PCI Express Multifunction I/O Board
Instruction Manual

Model 826 | Rev.3.0.10 | July 2017

SENSORAY | embedded electronics

Designed and manufactured in the U.S.A.

SENSORAY | p. 503.684.8005 | email: info@SENSORAY.com | www.SENSORAY.com

7313 SW Tech Center Drive | Portland, OR 97203

Table of Contents

Chapter 1: Preliminary

1.1 Revision history

Rev	Released	Description
3.0.0	10/2012	Initial release.
3.0.1	11/2012	Corrected S826_SafeWrenWrite() and S826_SafeWrenRead(). Added timing specs for DAC serializer.
3.0.2	03/2013	Modified behavior of S826_DioCapRead().
3.0.3	06/2013	Added hold registers; some rephrasing for clarity. FPGA version info now includes major and minor version numbers. Added DIO/counter input filter functions.
3.0.4	07/2013	Fixed typo in S826_AdcRead().
3.0.5	11/2013	Fixed broken link. Corrected explanations of safemode writes for DAC setpoint and range.
3.0.6	12/2015	Clarified use of +5V power out at DIO/counter connectors. Expanded specs for DAC, ADC and DIO. Corrected table in section 5.2, pin 26.
3.0.7	02/2016	Added storage temp to specs. Described heartbeat LED.
3.0.8	08/2016	Expanded explanations of external DIO pull-ups. Added specifications, including max input voltages when board is unpowered. Clarified: ExtIn can trigger snapshots regardless of value in counter's Mode register IM field. Correction: RS-422 receivers do not have on-board termination resistors. Added counter timing diagrams and descriptions.
3.0.9	06/2017	In S826_CounterModeWrite() example: changed S826_CounterModeWrite() to S826_CounterStateWrite(). Expanded and clarified specifications. Added application notes and code examples.
3.0.10	07/2017	Fixed missing cross-references. Clarified thread-safe behavior.

1.2 Handling instructions

The Model 826 circuit board contains electronic circuitry that is sensitive to Electrostatic Discharge (ESD).

Special care should be taken in handling, transporting, and installing circuit board to prevent ESD damage to the board. In particular:

- Do not remove the circuit board from its protective packaging until you are ready to install it.
- Handle the circuit board only at grounded, ESD protected stations.
- Remove power from the equipment before installing or removing the circuit board. In particular, disconnect all field wiring from the board before installing or removing the board from the backplane.

1.3 Limited warranty

Sensoray Company, Incorporated (Sensoray) warrants the Model 826 hardware to be free from defects in material and workmanship and perform to applicable published Sensoray specifications for two years from the date of shipment to purchaser. Sensoray will, at its option, repair or replace equipment that proves to be defective during the warranty period. This warranty includes parts and labor.

The warranty provided herein does not cover equipment subjected to abuse, misuse, accident, alteration, neglect, or unauthorized repair or installation. Sensoray shall have the right of final determination as to the existence and cause of defect.

As for items repaired or replaced under warranty, the warranty shall continue in effect for the remainder of the original warranty period, or for ninety days following date of shipment by Sensoray of the repaired or replaced part, whichever period is longer.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. Sensoray will pay the shipping costs of returning to the owner parts that are covered by warranty. A restocking charge of 25% of the product purchase price will be charged for returning a product to stock.

Sensoray believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, Sensoray reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition.

The reader should consult Sensoray if errors are suspected. In no event shall Sensoray be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, SENSORAY MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF SENSORAY SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. SENSORAY WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

Third party brands, names and trademarks are the property of their respective owners.

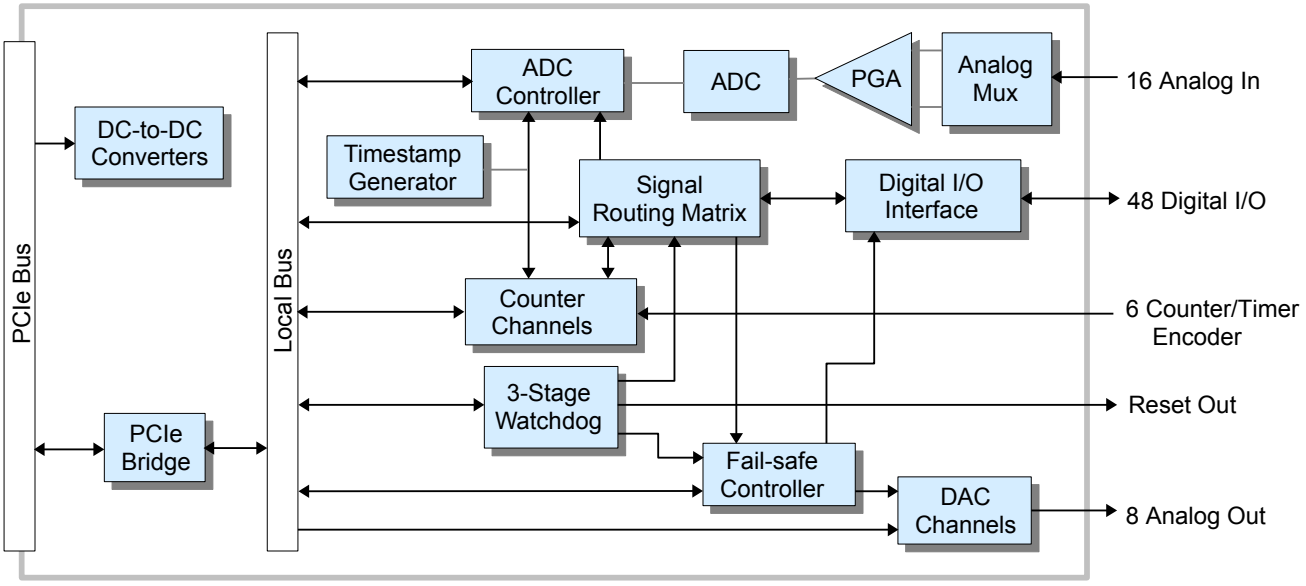
Chapter 2: Introduction

2.1 Overview

Model 826 is a PCI Express board that features an assortment of I/O interfaces commonly used in measurement and control applications:

- **48 bidirectional digital I/O channels** with edge detection and fail-safe output control.
- **Eight 16-bit analog outputs** (±10V, ±5V, 0-10V, 0-5V) with fail-safe output control.
- **Sixteen 16-bit differential analog inputs** (±10V, ±5V, ±2V, ±1V) with hardware and software triggering.
- **Six 32-bit counters**. Clock inputs may be driven from external quadrature-encoded (e.g., incremental encoder) or single-phase sources. Inputs accept differential RS-422 or standard TTL/CMOS single-ended signals. Auxiliary inputs and outputs can be internally routed to digital I/O channels.
- **Multistage watchdog timer** that can activate fail-safe outputs and generate service requests.
- **Fail-safe controller** switches outputs to safe states upon watchdog timeout or external trigger.
- **Signal routing matrix** allows software to interconnect interfaces without external wiring.

