애플리케이션 노트 AN-80
BridgeSwitch 제품군

BridgeSwitch 고장 통신 인터페이스

소개

이 애플리케이션 노트에서는 BridgeSwitch™ 고장 상태 통신 인터페이스 기능에 대한 소프트웨어 구현 안내서에 대해 설명합니다. 여기에는 BridgeSwitch 고장 상태 통신 인터페이스의 개요, 수신된 상태 업데이트를 캡처 및 처리하는 상태 기기, 레퍼런스 코드 및 해당 데이터 구조, UART 단자를 통한 상태 업데이트를 표시하는 소프트웨어 데모, 인버터 보드 내의 고장 보호 구현 예시가 포함되어 있습니다.

BridgeSwitch 고장 상태 통신 인터페이스

BridgeSwitch 디바이스는 내부 및 시스템 레벨 고장을 비롯한 상태 업데이트를 오픈 드레인 고장 출력을 통해 시스템 MCU에 전달할 수 있습니다. 이 디바이스는 7비트 단어 패턴 뒤에 홀수 패리티 비트를 사용하여 상태 업데이트를 보고합니다.

다음 섹션에서는 고장 버스 사양을 세부적으로 설명합니다.

하드웨어 구성

모든 감지된 상태 업데이트를 시스템 마이크로 컨트롤러에 전달하기 위해, 단일 와이어 버스에 연결되는 모든 고장 핀은 시스템 공급 전압 쪽으로 풀업되어 있습니다. 그림 1에는 단일 와이어 버스 구성으로 시스템 MCU에 연결된 일반적인 BridgeSwitch 디바이스 세 개가 나와 있습니다.

디바이스 ID 핀 연결을 사용하면 디바이스 ID 핀 연결에 따라 각 디바이스가 자체적으로 고유한 디바이스 ID를 할당할 수 있습니다. 이러한 디바이스 ID를 사용하면 상태 통신을 시작할 때 각각의 디바이스 ID 기간 t_{ID}에 대한 고장 버스를 아래로 내려 고장 상태가 감지된 실제 위치를 시스템 마이크로 컨트롤러에 전달할 수 있습니다.

표 1에는 디바이스 ID, 결과 디바이스 ID 시간 t_{ID}, 그리고 ID 핀 연결을 통해 각각의 ID를 프로그래밍하는 방법이 나열되어 있습니다.

디바이스 ID	t_{ID}	ID 핀 연결
1	40 µs	BPL 핀
2	60 µs	플로우팅
3	80 µs	SG 핀

표 1 – ID 핀을 통한 디바이스ID 선택

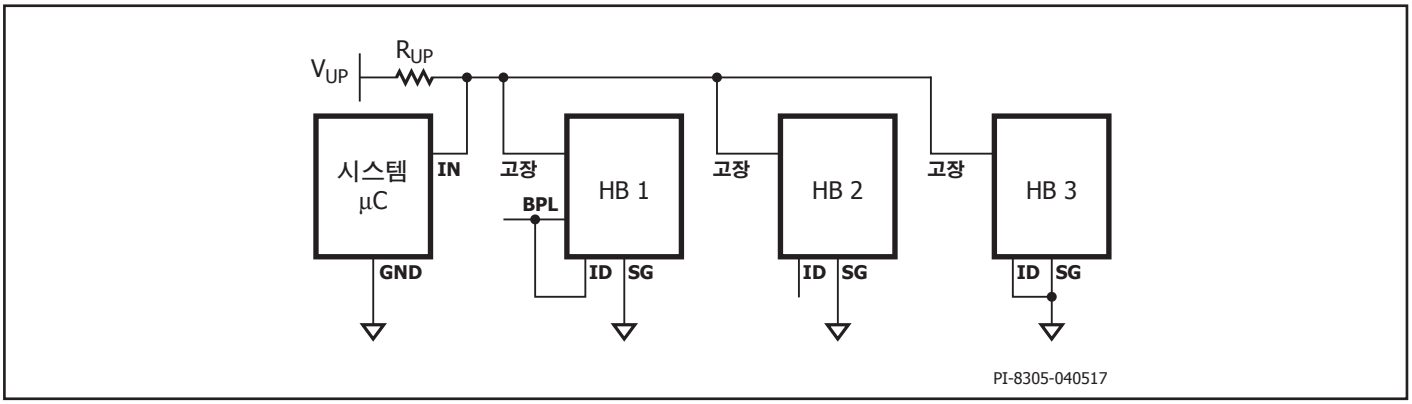

그림 1 – 디바이스 ID 프로그래밍을 통한 단일 와이어 상태 통신 버스

고장 상태 통신 버스 사양

상태 인코딩

7비트 단어 뒤에 오는 패리티 비트는 고장 정보를 인코딩합니다. 표 2에는 디바이스가 시스템 마이크로 컨트롤러에 전달할 수 있는 다양한 상태 업데이트의 인코딩이 요약되어 있습니다. 상태 단어는 동시에 발생할 수 없는 상태 변경 사항이 함께 그룹화된 블록 다섯 개로 구성됩니다. 따라서 고장 우선 순위 및 고장 보고 대기열에 신경쓰지 않고도 여러 가지 상태 업데이트를 시스템 마이크로 컨트롤러에 동시에 보고할 수 있습니다.

마지막 행(7비트 단어 "000 00 0 0")은 디바이스 준비 상태를 인코딩하며 작동 시작 성공 시퀀스를 시스템에 전달하는 데 사용됩니다. 이 행은 특정 고장이 해결되었을 때 전달되며, 고장이 발생하지 않은 경우에는 상태를 승인하여 시스템 MCU를 요청하기 위해 전송됩니다.

다음 중 한 가지 이유로 인해 고장 버스에서의 통신이 시작됩니다.

- 작동 시작 성공 후 미션 모드 통신이 준비됨.
- 디바이스 중 하나에 의해 고장 상태 레지스터 업데이트 통신이 시작됨.
- 시스템 마이크로 컨트롤러의 쿼리에 따른 전류 상태 통신.

고장	비트 0	비트 1	비트 2	비트 3	비트 4	비트 5	비트 6
HV 버스 OV	0	0	1				
HV 버스 UV 100%	0	1	0				
HV 버스 UV 85%	0	1	1				
HV 버스 UV 70%	1	0	0				
HV 버스 UV 55%	1	0	1				
시스템 열 고장	1	1	0				
S 드라이버가 준비되지 않음[1]	1	1	1				
LS FET 열 경고				0	0		
LS FET 써멀 셧다운				1	0		
HS 드라이버가 준비되지 않음[2]				1	1		
LS FET 과전류						1	
HS FET 과전류							1
디바이스가 준비됨(고장 없음)	0	0	0	0	0	0	0

참고:

1. XL 핀 오픈/회로 단락 고장, XL 핀 회로 단락에 대한 IPH 핀, 트림 비트 손상이 포함됩니다.
2. HS-LS 통신 단절, V_{BPH} 또는 내부 5V 레일 범위 이탈, XH 핀 오픈/회로 단락 고장이 포함됩니다.

표 2. 상태 단어 인코딩.

그림 2는 세가지 경우 모두에 대한 고장 인터페이스 통신 순서도를 보여줍니다.

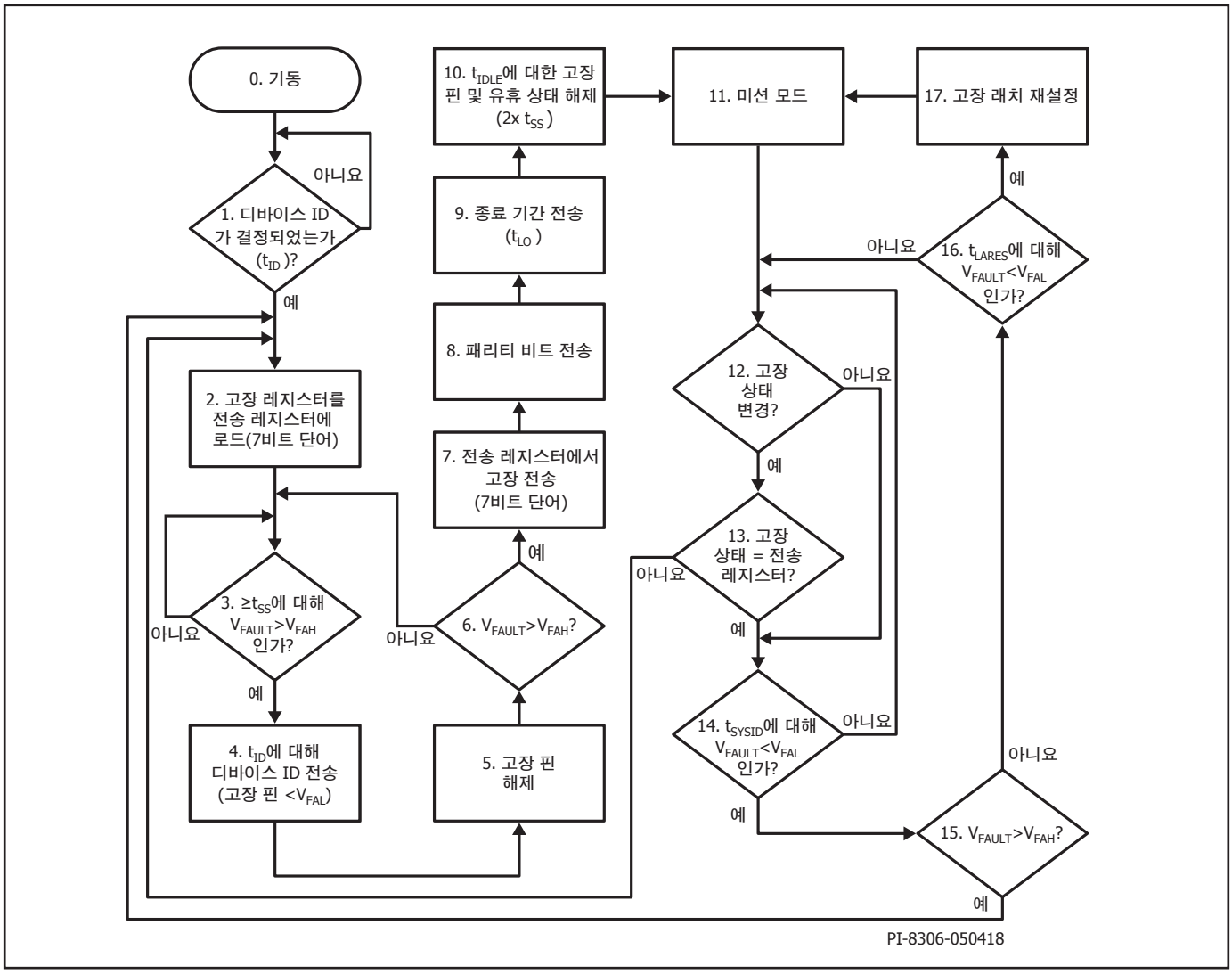

그림 2. 상태 통신 순서도.

