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

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

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




# THIS SPEC IS OBSOLETE

Spec No: 001-70630

Spec Title: EVENT DATA RECORDER WITH CONTROLLER AREA NETWORK USING PSOC(R) 3 AND NVSRAM - AN70630

Sunset Owner: Daniele Radaelli (DHR)

Replaced By: NONE

AN70630

# Event Data Recorder with Controller Area Network using PSoC® 3 and nvSRAM

**Author: Daniele Radaelli**

**Associated Project: Yes**

**Associated Part Family: CY8C34xx, CY8C36xx, CY8C38xx, CY14x101Q2A**

**Software Version: PSoC® Creator™ 1.0 SP2**

**Related Application Notes: For a complete list of the application notes, click here.**

OBSOLETE


## Abstract

AN70630 describes how to combine PSoC® 3 with Cypress nvSRAM to build an event data recorder (EDR), which is finding increasing application in automotive. PSoC 3, with its controller area network connectivity and efficient data transfer using direct memory access, helps free the CPU to execute other tasks.


## Contents


Introduction ..... 2
Functional Requirements for Basic EDR with CAN ..... 2
CAN node Event Data Recorder PSoC 3 solution ..... 3
- DMA-based versus Firmware-based data recording ..... 4
- PSoC 3 schematic implementation overview ..... 5
- CAN component configuration ..... 5
- Time keeper ..... 6
- Local Sensing ..... 8
- User Interface ..... 8
- Loop timer and sensor polling ..... 9
- Firmware architecture ..... 10
- Recording locally sensed data using DMA seamlessly with CAN messages ..... 11
- System commands ..... 12
- Example of using commands: replay recorded data ..... 13

nvSRAM interface block in PSoC 3 ..... 14
- Overview ..... 14
- Data Format ..... 14
- nvSRAM interface module schematic design ..... 15
- DMA flow and control for data recording ..... 15
- nvSRAM interface module public APIs ..... 19
- Maximum recordable time window ..... 20
- nvSRAM interface module memory map summary ..... 20

Demonstration with the PSoC 3 Development Kit, CAN Expansion Board Kit, and nvSRAM ..... 21
- Hardware Setup ..... 21
- Data recording ..... 23
- Sudden power loss and data retrieval ..... 24

Summary ..... 25
Related Application Notes ..... 25
Appendix: project setup with CY8CKIT-030 PSoC 3 DVK and nvSRAM SOIC-DIP adapter ..... 26

## Introduction

Systems that rely on electronic subsystems for their functionality use EDRs to store information about system status as it evolves after a critical event occurs. A critical event typically stops the system from functioning.

An EDR, such as the one described in this application note, monitors communication among electronic subsystems (controller area network nodes) and records all or selected information. Moreover, an EDR can store information collected and processed locally from various sensors.

Typically, EDRs use two external memory devices to perform their function: an SRAM and an EEPROM. The SRAM acts as a fast access circular buffer continuously recording system status data (for the last few tens of seconds). The EEPROM stores data transferred from the SRAM in case of a crash. Using only an SRAM would not maintain data after power is disconnected. Using only an EEPROM would not allow fast access continuous data recording, as in a circular buffer, due to its intrinsic latency. In addition, if used as a circular buffer, the EEPROM would wear out quickly because of the intrinsic limited number of write/erase cycles, causing the device to fail. Cypress's nvSRAM combines the two memories, SRAM and EEPROM, in a single IC and, therefore, supports continuous data recording, fast access, and non-volatile data store in case of a crash.

In a critical event, EEPROM-based EDR systems require the microcontroller actively to supervise data transfer from the buffer SRAM to the EEPROM. That does cause delays resulting from page buffering and slow EEPROM write times. A typical EEPROM with a 64-byte page buffer and 5-ms write cycle would take 80 ms to store 1 KB of data. However, Cypress's nvSRAM does not add a buffering delay, and all of the data is stored in the nonvolatile cells within 8 ms. If the critical event is also subject to an abrupt power outage, the EEPROM-based EDR system requires that backup power be provided for several tens of milliseconds so that the entire system can complete the data transfer, which requires large capacitors or backup batteries. On the contrary, a small amount of power is required for the Cypress nvSRAM alone; the power state of the rest of the system becomes irrelevant.

EDRs and data loggers are becoming more common in the automotive industry, and the National Highway Traffic Safety Administration (NHTSA) is developing regulations. The Cypress PSoC 3, with its CAN connectivity and efficient data transfer capabilities using direct memory access (DMA), when paired with Cypress's nvSRAM, can provide a compact and effective solution.

This application note does not discuss formal requirements for EDRs. Nor does it address all the issues related to EDRs and NHTSA regulations. However, the note describes an implementation based on PSoC 3 and nvSRAM that forms the basic building block of an EDR system attached to a CAN bus.

Furthermore, this application note demonstrates the functionality of an EDR using PSoC® 3 development kits CY8CKIT-001 or CY8CKIT-030, CAN expansion board CY8CKIT-017, an nvSRAM (CY14B101Q2A), and Vector CANalyzer® to transmit and receive CAN messages with a PC.

## Functional Requirements for Basic EDR with CAN

The main goal of an EDR is to record the evolving system status, for later analysis, after a critical event occurs. The following is a list of functional requirements to implement a basic CAN node data recorder system:

- Stored data must be nonvolatile, to ensure that data is not destroyed in case of power loss.
- System must support data recording of multiple CAN message IDs. No compromise should be required to select which message is recorded.
- System must support data recording of critical information captured through local sensors (for example, an accelerometer or GPS), complementing information available through CAN.
- System must add a timestamp to the recorded events. The timestamp must have sufficient granularity to ensure you can reconstruct a timeline of events.
- CPU intervention in recording data must be minimized to ensure maximum system availability during event data recording.
- System must accept commands through specific CAN message ID to control EDR operations.

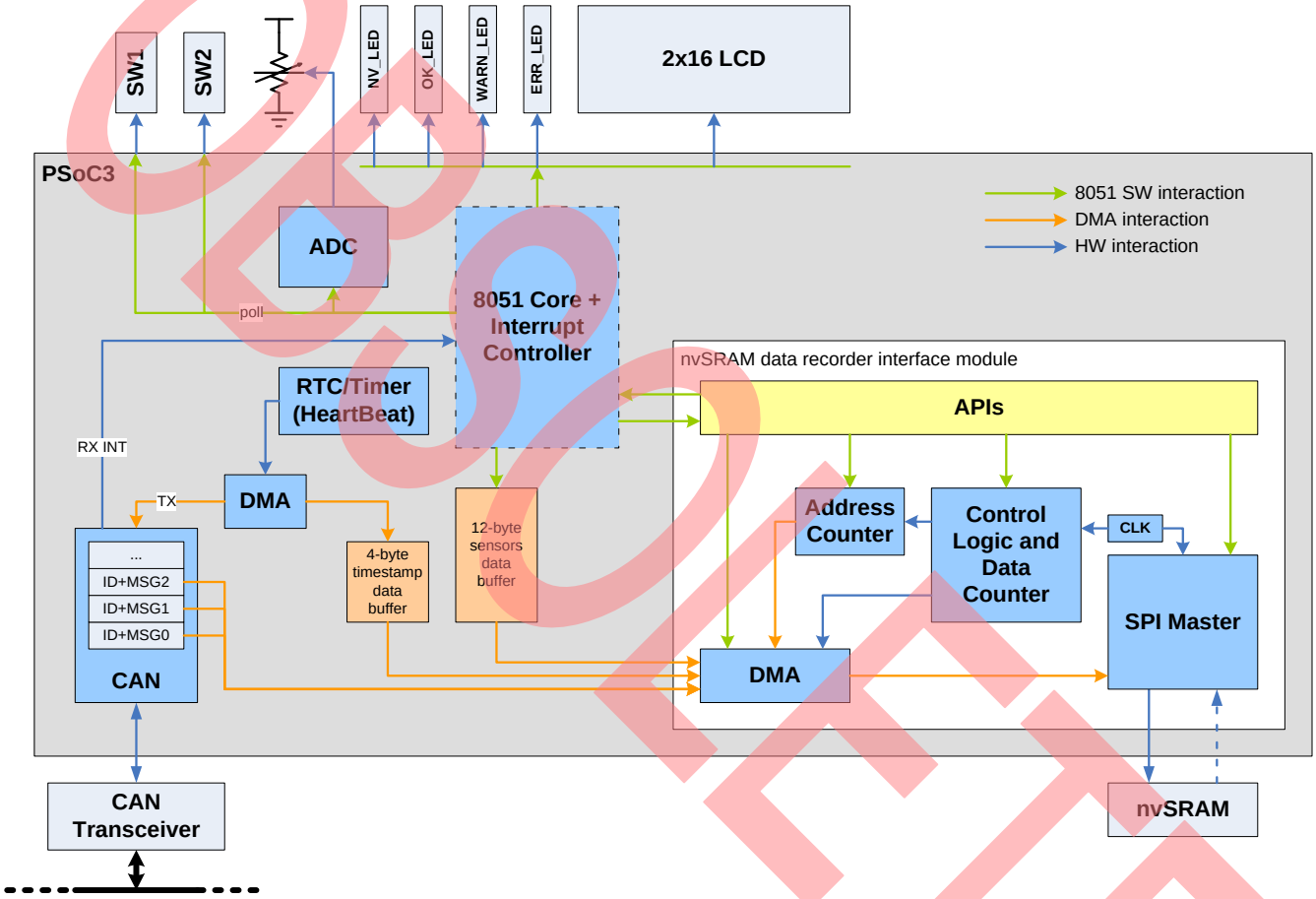
Requirements for data format, storage size, length of recorded data, data samples frequency, tamper detection and prevention, and other factors are specific to the system application and, therefore, will not be discussed in this note. However, some of these characteristics will be described in the implementation discussion.

OBSOLETE

## CAN node Event Data Recorder PSoC 3 solution

The block diagram in Figure 1 shows the architecture of an EDR using PSoC 3 and nvSRAM and addresses the basic requirements described in the previous section. The implementation uses the PSoC 3 kits CY8CKIT-001 (or CY8CKIT-030), CY8CKIT-017 (CAN/LIN expansion board), and an nvSRAM device (CY14B101Q2A).

Figure 1. Architecture of CAN node EDR based on PSoC 3 and nvSRAM.




The **nvSRAM data recorder interface module** encapsulates the operations of the interface with the nvSRAM. The module provides APIs through which the system firmware can configure the data sources for data logging and request specific action (log data, retrieve data, store, recall, and so on). The system application firmware does not need to account for SPI configuration, DMA setup, or the approach to write/read to/from the nvSRAM. The details of this module are described in a later section (nvSRAM interface block in PSoC 3 on page 14).

The **CAN module** is the link to the bigger system, specifically a car modules network. The configuration has four RX mailboxes and four TX mailboxes, which will be detailed later (CAN component configuration on page 5).

Three of the four RX messages will be recorded to nvSRAM and are set up to trigger an interrupt. The Interrupt Service Routine calls nvSRAM APIs to set the source of the data to be logged (the CAN message buffer register itself) and to start the data logging process through DMA.

The **RTC (real-time clock)/timer block** is the system's main watch, driven by an external 32768-Hz crystal. The RTC produces one pulse per second; therefore, a "helper" timer is added to increase the granularity of the time keeper to 1/64 of a second (15.625 ms). Every 15.625 ms, a DMA data transfer sequence is triggered to update a timestamp buffer memory (containing hr, min, sec, and x/64 seconds) and to transmit a timestamp message on

OBSOLETE

the CAN bus. Such a timestamp is added to each data logged record. The timestamp CAN message can be used for test/verification: After a power loss, an external node (emulated by a PC in the demonstration in this application note) knows at what time the node ceased to work; that time later can be compared to the timestamp of the last saved data in the EDR.

The **two switches (SW1 and SW2) and ADC** emulate possible local sensors in the data recorder (for example, an accelerometer, temperature sensor, or switches), and their data is recorded seamlessly to the nvSRAM with the data from the CAN bus.