Figure 1: Model 826 block diagram



Sensoray provides a free, comprehensive API (application programming interface) to facilitate the rapid development of polled, event-driven, or mixed-mode programs for Model 826. The API works in concert with the board's advanced architecture to support complex, high performance applications.

Standard headers are provided for connecting on-board peripherals to external circuitry. The board's low-profile headers allow it to fit comfortably into high-density systems, and an integral cable clamp keeps cables secure and organized.

On-board LEDs provide visual indications of the board's condition. The PWR indicator is lit when the on-board power supplies are operating. The HB (heartbeat) indicator flashes continuously when the board is operating normally. Six additional LEDs (E0-E5) indicate counter clock activity as explained in Status LEDs.

2.1.1 Timestamp generator

A timestamp generator is shared by the analog input system and counter channels. It is a free-running 32-bit binary counter that increments every microsecond (it does not keep track of the date or time-of-day). The counter start at zero counts upon board reset and overflows every 2^{32} microseconds (approximately 71.6 minutes). The counter is not writable, but it can be read by calling S826_TimestampRead.

The timestamp generator's current time is automatically appended to every counter and ADC sample so that application programs can know (to within 1 µs) when each sample was acquired. It is particularly useful for precisely measuring the time between hardware events, and calculation of elapsed time is simple (a single subtraction) as long as the time doesn't exceed 71.6 minutes. It can be used in a variety of ways, including measuring speed and capturing serial data.

2.1.2 Board reset

Upon power-up or hardware reset, all interfaces and output signals are forced to their default initial states. The initial states of the interfaces are discussed in the interface chapters.

The HB (heartbeat) LED will begin to flash continuously when the reset signal terminates. This indicates normal operation of the board's core logic circuitry but does not reflect the health of any I/O interfaces.

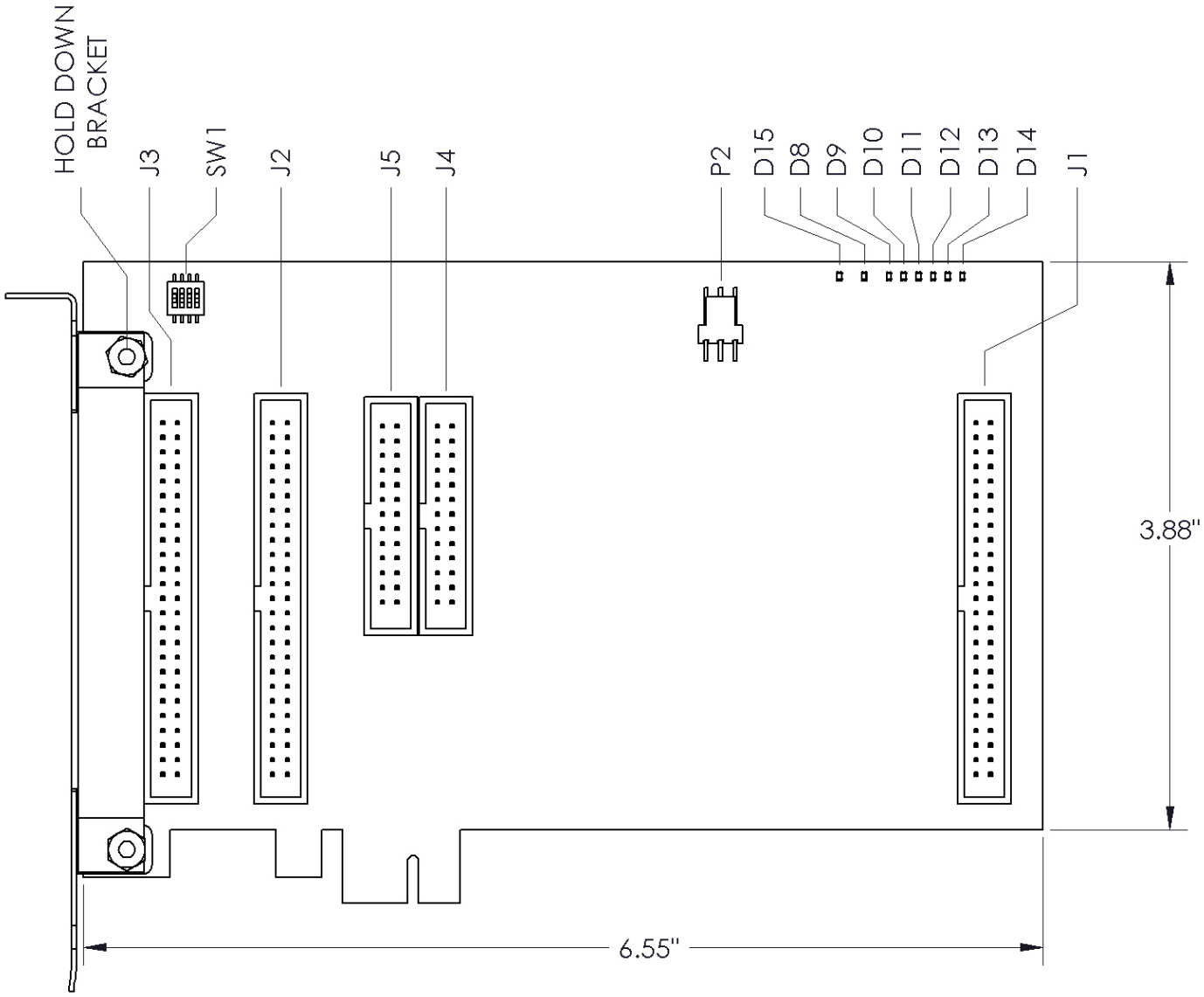
2.2 Hardware configuration

Up to sixteen model 826 boards can coexist in a single host computer. When more than one board is used, each board must be configured before installation. This is done by setting quad dip switch SW1 to assign to the board a unique identifier (board number) in the range 0 to 15.

If more than one 826 board will be plugged into a common backplane, the boards must be configured so that each board has a unique board number. Board numbers are typically contiguous, though this is not required. For example, board numbers 0 and 3 could be assigned in a two-board system, though it would be more common to assign board numbers 0 and 1. This table shows the relationship between board number and switch settings:

Board Number	SW1-4	SW1-3	SW1-2	SW1-1	Notes
0	OFF	OFF	OFF	OFF	Factory default
1	OFF	OFF	OFF	ON	
2	OFF	OFF	ON	OFF	
3	OFF	OFF	ON	ON	
4	OFF	ON	OFF	OFF	
5	OFF	ON	OFF	ON	
6	OFF	ON	ON	OFF	
7	OFF	ON	ON	ON	
8	ON	OFF	OFF	OFF	
9	ON	OFF	OFF	ON	
10	ON	OFF	ON	OFF	
11	ON	OFF	ON	ON	
12	ON	ON	OFF	OFF	
13	ON	ON	OFF	ON	
14	ON	ON	ON	OFF	
15	ON	ON	ON	ON	

2.3 Board layout



2.4 Cable installation

The 826 board should be connected to external circuitry with Sensoray cables, model 826C1 (26 conductor, for counters) and 826C2 (50 conductor, for analog and digital I/O), which are specifically designed for this purpose. These cables feature thin, flat cable and low profile headers that allow the board to fit into high-density systems when loaded with a complete complement of five cables.

