　狀態通訊流程圖。

狀態更新通訊總是從通訊裝置啟動的匯流排仲裁開始。如果匯流排已經靜默至少 80 µs，則此通訊透過下拉故障接腳傳送相應的裝置 ID 時間段 t_{ID}。在裝置贏得匯流排仲裁後，其發送當前故障暫存器 (7 位元字詞)，然後發送奇校驗位元和傳送結束訊號，如通訊流程圖 (圖 2) 所示。

位元串流計時

圖 3 描繪了 BridgeSwitch 用於狀態更新通訊的位元串流計時圖。兩個邏輯狀態在故障接腳上以兩個不同的電壓訊號高電平時間段編碼，然後是低電平時間段 t_{LO} (通常為 10 µs)。邏輯 “1” 以時間段 t_{BIT1} (通常為 40 µs) 編碼，而邏輯 “0” 以時間段 t_{BIT0} (通常為 10 µs) 編碼。表 3 列出了故障狀態通訊的計時表。

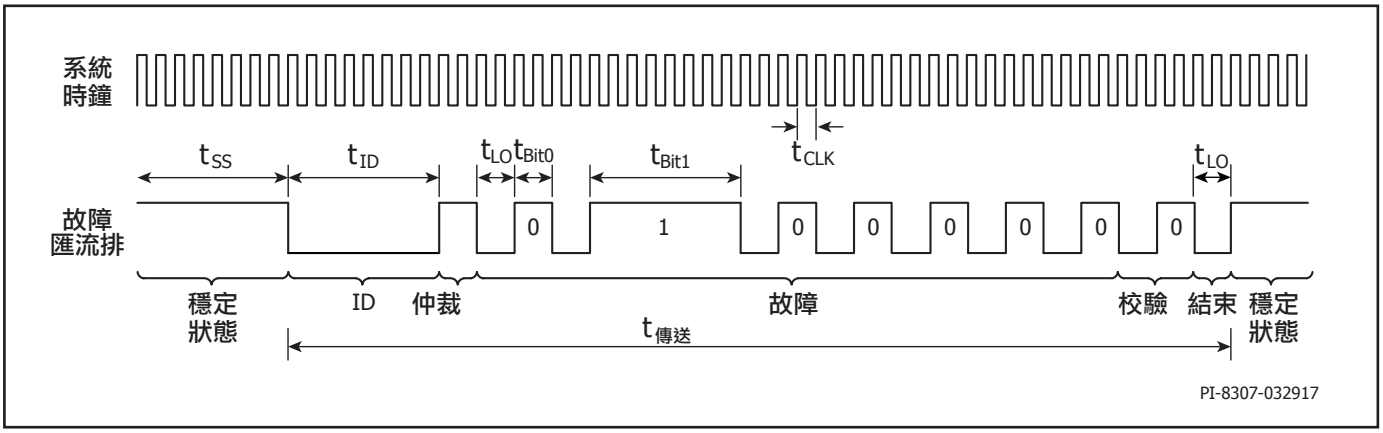

圖 3. 狀態通訊位元串流。

符號	說明	邏輯狀態	時間段 (典型值)
t_{ID}	裝置 ID	0	請參閱表 1
t_{LO}	低電平時間	0	10 µs
t_{Bit0}	邏輯 0	1	10 µs
t_{Bit1}	邏輯 1	1	40 µs

表 3. 位元串流計時表。

在每次完成傳送之後，裝置將在開始新通訊之前空閒 t_{IDLE} (通常為 2 × t_{SS} = 160 µs)。這使得匯流排上的其他裝置能夠傳達可能發生的狀態變更或回應系統微控制器發送的狀態查詢。

裝置僅通訊一次每個偵測到的狀態更新。它還會向系統 MCU 報告所有系統級故障的狀態變更。這包括 DC 匯流排欠壓和過壓條件以及外部溫度監測器故障。此外，裝置會報告裝置內部故障的所有狀態級變更，但 LS 電源 FREDFET 過熱關機除外。

BridgeSwitch 裝置還會監控故障匯流排，以取得系統微控制器處於任務模式後可能發送的指令。這可能是微控制器透過將匯流排拉低時間段 t_{SYSID} (通常為 160 µs) 進行的狀態更新查詢 (參見圖 2 中的步驟 15)。或者，這可能是重設裝置狀態暫存器的指令 (包括過溫關機鎖定)，以及透過將故障匯流排拉低 t_{LARES} 時間段 (2 倍的 t_{SYSID} =通常為 320 µs) 進入開機序列模式的指令（參見圖 2 中的步驟 17）。MCU 發送鎖定重設指令後，建議使用開機序列。表 4 總結了可用的系統微控制器指令。

匯流排下拉時間段	指令
t_{SYSID}	狀態查詢
$t_{LARES(2xTSYSID)}$	狀態暫存器，包括過溫鎖定重設和開機序列模式

表 4. 系統 MCU 指令。

軟體實施

本節介紹狀態機的實施，狀態機根據上一節中描述的狀態通訊規格從每個 BridgeSwitch 裝置擷取和處理狀態更新。

所呈現的範例使用基於中斷的實施。使用者必須根據其特定的應用程式要求來決定中斷優先級，例如馬達控制演算法或微控制器的類型。

系統 MCU 的周邊設備

為了演示故障匯流排通訊介面的實施，我們使用 Cypress PSoC Creator IDE 4.1 版開發參考代碼，並在 Cypress PSoC 4 MCU (CY8CKIT-042 PSoC Pioneer 套件) 上進行了測試。MCU 電路板提供內建的程式設計器和偵錯器，兩者透過 USB 接頭相接，實現與 PC 的通訊。透過如下方式展示狀態機：在故障偵測範例部分中描繪的 UART 主控台上列印接收的狀態更新。

故障狀態通訊匯流排在開放式汲極驅動模式下連線到單個雙向 MCU 接腳。該接腳連接一個定時器，後者擷取訊號的上升和下降邊緣。使用兩個 16 位元定時器/計數器區塊 *Bit_counter_timer* 和 *ID_counter_timer* (時鐘頻率為 12 MHz)，以基於中斷的方式處理故障訊號。*Bit_counter_timer* 從上升到下降邊緣擷取訊號，而 *ID_counter_timer* 從下降到上升邊緣擷取訊號。這兩個定時器擷取計數值，並在分別接收到故障訊號的下降邊緣和上升邊緣時產生中斷。故障狀態機例程處理每個接收的中斷。

軟體描述

軟體實施首先是將 *fault_bus_state* 變數初始化為空閒狀態 *(STEADY_STATE)*。一旦收到中斷，*fault_bus_state* 變數就會擷取故障訊號的狀態。初始化函數 *init_fault_bus_interrupt()* 初始化定時器/計數器擷取連接埠，並啟用上升邊緣和下降邊緣的擷取中斷。只要中斷服務例程 (ISR) 觸發，系統就會叫用 *fault_detect()* 函數。此函數是故障狀態機例程，用於擷取和處理收到的故障。主要軟體流程的高級檢視如圖 5 所示。

故障狀態機例程基本上是處理 ISR 事件並基於故障處理中的當前狀態更新 *fault_bus_state* 變數。故障狀態機的狀態可以是 *STEADY_STATE*、*ID_DET*、*ARBITRATION*、*T_LO 和 BIT_DETECT*。故障狀態機的詳細軟體流程圖如圖 6 所示。只要收到沒有校驗錯誤的完整故障狀態封包，它就會叫用故障處理函數 *fault_process()*，否則會重新進行同步，並在其中將 *fault_bus_state* 重設為 *STEADY_STATE*。