상태 업데이트 통신은 항상 통신 디바이스에 의해 시작된 버스 중재로 시작됩니다. 이는 버스가 최소한 80µs 동안 소음이 없는 경우 고장 핀을 풀다운하여 각각의 디바이스 ID 시간 t_{ID}를 전송합니다. 디바이스가 버스 중재에서 이기면 전류 고장 레지스터(7비트 단어) 뒤에 홀수 패리티 비트가 오며, 통신 순서도(그림 2)에 나와 있는 대로 전송 종료 신호가 전송됩니다.

비트 스트림 타이밍

그림 3은 BridgeSwitch가 상태 업데이트 통신에 사용하는 비트 스트림 타이밍 다이어그램을 보여줍니다. 두 가지 로직 상태는 고장 핀의 하이타임 기간, 그리고 그 뒤에 오는 로우타임 기간 t_{LO}(일반적으로 10 µs)인 두 가지 다른 전압 신호로 인코딩됩니다. 로직 "1"은 기간 t_{BIT1}(일반적으로 40 µs)로 인코딩되고 로직 "0"은 기간 t_{BIT0}(일반적으로 10 µs)으로 인코딩됩니다. 표 3에는 고장 상태 통신의 타이밍 표가 나와 있습니다.

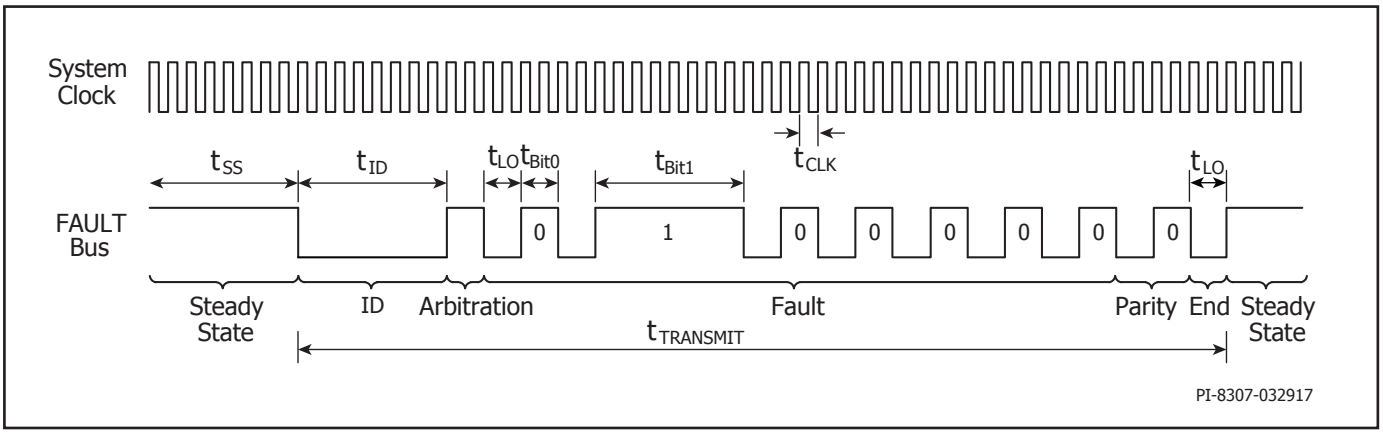

그림 3. 상태 통신 비트 스트림

기호	설명	로직 상태	기간(일반)
t_{ID}	디바이스 ID	0	표 1 참조
t_{LO}	로우타임	0	10 µs
t_{Bit0}	로직 0	1	10 µs
t_{Bit1}	로직 1	1	40 µs

표 3. 비트 스트림 타이밍 표

각각의 전송이 완료된 후 디바이스는 새로운 통신을 시작하기 전에 t_{IDLE}(일반적으로 2 × t_{SS} = 160µs) 동안 유휴 상태가 됩니다. 이로 인해 버스상의 다른 디바이스가 상태 변경을 전달하거나, 시스템 마이크로 컨트롤러에 의해 전송된 상태 쿼리에 응답할 수 있습니다.

디바이스는 각각의 감지된 상태 업데이트를 한 번만 전달합니다. 또한, 디바이스는 모든 시스템 레벨 고장의 상태 변경을 시스템 MCU에 보고합니다. 여기에는 DC 버스 저전압 및 고전압 상태, 그리고 외부 온도 모니터 고장이 포함됩니다. 디바이스는 LS 전력 FREDFET 써멀 셧다운을 제외하고, 디바이스 내부 고장의 모든 상태 레벨 변경도 보고합니다.

또한, BridgeSwitch 디바이스는 미션 모드일 때 시스템 마이크로 컨트롤러에서 전송한 명령에 대해 고장 버스를 모니터링합니다. 이는 버스를 t_{SYSID}(일반적으로 160 µs) 기간만큼 끌어내리는 것을 통한 마이크로 컨트롤러의 상태 업데이트 쿼리(그림 2의 15단계 참조)일 수 있습니다. 또는 이는 고장 버스를 t_{LARES}(2x t_{SYSID} = 일반적으로 320 µs) 기간만큼 끌어내려 과열 셧다운 래치를 포함한 디바이스 상태 레지스터를 재설정하고 작동 시퀀스 모드(그림 2의 17단계 참조)로 들어가기 위한 명령일 수 있습니다. 작동 시퀀스는 MCU가 래치 재설정 명령을 전송한 후 권장됩니다. 표 4에는 사용 가능한 시스템 마이크로 컨트롤러 명령이 요약되어 있습니다.

버스 풀다운 기간	명령
t_{SYSID}	상태 쿼리
$t_{LARES(2xTSYSID)}$	과열 래치 재설정을 포함한 상태 레지스터 및 가동 시퀀스 모드

표 4. 시스템 MCU 명령

소프트웨어 구현

이 섹션에서는 이전 섹션에서 설명한 상태 통신 사양을 기반으로 각 BridgeSwitch 디바이스에서 상태 업데이트를 캡처하고 처리하는 상태 기기의 구현에 대해 설명합니다.

제시된 예시에서는 차단 기반 구현을 사용합니다. 사용자는 모터 컨트롤 알고리즘 또는 마이크로 컨트롤러의 종류 같은 자신의 고유한 애플리케이션 요구 사항에 따라 차단 우선 순위를 정해야 합니다.

시스템 MCU 주변기기

고장 버스 통신 인터페이스 구현을 시연하기 위해 Cypress PSoC Creator IDE 버전 4.1을 사용하여 레퍼런스 코드를 개발하였으며 Cypress PSoC 4 MCU(CY8CKIT-042 PSoC Pioneer Kit)에서 테스트를 수행했습니다. MCU 보드는 USB 커넥터를 통해 접속하여 PC와 통신할 수 있는 온보드 프로그래머 및 디버거를 제공합니다. 상태 기기는 고장 감지 예시 섹션에 나와 있는 UART 콘솔에서 수신된 상태 업데이트를 출력하여 시연되었습니다.

고장 상태 통신 버스는 오픈 드레인 구동 모드에서 양방향 MCU 핀 하나에 연결됩니다. 이 핀은 신호의 상승 및 하강 엣지를 캡처하는 타이머에 부착됩니다. 고장 신호의 처리는 두 가지 16비트 타이머/카운터 블록인 *Bit_counter_timer* 및 12 MHz 클록의 *ID_counter_timer*를 사용하는 차단 기반 방식입니다. *Bit_counter_timer*는 상승 엣지에서 하강 엣지의 신호를 캡처하는 반면, *ID_counter_timer*는 하강 엣지에서 상승 엣지의 신호를 캡처합니다. 이러한 두 가지 타이머는 고장 신호의 하강 엣지와 상승 엣지가 각각 수신될 경우 카운트 값을 캡처하고 차단을 생성합니다. 고장 상태 기기 루틴은 각각의 수신된 차단을 처리합니다.

소프트웨어 설명

소프트웨어 구현은 *fault_bus_state* 변수를 유휴 상태 *(STEADY_STATE)*로 초기화하는 것으로 시작합니다. *fault_bus_state* 변수는 차단이 수신되면 고장 신호의 상태를 캡처합니다. 초기화 함수 *init_fault_bus_interrupt()*는 타이머/카운터 캡처 포트를 초기화하고, 상승 엣지와 하강 엣지 두 가지 모두에 대한 차단을 캡처할 수 있도록 합니다. 차단 서비스 루틴(ISR)이 트리거되는 경우에는 항상 *fault_detect()* 함수가 호출됩니다. 이 함수는 수신된 고장을 캡처하고 처리하는 고장 상태 기기 루틴입니다. 주요 소프트웨어 흐름의 상세한 보기는 그림 5에 나와 있습니다.