To install the cables (see above diagram):

- Loosen and remove the board's cable clamp (part of the hold-down bracket assembly).
- Pass each cable's low-profile end through the hold-down bracket (from left of bracket) and plug it into its connector.
- Install and tighten the cable clamp.

Chapter 3: Programming

A free software development kit (SDK) for Model 826 is available at Sensoray's web site. The SDK includes Linux and Windows device drivers, sample application programs, and a pre-built application program interface (API) for Windows (DLL) and Linux (static library). The API core and Linux source code is available for OEMs who need to port the API and driver to other operating systems.

This chapter provides an overview of the API and discusses functions that are used in all applications. Later chapters discuss API functions that are specific to the board's I/O systems.

3.1 Thread safety

All API functions are guaranteed to be thread and process safe when accessing non-shared hardware resources. For example, two threads may simultaneously call S826_DioOutputWrite to program digital I/O (DIO) output levels. This is guaranteed to be safe if the threads interact with mutually exclusive DIOs. However, if both threads attempt to program the same DIO, that DIO would be a shared resource and the function is not guaranteed to be safe.

3.1.1 Atomic read-modify-write

To facilitate high-performance thread-safe behavior, some API functions include a “mode” argument that specifies how a register write will be performed. The mode argument can be used to access bit-set and bit-clear functions available on various interface control registers.

mode	Operation
0	Write all data bits unconditionally. All register bits are programmed to explicit values.
1	Clear (program to '0') all register bits that have '1' in the corresponding data bit. All other register bits are unaffected.
2	Set (program to '1') all register bits that have '1' in the corresponding data bit. All other register bits are unaffected.

For example, if mode=2 and the data value is 0x00000003, register bits 0 and 1 will be set to '1' and all other register bits will retain their previous values.

3.2 Event-driven applications

Event-driven applications can be implemented by calling the API's blocking functions S826_CounterSnapshotRead, S826_AdcRead, S826_DioCapRead, and S826_WatchdogEventWait, which seamlessly manage hardware interrupts in conjunction with the device driver. Each of these functions can be used to block the calling thread (which allows other threads to do productive work) while it waits for hardware signal events. A blocking function will return immediately (without blocking) if the event occurs before the function is called.

All of the blocking functions include a “tmax” argument that specifies the maximum amount of time to wait for events:

tmax	Blocking function behavior
0	Return immediately even if event has not occurred (never blocks).
1 to 4294967294	Block until event occurs or tmax microseconds elapse, whichever comes first. This corresponds to times ranging from 1 microsecond to approximately 71.6 minutes.
S826_WAIT_INFINITE	Block until event occurs, with no time limit.

Note that non-realtime operating systems (e.g.,Windows) may not respond to events with deterministic timing. The responsiveness of such systems can depend on a number of factors including CPU loading, thread and process priorities, memory speed and capacity, core count and architecture, and clock frequency.

3.3 Error codes

Most of the API functions return an error code. These functions return zero if no errors are detected, otherwise a negative value will be returned that indicates the type of error that occurred.

Error Code	Value	Description
S826_ERR_OK	0	No errors.
S826_ERR_BOARD	-1	Invalid board number.
S826_ERR_VALUE	-2	Illegal argument value.
S826_ERR_NOTREADY	-3	Device was busy or unavailable, or blocking function timed out.
S826_ERR_CANCELLED	-4	Blocking function was canceled.
S826_ERR_DRIVER	-5	Driver call failed.
S826_ERR_MISSEDTRIG	-6	ADC trigger occurred while previous conversion burst was in progress.
S826_ERR_DUPADDR	-9	Two 826 boards are set to the same board number. Change DIP switch settings.
S826_ERR_BOARDCLOSED	-10	Addressed board is not open.
S826_ERR_CREATEMUTEX	-11	Failed to create internal mutex.
S826_ERR_MEMORYMAP	-12	Failed to map board into memory space.
S826_ERR_MALLOC	-13	Failed to allocate memory.
S826_ERR_FIFOOVERFLOW	-15	Counter channel's snapshot FIFO overflowed.
S826_ERR_OSSPECIFIC	-1xx	Error specific to the operating system. Contact Sensoray.

3.4 Open/close functions

3.4.1 S826_SystemOpen

The S826_SystemOpen function detects all Model 826 boards and enables communication with the boards.

```
int S826_SystemOpen(void);
```

Return Values

If the function succeeds, the return value is a set of bit flags indicating all detected 826 boards. Each bit position corresponds to the board number programmed onto a board's switches (see “Hardware configuration“). For example, bit 0 will be set if an 826 with board number 0 is detected. The return value will be zero if no boards are detected, or positive if one or more boards are detected without error.

If the function fails, the return value is an error code (always a negative value). S826_ERR_DUPADDR will be returned if two boards are detected that have the same board number.

Remarks

This function must be called by a process before interacting with any 826 boards. The function allocates system resources that are used internally by other API functions. S826_SystemClose must be called once, for each call to S826_SystemOpen, to free the resources when they are no longer needed (e.g., when the 826 application program terminates). The board's circuitry is not reset by this function; all registers and I/O states are preserved.

S826_SystemOpen should be called once by every process that will use the 826 API. In a multi-threaded process, call the function once before any other API functions are called; all threads belonging to the process will then have access to all 826 boards in the system.

Example

```
// Open the 826 API and list all boards ---------------

int id;  // board ID (determined by switch settings on board)
int flags = S826_SystemOpen();
if (flags < 0)
  printf("S826_SystemOpen returned error code %d", flags);
else if (flags == 0)
  printf("No boards were detected");
else {
  printf("Boards were detected with these IDs:");
  for (id = 0; id < 16; id++) {
    if (flags & (1 << id))
      printf(" %d", id);
  }
}
```

3.4.2 S826_SystemClose

The S826_SystemClose function frees all of the system resources that were allocated by S826_SystemOpen and disables communication with all 826 boards.

```
int S826_SystemClose();
```

Return Values

The return value is always zero.

Remarks

This function should be called when a process (e.g., an application program) has finished interacting with the board's I/O interfaces, to free system resources that are no longer needed. Any API functions that are blocking will return immediately, with return value S826_ERR_SYSCLOSED. The board's circuitry is not reset by this function; all registers and I/O states are preserved.