故障處理函數解碼接收到的故障狀態更新並叫用所需的動作。例如，在接收到過電流故障後關閉變頻器或在接收到熱警告狀態更新後降低變頻器輸出功率。故障狀態儲存在 *fault* 變數中，每次收到新的狀態更新時，該變數都會更新。使用者應根據收到的故障狀態和應用要求，提供必要的動作或決定 MCU 應執行的動作。表 5 列出了系統微控制器在接收到狀態更新後可以採取的範例性動作。*fault_process()* 函數軟體流程圖如圖 7 所示。

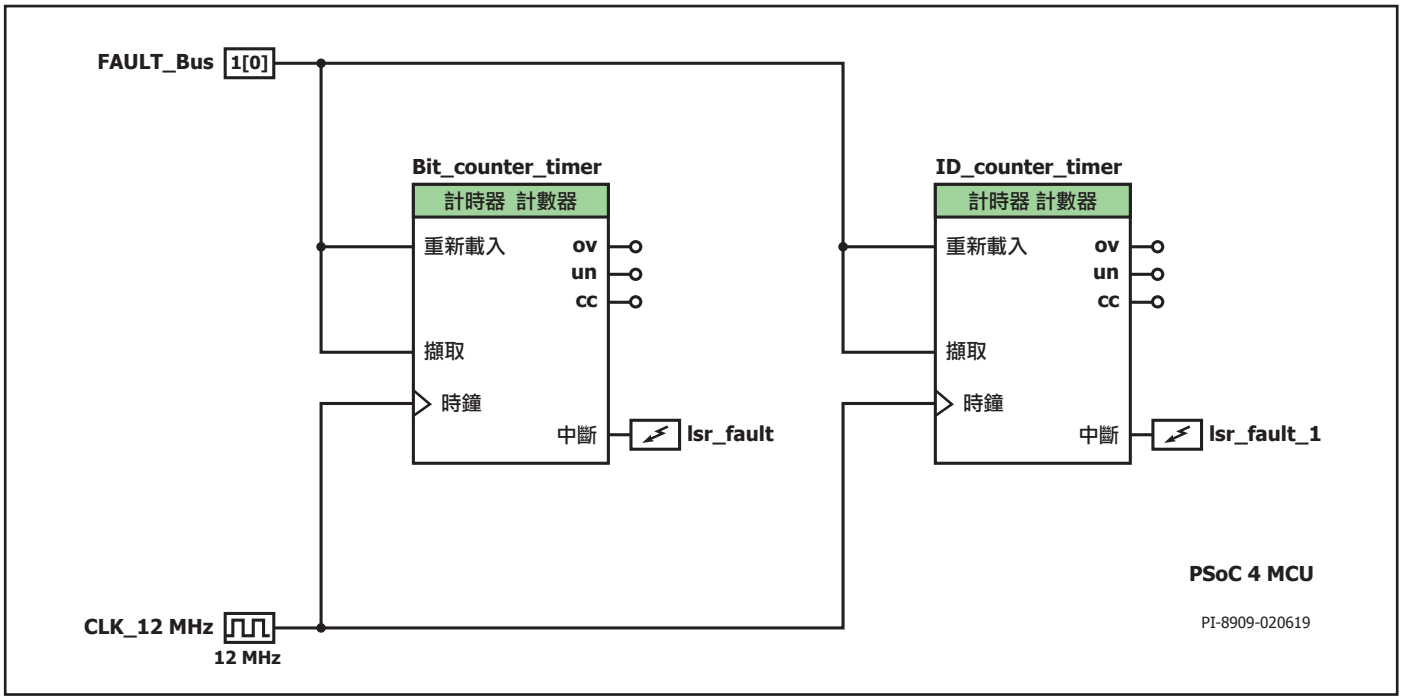

圖 4.　適用於 PSoC 4 MCU 的故障訊號處理用系統 MCU 周邊設備範例。

狀態查詢和鎖定重設指令
狀態查詢和鎖定重設指令的軟體實施支援以下範例使用案例：

狀態查詢指令：
每次想要在變頻器關閉一段時間後重新啟動變頻器（即傳送 PWM 訊號）時，MCU 就可以傳送狀態查詢。例如，在報告線路過壓或過流故障後。狀態查詢的主要目的是檢查所有裝置是否準備就緒或 MCU 是否必須啟動開機序列。圖 8 顯示了狀態查詢範例實施的流程圖。

狀態查詢例程檢查每個 BridgeSwitch 裝置的狀態，以確定以下哪個條件適用：

A. 每個裝置都回應「就緒」狀態 (不存在故障)：MCU 叫用 RESTART 指令重新啟動變頻器並傳送 PWM 訊號以控制 BridgeSwitch 的輸入。
B. 一個或多個裝置回應高壓側驅動器未就緒故障：MCU 叫用 START UP 指令，該指令啟動開機序列，將高壓側驅動器供應電壓 (相對於 HB 接腳的 V_{BPH}) 充電至其標準值。完成開機序列後，MCU 會跟進執行另一個狀態查詢指令。如果所有裝置備都回應「就緒」，則 MCU 叫用 RESTART 指令重新啟動變頻器。如果任何裝置均以「就緒」以外的狀態回應，則變頻器將保持關閉模式。

狀態查詢指令由 *status_query()* 函數處理，該函數在 t_{SYSID} = 160 µs 時將故障匯流排總線拉低 (參見表 4)。每個裝置都遵循此指令，並將連續傳送其狀態。在系統微控制器傳送狀態查詢後，偵測到的故障狀態將由 *process_status_query_command()* 函數 (置於故障匯流排狀態機函數內) 處理，該函數將儲存每個偵測到的裝置狀態並在 *status_query_action()* 函數中處理它們。*status_query_action()* 函數檢查收到的故障狀態，並按照表 5 中所述的條件提供動作。

鎖定重設指令：
MCU 可以在一個 (或所有) 裝置報告過溫故障 (並鎖定關閉) 後的某個時間傳送鎖定重設指令。圖 9 顯示了鎖定重設指令範例的流程圖。在鎖定重設指令執行後，建議使用開機序列。這確保了在切換恢復之前旁路高壓側電壓處於標準水平。

鎖定重設指令由 *latch_reset()* 函數處理，該函數將故障匯流排拉低 t_{LARES} = 320 µs (參見表 4) 以重設每個裝置的狀態。在系統微控制器傳送鎖定重設指令後叫用開機序列。請注意，每當傳送鎖定重設指令時，必須停用故障偵測功能。

故障	狀態字詞	軟體 動作/決定	附註
高電壓匯流排 OV	001 xxx x	關機	通常只有一個裝置監控 HV 匯流排，MCU 會關閉整個變頻器。
高電壓匯流排 UV 100%	010 xxx x	無	MCU 可以將馬達輸出增加到標準功率 – 前提是之前的狀態更新為 UV85、UV70 或 UV50。
高電壓匯流排 UV 85%	011 xxx x	警告	MCU 可以降低馬達輸出功率 (速度/扭力) 以減少變頻器負載。
高電壓匯流排 UV 70%	100 xxx x	警告	MCU 可以降低馬達輸出功率 (速度/扭力) 以減少變頻器負載。
高電壓匯流排 UV 55%	101 xxx x	警告	MCU 可以降低馬達輸出功率 (速度/扭力) 以減少變頻器負載。
系統過熱故障	110 xxx x	警告/關閉	取決於監控哪個外部元件溫度。
LS 驅動器未就緒	111 xxx x	關機	MCU 可能會在一段時間後嘗試重新啟動變頻器以檢查故障是否已得到清除。
LS FET 過熱警告	xxx 010 x	警告	MCU可以降低馬達輸出功率 (速度/扭力) 以減少變頻器負載或限制 PCB 溫度。
LS FET 過溫關閉	xxx 10x x	關機	鎖定關閉可能僅在一個裝置上發生，MCU 應該關閉整個變頻器，MCU 可以在冷卻期後重新啟動變頻器。
LS FET 過流	xxx xx1 x	關機	裝置自動關閉相應的 FREDFET，以保護馬達免受失速或過載情況的影響。MCU 關閉整個變頻器。
HS 驅動器未就緒	xxx 11x x	關機	MCU 可能會在一段時間 (幾秒) 後嘗試重新啟動變頻器以檢查故障是否已得到清除。
HS FET 過流	xxx xxx 1	關機	裝置自動關閉相應的 FREDFET，以保護馬達免受失速或過載情況的影響。MCU 關閉整個變頻器。
裝置就緒 (無故障)	000 000 0	無	在先前的狀態更新為 HV 匯流排 OV 或 LS/HS FET 過流後，MCU 可以重新啟動變頻器。如果先前的狀態更新為過熱警告，它還可以將馬達功率增加到標準值。