고장 상태 기기 루틴은 고장 처리의 현재 상태를 기준으로 기본적으로 ISR 이벤트를 처리하고 *fault_bus_state* 변수를 업데이트합니다. 고장 기기 상태는 *STEADY_STATE, ID_DET, ARBITRATION, T_LO* 및 *BIT_DETECT*입니다. 고장 상태 기기의 자세한 소프트웨어 흐름 다이어그램은 그림 6에 나와 있습니다. 고장 상태의 전체 패킷이 패리티 오류 없이 수신될 때마다 고장 처리 함수 *fault_process()*가 호출되며, 그밖에 *fault_bus_state*가 *STEADY_STATE*로 재설정되는 재동기화가 발생합니다.

고장 처리 함수는 수신된 고장 상태 업데이트를 디코딩하고 필수 작업을 호출합니다. 예를 들어, 과전류 고장이 수신된 후 인버터를 셧다운하거나 열 경고 상태 업데이트가 수신된 후 인버터 출력 전력을 줄입니다. 고장 상태는 *fault* 변수에 저장되는데, 이는 새로운 상태 업데이트가 수신될 때마다 업데이트됩니다. 사용자는 수신된 고장 상태 및 애플리케이션 요구 사항에 따라 필요한 조치를 제공하거나 MCU에서 수행해야 하는 작업을 결정해야 합니다. 표 5에는 상태 업데이트가 수신된 후 시스템 마이크로 컨트롤러에서 취할 수 있는 조치의 예시가 나와 있습니다. *fault_process()* 함수 소프트웨어 흐름 다이어그램은 그림 7에 나와 있습니다.

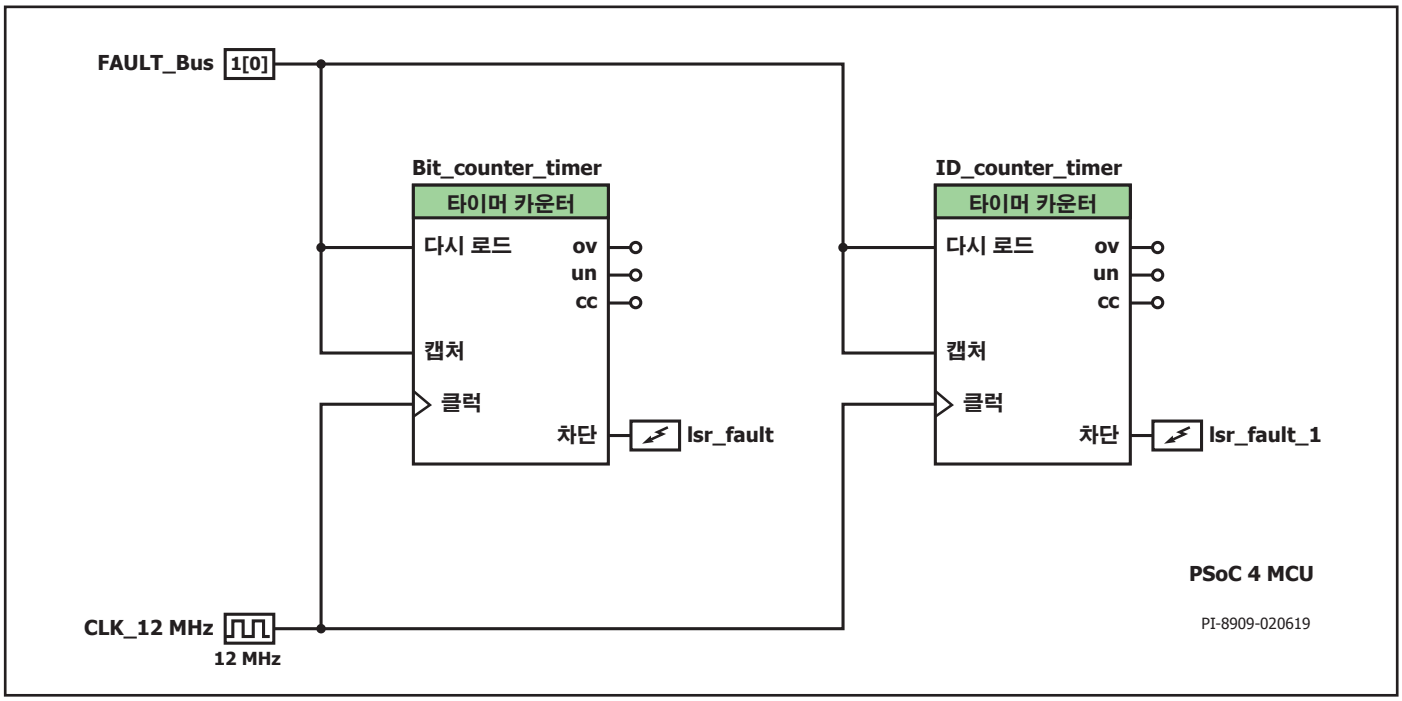

그림 4. PSoC 4 MCU를 통한 고장 신호 처리에 대한 시스템 MCU 주변 기기

상태 쿼리 및 래치 재설정 명령
상태 쿼리 및 래치 재설정 명령의 소프트웨어 구현은 아래와 같은 예시의 사용 사례를 지원합니다.

상태 쿼리 명령:
MCU는 인버터가 잠시 동안 꺼진 후 인버터를 재시작하려고 할 때마다 상태 쿼리를 전송(예: PWM 신호 전송)할 수 있습니다. 예: 보고된 라인 과전압 또는 과전류 고장 후. 상태 쿼리의 주요 목적은 모든 디바이스가 준비되었는지 또는 MCU가 가동 시퀀스를 시작해야 하는지 확인하기 위한 것입니다. 그림 8은 상태 쿼리 구현의 순서도 예시를 보여줍니다.

상태 쿼리 루틴은 각 BridgeSwitch 디바이스의 상태를 확인하여 다음 중 어떤 조건을 적용해야 할지 결정합니다.

A. 각 디바이스의 상태가 준비됨으로 응답함(고장 없음): MCU가 RESTART 명령을 호출하여 인버터를 리스타트하고 PWM 신호를 전송하여 BridgeSwitch의 입력을 제어합니다.
B. 하이 사이드 드라이브에 응답하는 하나 이상의 디바이스가 준비되지 않는 고장: MCU는 START UP 명령을 호출합니다. 이는 가동 시퀀스를 시작하여 하이 사이드 드라이버 공급 전압(HB 핀에 대한 V_{BPH})을 공칭 값으로 충전합니다. 가동 시퀀스가 완료된 후, MCU는 다른 상태 쿼리 명령을 연달아 수행합니다. 모든 디바이스가 준비됨으로 응답하면 MCU는 RESTART 명령을 호출하여 인버터를 리스타트합니다. 준비됨이 아닌 다른 상태로 응답하는 디바이스가 있을 경우, 인버터는 셧다운 모드로 유지됩니다.

상태 쿼리 명령은 고장 버스를 t_{SYSID} = 160µs(표 4 참조) 동안 끌어내리는 *status_query()* 함수에 의해 처리됩니다. 각 디바이스는 이 명령을 따르며 해당하는 상태를 연달아 전송합니다. 상태 쿼리가 시스템 마이크로 컨트롤러에 의해 전송된 후에는 감지된 고장 상태가 *process_status_query_command()* 함수(고장 버스 상태 기기 함수 내에 있음)에 의해 처리됩니다. 이 함수는 각 감지된 디바이스 상태를 저장하고 이를 *status_query_action()* 함수에서 처리합니다. *status_query_action()* 함수는 수신된 고장 상태를 확인하고 표 5에 설명된 대로 상태에 따른 조치를 제공합니다.

래치 재설정 명령:
MCU는 디바이스 중 하나(또는 모두)가 과열 고장(및 래치오프)을 보고한 경우 얼마 동안 래치 재설정 명령을 전송할 수 있습니다. 그림 9는 래치 재설정 명령의 순서도 예시를 보여줍니다. 가동 시퀀스는 래치 재설정 명령 후 권장됩니다. 이는 스위칭이 다시 시작되기 전에 바이패스 하이 사이드 전압이 공칭 레벨이 되도록 보장합니다.

래치 재설정 명령은 $_{LARES}$ = 320µs(표 4 참조) 동안 고장 버스를 끌어내려 각 디바이스 상태를 재설정하는 *latch_reset()* 함수에 의해 처리됩니다. 가동 시퀀스는 시스템 마이크로 컨트롤러에 의해 래치 재설정 명령이 전송된 후 호출됩니다. 래치 재설정 명령을 전송할 때마다 고장 감지 함수를 비활성화해야 합니다.

고장	상태 단어	소프트웨어 조치/결정	참고
고전압 버스 OV	001 xxx x	셧다운	일반적으로 디바이스 하나만 HV 버스를 모니터링하며, MCU는 전체 인버터를 셧다운합니다.
고전압 버스 UV 100%	010 xxx x	없음	MCU는 이전 상태 업데이트가 UV85, UV70 또는 UV50였던 경우 모터 출력을 공칭 전력으로 높이려고 할 수 있습니다.
고전압 버스 UV 85%	011 xxx x	경고	MCU는 모터 출력(속도/토크)을 낮추어 인버터 부하를 줄이려고 할 수 있습니다.
고전압 버스 UV 70%	100 xxx x	경고	MCU는 모터 출력(속도/토크)을 낮추어 인버터 부하를 줄이려고 할 수 있습니다.
고전압 버스 UV 55%	101 xxx x	경고	MCU는 모터 출력(속도/토크)을 낮추어 인버터 부하를 줄이려고 할 수 있습니다.
시스템 열 고장	110 xxx x	경고/셧다운	어떤 외부 부품 온도를 모니터링하는지에 따라 다릅니다.
LS 드라이버가 준비되지 않음	111 xxx x	셧다운	MCU는 고장이 해결되었는지 확인하기 위해 일정 시간 이후 인버터를 리스타트하려고 할 수 있습니다.
LS FET 열 경고	xxx 010 x	경고	MCU는 모터 출력(속도/토크)을 낮추어 인버터 부하를 줄이거나 PCB 온도를 제한하려고 할 수 있습니다.
LS FET 써멀 셧다운	xxx 10x x	셧다운	래칭 셧다운은 한 디바이스에서만 일어날 수 있으며, MCU는 전체 인버터를 셧다운해야 합니다. MCU는 냉각 꺼짐 기간 후 인버터 리스타트를 시도할 수 있습니다.
LS FET 과전류	xxx xx1 x	셧다운	디바이스는 각각의 FREDFET를 자동으로 꺼서 모터가 멈추거나 과부하 상태가 되지 않도록 보호합니다. MCU는 전체 인버터를 셧다운합니다.
HS 드라이버가 준비되지 않음	xxx 11x x	셧다운	MCU는 고장이 해결되었는지 확인하기 위해 일정 시간 후(몇 초) 인버터 리스타트를 시도할 수 있습니다.
HS FET 과전류	xxx xxx 1	셧다운	디바이스는 각각의 FREDFET를 자동으로 꺼서 모터가 멈추거나 과부하 상태가 되지 않도록 보호합니다. MCU는 전체 인버터를 셧다운합니다.
디바이스가 준비됨(고장 없음)	000 000 0	없음	MCU는 이전 상태 업데이트가 HV 버스 OV 또는 LS/HS FET 과전류였던 경우 인버터를 리스타트할 수 있습니다. 또한, MCU는 이전 상태 업데이트가 열 경고였던 경우 모터 전력을 공칭으로 높이려고 할 수 있습니다.