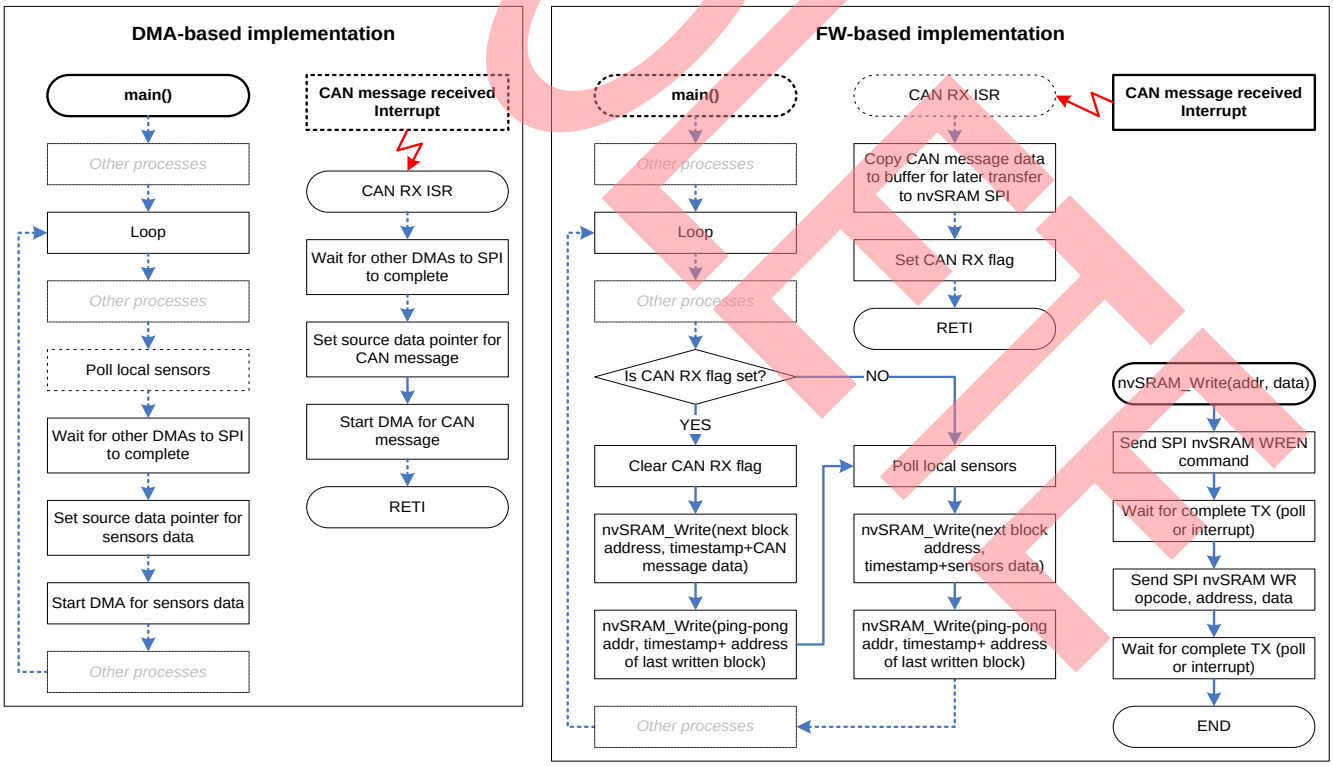
Finally, the **8051 core** performs supervisory tasks (updating LED statuses, writing messages to the LCD, and polling the sensors data), besides servicing the few interrupts in the system (CAN, ADC, RTC) during data logging. When the system is not recording data, the processor core can perform the tasks to read the data recorded in the nvSRAM, according to the commands received through CAN (using a dedicated RX mailbox). More details are explained in a later section on the recording and replaying of data (Example of using commands: replay recorded data 13).

## DMA-based versus Firmware-based data recording

The EDR leverages the efficient data transfer capabilities of PSoC 3 using the DMA engine. It is worth taking a quick high-level view of the microcontroller execution path, with and without the DMA support, to perform the data recording process to the nvSRAM through SPI communication.

Figure 2 shows the different flows in the two cases. When DMA is present, the microcontroller needs to initiate only the SPI transfer process by selecting the source of the data and starting the DMA, which will synchronize with the rest of the hardware to complete the data transmission. When DMA is not utilized, the microcontroller must send the data to the SPI hardware and interact with the hardware itself to perform the transfer. In this last case, the CPU is more involved in the data transfer, thus reducing the time for other tasks.

Figure 2. Data storing process - Main loop and interrupts flow comparison between implementations based on DMA and FW.

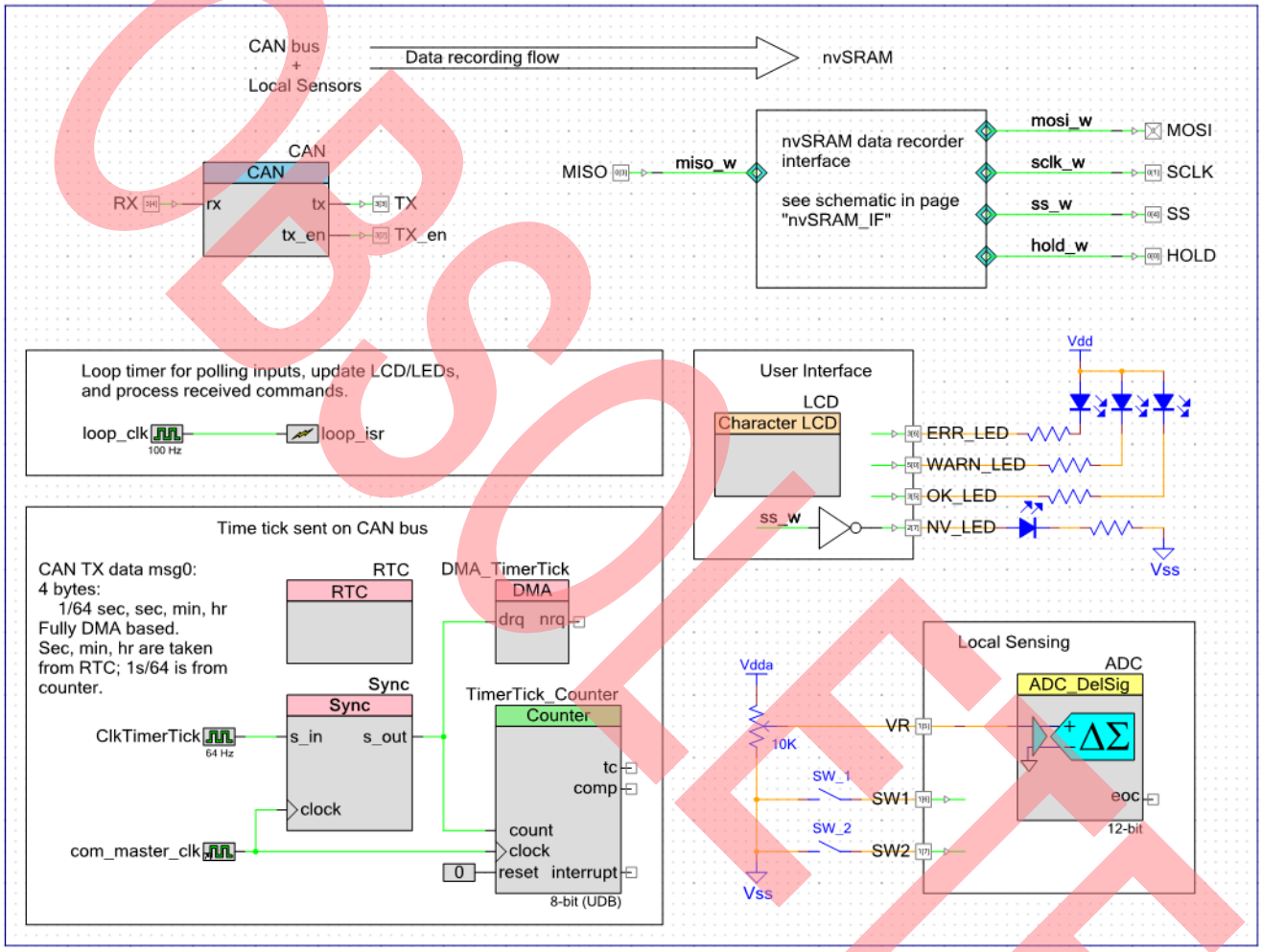



PROFIT

## PSoC 3 schematic implementation overview

Figure 3 shows the PSoC Creator® schematic of the EDR implementation. Note that the nvSRAM interface module is presented as just a block, and its details are described in a later section (nvSRAM interface block in PSoC 3 on page 14).

Figure 3. Data recorder PSoC® Creator schematic.




## CAN component configuration

The CAN component is configured with four RX and four TX mailboxes (see Figure 4 and Figure 5).

Three RX mailboxes handle messages recorded in the nvSRAM, and one RX mailbox acts as a “command register,” accepting a 4-byte message. The data recorder interprets that 4-byte message as a command (first byte) with parameters (3 bytes) to set up actions with the nvSRAM and the overall EDR system.

The “TimerTick” TX mailbox is a 4-byte message, containing the time information (hours, minutes, seconds, and n/64th of a second). This message, which is transmitted every 15.625 ms (1/64th of a second), serves as the EDR alive state communicated to other CAN nodes.

The two TX messages, “nvSRAM_Time_ID” and “nvSRAM_Data,” are transmitted as a pair over CAN every time the EDR receives a command to read one record from the nvSRAM. A record is a 16-byte block; therefore, two CAN messages are needed to transmit it. The “nvSRAM_Time_ID” contains the timestamp of the record (4 bytes) and the message ID (4 bytes). The message ID can be either the CAN message ID or an ID used to

OBSOLETE

indicate that the message content is data from local sensors. The “nvSRAM_Data” contains the recorded data for that message ID and timestamp. See also Data Format on page 14.

The “nvSRAM_LastGood” TX mailbox is for a message that is transmitted when the EDR receives a command to retrieve the last saved data (GET_NV_LAST, more commands later in System commands, page 12). An alternate implementation could be to set up the same information in an RTR message. That message contains the value of the address of the last nvSRAM 16-byte block written and the timestamp of that block.

In this project, the CAN baud rate is set at 500 kbps.

Figure 4. CAN RX buffers.

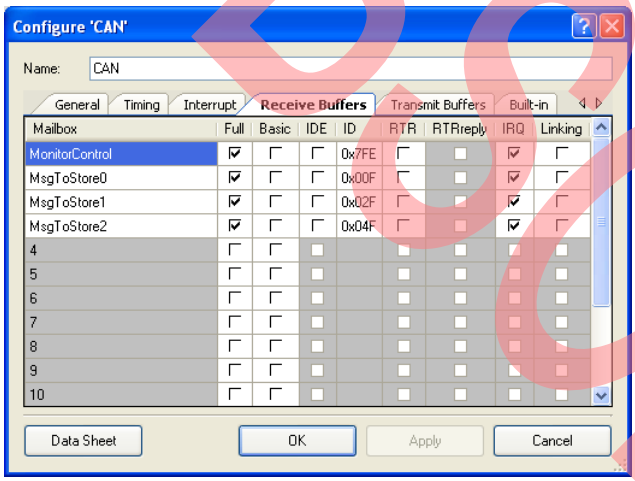
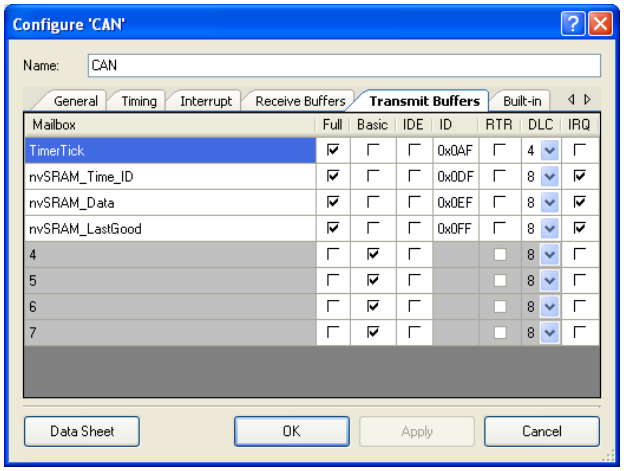



Figure 5. CAN TX buffers.

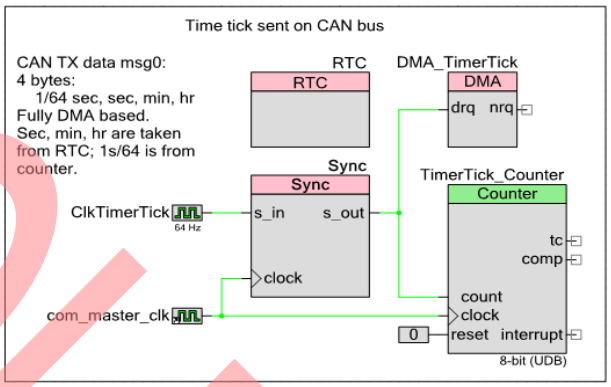



## Time keeper

The time keeper block generates the timestamp of the recorded data and sends a frequent message on the CAN bus as a fast “heartbeat” pulse to let the rest of the system know that the EDR is working.

In this project, the periodic message on the CAN bus is delivered every 15.625 ms. This information can be captured by a node emulated by a PC and later compared with the last saved data on the nvSRAM. As a result, you can evaluate the effectiveness of the solution and compare it with other approaches that rely on EEPROM or other nonvolatile memories.

Figure 6. Time keeper block schematic.




The real-time clock, based on the external 32768-Hz crystal, generates an interrupt every second. During that interrupt, the system date and time are updated.

The “ClkTimerTick” is a 64-Hz clock derived from the 32768-Hz crystal as well, making it synchronous with the RTC. The TimerTick_Counter input is synchronized to the com_master_clk clock with the Sync component.
On every rising edge of the ClkTimerTick clock, the TimerTick_Counter value is incremented (up to 63 and then reset to 0), and a DMA is triggered.

The DMA performs two tasks:

1. Transfer hr, min, sec from the RTC registers to the TimerTick CAN mailbox;
   transfer the TimerTick_Counter (n/64th of a second) value to the TimerTick CAN mailbox;
   set the CAN_SEND_MESSAGE bit in the TimerTick mailbox to transmit the message.
2. Copy the hr, min, sec from the RTC, and the n/64th of a second from the TimerTick_Counter to a 4-byte buffer in the PSoC 3 RAM. This timestamp will be attached to the data logged message.