表 5.　接收狀態更新後微控制器採取的範例性動作。

軟體流程圖

下面的流程圖說明了故障訊號處理軟體的高級檢視。

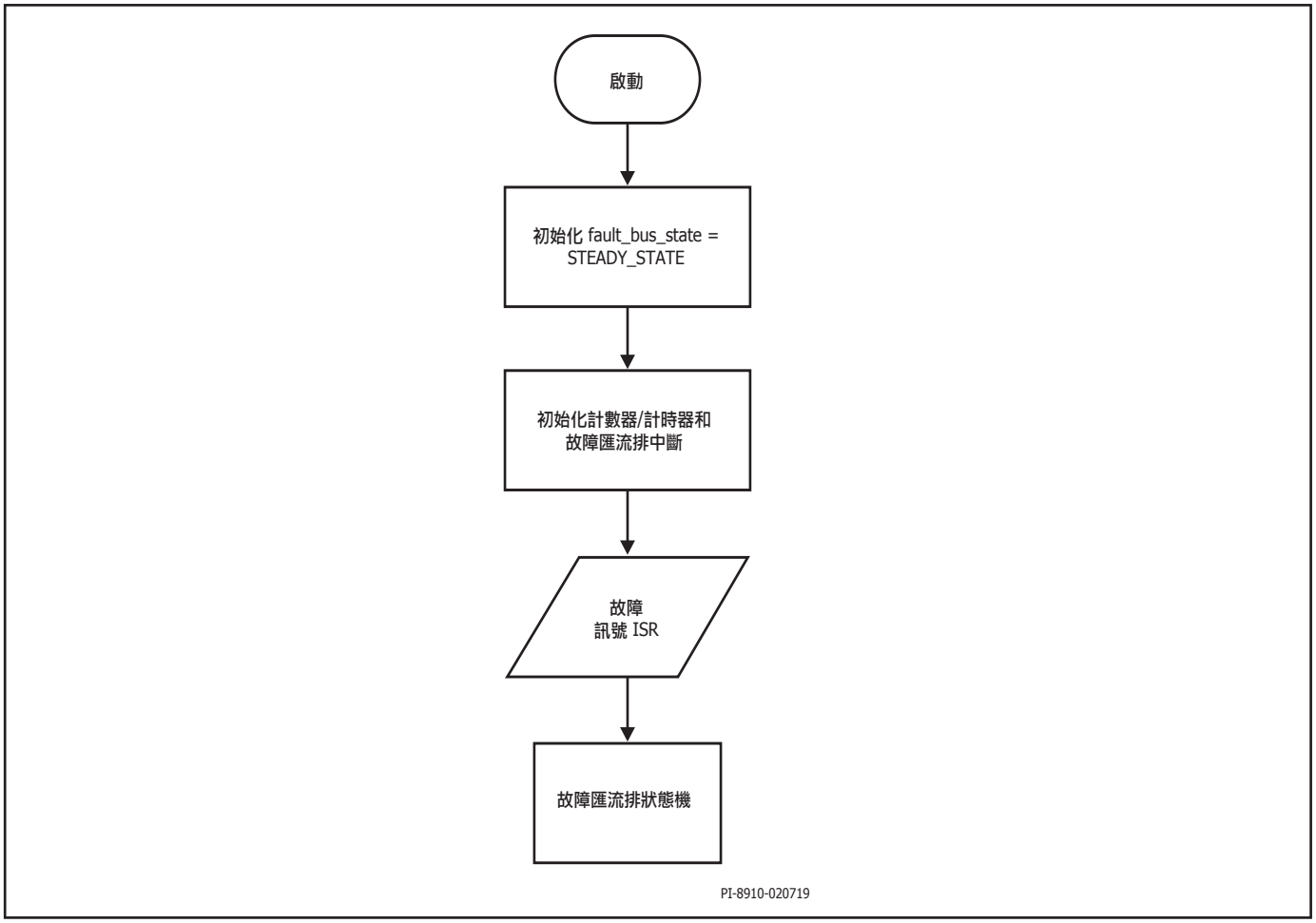

圖 5. 軟體中故障匯流排實施的高級檢視。

故障匯流排狀態機

圖 6 說明了故障匯流排狀態機的軟體流程圖。

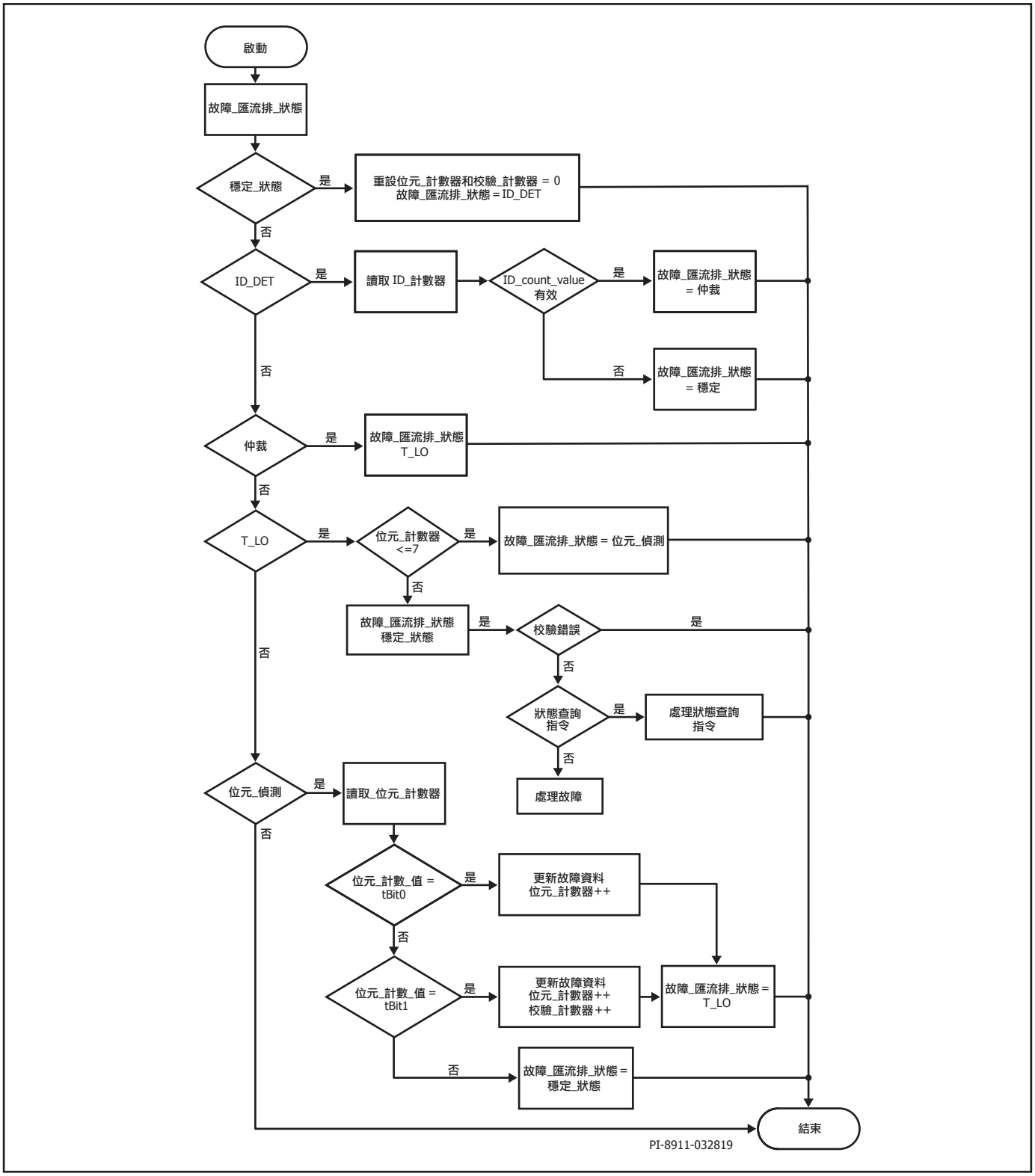

圖 6.　故障匯流排狀態機。

圖 7 說明了故障處理函數的軟體流程圖。

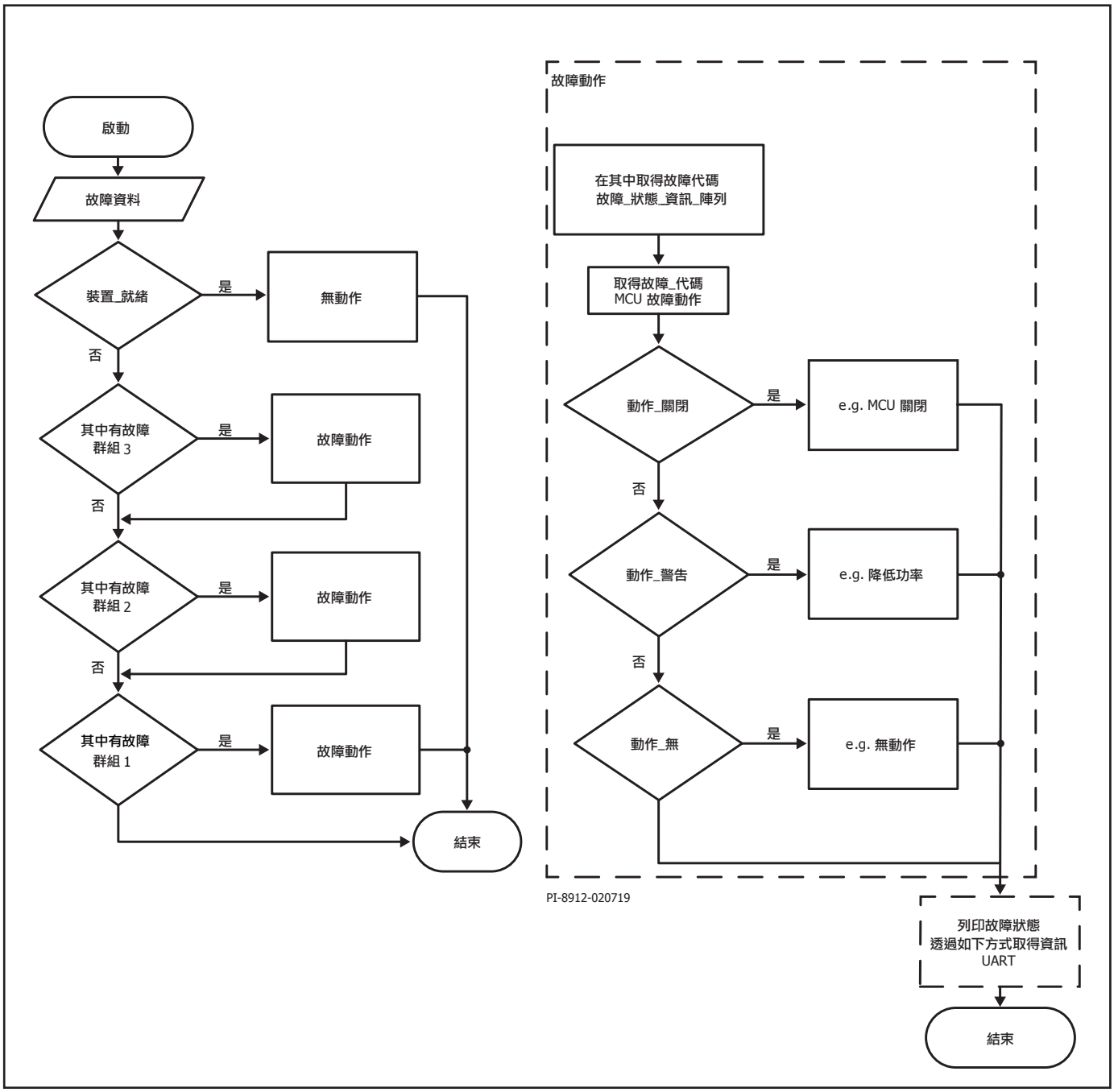

圖 7.　　故障處理函數。

收到的故障資料可能包含多種類型的狀態更新，而故障處理函數應該能夠處理每種故障類型。對於不能同時發生的故障位元，將它們組合在一起以確定故障類型。表 6 列出了狀態字詞群組。可在單個狀態字詞中同時報告來自 *GROUP1*、*GROUP2*、*低壓側 FET 過流*和*高壓側 FET 過流*的故障。

群組	故障	位元 0	位元 1	位元 2	位元 3	位元 4	位元 5	位元 6
GROUP1	HV 匯流排 OV	0	0	1				
	HV 匯流排 UV 100%	0	1	0				
	HV 匯流排 UV 85%	0	1	1				
	HV 匯流排 UV 70%	1	0	0				
	HV 匯流排 UV 55%	1	0	1				
	系統過熱故障	1	1	0				
	S 驅動器未就緒[1]	1	1	1				
GROUP2	LS FET 過熱警告				0	0		
	LS FET 過溫關閉				1	0		
	HS 驅動器未就緒[2]				1	1		
LS FET 過流							1	
HS FET 過流								1

表 6.　　故障狀態群組。

在此範例性實施中，將針對每個報告的故障類型叫用故障動作函數。故障動作函數從 *FAULT_STATUS_INFO_ARRAY* 中選擇對應於故障代碼的特定故障動作，MCU 隨後執行該故障動作。有關可能的特定動作的清單，請參閱表 5。報告狀態更新後的動作需要根據特定的應用要求進行調整。