표 5. 상태 업데이트 수신 후 마이크로 컨트롤러에서 수행한 조치의 예시

소프트웨어 순서도

아래 순서도에는 고장 신호 처리에 대한 소프트웨어의 상세한 보기가 나와 있습니다.

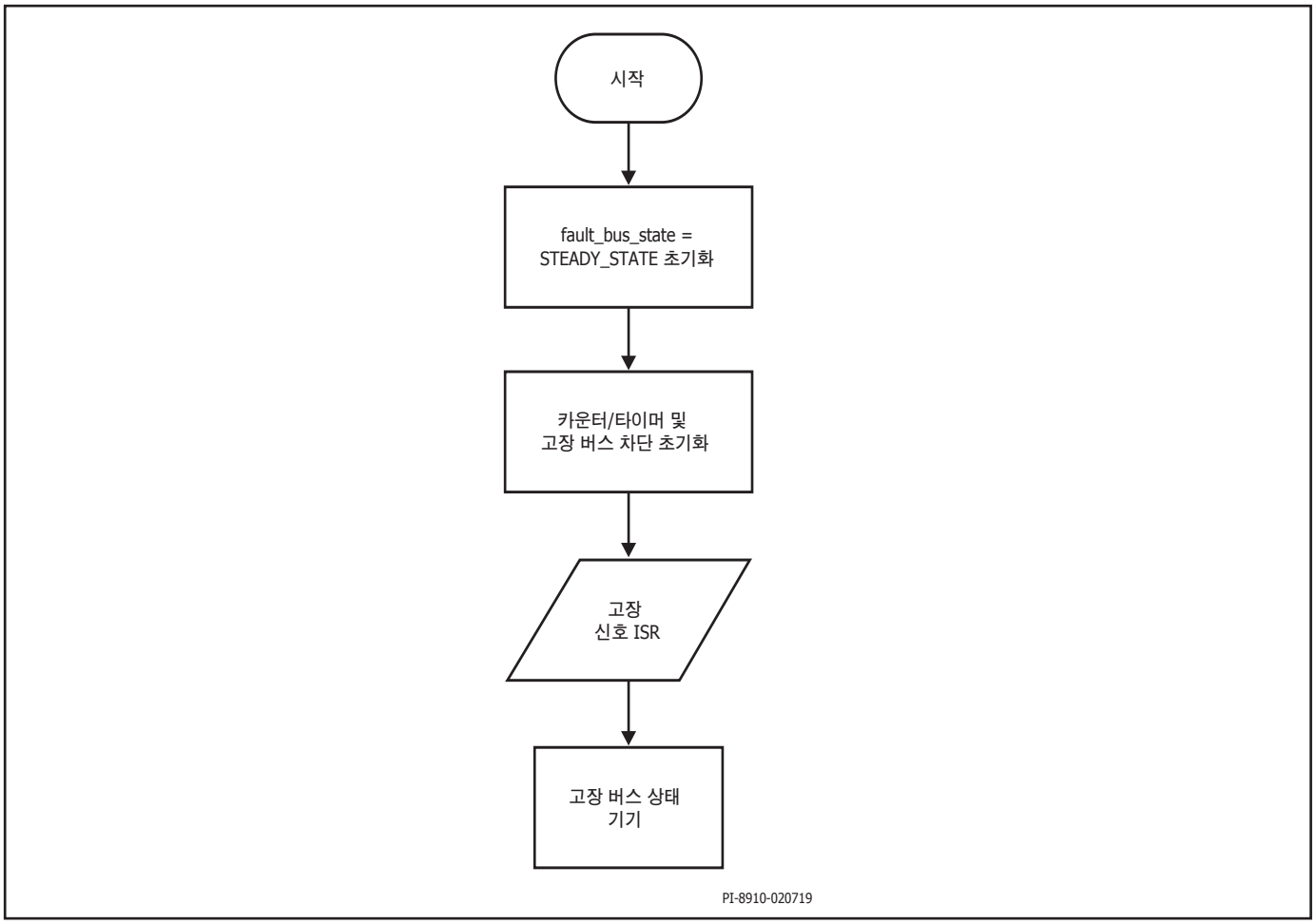

그림 5. 소프트웨어에서 고장 버스 구현의 상세한 보기

고장 버스 상태 기기

그림 6은 고장 버스 상태 기기의 소프트웨어 순서도를 보여줍니다.

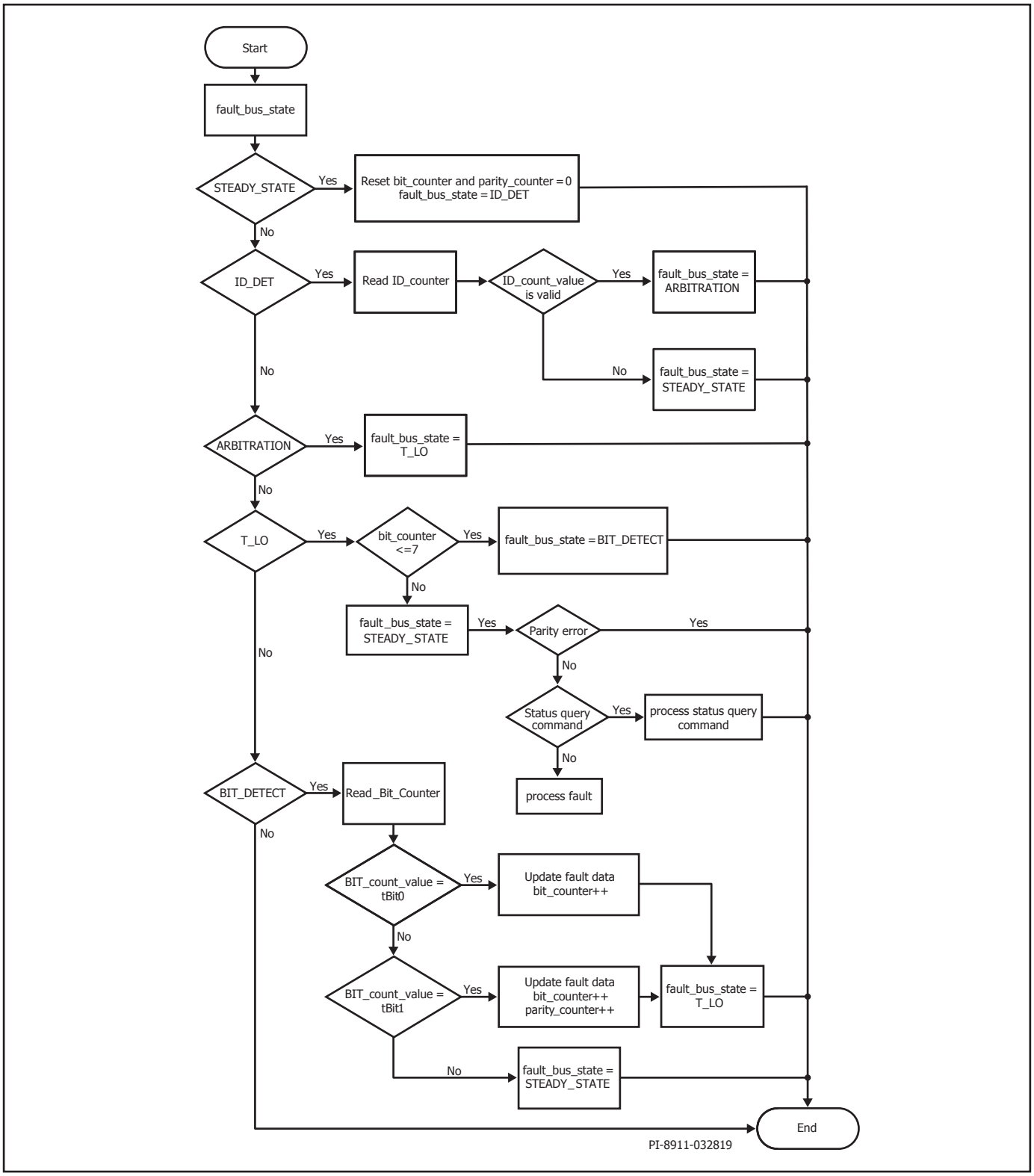

그림 6. 고장 버스 상태 기기

그림 7은 고장 처리 함수의 소프트웨어 순서도를 보여줍니다.