3.5 Status functions

3.5.1 S826_VersionRead

The S826_VersionRead function returns API, driver, and board version information.

```
int S826_VersionRead(
  uint board,       // board identifier
  uint *api,        // API version
  uint *driver,     // driver version
  uint *bdrev,      // circuit board version
  uint *fpgarev     // FPGA version
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

api
Pointer to a buffer that will receive the API version info. The hexadecimal value is formatted as 0xXXYYZZZZ, where XX = major version, YY = minor version, and ZZZZ = build.

driver
Pointer to a buffer that will receive the driver version info. The hexadecimal info is formatted as 0xXXYYZZZZ, where XX = major version, YY = minor version, and ZZZZ = build.

boardrev
Pointer to a buffer that will receive the version number of the 826's circuit board.

fpgarev
Pointer to a buffer that will receive the board's FPGA version info. The hexadecimal info is formatted as 0xXXYYZZZZ, where XX = major version, YY = minor version, and ZZZZ = build.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Example

```
// Read and display version numbers from board #0 --------------

#define DOTREV(N) ((N) >> 24) & 0xFF, ((N) >> 16) & 0xFF, (N) & 0xFFFF
uint api, drv, bd, fpga;
int errcode = S826_VersionRead(0, &api, &drv, &bd, &fpga);  // Read version info.
if (errcode == S826_ERR_OK) {                               // If no errors then
  printf("API version %d.%d.%d\n", DOTREV(api));            //   display info.
  printf("Driver version %d\n", DOTREV(drv));
  printf("Board version %d\n", bd);
  printf("FPGA version %d\n", DOTREV(fpga));
}
else
  printf(" S826_VersionRead returned error code %d", flags);
```

3.5.2 S826_TimestampRead

The S826_TimestampRead function returns the current value of the timestamp generator.

```
int S826_TimestampRead(
  uint board,       // board identifier
  uint *timestamp   // current timestamp
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

timestamp
Pointer to buffer that will receive the timestamp. The returned value is the contents of the timestamp generator at the moment the function executes.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function can be used to monitor elapsed times with microsecond resolution.

Example

This function will directly read the timestamp generator:

```
// Read the timestamp generator's current time.
uint CurrentTimestamp(uint board)
{
  uint t;
  S826_TimestampRead(board, &t);
  return t;
}
```

The following example shows a simple application of direct timestamp reads:

```
// Example: Use board0 to measure system Sleep() time ---------

uint t1, t0 = CurrentTimestamp(0);  // Get start time.
Sleep(100);                         // Sleep approximately 100 ms.
t1 = CurrentTimestamp(0);           // Get end time.
printf("Slept %d µs", t1 - t0);     // Display actual sleep time.
```

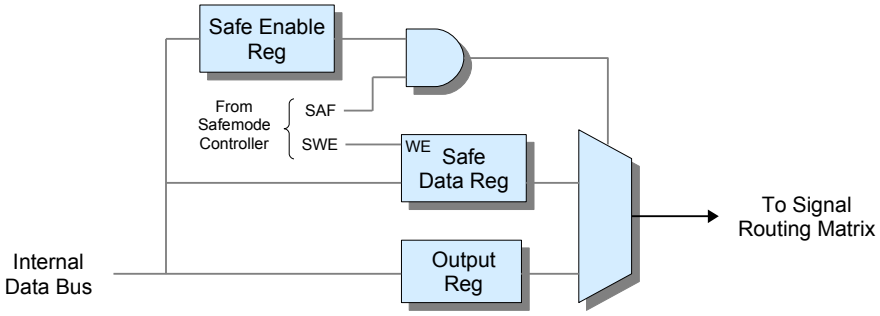
Chapter 4: Virtual outputs

4.1 Introduction

The board has six virtual digital output channels, numbered 0 to 5, that can be used by software to signal various interfaces on the board. The virtual output channels are physical circuits that are architecturally similar to the board's general-purpose digital output channels, but they cannot be routed to the board's headers or connected to external circuitry.

Each virtual output channel is connected to the board's internal signal routing matrix, which in turn routes the channel's output signal to other on-board interfaces under program control. A virtual output channel can be routed to the ExtIn input of one or more counter channels, or to the ADC trigger input, or to combinations of these.

Figure 2: Virtual output channel (1 of 6)



4.1.1 Safemode

Safemode is activated when the SAF signal (see Figure 2) is asserted. When operating in safemode, the virtual output state is determined by the Safe Enable and Safe Data registers: when Safe Enable equals '1' the pin will be driven to the fail-safe value stored in Safe Data; when Safe Enable equals '0' the pin will output the normal runmode signal from its output register.

Upon board reset, the Safe Enable register is set to '1' so that the virtual output will exhibit fail-safe behavior by default (i.e., it will output the contents of the Safe Data register when SAF is asserted). Fail-safe operation can be disabled for a virtual output channel by programming its Safe Enable register to '0'.

The Safe Data register is typically programmed once (or left at its default value) by the application when it starts, though it may be changed at any time if required by the application. It can be written to only when the SWE bit is set to '1'. See “Safemode controller” for more information about safemode.

4.2 API functions

The virtual output API functions employ individual bits to convey information about the virtual output channels, wherein each bit represents the information for one channel. The information is organized into a single quadlet as follows (e.g., “v5” = virtual output channel 5):

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIO	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	v5	v4	v3	v2	v1	v0

4.2.1 S826_VirtualWrite

The S826_VirtualWrite function programs the virtual output registers.

```
int S826_VirtualWrite(
  uint board,     // board identifier
  uint data,      // data to write
  uint mode       // 0=write, 1=clear bits, 2=set bits
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Normal (runmode) output data for the six virtual channels (see Section 4.2).

mode
Write mode for data: 0 = write, 1 = clear bits, 2 = set bits (see “Atomic read-modify-write”).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

In mode zero, this function will unconditionally write new values to all virtual output registers. In modes one and two, the function will selectively clear or set any arbitrary combination of output registers; data bits that contain logic '1' indicate virtual output registers that are to be modified, while all other output registers will remain unchanged. Modes one and two can be used to ensure thread-safe operation as they atomically set or clear the specified bits.

4.2.2 S826_VirtualRead

The S826_VirtualRead function reads the programmed states of all virtual output registers.

```
int S826_VirtualRead(
  uint board,    // board identifier
  uint *data     // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to a buffer (see Section 4.2) that will receive the output register states.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function returns the output register states.

4.2.3 S826_VirtualSafeWrite

The S826_VirtualSafeWrite function programs the virtual output channel Safe registers.

```
int S826_VirtualSafeWrite(
  uint board,     // board identifier
  uint data,      // safemode data
  uint mode       // 0=write, 1=clear bits, 2=set bits
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to data array (see Section 4.2) to be programmed into the Safe registers.

mode
Write mode for data: 0 = write, 1 = clear bits, 2 = set bits (see "Atomic read-modify-write").

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

4.2.4 S826_VirtualSafeRead

The S826_VirtualSafeRead function returns the contents of the virtual output channel Safe registers.