本文件中的「故障偵測範例」段落透過在 UART 主控台中顯示故障狀態來演示狀態更新解碼。該段落還說明了 MCU 在三相變頻器電路板中採取的具體措施。

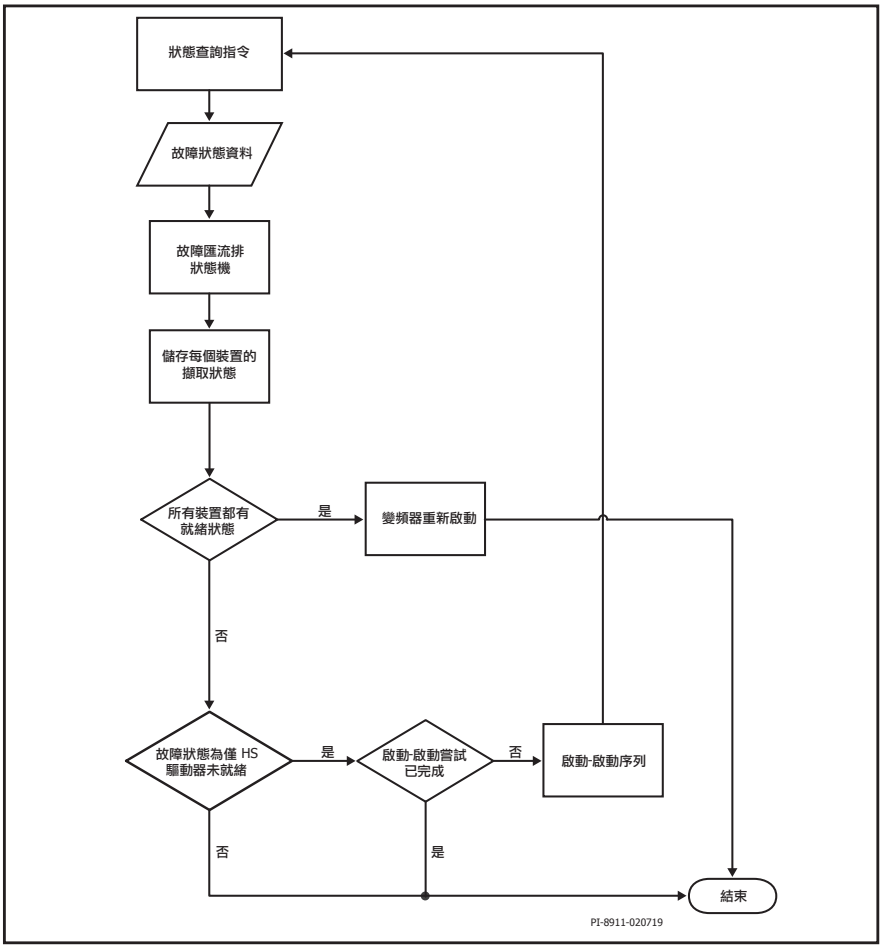

圖 8.　狀態查詢指令處理函數。

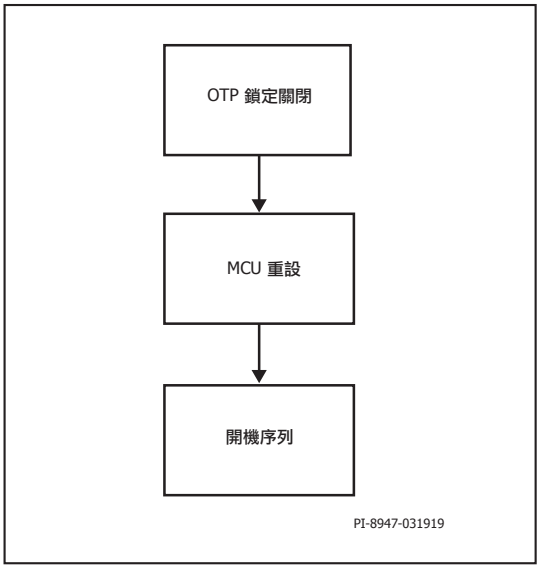

圖 9.　鎖定重設指令函數。

參考代碼資料結構

故障狀態機的狀態
以下是各種故障匯流排狀態，其確定偵測過程中的當前故障訊號狀態。

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

Fault_Status_Info_Array
此 FAULT_STATUS_INFO 陣列是解碼故障狀態更新後的特定動作清單。

```
{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，過壓條目 {HV_BUS_OV，ACTION_SHUTDOWN} 表示在發生此錯誤時 MCU 應關閉系統。

在此實施中，故障狀態更新後的特定動作是：
ACTION_SHUTDOWN,
ACTION_WARNING,
ACTION_NONE,

故障狀態代碼
可能發生的故障狀態是：

```
//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
該結構包含所發生故障的故障代碼和裝置 ID。

dev_id
故障

參考代碼

使用已在 CY8CKIT-042 PSoC Pioneer 套件裝置上測試的 PSoC Creator IDE 4.1 版開發此範例代碼，其中 CY8CKIT-042 PSoC Pioneer 套件裝置帶有 DER-654 參考設計變頻器電路板。下面的代碼給出了與故障訊號處理相關的參考函數。顯示的參考代碼不包括透過 UART 主控台列印故障狀態資訊的代碼片段（有關詳細資訊，請參閱「附註」段落）。有關所使用的其他變數的定義，請參閱提供的代碼檔案。

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

下面的代碼顯示了與本文件中描述的狀態查詢和鎖定重設指令範例實施相關的參考函數。對狀態查詢和鎖定重設指令的叫用應在實際實施中單獨處理，具體取決於每個使用者的使用案例。有關所使用的變數的定義，請參閱提供的代碼檔案。