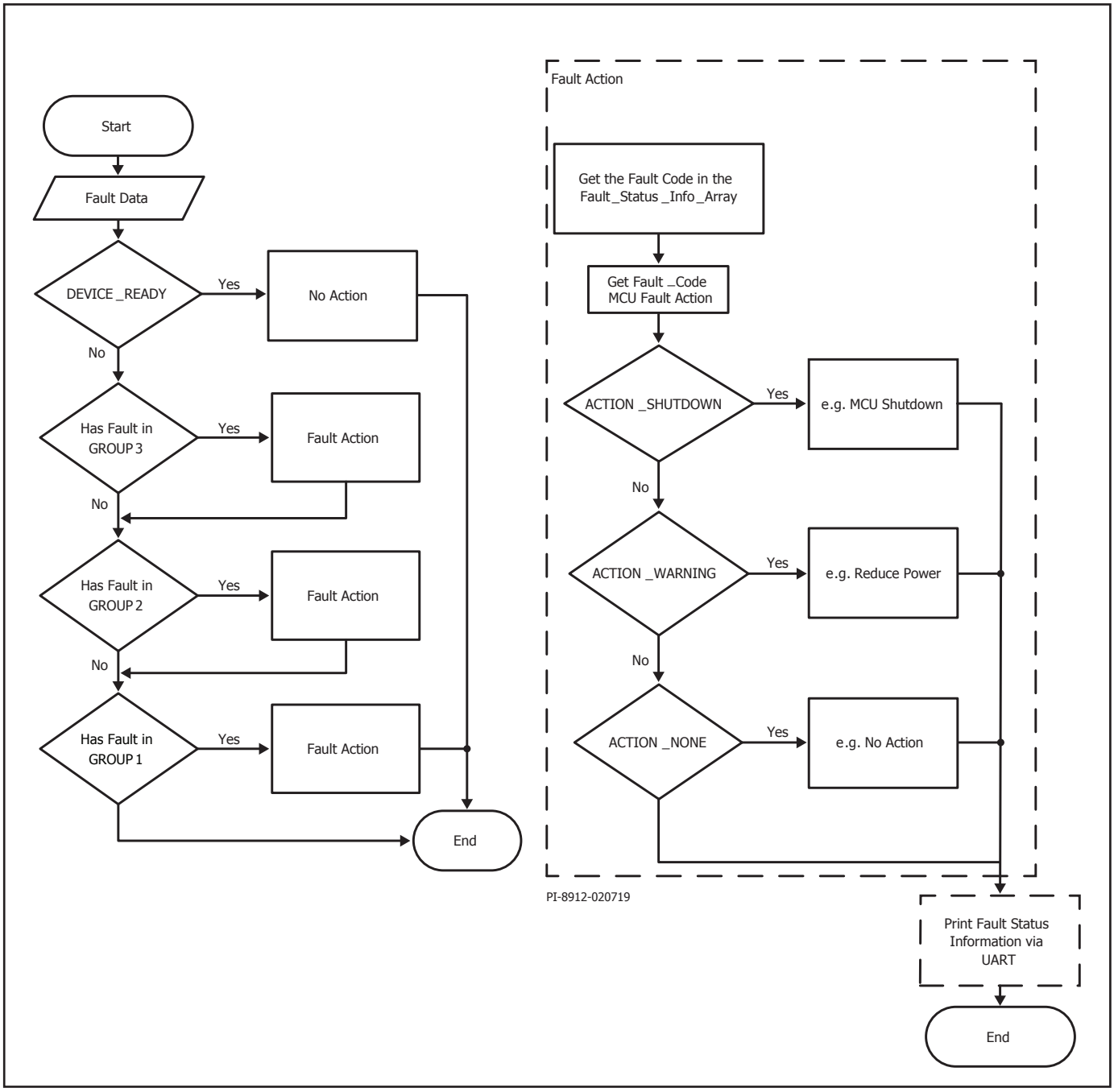

그림 7. 고장 처리 함수

수신된 고장 데이터에는 상태 업데이트 유형이 두 개 이상 포함될 수 있으며 고장 처리 함수는 각각의 고장 유형을 처리할 수 있어야 합니다. 동시에 발생할 수 없는 고장 비트는 함께 그룹화되어 고장 유형을 결정합니다. 표 6에는 상태 단어 그룹이 나와 있습니다. *GROUP1, GROUP2, 로우 사이드 FET 과전류* 및 *하이 사이드 FET 과전류*의 고장은 단일 상태 단어 내에서 동시에 보고될 수 있습니다.

그룹	고장	비트 0	비트 1	비트 2	비트 3	비트 4	비트 5	비트 6
GROUP1	HV 버스 OV	0	0	1				
	HV 버스 UV 100%	0	1	0				
	HV 버스 UV 85%	0	1	1				
	HV 버스 UV 70%	1	0	0				
	HV 버스 UV 55%	1	0	1				
	시스템 열 고장	1	1	0				
	S 드라이버가 준비되지 않음[1]	1	1	1				
GROUP2	LS FET 열 경고				0	0		
	LS FET 써멀 셧다운				1	0		
	HS 드라이버가 준비되지 않음[2]				1	1		
LS FET 과전류							1	
HS FET 과전류								1

표 6. 고장 상태 그룹

이 구현 예시에서는 고장 유형이 보고될 때마다 고장 조치 함수가 호출됩니다. 고장 조치 함수는 *FAULT_STATUS_INFO_ARRAY*에서 고장 코드에 해당하는 특정 고장 조치를 선택하며, 그런 다음 MCU가 이를 이어서 실행합니다. 표 5에서 특정 조치 목록을 참조하십시오. 보고된 상태 업데이트에 따른 조치는 특정 애플리케이션 요구 사항에 따라 조정해야 합니다.

이 문서의 오류 감지 예시 단락에는 UART 콘솔을 통해 고장 상태를 표시하여 상태 업데이트 디코딩을 시연합니다. 이 단락에서는 3상 인버터 보드에서 MCU가 수행하는 특정 조치에 대해서도 설명합니다.

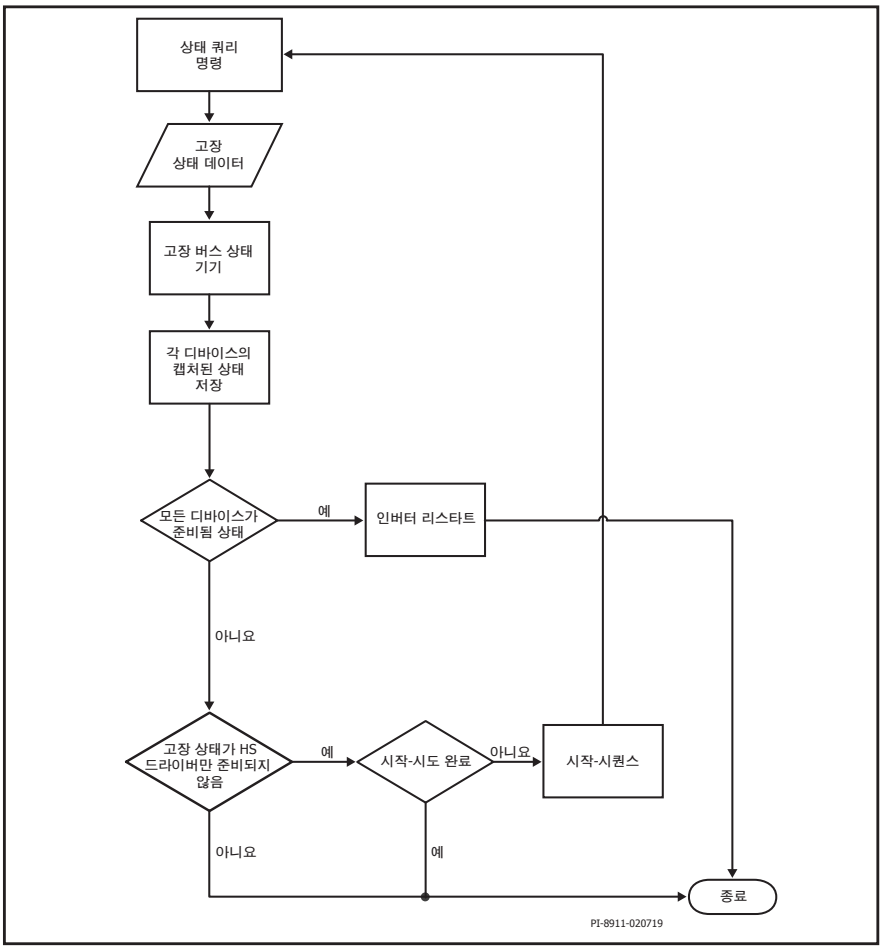

그림 8. 상태 쿼리 명령 처리 함수

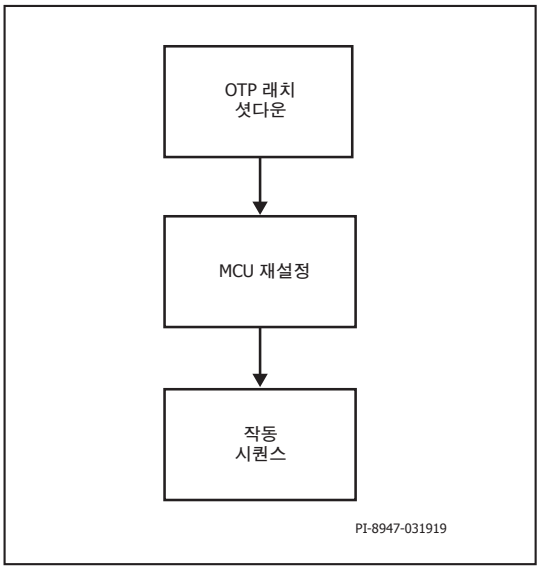

그림 9. 래치 재설정 명령 함수

참조 코드 데이터 구조

고장 상태 기기 상태
다음은 감지 프로세스 중 전류 고장 신호 상태를 결정하는 고장 버스 상태입니다.

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

Fault_Status_Info_Array
이 FAULT_STATUS_INFO 배열은 디코딩된 고장 상태 업데이트에 따른 특정 조치 목록입니다.

```
{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},
```

예를 들어 과전압 {HV_BUS_OV, ACTION_SHUTDOWN}은 이 오류가 발생할 경우 MCU가 시스템을 셧다운해야 한다는 것을 나타냅니다.