```
int S826_VirtualSafeRead(
  uint board,    // board identifier
  uint *data     // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to a buffer (see Section 4.2) that will receive the Safe register states.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

4.2.5 S826_VirtualSafeEnablesWrite

The S826_VirtualSafeEnablesWrite function programs the virtual output channel Safe Enable registers.

```
int S826_VirtualSafeEnablesWrite(
  uint board,       // board identifier
  uint enables      // safemode enables
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enables
Pointer to array of values (see Section 4.2) to be programmed into the Safe Enable registers.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

4.2.6 S826_VirtualSafeEnablesRead

The S826_VirtualSafeEnablesRead function returns the contents of the virtual output channel Safe Enable registers.

```
int S826_VirtualSafeEnablesRead(
  uint board,      // board identifier
  uint *enables    // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enables
Pointer to a buffer (see Section 4.2) that will receive the Safe Enable register contents.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

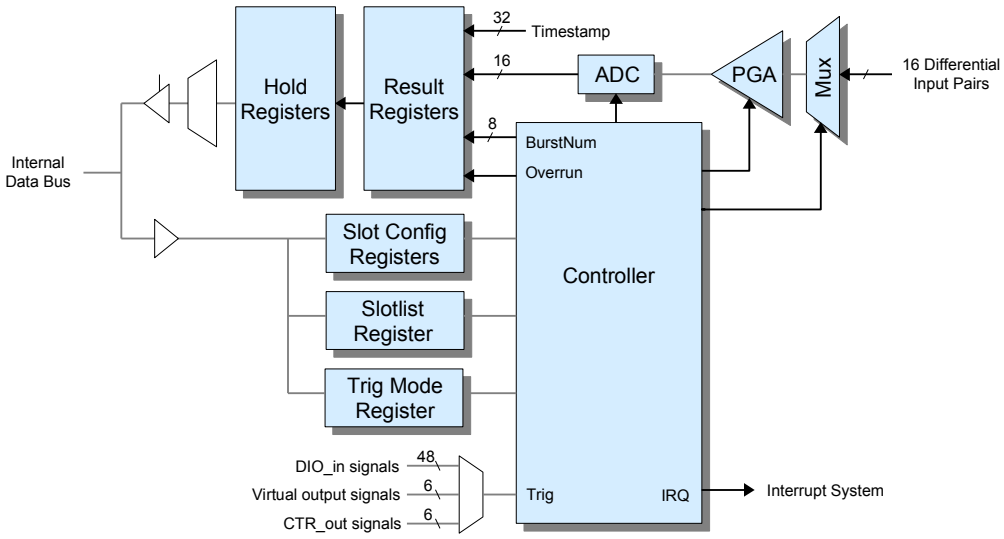
Chapter 5: Analog inputs

5.1 Introduction

The board's analog input system consists of the following major elements:

- **Analog multiplexer (Mux)** - selects one of 16 differential input pairs (channels) for conversion.
- **Programmable gain amplifier (PGA)** - applies voltage gain of 1, 2, 5, or 10 to the selected input.
- **Analog-to-digital converter (ADC)** - performs an analog-to-digital conversion in three microseconds or less.
- **Controller** - manages operation of the analog input system.

Figure 3: Analog input system



Analog-to-digital (A/D) conversions are performed in bursts of up to 16 conversions. Each conversion takes place in a timeslot (called a *slot*), which is a reserved time interval within a burst. During a burst, slots are processed in numerical order beginning with slot 0 and ending with slot 15.

Every slot has three attributes that are programmed into its slot configuration register via the S826_AdcSlotConfigWrite function:

- **Analog channel number:** designates the analog input channel that will be digitized when the slot is processed. Any of the 16 channels may be assigned to any slot. A channel may be assigned to two or more slots if desired.
- **PGA gain:** specifies the analog gain to apply to the analog input signal.
- **Settling time:** specifies the delay from analog multiplexer switching to start of digitization.

The slotlist register designates each slot as either active or inactive; this is programmed via the S826_AdcSlotlistWrite function. When a burst occurs, each slot is processed according to the slotlist. An active slot is processed by performing one A/D conversion; its total time is the sum of its settling time plus a fixed conversion time. Inactive slots are skipped and consume no time during a burst (the slot configuration register is ignored for an inactive slot). The total processing time of each slot is configurable, from zero (inactive slot) to approximately 335 milliseconds, in one-microsecond increments.

ADC conversions are disabled upon board reset. Typically, an application program will configure the ADC system (by programming slotlist and slot configuration registers) before enabling conversions, though this is not required. The slotlist and slot configuration registers may be reprogrammed at any time, even when conversions are enabled.

5.1.1 Triggering

The trigger mode register determines which of two triggering modes, *triggered* or *continuous*, will be used to initiate conversion bursts. In triggered mode each burst must be initiated by a trigger signal, whereas in continuous mode the trigger signal is ignored and conversion bursts will automatically execute one after another with no idle time between bursts. The trigger mode and trigger signal source are programmed by calling S826_AdcTrigModeWrite.

When operating in triggered mode, S826_AdcTrigModeWrite selects one of the following signals to serve as the trigger:

- Any of the 48 general purpose digital I/O (DIO) channels. This enables an external digital signal to trigger bursts. When a DIO channel is used to trigger A/D conversions, the timing of the trigger signal is constrained by the DIO subsystem. See DIO “Pin timing” for more information.
- Any of the six counter ExtOut signals. This enables a counter to periodically trigger ADC bursts. The selected counter output is internally routed to the ADC trigger input; no external wiring is required.
- Any of the six virtual digital outputs. This enables software to trigger ADC bursts by writing to a virtual output. See Virtual outputs for more information. The selected virtual output is internally routed to the ADC trigger input; no external wiring is required.

5.1.2 Burst counter

The controller maintains an 8-bit burst counter that indicates the number of conversion bursts since the ADC was enabled. The burst counter is zeroed upon board reset and whenever the ADC is disabled. The counter increments at the end of each burst and overflows to zero when incrementing from 255. The burst count is passed to the host along with every ADC sample so that the program can determine which burst a sample belongs to.

In triggered mode, the burst count can be used to keep track of the number of received triggers. If a trigger occurs while a burst is in progress, a MissedTrigger flag is set and the trigger will be ignored. After this happens, the burst count will no longer accurately indicate the number of received triggers (accuracy can be restored by disabling and then re-enabling ADC conversions).