In the associated project, the function that performs the DMA configuration is DMA_TimerTick_Setup():

PROFIT

Code 1. DMA_TimerTick_Setup() function.

```
/*******************************************************************************
* Function Name:   DMA_TimerTick_Setup
********************************************************************************
* Summary:
*  This function initializes the DMA channel, transferring the RTC time and 1s/64
*  to the CAN TX msg and to an internal SRAM buffer.
* Parameters:
*  void
* Return:
*  DMA channel value
*******************************************************************************/

static uint8 DMA_TimerTick_Setup(void)
{
  uint8 DMA_TT_Ch;
  static uint8 cmd_tx_data[] = {(uint8)CAN_SEND_MESSAGE, 0u, 0u, 0u};

  /* Initialize DMA: burstCount=0 (whole transfer in one burst)    */
  /*                 requestPerBurst=0 (auto)                      */
  /*                 upperSrcAddress=0 (upper 16 bits source addr) */
  /*                 upperDestAddress=0 (upper 16 bits dest addr)  */
  DMA_TT_Ch = DMA_TimerTick_DmaInitialize(0u, 0u, 0u, 0u);

  /* Allocate TDs for different data - TD variables are declared at the module level */
  DMA_TT_TD_data = CyDmaTdAllocate();
  DMA_TT_TD_tick = CyDmaTdAllocate();
  DMA_TT_TD_send = CyDmaTdAllocate();
  DMA_TT_TD_tb[0] = CyDmaTdAllocate();
  DMA_TT_TD_tb[1] = CyDmaTdAllocate();

  /*******************************************************************
  * Configure TDs: TD handle, bytes to be transferred, next TD, options
  *  AND Setup TDs addresses: TD handle, source address, dest address
  *******************************************************************/
  /* Transfer sec, min, hour with one TD.  Assumes that sec, min, hour are in
  *  consecutive addresses */
  (void) CyDmaTdSetConfiguration(DMA_TT_TD_data, 3u, DMA_TT_TD_tick, TD_INC_DST_ADR | \
                                                                     TD_INC_SRC_ADR | \
                                                                     TD_AUTO_EXEC_NEXT);
  (void) CyDmaTdSetAddress(DMA_TT_TD_data, LO16((uint32)(&RTC_currentTimeDate.Sec)), \
                                           LO16((uint32)(&CAN_TX[0].txdata.byte[1u])));

  /* Transfer 1/64 sec value */
  (void) CyDmaTdSetConfiguration(DMA_TT_TD_tick, 1u, DMA_TT_TD_send, TD_AUTO_EXEC_NEXT);
  (void) CyDmaTdSetAddress(DMA_TT_TD_tick,LO16((uint32)(TimerTick_Counter_COUNTER_LSB_PTR)),\
                                          LO16((uint32)(&CAN_TX[0].txdata.byte[0u])));

  /* Send message command */
  (void) CyDmaTdSetConfiguration(DMA_TT_TD_send, 4u, DMA_TT_TD_tb[0], TD_AUTO_EXEC_NEXT);
  (void) CyDmaTdSetAddress(DMA_TT_TD_send, LO16((uint32)(cmd_tx_data)), \
                                           LO16((uint32)(&CAN_TX[0u].txcmd)));

  /* Copy time to buffer: rtc_time_buf[1] = sec
  *                       rtc_time_buf[2] = min
  *                       rtc_time_buf[3] = hr */
  (void) CyDmaTdSetConfiguration(DMA_TT_TD_tb[0], 3u, DMA_TT_TD_tb[1], TD_INC_DST_ADR | \
                                                                       TD_INC_SRC_ADR | \
                                                                       TD_AUTO_EXEC_NEXT);
```

OBSOLETE

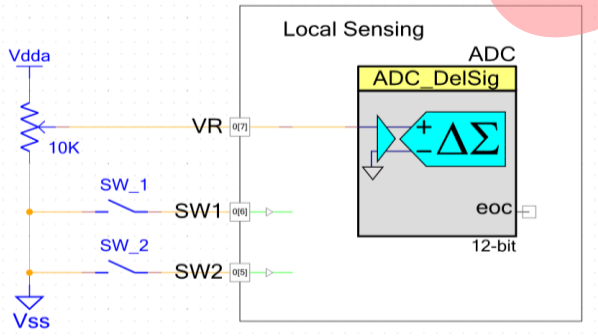
```
    (void) CyDmaTdSetAddress(DMA_TT_TD_tb[0], LO16((uint32)(&RTC_currentTimeDate.Sec)), \
                                              LO16((uint32)(&rtc_time_buf[1])));

    /* Copy time to buffer: rtc_time_buf[0] = 1/64 sec */
    (void) CyDmaTdSetConfiguration(DMA_TT_TD_tb[1], 1u, DMA_TT_TD_data, 0u);
    (void)CyDmaTdSetAddress(DMA_TT_TD_tb[1],LO16((uint32)(TimerTick_Counter_COUNTER_LSB_PTR)),\
                                            LO16((uint32)(&rtc_time_buf[0])));

    /* Set initial TD */
    (void) CyDmaChSetInitialTd(DMA_TT_Ch, DMA_TT_TD_data);

    return DMA_TT_Ch;
}
```

## Local Sensing

The local sensing block emulates the sampling of an analog signal from a sensor and the monitoring of two digital signals. A potentiometer and two switches on the PSoC 3 CY8CKIT-001 and CY8CKIT-030 development kits simulate the local sensor and the digital signals.

In this example, both the ADC (converting the potentiometer voltage) and the digital pins (connected to the switches) are sampled periodically with a polling method (every 10 ms). An interrupt-based design can be implemented, too.

Figure 7. Local sensing schematic.

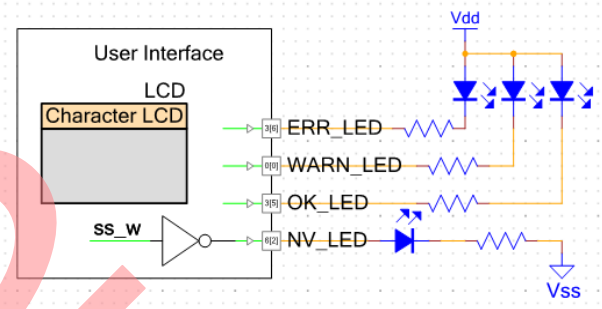



System requirements determine the choice between polling and interrupt-based design. Specific system decisions are beyond the scope of this application note.

## User Interface

The user interface consists of messages on an LCD and signals on LEDs.

Figure 8. User Interface block schematic.




The LCD displays messages that depend on the state of the data recorder and on the command received (see System commands on page 12).

The first row of the LCD shows a response to the command received. If one row is not sufficient to display the response, the second row also is used.

In most cases, the second row shows the 12-bit ADC value, the state of the switches, and the time. The state of the two switches is implemented as a toggle so that one push-release will be read as “ON” and the next push-release as “OFF”. When a button is “ON”, its number (“1” or “2”) will show on the LCD.

Below is a description of the different LCD states:

- Figure 9: When in standby (at power up or when the data recording/reading is stopped), the first row maintains the data from the last command, and the second row shows the ADC value, the state of the switches, and the time.
- Figure 10: While the data logger is actively storing data, the first row displays the number of received CAN messages to store and the number of data blocks stored in the nvSRAM. The two numbers are different when the local sensors data also is stored.

OBSOLETE

- Figure 11: When the EDR receives a command to retrieve the last saved address (GET_NV_LAST), the first row displays hex values of the storage location of the last saved address ("01" or "02", see also DMA flow and control for data recording on page 15), the block address value (3-byte), and the timestamp value at that address (4-byte). See Figure 29 and associated text on page 25 for an example.
- Figure 12: When the EDR receives a command to read the nvSRAM records, the first row displays the 17-bit address being read, the timestamp at that address (only min, sec, n/64th of a second), and the message ID (because of character limitations in the LCD used, only 11-bit IDs are displayed, or a 3-bit value to indicate local sensors). The second row contains the 8-byte message value.
- Figure 13: When the EDR receives a command to read a revision of the nvSRAM module firmware, the first row displays its value. The second row displays the ADC value, the state of the switches, and the time.

The ERR_LED, WARN_LED, and OK_LED signals are controlled in firmware; they are updated every 10 ms and whenever a CAN message is received. The ERR_LED and WARN_LED statuses are tied to the CAN error count maintained in the CAN hardware block. The OK_LED is pulsed every time a message is received successfully.

The NV_LED signal is connected internally in PSoC 3 to the SPI Slave Select pin, so that every SPI communication prompts the external LED to pulse.

Figure 9. LCD in standby.

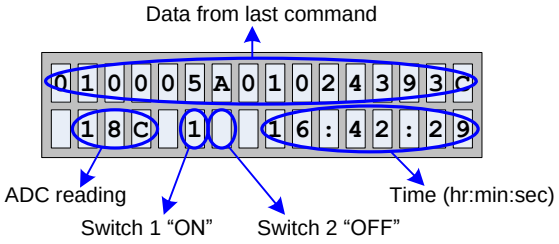



Figure 10. LCD during data recording.

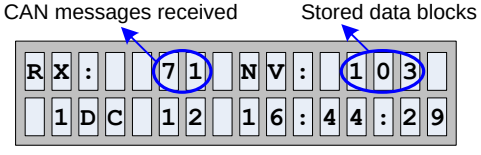



Figure 11. LCD showing last saved address.

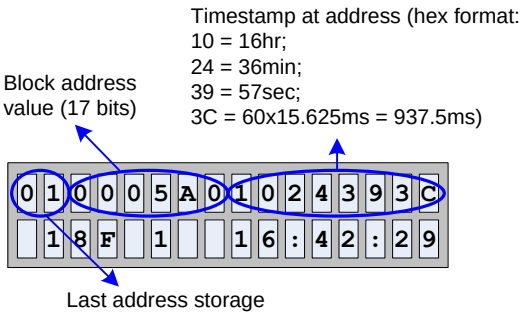



Figure 12. LCD during nvSRAM read.

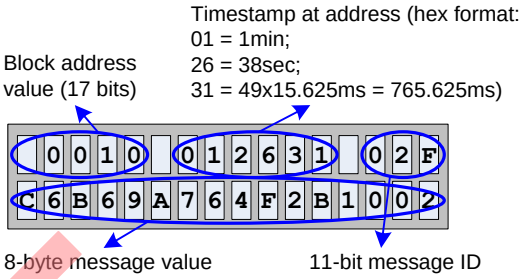
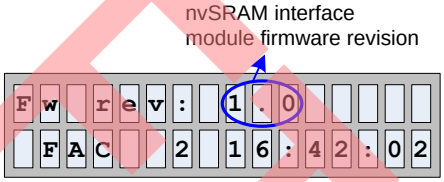


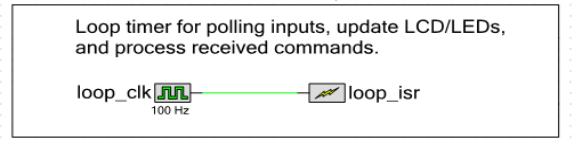

Figure 13. LCD displaying firmware revision.




## Loop timer and sensor polling

The loop timer is a clock tied to an interrupt component. Its associated interrupt service routine (ISR) sets a flag that is read in the main loop. Every time the flag is set, the main loop executes polling and processing tasks after clearing the flag.

Figure 14. Loop Timer.




OBSOLETE

## Firmware architecture

The nvSRAM data logger firmware is composed of three layers (excluding PSoC Creator components layer):

1. A main function at the top layer controls the overall flow (initialization; loop: check commands, poll inputs, update outputs/UI, poll low-voltage detect circuit). See Figure 15.
2. A system functions module (system_func.c/h) includes the functions called by main and local helper functions. See Figure 16.
3. The nvSRAM data recorder interface module at the bottom layer provides the public APIs used by the system functions module to access the nvSRAM interface (nvSRAM interface block in PSoC 3 on page 14).

Figure 15. main() firmware flow.