이러한 구현에서 고장 상태 업데이트에 따른 특정 조치는 다음과 같습니다.
ACTION_SHUTDOWN,
ACTION_WARNING,
ACTION_NONE,

고장 상태 코드
일어날 수 있는 고장 상태는 다음과 같습니다.

```
//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
이 구조에는 발생한 고장의 고장 코드와 디바이스 ID가 포함됩니다.

dev_id
fault

참조 코드

이 예시 코드는 PSoC Creator IDE 버전 4.1를 사용하여 개발하였으며 DER-654 참조 설계 인버터 보드가 장착된 CY8CKIT-042 PSoC Pioneer Kit 디바이스에서 테스트되었습니다. 아래 코드는 고장 신호 처리와 관련된 참조 함수를 나타냅니다. 표시된 참조 코드에는 UART 콘솔을 통해 고장 상태 정보를 출력하는 코드 스니펫은 제외됩니다(자세한 내용은 참고 부분 참조). 사용된 다른 변수의 정의를 보려면 제공된 코드 파일을 참조하십시오.

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

아래 코드는 이 문서에서 설명된 상태 쿼리 및 래치 재설정 명령의 구현 예시와 관련된 참조 함수를 나타냅니다. 상태 쿼리 및 래치 재설정 명령에 대한 호출은 각 사용자의 사용 사례에 따라 실제 구현에서 별도로 처리되어야 합니다. 사용된 변수의 정의를 보려면 제공된 코드 파일을 참조하십시오.

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

   }
```

고장 감지 예시

표시된 고장 감지 예시 및 예시 조치에 따라 마이크로 컨트롤러에서 내린 결정은 위에서 자세히 다룬 참조 코드를 사용하여 표 5에 나와 있습니다.

UART 단자는 고장 상태 정보를 표시하여 고장 상태 기기를 보여줍니다. 표시되는 정보의 형식은 *[디바이스 ID, 고장, 조치]*입니다. 예를 들어, *W, STS, S*라는 UART 메시지는 디바이스 W(디바이스 1, 2, 3은 각각 U, V, W에 지정됨)로부터 상태 업데이트를 나타내며, 고장 상태는 시스템 써멀 셧다운(STS)이고, MCU 조치는 셧다운(S)입니다.

그림 10에는 보고된 시스템 열 고장이 나와 있으며 인버터 셧다운에 대한 예시 조치가 나와 있습니다(그림 11의 UART 열 출력 참조).

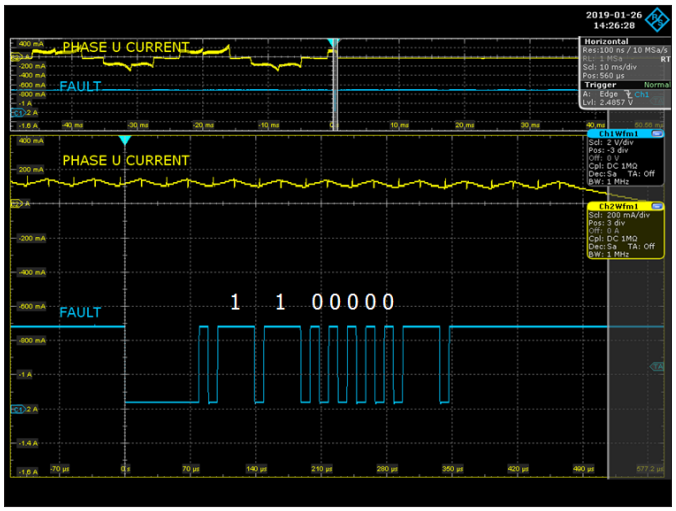

그림 10. 시스템 온도 상태가 고장으로 수신된 후의 인버터 셧다운 예시

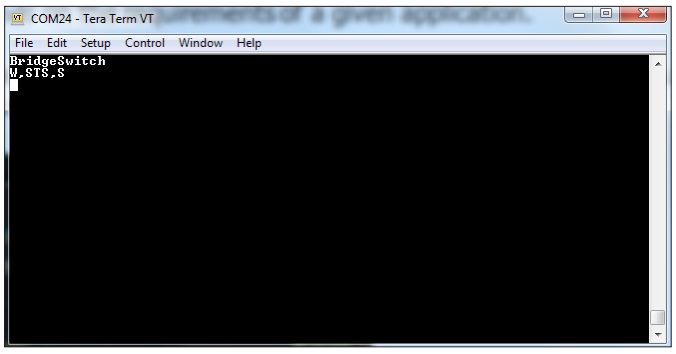

그림 11. 시스템 열 상태가 고장으로 수신된 후의 UART 단자 출력

그림 12에는 보고된 로우 사이드 과전류 고장 및 인버터 셧다운에 대한 예시 조치가 나와 있습니다(그림 13의 UART 단자 출력 참조).

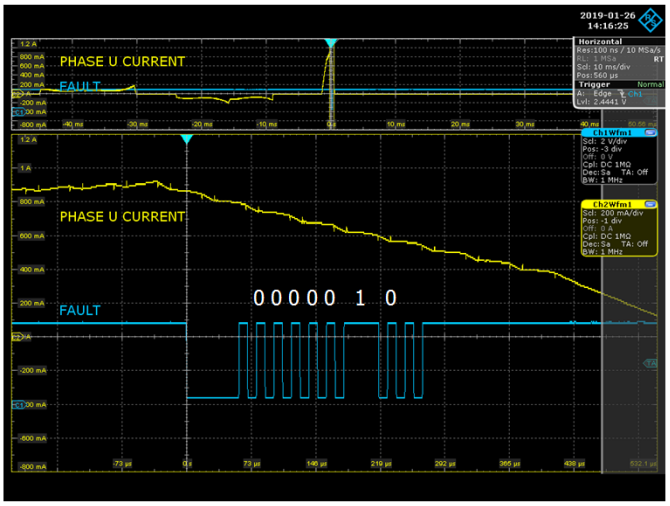

그림 12. 로우 사이드 과전류 상태가 고장으로 수신된 후의 인버터 셧다운 예시

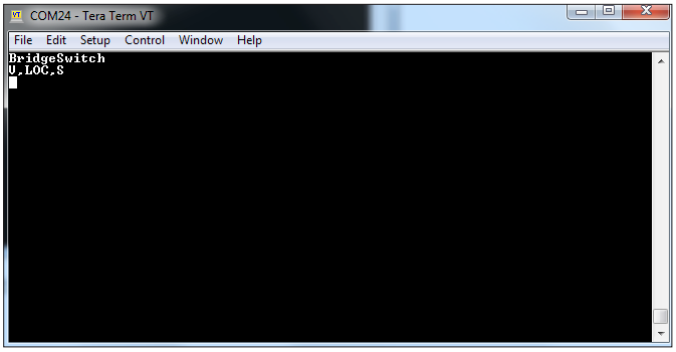

그림 13. 로우 사이드 과전류 상태가 고장으로 수신된 후의 UART 단자 출력

그림 14에는 경고 상태와 함께 보고된 고전압 버스 UV85 고장이 나와 있습니다. 이 구현 예시에서는 MCU가 특정 조치를 취하지 않았으나 경고 상태만 표시되었습니다. 그림 15의 UART 단자를 참조하십시오.

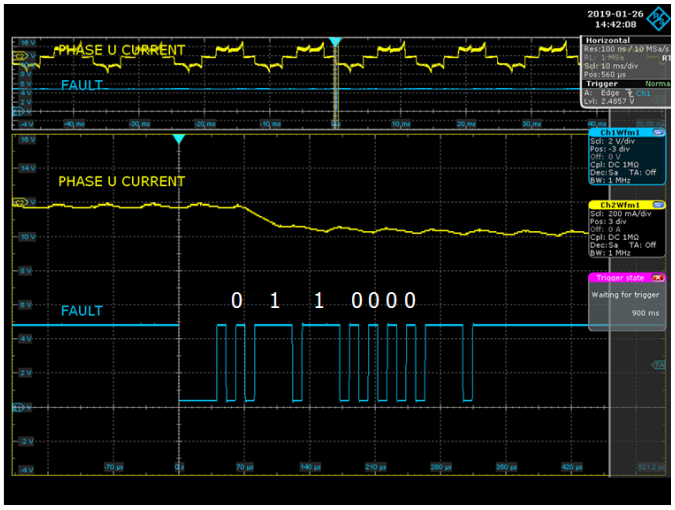

그림 14. 고전압 버스 UV85 수신 후 경고 상태

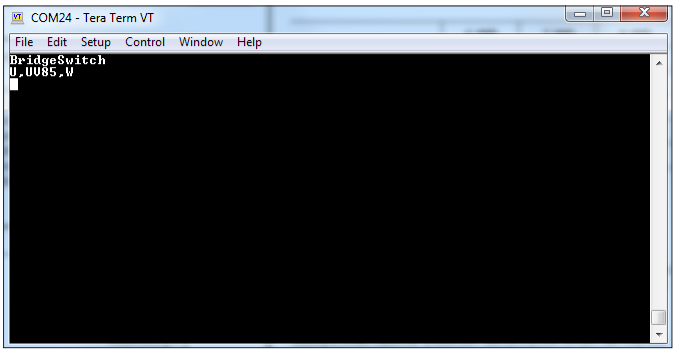

그림 15. 고전압 버스 UV85 수신 후 UART 단자 출력

그림 16에는 보고된 고전압 버스 과전압 및 인버터 셧다운에 대한 예시 조치가 나와 있습니다(그림 17의 UART 단자 출력 참조).