5.1.3 Result registers

Each slot has a result register that stores the slot's most recently acquired result, which is a set of four values: ADC output data, timestamp (which indicates the time the result was acquired), burst count, and overrun flag. Each slot also has a hold register that caches a result while it is being read by the host computer. The hold register ensures that the result's four component values will remain correlated if a new result is captured while the previous result is being read.

During a burst, the controller processes each slot by performing a sequence of operations. First it switches the analog multiplexer to the desired differential input pair and programs the gain and then, if the slot has a non-zero settling time, it waits for the settling time to elapse. When the settling time has elapsed, the controller starts an analog-to-digital conversion. At the moment the conversion completes, the four component values that comprise the result are simultaneously sampled and copied to the result register.

A new result will always overwrite the previous result, even if the previous result has not been read. If the previous result has not yet been read when a new result is written, the overrun flag will be set to '1'; otherwise it will be set to '0'.

Sixteen hardware status flags (one per slot) indicate unread results. A status flag is set when the controller writes a new result to the associated result register, and reset when the program reads the result. Results are read by calling the S826_AdcRead function. The S826_AdcStatusRead function can be called to check the status flags without returning results or altering status flags.

5.2 Connector J1

All analog input and output signals are available at connector J1.

J1 Pinout

Pin	Name	Function
1	GND	Power supply common
2	GND	Power supply common
3	-AIN0	Analog input (-) channel 0
4	+AIN0	Analog input (+) channel 0
5	-AIN1	Analog input (-) channel 1
6	+AIN1	Analog input (+) channel 1
7	-AIN2	Analog input (-) channel 2
8	+AIN2	Analog input (+) channel 2
9	-AIN3	Analog input (-) channel 3
10	+AIN3	Analog input (+) channel 3
11	-AIN4	Analog input (-) channel 4
12	+AIN4	Analog input (+) channel 4
13	-AIN5	Analog input (-) channel 5
14	+AIN5	Analog input (+) channel 5
15	-AIN6	Analog input (-) channel 6
16	+AIN6	Analog input (+) channel 6
17	-AIN7	Analog input (-) channel 7
18	+AIN7	Analog input (+) channel 7
19	GND	Power supply common
20	GND	Power supply common
21	-AIN8	Analog input (-) channel 8
22	+AIN8	Analog input (+) channel 8
23	-AIN9	Analog input (-) channel 9
24	+AIN9	Analog input (+) channel 9
25	-AIN10	Analog input (-) channel 10

Pin	Name	Function
26	+AIN10	Analog input (+) channel 10
27	-AIN11	Analog input (-) channel 11
28	+AIN11	Analog input (+) channel 11
29	-AIN12	Analog input (-) channel 12
30	+AIN12	Analog input (+) channel 12
31	-AIN13	Analog input (-) channel 13
32	+AIN13	Analog input (+) channel 13
33	-AIN14	Analog input (-) channel 14
34	+AIN14	Analog input (+) channel 14
35	-AIN15	Analog input (-) channel 15
36	+AIN15	Analog input (+) channel 15
37	GND	Power supply common
38	GND	Power supply common
39	GND	Power supply common
40	GND	Power supply common
41	AOUT4	Analog output channel 4
42	AOUT0	Analog output channel 0
43	AOUT5	Analog output channel 5
44	AOUT1	Analog output channel 1
45	AOUT6	Analog output channel 6
46	AOUT2	Analog output channel 2
47	AOUT7	Analog output channel 7
48	AOUT3	Analog output channel 3
49	GND	Power supply common
50	GND	Power supply common

5.3 API functions

5.3.1 S826_AdcSlotConfigWrite

The S826_AdcSlotConfigWrite function configures a timeslot.

```
int S826_AdcSlotConfigWrite(
  uint board,    // board identifier
  uint slot,     // timeslot number
  uint chan,     // analog input channel number
  uint tsettle,  // settling time in microseconds
  uint range     // input range code
);
```

Parameters

board

826 board number. This must match the settings of the board's dip switches as described in section 2.2.

slot

Timeslot number in the range 0 to 15.

chan
Analog input channel number in the range 0 to 15.

tsettle
Microseconds to allow the analog input to settle before conversion, in the range 0 to 335,544.

range
Enumerated value that specifies the analog input voltage range. This determines the analog gain that will be applied to the input signal before digitization.

range	PGA Gain	Analog Input Range	Notes
0	x1	-10V to +10V	Default upon reset
1	x2	-5V to +5V	
2	x5	-2V to +2V	
3	x10	-1V to +1V	

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

The settings established by this function will be used the next time the slot is converted and remain in effect until changed by another call to this function or a board reset.

The maximum total time for each slot is *tsettle*+3 µs. Sufficient settling time must be allowed when different channels are being digitized so that each input signal has time to stabilize before digitization. If a single channel is being digitized in multiple, consecutive slots, only the first of its slots must allow for settling time; subsequent slots may have zero settling time because the channel will already be settled. Similarly, if only one channel is being digitized (even if it is being digitized multiple times in different slots), all settling times may be set to zero because no channel switching will occur.

The ADC data latency is greater than the slot time because an additional 1µs is needed to fetch the digitized data after each conversion. The data latency is only incurred once per burst, because the ADC controller always fetches data from the previous conversion while the next conversion is in progress. Consequently, the elapsed time from hardware trigger to burst completion (in microseconds) is

$$t_{burst} \leq 1 + 3 \cdot NumTimeslots + \sum t_{settle}$$

5.3.2 S826_AdcSlotConfigRead

The S826_AdcSlotConfigRead function returns the configuration of a timeslot.

```
int S826_AdcSlotConfigRead(
  uint board,     // board identifier
  uint slot,      // timeslot number
  uint *chan,     // analog input channel number
  uint *tsettle,  // settling time in microseconds
  uint *range     // input range code
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

slot
Timeslot number in the range 0 to 15.

chan
Buffer that will receive the analog input channel number.

tsettle
Buffer that will receive the analog settling time in microseconds.

range
Buffer that will receive the analog input voltage range code, as specified in S826_AdcSlotConfigWrite.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function returns the settings established by a board reset or previous call to S826_AdcSlotConfigWrite.

5.3.3 S826_AdcSlotlistWrite

The S826_AdcSlotlistWrite function programs the conversion slot list.

```
int S826_AdcSlotlistWrite(
  uint board,      // board identifier
  uint slotlist,   // conversion slot list
  uint mode        // write mode
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

slotlist
List of slots to be digitized during subsequent conversion bursts, one bit per slot. Bits 0 to 15 correspond to slots 0 to 15, respectively. Each bit that is set to logic '1' will cause the corresponding slot to be digitized, while '0' will cause the slot to be skipped.

mode
Write mode for slotlist: 0 = write, 1 = clear bits, 2 = set bits (see “Atomic read-modify-write”).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

The settings established by this function will be used the next time a slot is digitized and remain in effect until changed by another call to this function or a board reset.