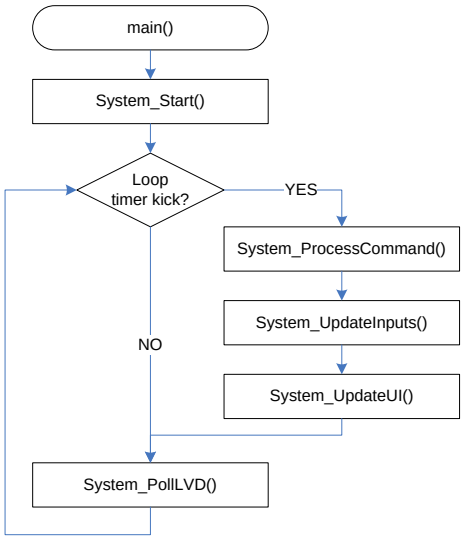
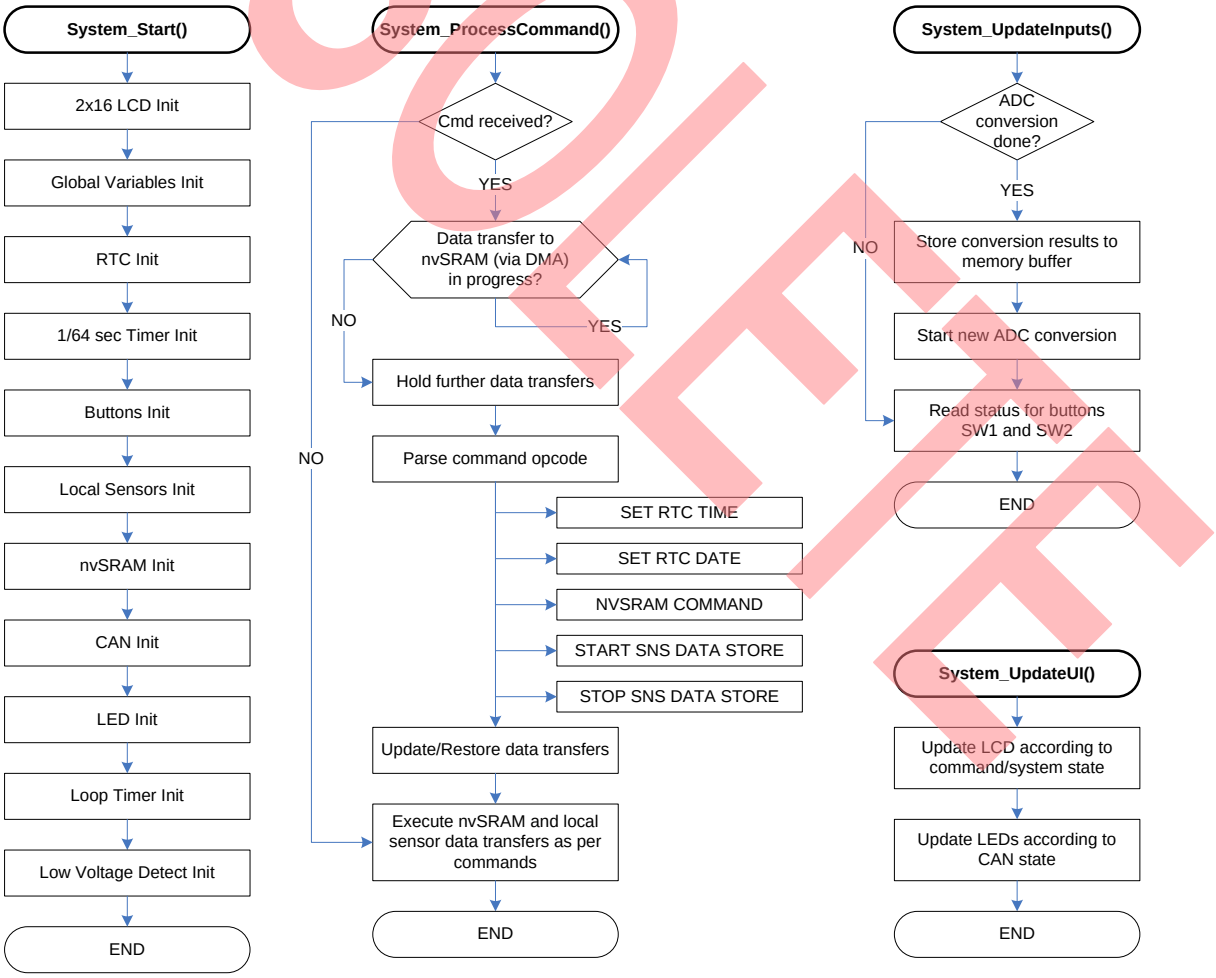



Figure 16. System functions module public APIs flowchart.




OBSOLETE

Four interrupt sources are enabled:

1. loop_isr: This ISR is triggered by the loop timer clock, and it signals the execution of the main loop once. It is currently set so that the main loop is executed every 10 ms.
2. CAN_isr: This is the CAN interrupt routine, which is executed every time a message is received. If the message is a command (ID=0x7FE), then the CAN interrupt routine sets a flag for the main loop to process the command. If the message received is meant to be stored in nvSRAM, then the ISR calls nvSRAM interface APIs to initiate a DMA transfer from CAN to the SPI nvSRAM device. Note that a command (ID=0x7FE) may be received while a DMA transfer is in progress. For that reason, the function System_ProcessCommand() shown in Figure 16 blocks until the existing transfer is complete.
3. RTC_isr: This is a once-per-second interrupt that updates the time and date.
4. ADC_IRQ: The ADC ISR stops the conversion of the potentiometer voltage. It restarts when the main loop polls the inputs.

Global variables include:

- Flags (command received, CAN message received, loop timer kick, one second tick)
- Data arrays for CAN transmission
- CAN command and parameters buffer for value received
- Two counters for messages received from CAN and transmitted to nvSRAM, for display on the LCD.

## Recording locally sensed data using DMA seamlessly with CAN messages

The EDR based on PSoC 3 can record locally acquired sensors data seamlessly with CAN messages using the nvSRAM interface block APIs.

In the project associated with this application note, PSoC 3 periodically polls the ADC and the buttons and then stores the respective ADC result and buttons status in local variables. When PSoC 3 receives a CAN command message asking to store sensors data, it fills a 12-byte buffer with the ADC and buttons data (4 bytes for ID and 8 bytes for data). PSoC 3 then initiates data recording through the nvSRAM interface using the nvSRAM block APIs (see also nvSRAM interface module public APIs on page 19). DMA then performs data recording, transferring the data from the 12-byte buffer (containing the sensors data) to the nvSRAM, as described in DMA flow and control for data recording on page 15.

The implementation of that process follows:

Code 2. System_StoreSensorData() function.

```
static void System_StoreSensorsData(void)
{
  uint8 interruptStatus;

  /* Kick the store to nvSRAM if enabled
and if a new sample is available*/
  if((0u!=set_dma_sns)&&(0u!=sns_updated))
  {
    sns_updated = 0u;
    sensors_data[3]  = SNS_ID3;
    sensors_data[2]  = SNS_ID2;
    sensors_data[1]  = SNS_ID1;
    sensors_data[0]  = SNS_ID0;
    sensors_data[7]  =(uint8)(adc_data>>8);
                        /* MSB first */
    sensors_data[6]  = (uint8)(adc_data);
    sensors_data[5]  = button1;
    sensors_data[4]  = button2;
    sensors_data[11] = 0u;
    sensors_data[10] = 0u;
    sensors_data[9]  = 0u;
    sensors_data[8]  = 0u;

    interruptStatus = \
                    CyEnterCriticalSection();
    NVSRAM_WaitRecordComplete();
    NVSRAM_SetSourceData(sensors_data);
    (void) NVSRAM_RecordDataAction( \
                              record_action);
    CyExitCriticalSection(interruptStatus);
    […]
  }
}
```

Data records have a 4-byte timestamp attached; however, PSoC 3 does not set the 4-byte timestamp data pointer (used by the DMA process) in the code above. The DMA source location for the timestamp is common for both locally sensed data and CAN messages being recorded. Therefore, PSoC 3 needs to set the timestamp source location only once in the initialization code with the `NVSRAM_SetSourceTimeStamp` API.

The sensors data preparation and DMA request occurs only if PSoC 3 has polled the sensors (`sns_updated`) and if it has received the command to store sensors data (`set_dma_sns`).

After preparing the data in the 12-byte buffer, the code execution enters a critical section. Within that section, the firmware triggers the DMA to transfer the data to nvSRAM. Such a critical section is needed to keep the DMA process from conflicting with the DMA triggered in the CAN interrupt routine.

OBSOLETE

**Note** Since the CAN RX message ID register is 4 bytes and does not use the 3 least significant bits, the “message ID” for the sensors data has been set to 0x00000007.

## System commands

The CAN message ID 0x7FE sends commands to the data recorder. It contains the command code and relevant parameters (total of 4 bytes) transmitted to the nvSRAM data recorder. The following table summarizes the command names, opcodes, and description.

Table 1. System commands summary table.

| Command Name | Opcode (CAN byte 0) | Parameters (3 bytes) | Description |
|---|---|---|---|
| SET_RTC_TIME[1] | 0x01 | Hr, min, sec (byte 1, 2, 3) | Set Real Time Clock time |
| SET_RTC_DATE[1] | 0x02 | Yr, mo, day (byte 1, 2, 3) | Set Real Time Clock date |
| START_NV_STORE[2] | 0x03 | 17-bit starting address (MSB = byte 1) | Initiates the data logging. The initial address provided is 16-byte aligned automatically for writing to nvSRAM (the last 4 bits of the 3-byte parameter are ignored) |
| STOP_NV_STORE[2] | 0x04 | Don’t care | Terminates the data logging |
| GET_NV_LAST[2] | 0x05 | Don’t care | Returns (on LCD and on CAN message ID 0x0FF) the address of the last stored data and its timestamp. This command will terminate the data logging. |
| START_NV_READ[2] | 0x06 | 17-bit starting address (MSB = byte 1) | Initiates the data retrieval. The initial address provided is 16-byte aligned automatically.<br>Reports the 16-byte record at the initial address.<br>Read data is transmitted on CAN message IDs 0x0DF (timestamp + ID) and 0x0EF (data)<br>This command will terminate the data logging. |
| NEXT_NV_READ[2] | 0x07 | Don’t care | This command requires the system to be in nvSRAM read mode via the START_NV_READ command.<br>Reports the next 16-byte record.<br>Read data is transmitted on CAN message IDs 0x0DF (timestamp + ID) and 0x0EF (data) |
| PREV_NV_READ[2] | 0x08 | Don’t care | This command requires the system to be in nvSRAM read mode via the START_NV_READ command.<br>Reports the previous 16-byte record.<br>Read data is transmitted on CAN message IDs 0x0DF (timestamp + ID) and 0x0EF (data) |
| STOP_NV_READ[2] | 0x09 | Don’t care | Terminates the read process. Read commands are then ignored until a new START_NV_READ command is received |
| FORCE_SW_STORE[2] | 0x0A | Don’t care | Forces a nvSRAM software store by sending the Store command to the nvSRAM via SPI.<br>After execution, data logging will continue as before this command request. |
| FORCE_SW_RECALL[2] | 0x0B | Don’t care | Forces a nvSRAM recall by sending the Recall command to the nvSRAM via SPI.<br>After execution, data logging will continue as before this command request. |

[1] This is an application-specific system command.

[2] This is part of the nvSRAM recorder interface module and is exposed (with a user-defined offset) to the higher-level application.

OBSOLETE

| Command Name | Opcode (CAN byte 0) | Parameters (3 bytes) | Description |
|---|---|---|---|
| GET_FW_VERSION[2] | 0x20 | Don't care | Returns the nvSRAM interface module firmware revision. This is displayed on the LCD.<br>Data logging and nvSRAM reading must be stopped before requesting this command (with STOP_NV_STORE and STOP_NV_READ). |
| START_SNS_STORE[1] | 0x35 | Don't care | Starts recording local sensors data to nvSRAM. Note that START_NV_STORE command must be issued first. |
| STOP_SNS_STORE[1] | 0x36 | Don't care | Stops recording local sensors data. |

## Example of using commands: replay recorded data

Recorded data is retrieved by sending the proper commands with CAN message ID 0x7FE.

A typical procedure to read the logged data is as follows (from a requesting CAN node):

1. Send message ID 0x7FE with command (data byte 0) set to GET_NV_LAST (0x05 in this case, or 0x03 + any offset if the data logger application implements other commands besides those implemented in the nvSRAM module). After this message is received, the data logger sends a message ID 0x0FF with an 8-byte data: byte 0 is the storage location of the last saved address ("01" or "02", see also DMA flow and control for data recording on page 15); bytes 1-3 contain the 17-bit nvSRAM memory address of the last valid saved data; bytes 4-7 contain the timestamp for the record at the reported memory address (hr, min, sec, n/64th of a second).
2. Send message ID 0x7FE with command (data byte 0) set to START_NV_READ (0x06 in this case, or 0x04 + offset), and the starting address (byte 1-3). The starting address is a 17-bit value; however, due to the 16-byte alignment configuration of the data logger, the last 4 bits are ignored. Since the data logger writes to the nvSRAM as a circular buffer, a good starting address is the one after the last valid saved address indicated by the command executed in step 1.
When the data logger receives this command, it sends back two messages: message ID 0x0DF contains the timestamp and the message ID of the data at this address, and message ID 0x0EF contains the data.
3. Send message ID 0x7FE with command set to NEXT_NV_READ (0x07 in this case, or 0x05 + offset) or PREV_NV_READ (0x08 in this case, or 0x06 + offset) to read recorded events forward or backward, respectively. As in step 2, when the data logger receives this command, it sends back two messages: message ID 0x0DF contains the timestamp and the message ID of the data at this address, and message ID 0x0EF contains the data.
4. When all the information has been retrieved, send message ID 0x7FE with command set to STOP_NV_READ (0x09 in this case, or 0x07 + offset). When the data logger receives this command, it exits "read mode," and read commands have no effect unless a new START_NV_READ command is transmitted.