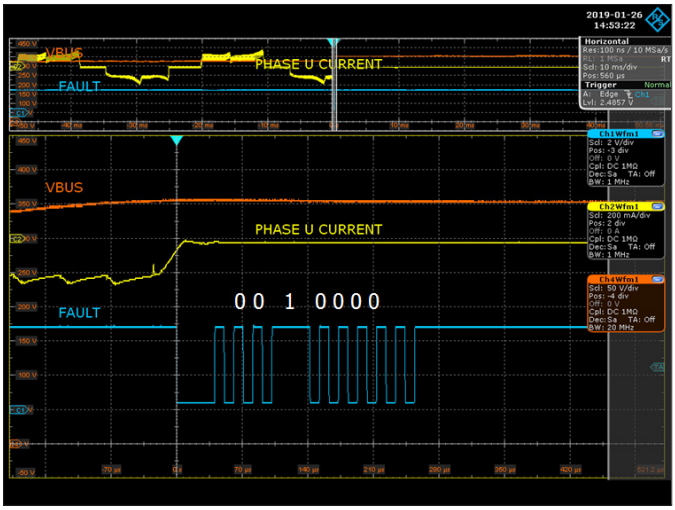

그림 16. 고전압 버스 과전압 수신 후 인버터 셧다운 예시

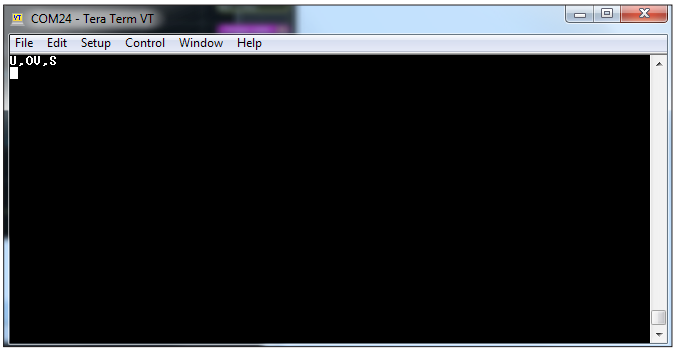

그림 17. 고전압 버스 과전압 수신 후 UART 단자 출력

MCU 명령 예시

그림 18은 라인 과전압 고장 상태에 의해 발생한 셧다운에 따른 상태 쿼리 명령 후의 인버터 리스타트를 보여줍니다.

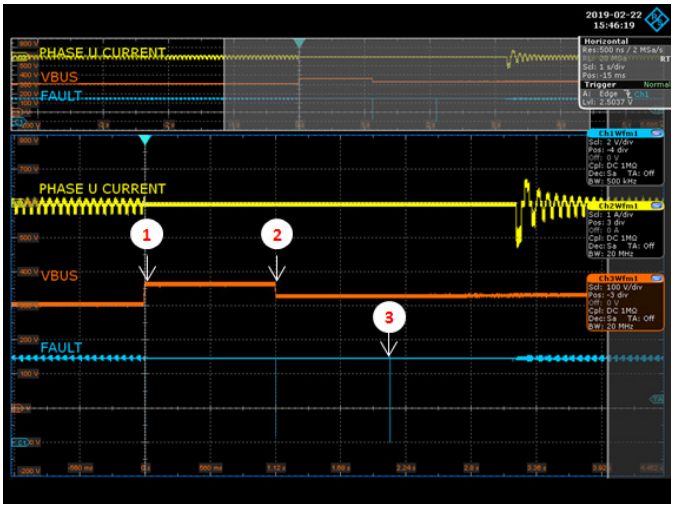

그림 18. 라인 과전압 상태 후 상태 쿼리 명령

(1) 라인 과전압이 발생하고 그림 19에 나와 있는 것처럼 인버터가 OV 상태가 되면서 셧다운됩니다. (2) 과전압 상태가 해결되고 (3)에서 시스템 마이크로 컨트롤러가 상태 쿼리 명령을 전송하여 디바이스 상태를 확인합니다. 그림 20은 상태 쿼리 명령 및 모든 세 가지 디바이스의 각 상태 보고서를 보여줍니다. 모든 디바이스가 준비됨으로 보고되었으며 시스템 마이크로 컨트롤러가 인버터를 리스타트합니다.

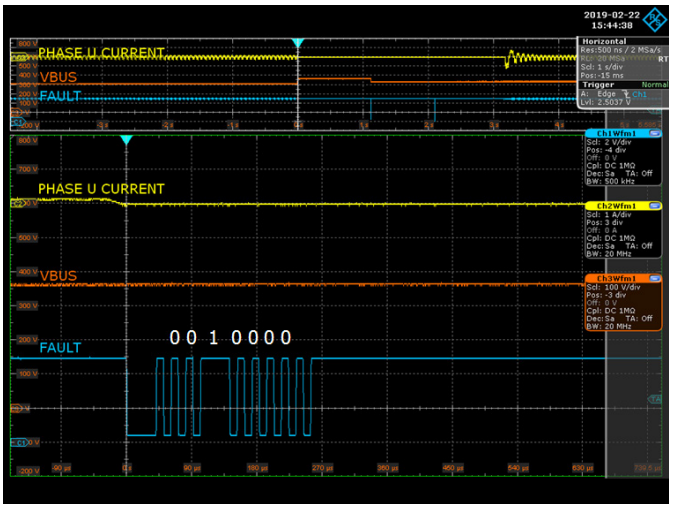

그림 19. 라인 과전압 상태 후 인버터 셧다운

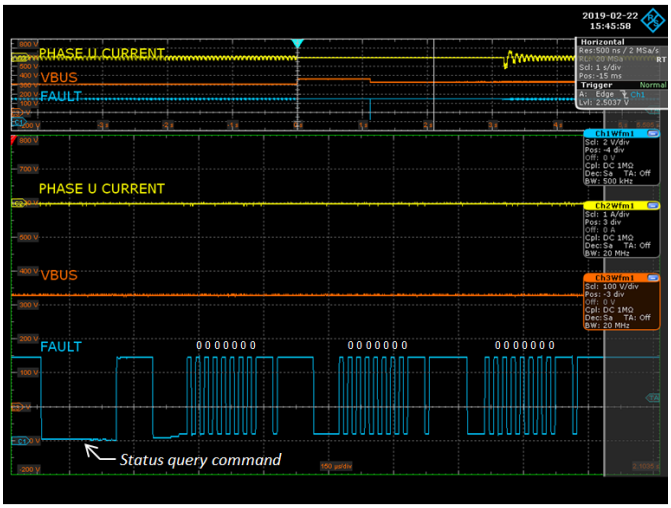

그림 20. 라인 과전압 상태 후 상태 쿼리 명령 (모든 디바이스가 준비됨으로 보고됨)

그림 21은 하이 사이드 드라이버가 준비되지 않음 고장 (1)으로 인해 발생한 셧다운에 따른 상태 쿼리 (2) 명령 후의 인버터 리스타트를 보여줍니다.

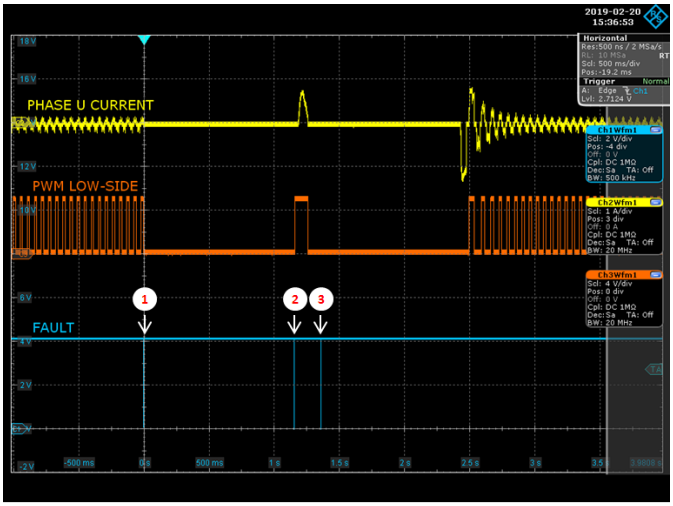

그림 21. 하이 사이드 드라이버가 준비되지 않음 고장 후 상태 쿼리 명령

이 예시에서 보고된 상태 업데이트는 하이 사이드 드라이버만 준비되지 않음 고장입니다. MCU는 기동 루틴(예: 100ms 동안 하이 로직을 로우 사이드 PWM 입력 INL에 적용)을 명령합니다. (3)에서 MCU는 다른 상태 쿼리 명령을 전송하여 모든 디바이스가 준비되었는지 확인합니다. 이 예시에서는 모든 고장이 해결되었으며 모든 디바이스가 준비되었습니다. MCU는 인버터 리스타트를 시작하고 PWM 신호를 BridgeSwitch 컨트롤 입력 INL 및 /INH에 전송합니다.

그림 22는 해당하는 하이 사이드 드라이버가 준비되지 않음 고장 상태인 것을 보여주며, 그림 23은 상태 쿼리 명령 및 기동 시퀀스 시도 후 모든 세 가지 디바이스의 각 상태 보고서를 보여줍니다. 모든 디바이스가 준비됨으로 보고되었으며 시스템 마이크로 컨트롤러가 인버터를 리스타트합니다.