In mode zero, this function will unconditionally write a new slotlist. In modes one and two, the function will selectively set or clear any arbitrary combination of slots; slotlist bits that contain logic '1' indicate slots that are to be modified, while all other slots will remain unchanged. Modes one and two can be used to guarantee thread-safe operation as they atomically enable or disable the specified slots without affecting any other slots.

5.3.4 S826_AdcSlotlistRead

The S826_AdcSlotlistRead function returns the conversion slot list.

```
int S826_AdcSlotlistRead(
  uint board,       // board identifier
  uint *slotlist    // conversion slot list
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

slotlist
Pointer to a buffer that will receive the slotlist. In the received value, bits 0 to 15 correspond to slots 0 to 15. For each bit: '1' = active (slot will be digitized during bursts), '0' = inactive (slot will be skipped during bursts).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function returns the current slotlist, which was previously programmed by S826_AdcSlotlistWrite or cleared by a board reset.

5.3.5 S826_AdcTrigModeWrite

The S826_AdcTrigModeWrite function programs the ADC triggering mode.

```
int S826_AdcTrigModeWrite(
  uint board,      // board identifier
  uint trigmode    // hardware trigger mode
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

trigmode
Hardware trigger configuration:

Bit	Function	Description
31-8	Reserved	Set all bits to 0.
7	Trigger enable	Set to 1 to enable hardware triggering (triggered mode), or 0 to disable hardware triggering (continuous mode).
6	Trigger polarity	1 selects rising edge; 0 selects falling edge. This is ignored if hardware triggering is disabled.
5-0	Trigger source	Hardware trigger signal source (ignored if hardware triggering is disabled): 0-47 = DIO channel 0-47. 48-53 = ExtOut signal from counter channel 0-5. 54-59 = Virtual digital output channel 0-5.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

The settings established by this function take effect upon the next conversion burst and remain in effect until changed by another call to this function or a board reset. Hardware triggering is disabled upon board reset.

5.3.6 S826_AdcTrigModeRead

The S826_AdcTrigModeRead function reads the ADC triggering mode.

```
int S826_AdcTrigModeRead(
  uint board,      // board identifier
  uint *trigmode   // hardware trigger mode
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

trigmode
Pointer to a buffer that will receive the hardware trigger configuration. See S826_AdcTrigModeWrite for the format of this value.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

5.3.7 S826_AdcEnableWrite

The S826_AdcEnableWrite function enables or disables ADC conversions.

```
int S826_AdcEnableWrite(
  uint board,    // board identifier
  uint enable    // enable conversions when true
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enable
Set to 1 to enable, or 0 to disable ADC conversion bursts. Conversion bursts are disabled by default upon board reset.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

When enable is false, any conversion burst that is currently in progress will be terminated and all results and overrun status flags are reset. When enable is true, the resulting behavior depends on the trigger mode:

Continuous mode
The first conversion burst will begin immediately, followed by additional bursts. Each subsequent burst begins immediately after the preceding burst ends. Back-to-back bursts will continue until ADC conversions are disabled.

Triggered mode
A single conversion burst will begin in response to the next trigger and conversions will cease upon completion of that burst. Each subsequent trigger will cause another single burst. Triggers have no effect when ADC conversions are disabled.

5.3.8 S826_AdcEnableRead

The S826_AdcEnableRead function returns the enable status of the ADC system.

```
int S826_AdcEnableRead(
  uint board,    // board identifier
  uint *enable   // enable status
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enable
Buffer that will receive the ADC system enable status: 1 = enabled, 0 = disabled.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function returns the ADC system's enable status that was previously configured by a board reset or a call to S826_AdcEnableWrite.

5.3.9 S826_AdcStatusRead

The S826_AdcStatusRead function reads the ADC conversion status.

```
int S826_AdcStatusRead(
  uint board,       // board identifier
  uint *status      // conversion status
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

status
Pointer to a buffer that will receive the conversion status for all slots. Each bit corresponds to one slot; bits 0 to 15 correspond to slots 0 to 15. A bit will be set to '1' if new (unread) data is available in the slot's result register, or '0' when the result register is empty (or has been read).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function returns status information without altering the state of the ADC system.

5.3.10 S826_AdcRead

The S826_AdcRead function fetches ADC data from one or more timeslots.

```
int S826_AdcRead(
  uint board,       // board identifier
  int buf[16],      // pointer to adc result buffer
  uint tstamp[16],  // pointer to timestamps buffer
  uint *slotlist,   // pointer to slotlist
  uint tmax         // maximum time to wait
);
```

Parameters

board

826 board number. This must match the settings of the board's dip switches as described in section 2.2.

buf

Pointer to a buffer that will receive ADC results for the sixteen possible slots. The buffer must be large enough to accommodate sixteen values regardless of the number of active slots. Each slot is represented by a 32-bit value, which is stored at buf[SlotNumber]:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BSTNUM								V	0000000							ADCVAL															

Field	Description
BSTNUM	Burst number. This indicates the ADC burst number corresponding to the ADC data. It can be used to time-correlate the ADC data if V = '1'. The burst number is incremented at the end of each burst; it restarts at zero at the end of burst number 255.
V	Data Overwritten flag. When set to '1' this indicates the previous ADC result was overwritten by a new result before it was read.
ADCVAL	ADC data, expressed as 16-bit signed integer:

Analog Voltage	ADCVAL
-10V to +10V	0x8000 to 0x7FFF
-5V to +5V	0x8000 to 0x7FFF
-2V to +2V	0x8000 to 0x7FFF
-1V to +1V	0x8000 to 0x7FFF

tstamp

Pointer to a buffer that will receive the timestamps corresponding to ADC results. The buffer must be large enough to accommodate sixteen values regardless of the number of active slots. Each timestamp indicates the moment in time that its corresponding ADCVAL was acquired. For any given slot, the slot's timestamp will be stored in tstamp[SlotNumber]. The application may set this to NULL if timestamps are not needed.

slotlist

Pointer to a buffer containing bit flags, one bit per slot, that indicate slots of interest. Bits 0-15 correspond to slots 0-15. Before calling the function, set to '1' all bits that correspond to slots of interest. When the function returns, the buffer contents will have been changed so that '1' indicates a slot has new data in buf and '0' indicates no new data.

tmax

Maximum time, in microseconds, to wait for ADC data. See “Event-driven applications” for details.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function reads the result registers of an arbitrary set of slots and copies the results to buf. As many as sixteen results (one per slot) may be copied to buf each time the function is called. Over-range and under-range conditions are indicated by maximum or minimum data values for the selected input range, respectively; there are no special status flags that indicate these conditions.

Before calling the function, one or more slotlist bits must be set to indicate the slots of interest. Only new, unread results are copied to buf. If a slot is not of interest or has not been converted since it was last read, its buf value will not change. In all cases (even when the function returns an error), when the function returns, slotlist will indicate the slots of interest that have new buf values.