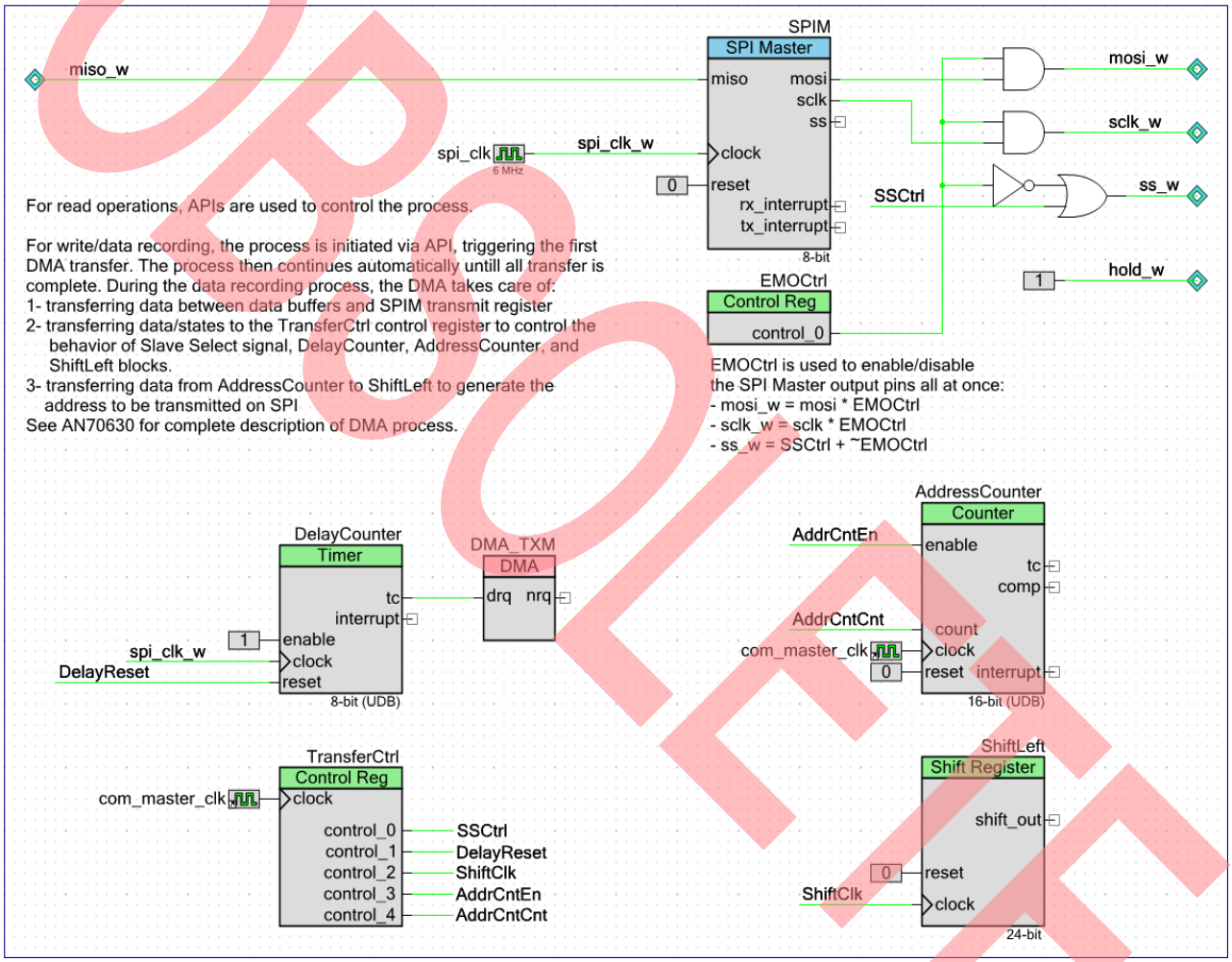
OBSOLETE

# nvSRAM interface block in PSoC 3

## Overview

The nvSRAM data recorder interface driver module consists of both hardware (digital blocks and DMA) and firmware (APIs) components. Figure 17 below shows the PSoC Creator schematic. The files `nvsram_if.c` and `nvsram_if.h` in the associated project contain the entire API implementation and the public interface.

Figure 17. nvSRAM recorder interface PSoC Creator schematic.




## Data Format

Before entering the details of the schematic and the architecture of the nvSRAM interface block, let's understand how a data record is structured.

Each record consists of 16 bytes, whose content is as follows (bytes start on the 16-byte aligned address):

Table 2. Data format of records stored in nvSRAM.

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
|---|---|---|---|---|---|---|---|
| HR | MIN | SEC | N/64$^{th}$ sec | Msg ID MSB | Msg ID B2 | Msg ID B1 | Msg ID LSB |
| **Byte 8** | **Byte 9** | **Byte 10** | **Byte 11** | **Byte 12** | **Byte 13** | **Byte 14** | **Byte 15** |
| Data Hi MSB | Data Hi B2 | Data Hi B1 | Data Hi LSB | Data Lo MSB | Data Lo B2 | Data Lo B1 | Data Lo LSB |

OBSOLETE

The above table represents the data record used in this implementation, for both data stored to nvSRAM and nvSRAM data transmitted through CAN.

When handling a data block (16 bytes), the nvSRAM interface module expects only pointers to a 4-byte array (for the "timestamp" field) and to a 12-byte array (for the "ID+data" field). No restriction is imposed on the type of data on those two arrays. The two APIs used to set such pointers are `NVSRAM_SetSourceTimeStamp()` and `NVSRAM_SetSourceData()` respectively. The higher-level application can direct the data logger to any type of data, as long as the array sizes are not violated.

In the PSoC 3 RTC component, the RTC_TIME_DATE structure contains hours, minutes, and seconds in three consecutive bytes; the CAN ID and message data are in 12 consecutive bytes in the CAN hardware. As a result, simple settings for the 4-byte and 12-byte array pointers are enabled.

**Note** Since the data logged is automatically aligned to 16-byte addresses, the nvSRAM interface module APIs allow access to the external nvSRAM only at 16-byte aligned addresses.

## nvSRAM interface module schematic design

The main goal of the nvSRAM interface module is to control and execute the complete data recording process through DMA. The process would be independent of the CPU, freeing it to perform other tasks.

Given the importance of storing required information without software delays when a system fails, writing to the nvSRAM must be software-independent as much as possible. However, retrieving data from the nvSRAM is not as critical, because that action is expected in a controlled environment for an EDR system. Therefore, in this implementation, data retrieval (read from nvSRAM) is carried out in firmware.

In addition, during a power outage, the PSoC 3 may see the power drop before the nvSRAM does, or vice versa. In both cases, the data link must be protected and shut down gracefully to prevent a potential loss of data.

Let's examine the tasks accomplished, block by block:

- **SPIM**: The SPI Master is the main communication block, and it controls transfers of data bytes.
- **DelayCounter**: Because the SPIM has a hardware FIFO buffer with only 4 bytes, DMA transfers need to be controlled so as not to overflow the buffer. Therefore, the DMA is set up to transfer only one byte at a time to the SPIM, and the DelayCounter counts a specified number of SPI clocks before triggering the next DMA transfer.
- **AddressCounter**: This counter increments by 1 for every 16-byte data block transferred to nvSRAM through SPI. Since the nvSRAM memory used is 1 Mbit, arranged in 128k × 8, the total number of available 16-byte blocks is 8192. However, the last two "blocks" are used to keep track of the last saved address (see DMA flow and control for data recording), so only 8190 blocks are available. For that reason, the AddressCounter has a period set to 8190. Once it reaches 8190, the next address is set to 0, effectively using the nvSRAM as a circular buffer.
- **ShiftLeft**: This shift register sets the nvSRAM address. The current record block count is transferred (via DMA) from the AddressCounter to ShiftLeft, shifted left by 4 (to account for the 16-byte data alignment), and then sent over SPI as memory address (via DMA, as well).
- **DMA_TXM**: This instantiates the DMA channel that will handle the complete data store process.
- **TransferCtrl**: This control register controls signals to sequence the data transfer through SPI. This register is updated through DMA during data recording.
  - SSCtrl: Slave Select control signal. Set low during SPI transactions.
  - DelayReset: Resets the DelayCounter, which is out of reset only for the duration of data recording for each block of data (16 bytes).
  - ShiftClk: Clock input to the ShiftLeft component, which generates the 17-bit nvSRAM memory address.
  - AddrCntEn: Enable/disable the AddressCounter.
  - AddrCntCnt: Count pulse to the AddressCounter. A pulse is sent every time a 16-byte data block is transmitted to nvSRAM.
- **EMOCtrl** (EMergency Off Control): In case PSoC 3 sees a voltage drop before the nvSRAM does, PSoC 3 can prevent further SPI communication by forcing MOSI and SCLK low and SS high. This action removes the risk of an undefined voltage on such pins, a situation that could result in issuing commands or corrupt data in the nvSRAM. When a voltage drop occurs, it is sufficient for PSoC 3 to write '0' to this control bit to shut down all three pins. When the nvSRAM sees the voltage drop, it automatically triggers an AUTOSTORE event, its data is stored to the non-volatile cells, and its SPI interface is disabled.

**Note** The hold_w signal controls the HOLDb pin of the nvSRAM to suspend serial operation. This functionality is not used in the associated project; therefore, the signal is set to logic high ('1') to ensure it is always inactive.

## DMA flow and control for data recording

To execute a write operation to an nvSRAM, the master device (PSoC 3 in this case) first must send a WREN (write enable) command and then a WR (write) command,

OBSOLETE

followed by the destination address (3 bytes) and the data (the address increments automatically at each subsequent byte). The figures below are taken from the nvSRAM datasheet (CY14B101Q) and show the WREN command and the write timing for reference.

Figure 18. nvSRAM WREN instruction.

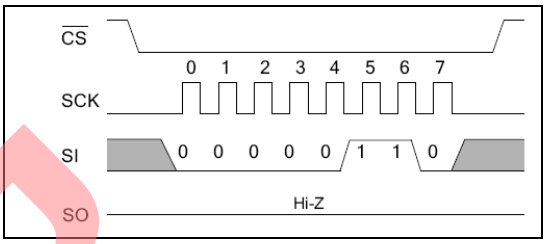
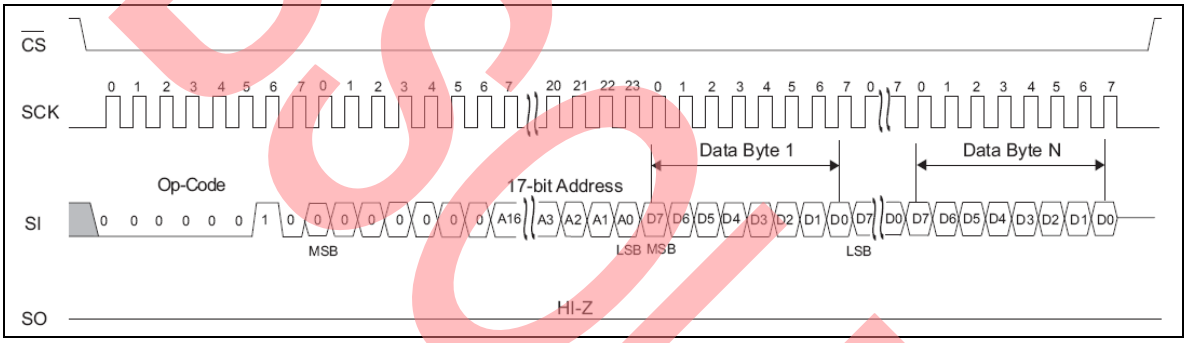



Figure 19. nvSRAM multi-byte write.




The design of the DMA channel to perform the full data recording process involves structuring a sequence for:

- sending the 16-byte data block to the SPI module for transmission to the nvSRAM;
- and controlling the SPI communication itself with nvSRAM-specific commands, memory address, and control signals (information that is not part of the data that needs to be recorded).

Therefore, the DMA will have to control the Slave Select pin (SS) and set the proper memory address on the SPI bus. Moreover, the DMA cannot send all the data (16-byte) to the SPI module at once, because the SPI has only a 4-byte buffer.

The DMA design then will require the support of:

- A timer/counter (DelayCounter) to count a specific number of SPI clocks before requesting the next transaction in the DMA chain (each transaction executes and sends the next byte or control signals).
- A counter (AddressCounter) to count the 16-byte data blocks being transmitted and, thus, set the memory address properly (with the help of the ShiftLeft block).
- A control register (TransferCtrl) to set/clear control signals through DMA writes to it.

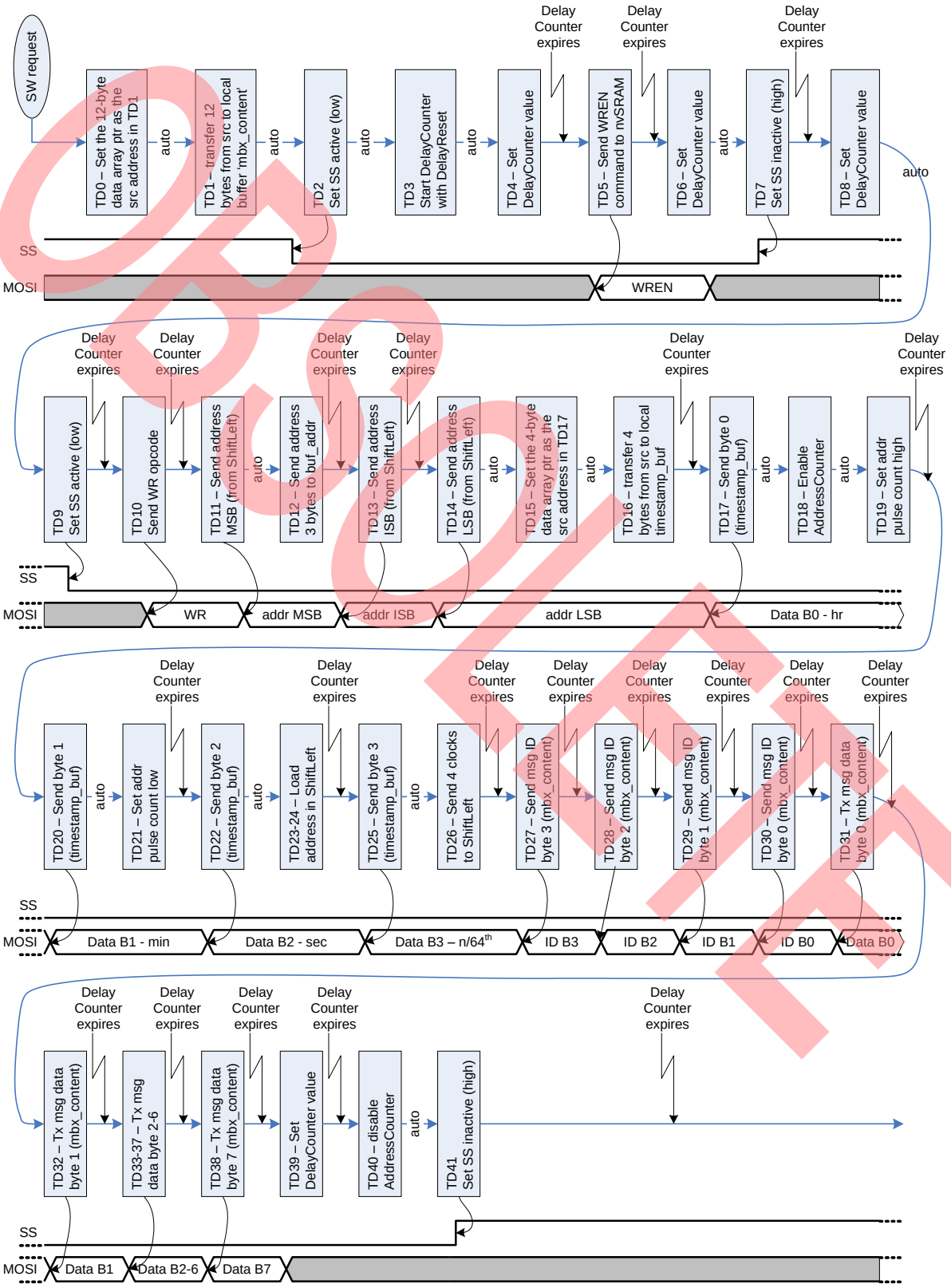
In addition, the DMA sequence implements a mechanism to maintain a record of the address of the last valid stored location. Such a mechanism invalidates a partial record received during a power loss. This mechanism works as follows:

After each 16-byte block write, two blocks (addresses 0x01FFE0 and 0x01FFF0) are used in ping-pong mode to store the address and the timestamp of the block that was just written. The timestamp is recorded as a validation token. When PSoC 3 reads back the data, the ping-pong locations identify the last valid stored record. The timestamp in the ping-pong locations is compared with the one recorded at the address pointed by the ping-pong locations themselves. Any difference would indicate that the address in the ping-pong location is not a valid last stored location. This mechanism guarantees that one of the two records points to the last valid stored location.