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

   }
```

故障偵測範例

所展示的故障偵測範例和由微控制器做出的決定遵循表 5 中列出的範例性動作 (使用上面詳述的參考代碼)。

UART 終端顯示故障狀態資訊以說明故障狀態機。顯示的資訊格式為 *[裝置 ID，故障，動作]*。例如，*W，STS 和 S* 的 UART 訊息表示來自裝置 W 的狀態更新 (裝置 1、2 和 3 分別指定為 U、V 和 W)，故障狀態是系統過熱關機 (STS)，而 MCU 動作是關閉 (S)。

圖 10 描述了報告的系統過熱故障和關閉變頻器的範例性動作 (參見圖 11 中的 UART 終端輸出)。

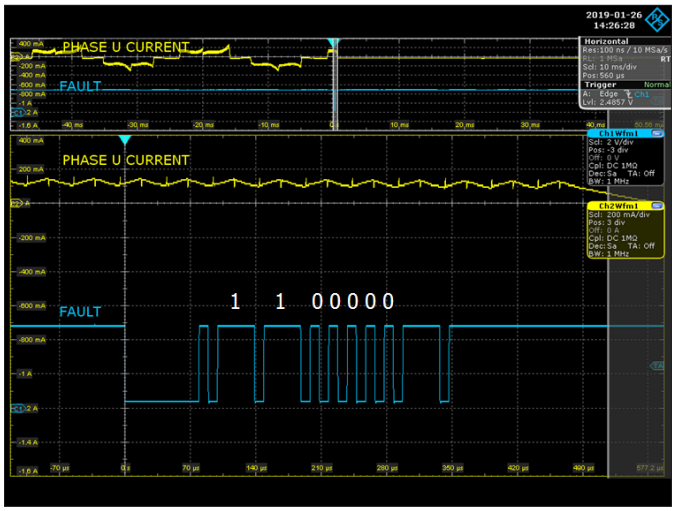

圖 10.　系統溫度狀態故障後的變頻器關閉範例。

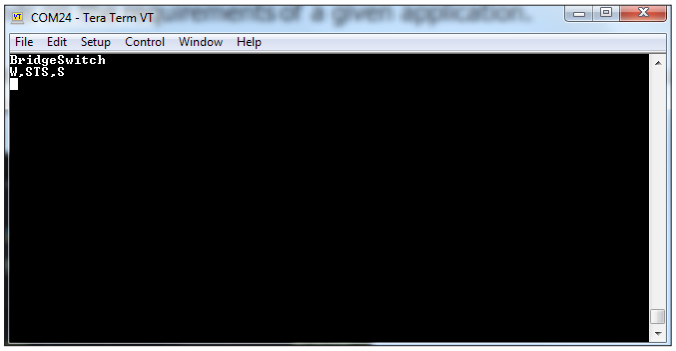

圖 11.　收到系統過熱故障後的 UART 終端輸出。

圖 12 描述了報告的低壓側過流故障和關閉變頻器的範例性動作 (參見圖 13 中的 UART 終端輸出)。

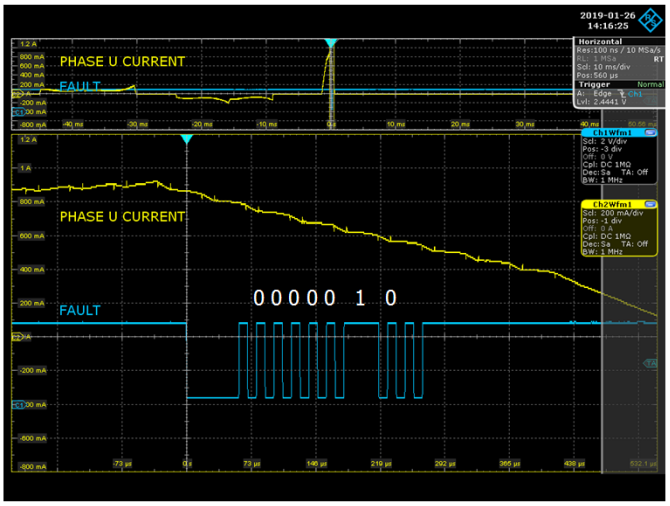

圖 12.　接收低壓側過流故障後的變頻器關閉範例。

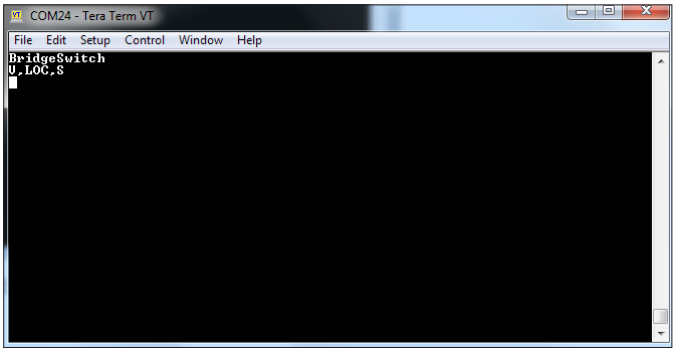

圖 13.　接收低壓側過流故障後的 UART 終端輸出。

圖 14 描繪了報告的具有警告狀態的高電壓匯流排 UV85 故障。在此範例性實施中，MCU 沒有採取特定動作，而是僅顯示警告狀態。請參閱圖 15 中的 UART 終端。

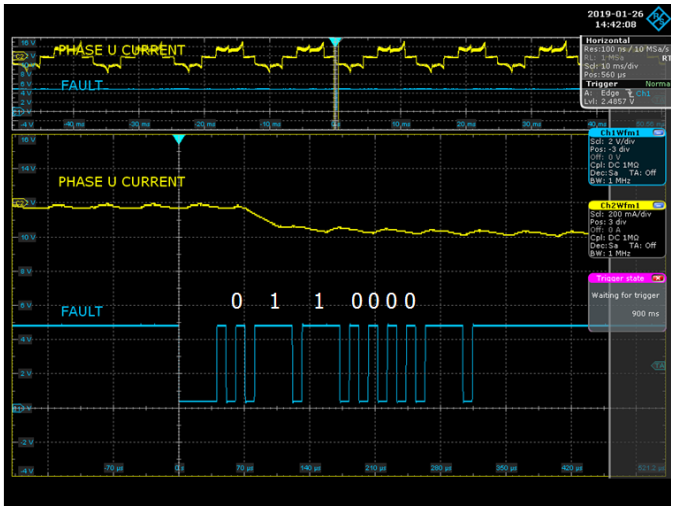

圖 14.　接收高電壓匯流排 UV85 後的警告狀態。

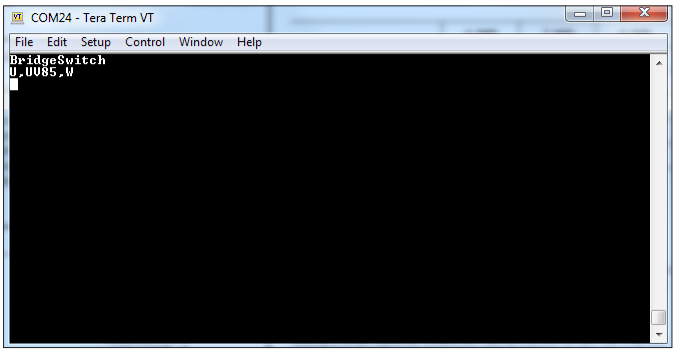

圖 15.　接收高電壓匯流排 UV85 後的 UART 終端輸出。

圖 16 描述了報告的高電壓匯流排過壓和關閉變頻器的範例性動作 (參見圖 17 中的 UART 終端輸出)。

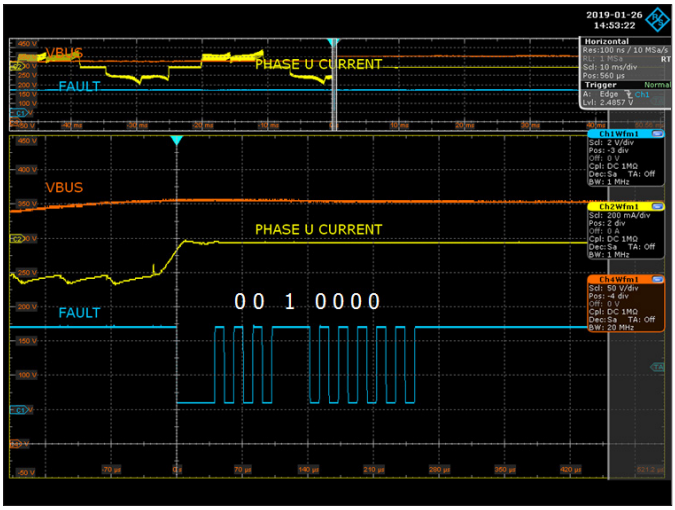

圖 16.　接收高電壓匯流排過壓後的變頻器關閉範例。

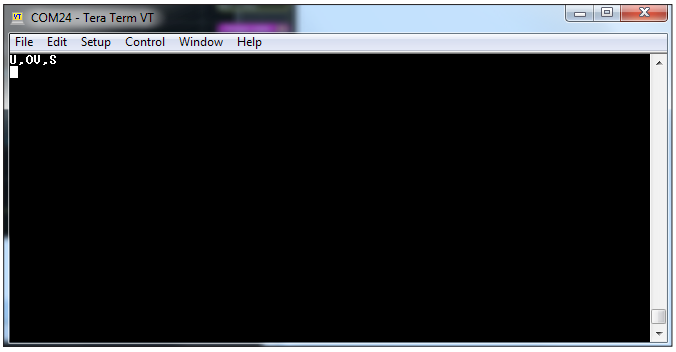

圖 17.　接收高電壓匯流排過壓後的 UART 終端輸出。

MCU 指令範例

圖 18 描繪了在根據線路過壓故障條件引起的關機執行狀態查詢指令之後的變頻器重新啟動。

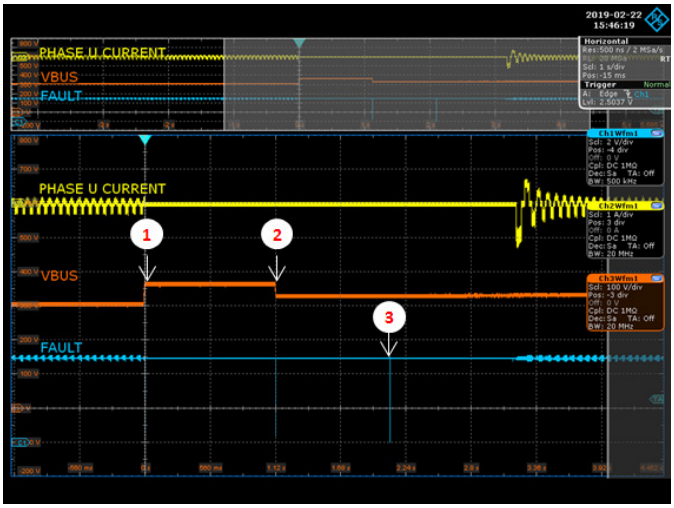

圖 18.　線路過壓條件之後的狀態查詢指令。

(1) 發生線路過壓，變頻器關閉，其 OV 狀態如圖 19 所示。(2) 過壓狀態已清除，並且在 (3) 時，系統微控制器傳送狀態查詢指令以檢查裝置狀態。圖 20 顯示了狀態查詢指令以及來自所有三個裝置的相應狀態報告。所有裝置都報告「就緒」，系統微控制器重新啟動變頻器。

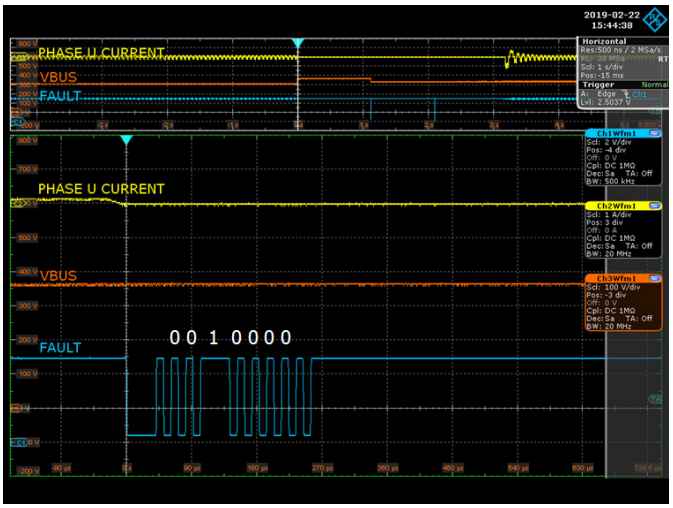

圖 19.　線路過壓條件之後的變頻器關閉。

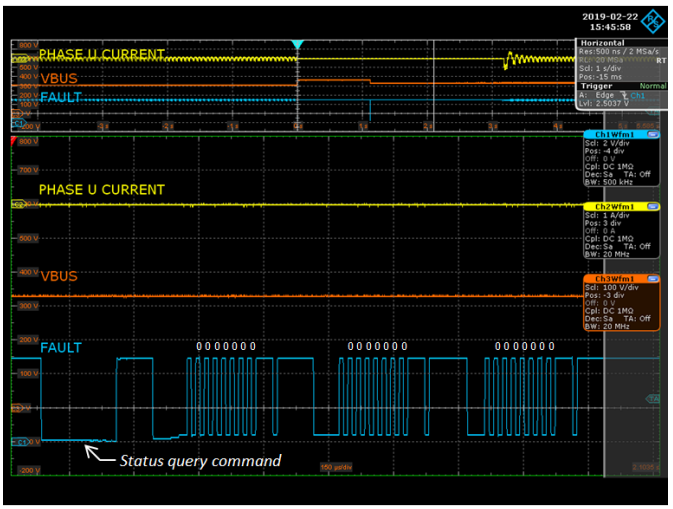

圖 20.　線路過壓條件之後的狀態查詢指令 (所有裝置都報告「就緒」)

圖 21 描繪了在根據高壓側驅動器未就緒故障 (1) 引起的關機執行狀態查詢 (2) 指令之後的變頻器重新啟動。