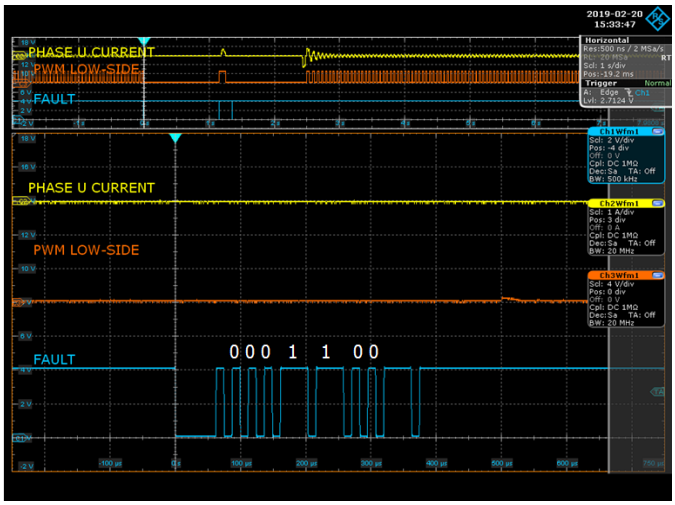

그림 22. 하이 사이드 드라이버가 준비되지 않음 고장 후 인버터 셧다운

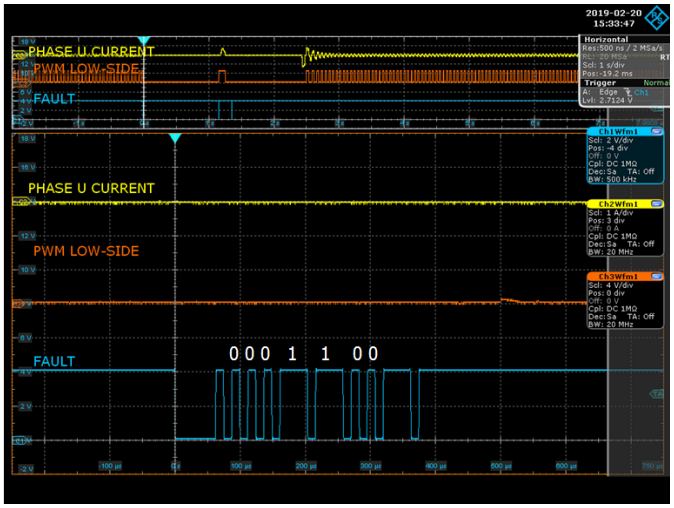

그림 23. 기동 시퀀스 후 상태 쿼리 명령

그림 24는 과열 고장으로 인해 발생한 래칭 셧다운에 따른 래치 재설정 명령을 보여줍니다. MCU는 래치 재설정 명령을 전송한 후 전체 작동 시퀀스를 적용합니다.

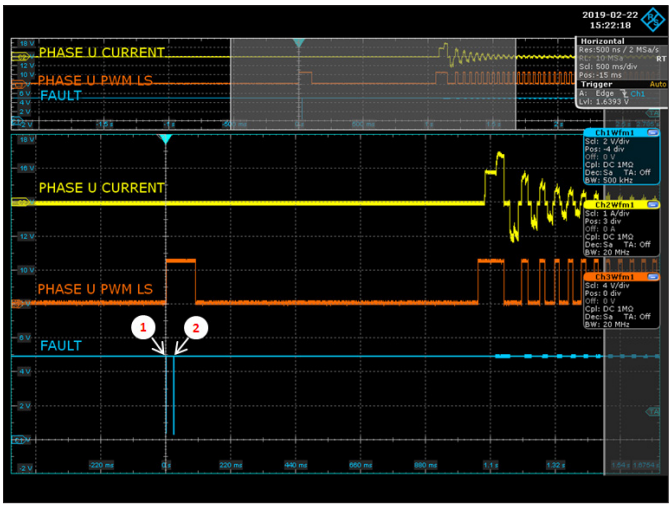

그림 24. 래칭 과열 보호 후 래치 재설정 명령 및 작동 시퀀스

그림 25는 래치 재설정 명령 (1) 및 하이 사이드 준비되지 않음의 기본 상태 보고서를 보여줍니다. (래치 재설정 명령 호출 시 오류 감지는 비활성화됨). MCU는 래치 재설정 명령을 전송한 후 기동(작동) 시퀀스를 적용합니다. 그림 26는 모든 디바이스가 준비됨 (2) 상태로 보고되었으며 인버터가 시작되었음을 보여줍니다.

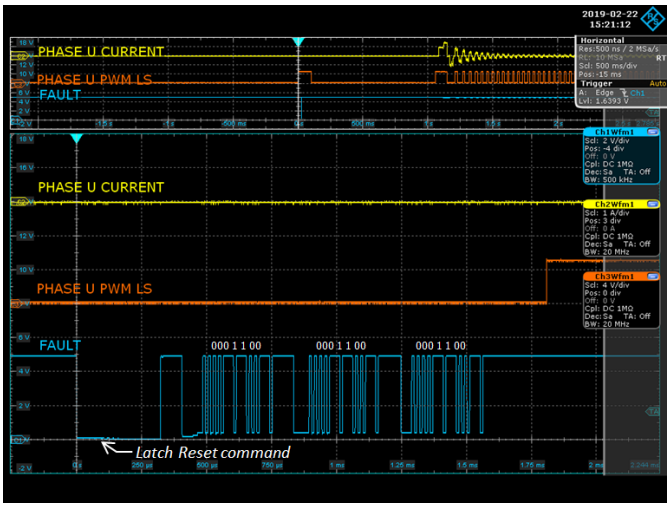

그림 25. 래치 OTP 후 래치 재설정 명령 및 작동 시퀀스

그림 26은 모든 디바이스가 준비됨 상태로 보고되었으며 작동 시퀀스가 성공적으로 완료되었음을 보여줍니다.

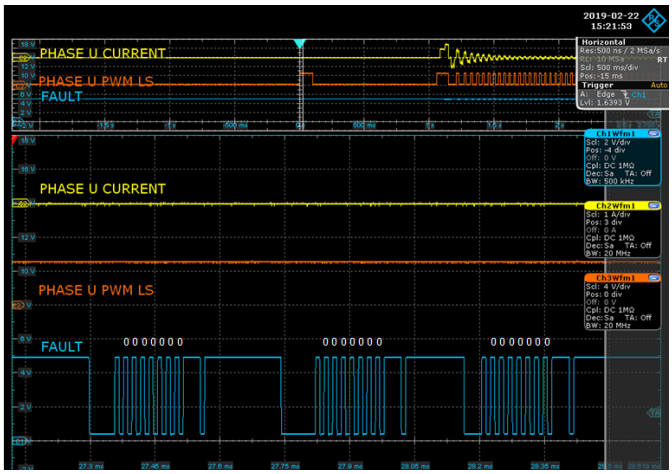

그림 26. 기동 시퀀스 후 디바이스 상태

예시 코드 라이브러리

아래 링크를 사용하여 BridgeSwitch 제품 페이지(www.power.com)에서 예시 코드 라이브러리를 다운로드할 수 있습니다.

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

참고

이 애플리케이션 노트에서는 디버깅을 목적으로 UART 콘솔을 통해 고장 정보를 표시하는 것에 대해 설명합니다. MCU에 가해지는 부하를 제한하려면 폴링된 방식으로 표시 실행을 구현해야 합니다. 표시되는 정보의 양을 최소화하면 부하도 줄어듭니다.

개정	참고	날짜
A	최초 출시.	04/19

최신 업데이트에 대한 자세한 내용은 당사 웹사이트를 참고하십시오. www.power.com
파워 인테그레이션스(Power Integrations)는 안정성 또는 생산성 향상을 위하여 언제든지 당사 제품을 변경할 수 있는 권한이 있습니다. 파워 인테그레이션스(Power Integrations)는 본 문서에서 설명하는 디바이스나 회로 사용으로 인해 발생하는 어떠한 책임도 지지 않습니다. 파워 인테그레이션스(Power Integrations)는 어떠한 보증도 제공하지 않으며 모든 보증(상품성에 대한 묵시적 보증, 특정 목적에의 적합성 및 타사 권리의 비침해를 포함하며 이에 국한되지 않음)을 명백하게 부인합니다.

특허 정보
본 문서에서 설명하는 제품 및 애플리케이션(제품의 외부 트랜스포머 구성 및 회로 포함)은 하나 이상의 미국 및 해외 특허 또는 파워 인테그레이션스(Power Integrations)에서 출원 중인 미국 및 해외 특허에 포함될 수 있습니다. 파워 인테그레이션스(Power Integrations)의 전체 특허 목록은 www.power.com에서 확인할 수 있습니다. 파워 인테그레이션스(Power Integrations)는 고객에게 www.power.com/ip.htm에 명시된 특정 특허권에 따른 라이센스를 부여합니다.

수명 유지 장치 사용 정책
파워 인테그레이션스(Power Integrations)의 제품은 파워 인테그레이션스(Power Integrations) 사장의 명백한 문서상의 허가가 없는 한 수명 유지 장치 또는 시스템의 핵심 부품으로 사용할 수 없습니다. 자세한 정의는 다음과 같습니다.

1. 수명 유지 장치 또는 시스템이란 (i) 신체에 외과적 이식을 목적으로 하거나, (ii) 수명을 지원 또는 유지하고, (iii) 사용 지침에 따라 올바로 사용하는 경우에도 작동이 실패하여 사용자에게 상당한 부상 또는 사망을 초래할 수 있는 장치 또는 시스템입니다.
2. 핵심 부품이란 부품의 작동이 실패하여 수명 유지 장치 또는 시스템의 작동이 실패하거나, 해당 장치 또는 시스템의 안전성 및 효율성에 영향을 줄 수 있는 수명 유지 장치 또는 시스템에 사용되는 모든 부품입니다.

파워 인테그레이션스(Power Integrations) 전 세계 판매 지원 지역

본사
5245 Hellyer Avenue
San Jose, CA 95138, USA
본사 전화: +1-408-414-9200
고객 서비스:
전 세계: +1-65-635-64480
북미: +1-408-414-9621
이메일: usasales@power.com

중국(상하이)
Rm 2410, Charity Plaza, No. 88
North Caoxi Road
Shanghai, PRC 200030
전화: +86-21-6354-6323
이메일: chinasales@power.com

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

독일(AC-DC/LED 판매)
Einsteinring 24
85609 Dornach/Aschheim
Germany
전화: +49-89-5527-39100
이메일: eurosales@power.com

독일(게이트 드라이버 판매)
HellwegForum 1
59469 Ense
Germany
전화: +49-2938-64-39990
이메일: igbt-driver.sales@power.com

인도
#1, 14th Main Road
Vasanthanagar
Bangalore-560052 India
전화: +91-80-4113-8020
이메일: indiasales@power.com

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

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

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

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

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

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