The diagrams in Figure 20 and Figure 21 describe the DMA flow for the data logging process.

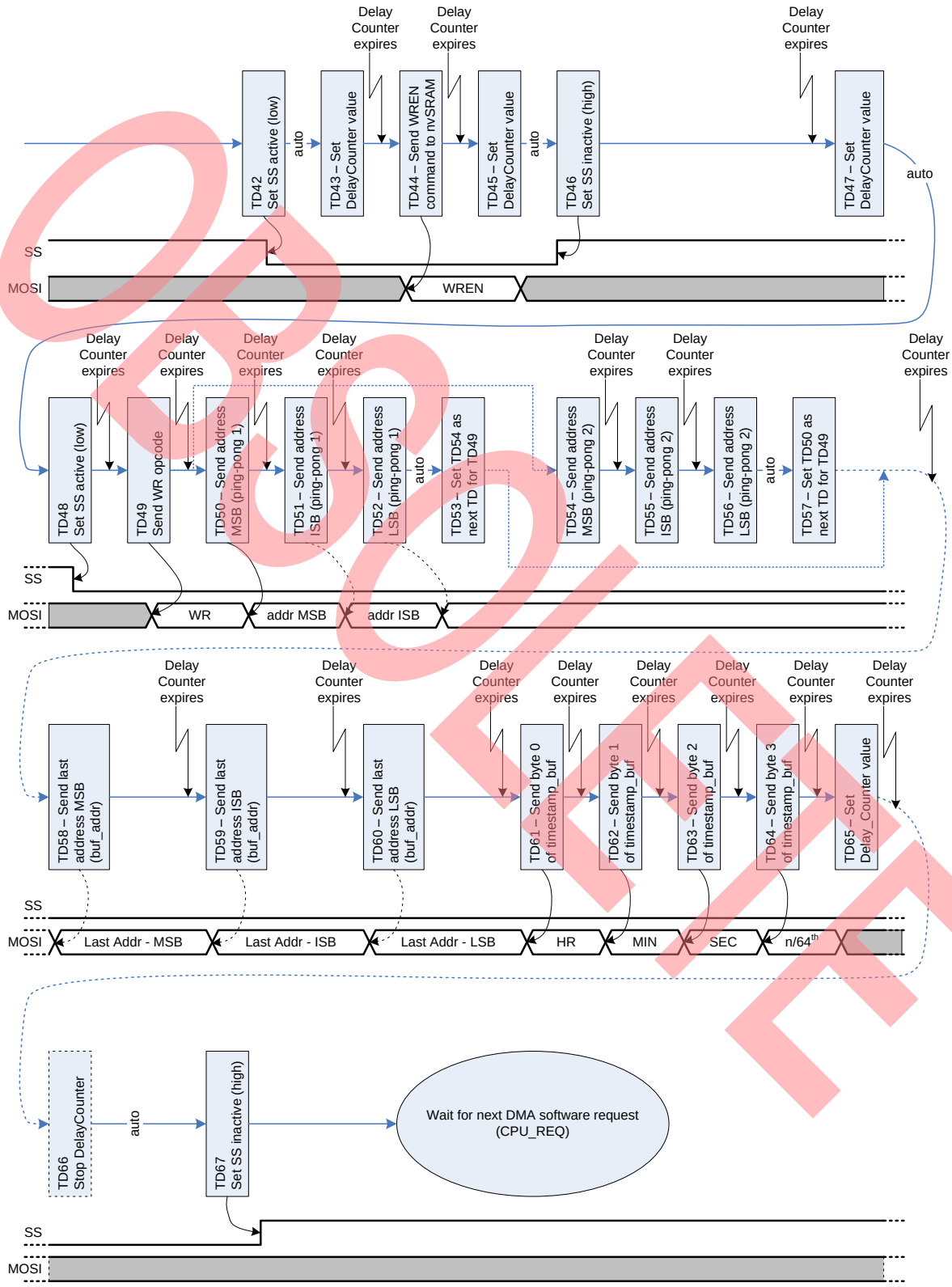
OBSOLETE

Figure 20. DMA flow and control sequence for data logging – page 1 of 2.




OBSOLETE

Figure 21. DMA flow and control sequence for data logging – page 2 of 2.




DRAFT

## nvSRAM interface module public APIs

The nvSRAM data recorder interface driver module (in the associated project) currently offers the following APIs.

### NVSRAM_IF_Start()

Prototype: `void NVSRAM_IF_Start(void)`

This function initializes the nvSRAM interface module, including DMA, interrupts, PSoC 3 components used, and variables. Specifically, the following actions are performed:

- Initialize module global variables
- Initialize control registers
- Verify RDYb bit status to make sure that the nvSRAM has completed its power up sequence
- Initialize DelayCounter, AddressCounter, ShiftLeft, and SPIM modules
- Set SPIM clock frequency
- Initialize and set up DMA channel and enable it

### NVSRAM_IF_Stop()

Prototype: `void NVSRAM_IF_Stop(void)`

The stop function resets the nvSRAM interface module to a global Stop state, including DMA, interrupts, PSoC 3 components used, and variables.

### NVSRAM_RecordDataAction()

Prototype: `cystatus NVSRAM_RecordDataAction (uint8 request) CYREENTRANT`

This function, which initiates or terminates data recording, sets the initial status of the TransferCtrl register and replicates the functionality of `CyDmaChSetRequest`. The function is defined as re-entrant due to its use in the main software loop and in the CAN interrupt routine.

### NVSRAM_SetSourceData()

Prototype: `void NVSRAM_SetSourceData(uint8 * const p_src_data) CYREENTRANT`

This function sets the source address of the data (12 bytes) to be recorded.

Parameters: `p_src_data` = pointer to the source data

The function is defined as re-entrant for the same reason cited in the previous function.

### NVSRAM_SetSourceTimeStamp()

Prototype: `void NVSRAM_SetSourceTimeStamp (uint8 * const p_src_time)`

This function is used to set the source address of the timestamp (4 bytes) to be recorded.

Parameters: `p_src_time` = pointer to the source timestamp

### NVSRAM_WaitRecordComplete()

Prototype:
`void NVSRAM_WaitRecordComplete(void)`

This function blocks until the full DMA transaction for one data block recording is complete.

### NVSRAM_Action()

Prototype: `void NVSRAM_Action(u_32b_t const * const p_cmd, act_result_t * const p_act)`

This function sets the task of the nvSRAM data recorder interface module. A state machine is implemented to control the system actions (start/stop store, set RTC, retrieve data, etc.)

Parameters:

`p_cmd` = pointer to 32-bit data (byte[0] = command; byte[1..3] = parameters)

`p_act` = pointer to structure containing array of four 32-bit return data and a 16-bit flags variable

The following commands are implemented (see also System commands on page 12):

```
#define START_NV_STORE      ((uint8) 0x01)
/*command is followed by starting address*/

#define STOP_NV_STORE       ((uint8) 0x02)
#define GET_NV_LAST         ((uint8) 0x03)
/* command to get the address of the last
good valid data in nvSRAM */

#define START_NV_READ       ((uint8) 0x04)
/*command is followed by starting address*/

#define NEXT_NV_READ        ((uint8) 0x05)
/* this command will trigger the next 16-
byte nvSRAM data transmission to CAN */

#define PREV_NV_READ        ((uint8) 0x06)
#define STOP_NV_READ        ((uint8) 0x07)
#define FORCE_SW_STORE      ((uint8) 0x08)
#define FORCE_SW_RECALL     ((uint8) 0x09)
#define GET_FW_VERSION      ((uint8) 0x1E)
```

The following flags are reported in the results structure (`p_act`):

```
#define F_GET_NV_LAST     ((uint16) 0x0001)
/*flag GET_NV_LAST command received*/

#define F_START_NV_READ   ((uint16) 0x0002)
/*flag START_NV_READ command received*/

#define F_NEXT_NV_READ    ((uint16) 0x0004)
/*flag NEXT_NV_READ command is received*/

#define F_PREV_NV_READ    ((uint16) 0x0008)
/*flag PREV_NV_READ command is received*/
```

OBSOLETE

```
#define F_RESTORE_REC_OK  ((uint16) 0x0010)
/*flag OK to restore the recording action
request*/

#define F_FORCE_REC_START ((uint16) 0x0020)
/*force the block data recording to start*/

#define F_LAST_GOOD_1     ((uint16) 0x0040)
/*flag last good location 1 is good*/

#define F_LAST_GOOD_2     ((uint16) 0x0080)
/*flag last good location 2 is good*/

#define F_ERR_START_NV    ((uint16) 0x0100)
/*error flag for failed start (address)*/

#define F_ERR_READ_NV     ((uint16) 0x0200)
/*error flag failed read start (address)*/

#define F_NV_STORE_ON     ((uint16) 0x0400)
/*flag NV store enabled*/

#define F_FW_VERSION      ((uint16) 0x0800)
/*flag FW revision has been retrieved*/

#define F_START_NV_READ_RPT ((uint16)
0x1000)
/* flag START_NV_READ is re-issued during a
read */
```

#### NVSRAM_GetCurrentRdAddr()

Prototype:
`uint3 NVSRAM_GetCurrentRdAddr(void)`

This function returns the current nvSRAM read address. This is for informational purposes, because the read address can be modified only with the START_NV_READ command.

Return: current read address

#### NVSRAM_TripEMO()

Prototype: `void NVSRAM_TripEMO(void)`

This function sets the hardware to block the SPI interface.

#### NVSRAM_ResetEMO()

Prototype: `void NVSRAM_ResetEMO(void)`

This function resets the EMO and allows normal operation.

#### NVSRAM_WaitDevRdy()

Prototype: `void NVSRAM_WaitDevRdy(void)`

This function blocks until the RDYb bit in the nvSRAM status register is 0, indicating that the device is ready (STORE or RECALL process are completed).

### Maximum recordable time window

The nvSRAM module is currently configured to work with the Cypress 1-Mbit nvSRAM (128k × 8), which translates into 8190 16-byte blocks (or 8190 data samples; 2 blocks are used to store the address of the last saved data). Therefore, depending on the frequency of the messages (both from CAN and from local sensor data) that need to be recorded, the recordable time changes.

For example, with only CAN messages every 10 ms, 81.9 seconds of data can be recorded.

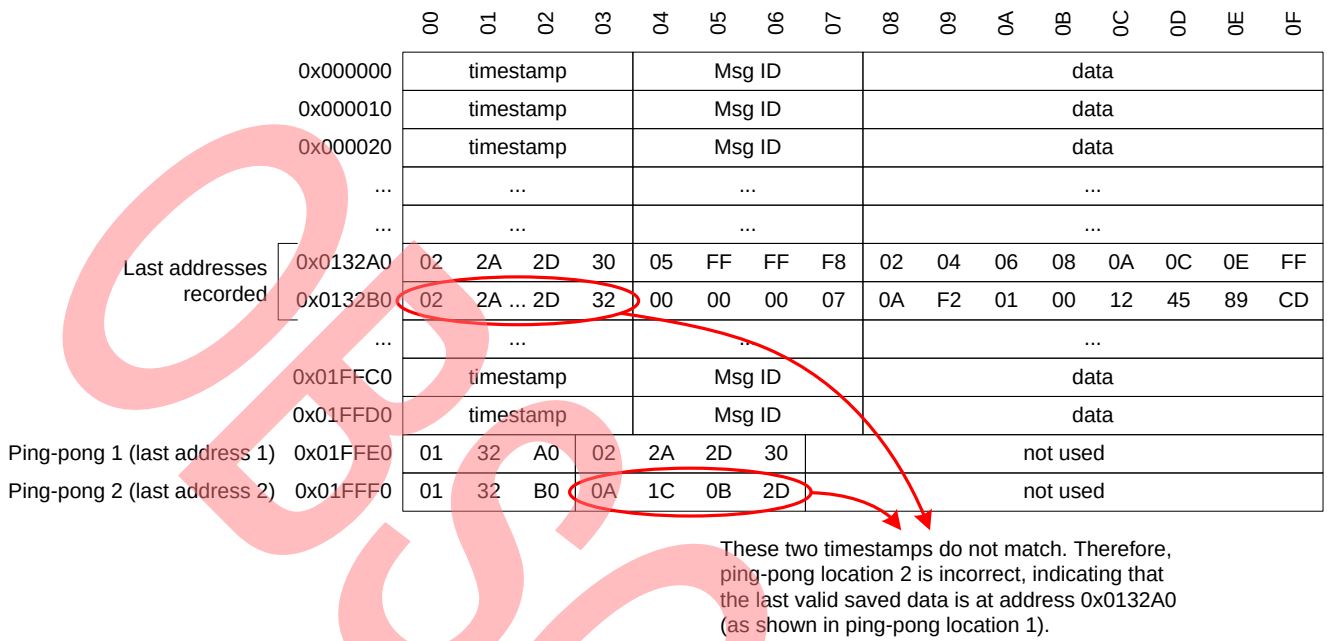
### nvSRAM interface module memory map summary

The following diagram (Figure 22) illustrates how the nvSRAM data recorder interface module maps the content of the nvSRAM memory itself. An example of the data content and last saved address is shown.