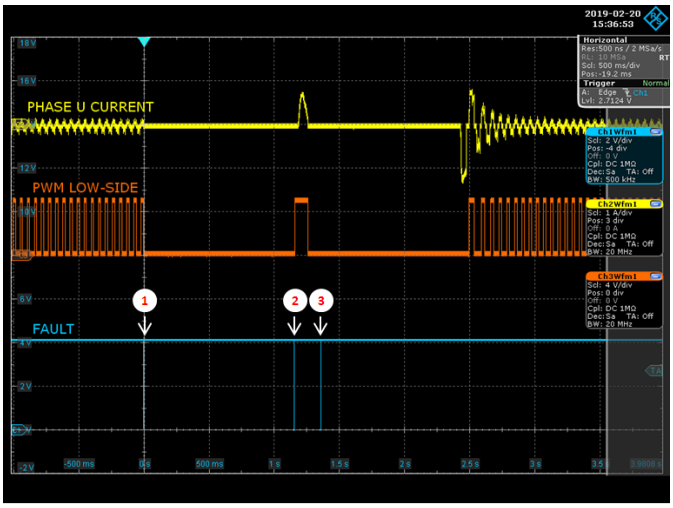

圖 21.　高壓側驅動器未就緒故障之後的狀態查詢指令。

在此範例中，報告的狀態更新僅是高壓側驅動器未就緒故障。MCU 將執行啟動例程 (即將邏輯高壓側施加到低壓側 PWM 輸入 INL，持續時間為 100 ms)。在 (3) 時，MCU 傳送另一個狀態查詢指令以檢查所有裝置是否都「就緒」。在此範例中，所有故障已得到清除且所有裝置都「就緒」。MCU 發起變頻器重新啟動並將 PWM 訊號傳送到 BridgeSwitch 控制輸入 INL 和 /INH。

圖 22 顯示了相應的高壓側驅動器未就緒故障狀態，圖 23 則顯示了狀態查詢指令以及嘗試執行啟動序列後所有三個裝置的相應狀態報告。所有裝置都報告「就緒」，系統微控制器重新啟動變頻器。

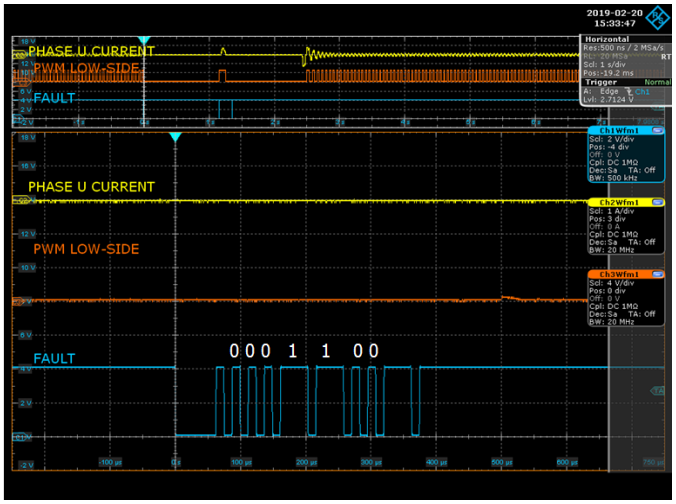

圖 22.　高壓側驅動器未就緒故障之後的變頻器關閉。

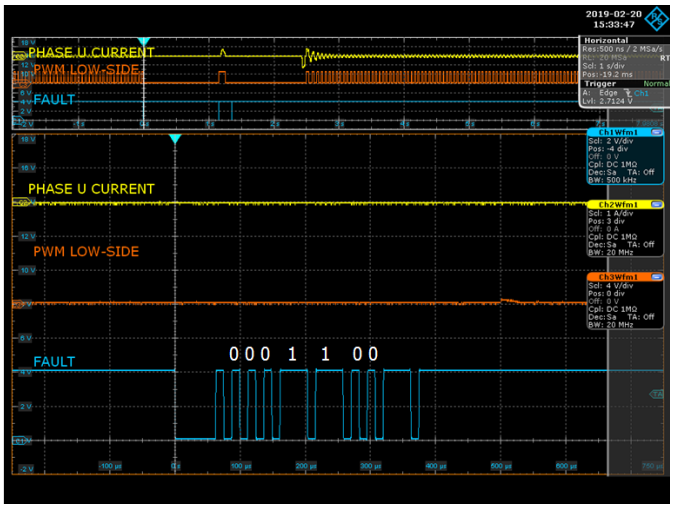

圖 23.　執行開機序列之後的狀態查詢指令。

圖 24 描繪了由過溫故障引起的鎖定關閉之後的鎖定重設指令。傳送鎖定重設指令後，MCU 套用完整的開機序列。

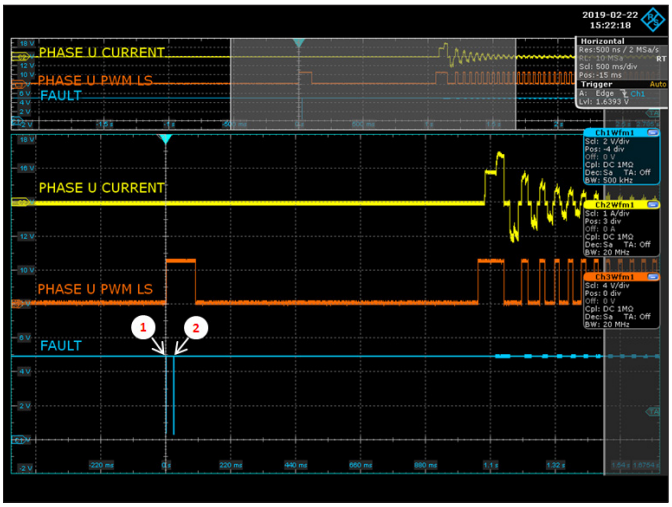

圖 24.　鎖定過溫保護後的鎖定重設指令和開機序列。

圖 25 顯示了鎖定重設指令 (1) 和高壓側未就緒的預設狀態報告 (請注意，應在調用鎖定重設指令時停用故障偵測）。傳送鎖定重設指令後，MCU 套用啟動 (開機)序列。圖 26 描繪了報告「就緒」的所有裝置 (2)，然後啟動變頻器。

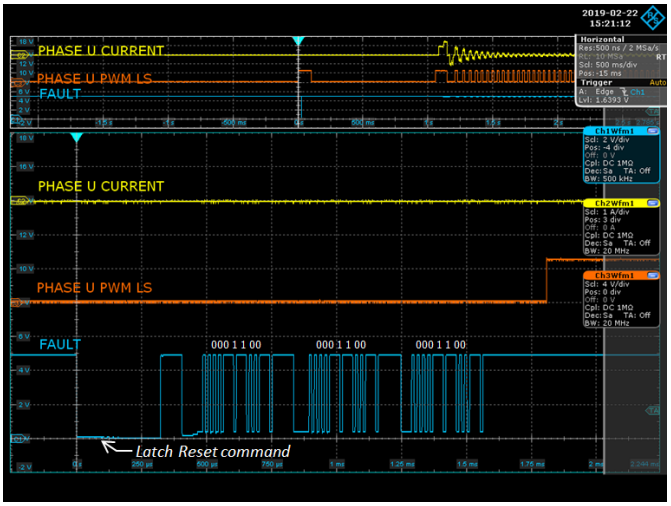

圖 25.　鎖定 OTP 之後的鎖定重設指令和開機序列。

圖 26 描繪了對所有報告「就緒」狀態的裝置成功完成開機序列。

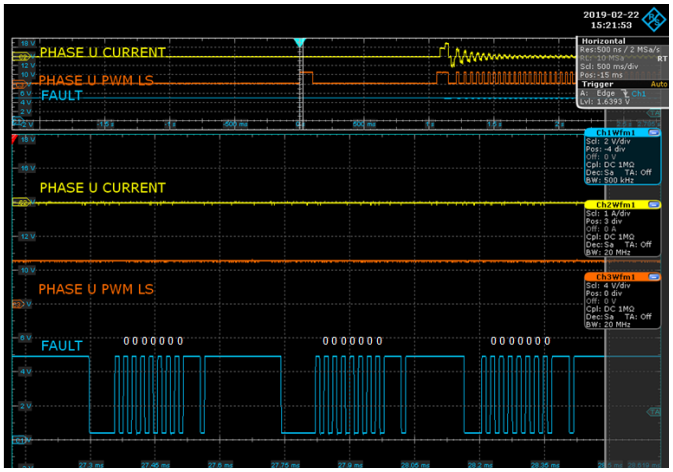

圖 26.　執行啟動序列之後的裝置狀態。

範例代碼庫

可以使用以下連結，從 BridgeSwitch 產品頁面 (www.power.com) 下載範例代碼庫：

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

附註

本應用說明描繪如何透過 UART 主控台顯示故障資訊以進行偵錯。應以輪詢方式實施顯示執行，以限制 MCU 上的負載。最大限度地減少顯示的資訊量也可以減少負載。

修訂	附註	日期
A	初始版本。	04/19

如需最新更新，請瀏覽我們的網站：www.power.com
Power Integrations 保有隨時對其產品進行變更以提升可靠性或可製造性的權利。Power Integrations 對因使用此處所述的任何裝置或電路所造成的損失概不負責。Power Integrations 在本文中不提供任何保證，並明確否認所有保證，包括但不限於對適售性、特定目的之適用性以及不侵犯第三方權利的默示保證。

專利資訊
Power Integrations 的一項或多項美國及國外專利 (或可能正在申請的美國及國外專利) 可能涵蓋本文件中所示的產品和應用 (包括產品外部的變壓器結構和電路)。www.power.com 上提供了 Power Integrations 專利的完整清單。Power Integrations 將某些特定專利授權給客戶，詳情請參閱 www.power.com/ip.htm。

生命支援政策
未經 Power Integrations 總裁明確的書面許可，不可將 Power Integrations 產品用作生命支援裝置或系統的關鍵元件。具體說明如下：