Memory blocks at addresses 0x000000 to 0x01FFD0 contain the recorded data, and locations 0x01FFE0 and 0x01FFF0 contain the address and timestamp of the last two saved locations.

OBSOLETE

Figure 22. nvSRAM memory map as implemented in the nvSRAM data recorder interface module.




OBSOLETE

## Demonstration with the PSoC 3 Development Kit, CAN Expansion Board Kit, and nvSRAM

The associated project uses the PSoC 3 development kit (DVK – CY8CKIT-001), the CAN expansion board kit (EBK – CY8CKIT-017), and the CY14B101Q2A SPI nvSRAM. Vector CANalyzer monitors the CAN bus, transmits commands and messages, and receives the TimerTick and data read messages.

The TimerTick message is useful because it shows on the PC the last time tick sent at the moment the power was lost. After the power is plugged back in, this timestamp can be compared with that recorded in the last valid address location. As a result, you can assess the time between the last saved event and the power loss.

The test/demo is configured to send sinewaves as the content of the messages transmitted from the PC (messages are transmitted every 25 ms). At the same time, the potentiometer and the buttons on the PSoC 3 DVK are exercised to emulate signals coming from local sensors. The sinewaves are set up so that each byte in each of the three CAN messages being recorded changes in a sinusoidal fashion.

### Hardware Setup

The CY8CKIT-017 CAN/LIN EBK is connected to port A of the CY8CKIT-001 DVK. The nvSRAM is mounted on an SOIC-to-DIP adapter (for example, SchmartBoards part # 204-0004-01) and placed in the prototype area (see Figure 24 and Figure 25).

The PSoC 3 and nvSRAM are powered at 3.3 V, while the CAN transceiver on the CY8CKIT-017 is powered at 5 V. See appendix on page 26 for a setup with CY8CKIT-030.

The jumper settings on the DVK and EBK are as follows:

- CY8CKIT-001 settings:
  - VDD SELECT set to 3.3 V
  - J6 and J7 set to VDD
  - J8 set to VREG
  - J11 (VR_PWR) connected
  - All VDDIO jumpers (J2, J3, J4, and J5) set to VDD
  - LCD POWER (J12) set to ON
- CY8CKIT-017 settings:
  - JP2 jumper in place
  - JP6 jumper between Vdd and V5_0

The pictures below show the hardware setup for test/demo.

Figure 23. Project pin assignment for CY8CKIT-001.

| Alias | Name | Pin | Lock |
|---|---|---|---|
| | MISO | P0[3] | ☑ |
| | MOSI | P0[2] | ☑ |
| | SCLK | P0[1] | ☑ |
| | SS | P0[4] | ☑ |
| | HOLD | P0[0] | ☑ |
| | RX | P3[4] | ☑ |
| | TX | P3[3] | ☑ |
| | TX_en | P3[2] | ☑ |
| | \LCD:LCDPort\[6:0] | P2[6:0] | ☑ |
| | ERR_LED | P3[6] | ☑ |
| | WARN_LED | P5[0] | ☑ |
| | OK_LED | P3[5] | ☑ |
| | NV_LED | P2[7] | ☑ |
| | VR | P1[5] | ☑ |
| | SW1 | P1[6] | ☑ |
| | SW2 | P1[7] | ☑ |

Figure 24. CY8CKIT-001 prototype area connections.

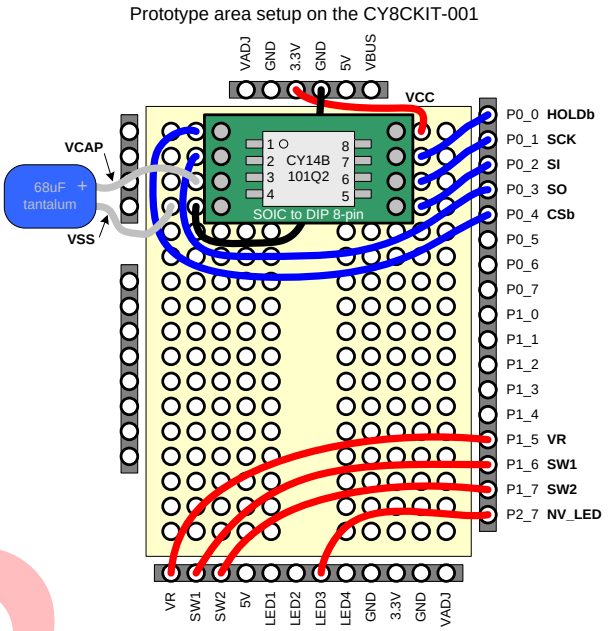
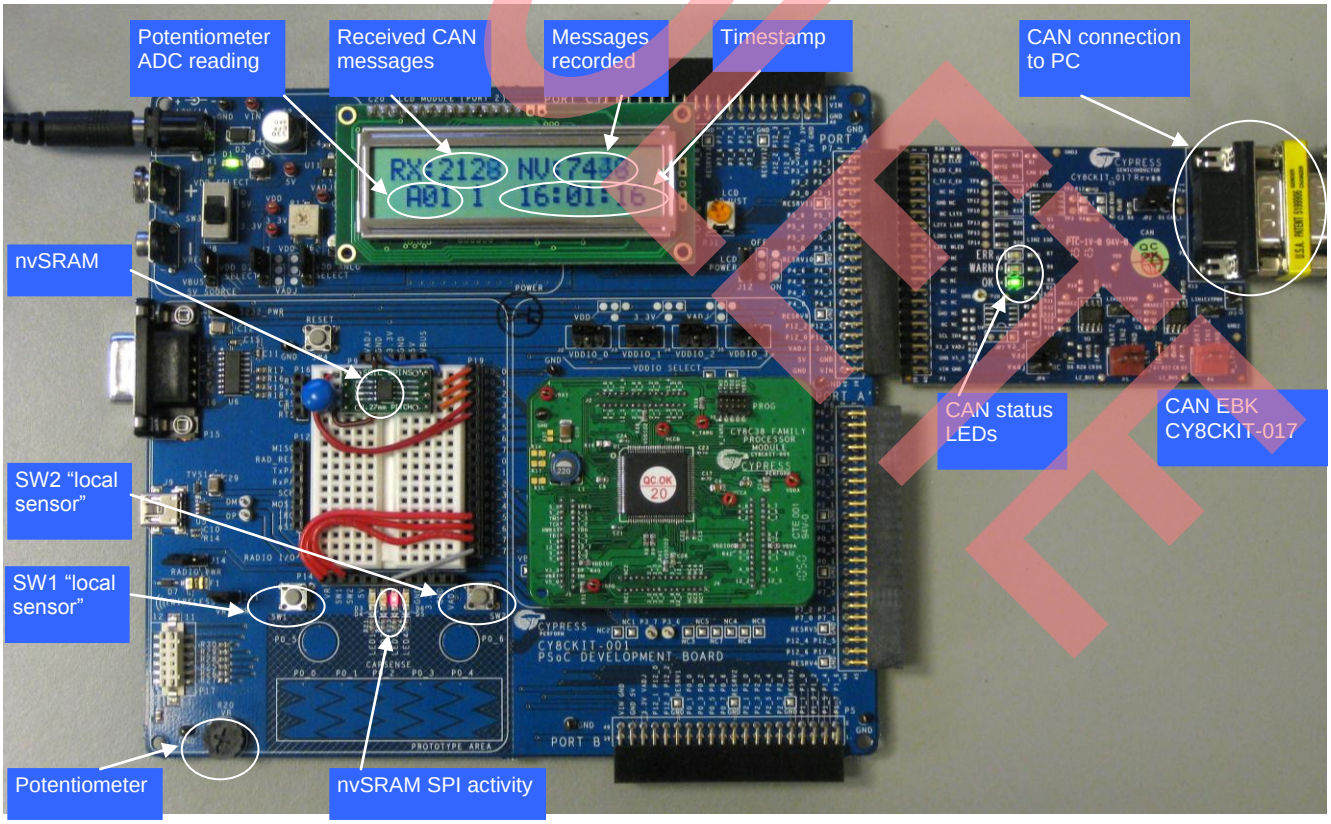



Figure 25. nvSRAM data recorder setup with PSoC 3 DVK CY8CKIT-001, CAN EBK, and nvSRAM.




OBSOLETE

## Data recording

After powering up the data recorder and connecting the Vector CANalyzer, two command messages are sent to start the data recording of CAN messages and of the "local sensors" samples (switches SW1 and SW2, and potentiometer value). The commands to start the data recording are as follows (see also System commands, page 12).

Table 3. Commands sent through CAN to start data recording.

| CAN Message ID | Message (4 bytes) | Command |
|---|---|---|
| 0x7FE | byte0: 0x03<br>byte1: 0x00<br>byte2: 0x00<br>byte3: 0x00 | START_NV_STORE |
| 0x7FE | byte0: 0x35<br>byte1: 0x00<br>byte2: 0x00<br>byte3: 0x00 | START_SNS_STORE |

After data recording begins, the PC sends messages containing sinusoidal values, and the switches and potentiometer are exercised to emulate signals coming from local sensors.

The messages ("sinewaves") sent from the PC (using Vector CANalyzer) are shown in Figure 26.

Each byte for each message ID changes over time to represent a sinewave.

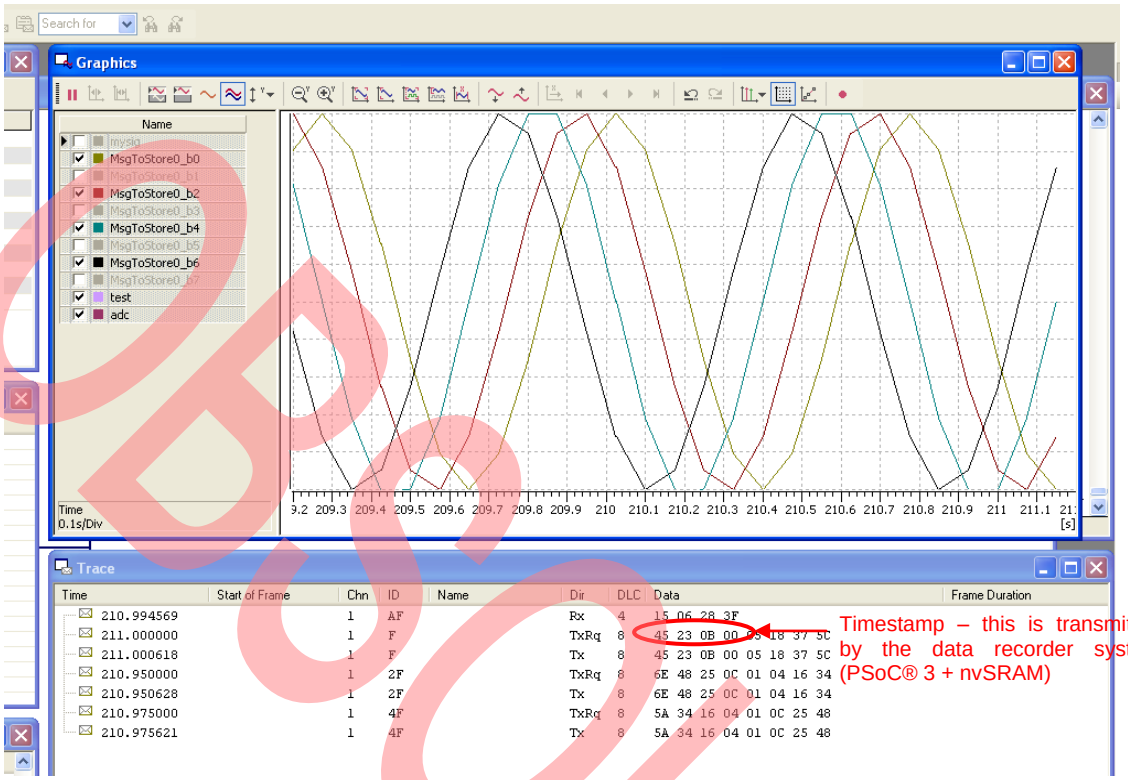
For example, Figure 27 shows a graphic visualization of byte-0, byte-2, byte-4, and byte-6 for message ID 0x00F. Also shown is a trace of the three messages sent to the nvSRAM data recorder every 25 ms.