1. 生命支援裝置或系統係指 (i) 透過外科手術植入人體的裝置，或 (ii) 支援或維持生命的裝置，以及 (iii) 根據合理推斷，遵循使用指示正確使用而無法正常執行功能時，會導致使用者重大傷害或死亡的裝置。
2. 關鍵元件係指生命支援裝置或系統中，根據合理推斷，無法正常執行功能時會導致生命支援裝置或系統出現故障，或是影響其安全或有效性的任何元件。

Power Integrations 全球銷售支援地點

全球總部
5245 Hellyer Avenue
San Jose, CA 95138, USA
總機：+1-408-414-9200
客戶服務：
全球：+1-65-635-64480
美洲：+1-408-414-9621
電子郵件：usasales@power.com

中國 (上海)
中國上海漕溪北路 88 號
聖愛廣場 2410 室
郵遞區號：200030
電話：+86-21-6354-6323
電子郵件：chinasales@power.com

中國 (深圳)
17/F, Hivac Building, No. 2, Keji Nan 8th Road, Nanshan District,
Shenzhen, China, 518057
電話：+86-755-8672-8689
電子郵件：chinasales@power.com

德國 (AC-DC/LED 銷售)
Einsteinring 24
85609 Dornach/Aschheim
Germany
電話：+49-89-5527-39100
電子郵件：eurosales@power.com

德國 (閘極驅動器銷售)
HellwegForum 1
59469 Ense
Germany
電話：+49-2938-64-39990
電子郵件：
igbt-driver.sales@power.com

印度
#1, 14th Main Road
Vasanthanagar
Bangalore-560052 India
電話：+91-80-4113-8020
電子郵件：indiasales@power.com

義大利
Via Milanese 20, 3rd.Fl.
20099 Sesto San Giovanni (MI) Italy
電話：+39-024-550-8701
電子郵件：eurosales@power.com

日本
Yusen Shin-Yokohama 1-chome Bldg.
1-7-9, Shin-Yokohama, Kohoku-ku
Yokohama-shi,
Kanagawa 222-0033 Japan
電話：+81-45-471-1021
電子郵件：japansales@power.com

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

新加坡
51 Newton Road
#19-01/05 Goldhill Plaza
Singapore, 308900
電話：+65-6358-2160
電子郵件：
singaporesales@power.com

台灣
台灣台北市內湖區1
318 號 5 樓
郵遞區號：11493
電話：+886-2-2659-4570
電子郵件：
taiwansales@power.com

英國
Building 5, Suite 21
The Westbrook Centre
Milton Road
Cambridge
CB4 1YG
電話：+44 (0) 7823-557484
電子郵件：eurosales@power.com