Figure 26. "Sinewaves" messages sent to the nvSRAM data recorder.




CONFIDENTIAL

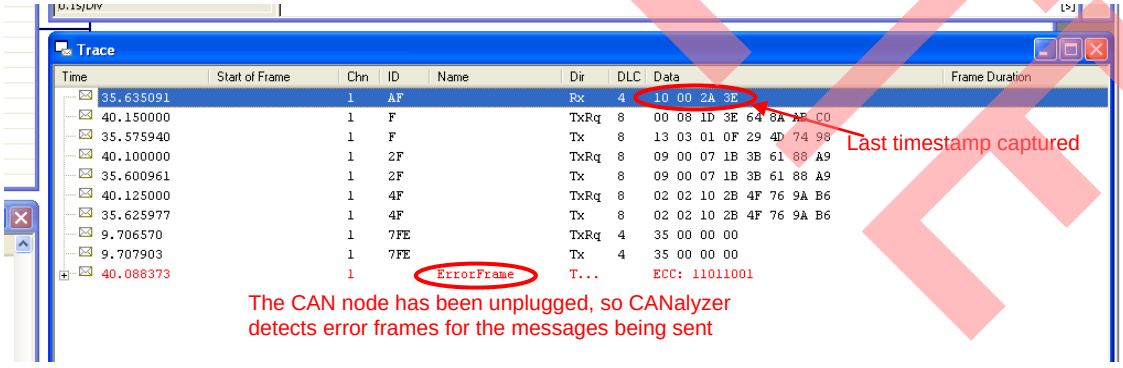
Figure 27. Graphics and Trace for the signals sent to the nvSRAM data recorder.




## Sudden power loss and data retrieval

When power is suddenly removed or lost, as happens during a critical event, the last timestamp sent by the nvSRAM data recorder is maintained in the Trace window.

Figure 28. Trace window showing the last timestamp sent before power loss.




After restoring power and sending the GET_NV_LAST command to the nvSRAM data recorder (message ID: 0x7FE; command: 0x05, 0x00, 0x00, 0x00), the address and timestamp of the last saved data appear on the LCD.

OBSOLETE

Figure 29. First row of the LCD indicates the reading following the command GET_NV_LAST.




Reading the first row of the LCD, from left to right, shows:

- '02': This data comes from ping-pong location 2
- '006AF0': This is the address of the last saved location before the power loss.
- '10002A3E': This is the timestamp recorded at location '006AF0'.

The timestamp recorded in nvSRAM at the last saved address corresponds to the last transmitted timestamp shown in Figure 28. Therefore, in this case, events up to 15.625 ms (at most) prior to the power loss event were recorded. Timestamps are sent every 15.625 ms (1/64th of a second).

You can read address '006AF0' by sending a START_NV_READ command and then the address itself (message ID: 0x7FE; command: 0x06, 0x00, 0x6A, 0xF0). The nvSRAM data recorder then transmits two messages with timestamp, message ID, and message data. This information also is shown on the LCD.

Figure 30. Trace showing the time+ID and data at address 0x006AF0, sent by the data recorder.

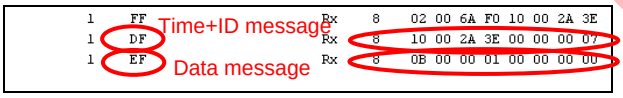



Figure 31. LCD shows the address of the data, a compact view of the timestamp, the ID of the message (7 = sensor data), and the data itself.




The information displayed is as follows:

- '6AF0': This is the memory address (hex format) being read.
- '002A3E': This is the timestamp (each character is a hex value) displaying min, sec, n/64th of a second.
- '7': This is the message ID. In this case, '7' refers to local sensors data (that is, ADC + buttons)
- '0B00': Potentiometer hex value (ADC 12-bit result)
- '00': Button 1 was not pressed at that time
- '01': Button 2 was pressed at that time
- '00000000': The last 4 bytes for message ID '7' are unused.

From that point, the user can access all the other events by sending the NEXT_NV_READ and PREV_NV_READ commands.

## Summary

Event data recorders are becoming increasingly common in automotive applications. With its CAN connectivity and efficient data transfer using DMA, Cypress PSoC 3 offers a compact and effective solution when paired with Cypress nvSRAM. Although this application note and associated example project do not address a full EDR implementation that complies with NHTSA regulations, they do demonstrate a workable EDR solution. Minimal help from the CPU is needed to record multiple CAN messages and locally sensed data. The nvSRAM performance and functionality allow safe storage of the data close in time to the power outage. Moreover, the nvSRAM SPI interface driver is implemented in PSoC 3 to support different data recording scenarios.

## Related Application Notes

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

AN52701 - PSoC® 3 and PSoC® 5 - How to Enable CAN Bus Communication

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

AN64574 - Designing with Serial Peripheral Interface (SPI) nvSRAM

AN43593 - Storage Capacitor (VCAP) Options for Cypress nvSRAM

## About the Author


| | |
|---|---|
| Name: | Daniele Radaelli |
| Title: | Systems Engineer |
| Contact: | dhr@cypress.com |


OBSOLETE

# Appendix: project setup with CY8CKIT-030 PSoC 3 DVK and nvSRAM SOIC-DIP adapter

The project associated with this application note also can be built to operate with the CY8CKIT-030 PSoC 3 development kit.

The CY8CKIT-017 CAN/LIN expansion board is connected to Port E of the CY8CKIT-030 DVK. The nvSRAM (CY1B101Q2) is mounted on an SOIC-to-DIP adapter (for example, SchmartBoard|ez 1.27-mm pitch SOIC-to-DIP adapter part number 204-0004-01 or DigiKey part number 309-1098-ND) and placed in the prototype area (see Figure 33 and Figure 34).

The PSoC 3 and nvSRAM are powered at 3.3 V, while the CAN transceiver on the CY8CKIT-017 is powered at 5 V.

The jumper settings on the DVK and EBK are as follows:

- CY8CKIT-030 settings:
  - VDDA/VDDD SEL (J10 and J11) set to 3.3 V
  - J30 (POT_PWR) connected
- CY8CKIT-017 settings:
  - JP2 jumper in place
  - JP6 jumper between Vdd and V5_0

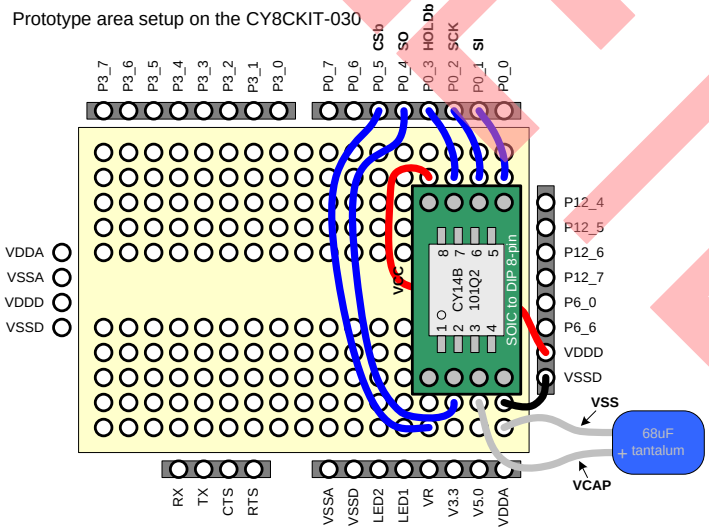
A CAN analyzer (for example, Vector CANalyzer with CANcardXL PCMCIA card) needs to be installed on a PC and connected to the P2 connector on the CAN/LIN Expansion Board (CY8CKIT-017).

The pictures below show the hardware setup used for test/demo.

Figure 32. Project pin assignment for CY8CKIT-030.

| Alias | Name | Pin | Lock |
|---|---|---|---|
| | MISO | P0[4] | ☑ |
| | MOSI | P0[1] | ☑ |
| | SCLK | P0[2] | ☑ |
| | SS | P0[5] | ☑ |
| | HOLD | P0[3] | ☑ |
| | RX | P3[4] | ☑ |
| | TX | P3[3] | ☑ |
| | TX_en | P3[2] | ☑ |
| | \LCD:LCDPort\[6:0] | P2[6:0] | ☑ |
| | ERR_LED | P3[6] | ☑ |
| | WARN_LED | P0[0] | ☑ |
| | OK_LED | P3[5] | ☑ |
| | NV_LED | P6[2] | ☑ |
| | VR | P6[5] | ☑ |
| | SW1 | P6[1] | ☑ |
| | SW2 | P15[5] | ☑ |

Figure 33. CY8CKIT-030 prototype area connections.




Capacitor is 68 µF, 10 V (or higher), Tantalum.

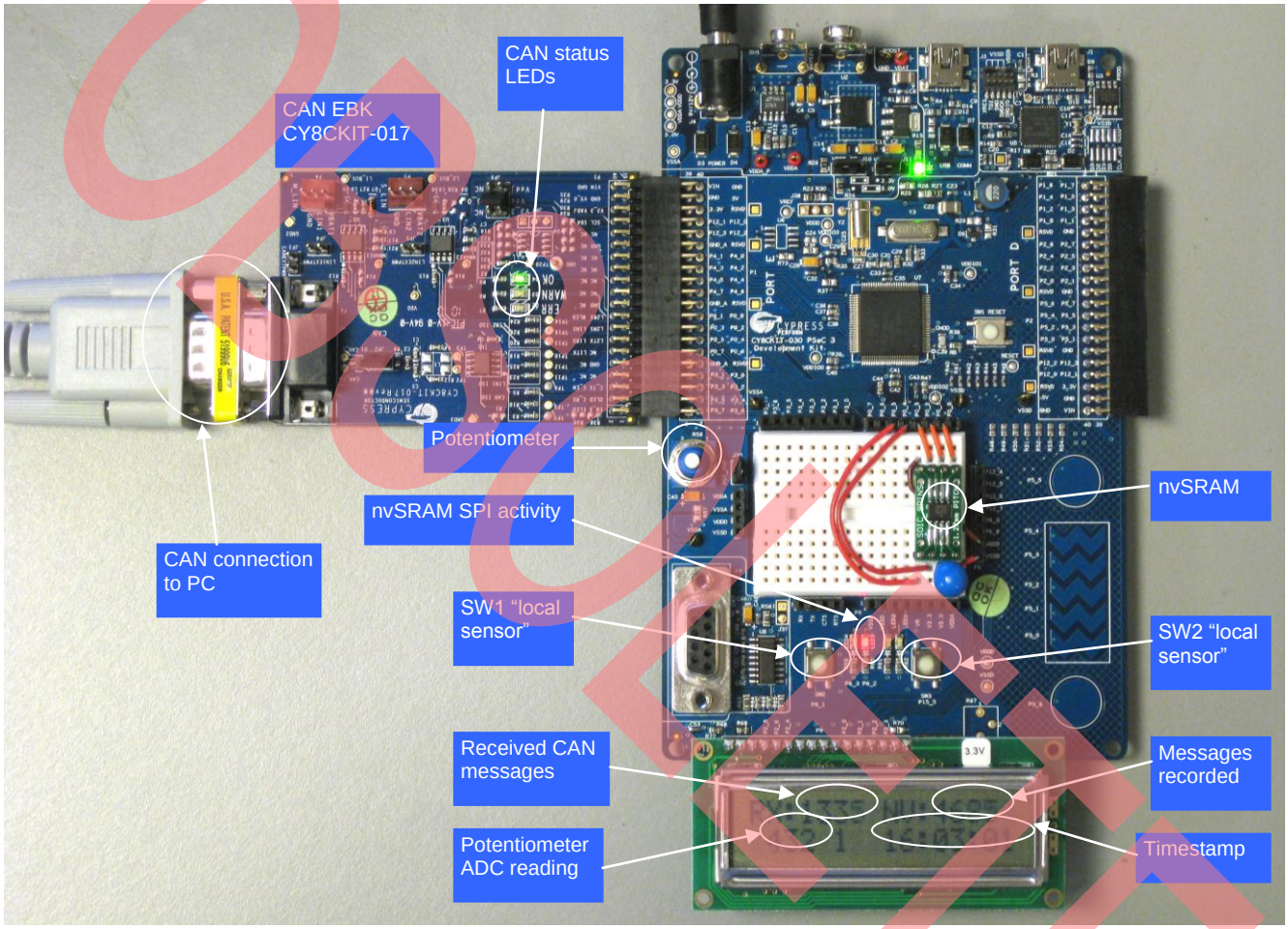
**Note** In the CY8CKIT-030, you do not need to connect the potentiometer, the switches, and the NV_LED

OBSOLETE

(nvSRAM SPI activity indicator, connected to CSb signal inside PSoC 3), because they are already hardwired on the board:

- VR is connected to P6[5]
- SW1 is connected to P6[1]
- SW2 is connected to P15[5]
- NV_LED (LED3) is connected to P6[2]

Figure 34. nvSRAM data recorder setup with PSoC 3 DVK CY8CKIT-030, CAN EBK, and nvSRAM.




Confidential

## Document History

Document Title: Event Data Recorder with Controller Area Network using PSoC® 3 and nvSRAM – AN70630

Document Number: 001-70630

| Revision | ECN | Orig. of Change | Submission Date | Description of Change |
|---|---|---|---|---|
| ** | 3450562 | DHR | 12/1/2011 | New Application Note |
| *A | 3974564 | DHR | 04/19/2013 | Obsolete document. |

OBSOLETE

## Worldwide Sales and Design Support

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

### Products

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

### PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

PSoC is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

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

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

© Cypress Semiconductor Corporation, 2011-2013. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

OBSOLETE