The function will operate in either blocking or non-blocking mode depending on the value of tmax. If tmax is zero, the function will return immediately. If tmax is greater than zero, the calling thread will block until all requested data is available or tmax elapses. In either case, the function will copy to buf all data from slots of interest that have a new result. The function will return S826_ERR_NOTREADY if any of the requested slot data is unavailable.

This function will return S826_ERR_CANCELLED if, while it is blocking, another thread calls S826_AdcWaitCancel to cancel waits on any of the slots of interest. It will return immediately if the wait criteria is completely satisfied due to the wait cancellations, otherwise it will continue to block and return when all remaining wait criteria is satisfied. S826_ERR_BOARDCLOSED will be returned immediately if S826_SystemClose executes while this function is blocking. In either case, no result data will be copied to buf.

In triggered mode, the board's internal MissedTrigger error flag will be set if a trigger occurs while an ADC burst is in progress. When the MissedTrigger flag is active, S826_AdcRead will wait for the requested slot data to arrive and then it will clear the MissedTrigger flag and return S826_ERR_MISSEDTRIG. If multiple threads are waiting in S826_AdcRead, only the first thread to return will receive S826_ERR_MISSEDTRIG; the other waiting threads will not receive this error.

Thread-safe operation is guaranteed only if the slots of interest for any given thread do not coincide with those of another thread. For example, thread safety would be assured if one thread designates slots 1 and 3-5 as slots of interest while another thread designates slots 2 and 9. Thread safety would be compromised, though, if both threads shared a common slot of interest.

5.3.11 S826_AdcWaitCancel

The S826_AdcWaitCancel function cancels a blocking wait on one or more slots.

```
int S826_AdcWaitCancel(
  uint board,      // board identifier
  uint slotlist    // slots to cancel
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

slotlist
Set of bit flags, one bit per slot, that indicate slots for which waiting is to be canceled. Bits 0-15 correspond to slots 0-15.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function cancels blocking for an arbitrary set of slots so that another thread, which is blocked by S826_AdcRead while waiting for adc data to arrive, will return immediately with S826_ERR_CANCELLED.

5.4 Application notes

5.4.1 Handling ADC interrupts

An interrupt request (IRQ) is generated when the ADC completes a conversion burst. These IRQs are managed by the blocking function S826_AdcRead, which configures ADC interrupts and handles the resulting IRQs as required. To wait for the next IRQ simply call S826_AdcRead; the function will return when ADC data is available, and will block and allow other threads to run while ADC data is unavailable.

The following example shows how this works. In this example, only slot 0 is being used, and timestamps are not used. Note that the ADC (and its trigger source) must have been previously configured and enabled (see next example).

```
void AdcHandler(void)
{
  int errcode;
  int slotval[16];  // buffer must be sized for 16 slots
  while (1) {
    uint slotlist = 1;  // only slot 0 is of interest in this example
    errcode = S826_AdcRead(0, adcdata, NULL, &slotlist, S826_WAIT_INFINITE); // wait for IRQ
    if (errcode != S826_ERR_OK)
      break;
    printf("Raw adc data = %d", slotval[0] & 0xFFFF);
  }
}
```

5.4.2 Periodic ADC conversions (self-paced)

The ADC can be used to periodically acquire samples without using hardware triggering (e.g., from a counter or external signal). To do this, configure the ADC for continuous triggering mode and use the slot settling times to control the sampling rate. Note that the sampling period will have a small amount of jitter (≤ 1 µs) because ADC conversion time is 2-3 µs.

This example shows how to acquire approximately 20 samples per second from analog input 0:

```
#define SAMPLING_PERIOD 50000         // Sampling period in microseconds (50000 = 20 samples/s).
#define TSETTLE SAMPLING_PERIOD - 3;  // Compensate for nominal ADC conversion time.

// Configure the ADC subsystem and start it running
S826_AdcSlotConfigWrite(board, 0, 0, TSETTLE, S826_ADC_GAIN_1); // measuring ain 0 on slot 0
S826_AdcSlotlistWrite(board, 1, S826_BITWRITE);                 // enable slot 0
S826_AdcTrigModeWrite(board, 0);                                // trigger mode = continuous
S826_AdcEnableWrite(board, 1);                                  // start adc conversions

while (1) {
  AdcHandler();   // Handle periodic ADC interrupts (using code from earlier example)
}
```

5.4.3 Periodic ADC conversions (triggered)

A counter can be used to periodically trigger ADC bursts. To set this up, configure the counter as a periodic timer that outputs a pulse at the end of each period. The output pulse need not be routed to a DIO pin because it can be internally routed to the ADC's trigger input. Also, it's not necessary to generate counter snapshots because the ADC will notify software when a conversion has completed.

The following code shows how to digitize all 16 analog inputs in a burst, at a rate of 100 bursts per second (10,000 µs/burst). The bursts are triggered by counter0, which must be configured to output pulses at 100 Hz.

```
// Use counter0 to periodically trigger ADC conversions -------------

void StartAdc16(uint period)  // Configure/start ADC and trigger generator
{
  int i;
  for (i = 0; i < 16; i++)  // Configure all timeslots: 1 slot per AIN; 20us/slot settling time.
    S826_AdcSlotConfigWrite(0, i, i, 20, S826_ADC_GAIN_1);
  S826_AdcSlotlistWrite(0, 0xFFFF, S826_BITWRITE); // Enable all 16 timeslots.
  S826_AdcTrigModeWrite(0, 0xB0);                  // Hardware triggered, source = counter0 ExtOut.
  S826_AdcEnableWrite(0, 1);                       // Enable ADC conversions.
  CreateHwTimer(0, 0, period);                     // Create and start the trigger generator.
}

int ReadAdc16(int *adcbuf)
{
  uint slotlist = 0xFFFF;     // Wait for ADC burst completion.
  return S826_AdcRead(0, adcbuf, NULL, &slotlist, S826_WAIT_INFINITE);
}
```

In the following example, all 16 AINs are digitized and processed ten times per second:

```
int adcbuf[16];       // sample buffer -- always set size=16 (even if fewer samples needed)
StartAdc16(100000);   // Configure adc; start adc and trigger generator.
while (ReadAdc16(adcbuf) == S826_ERR_OK) {    // Repeat forever:

  // TODO: PROCESS ADC SAMPLES IN ADCBUF[]

}
```

5.4.4 Oversampling

Oversampling is a useful technique for reducing noise and increasing resolution. Model 826 facilitates oversampling by allowing an analog input (AIN) to be measured multiple times in a single ADC burst. After the burst, the samples can be averaged to obtain an enhanced sample value.

The following code shows how to oversample two AINs. In each burst, AIN0 is digitized eight times and then AIN1 is digitized eight times. To set this up, each AIN is assigned to eight contiguous slots. For each AIN, the first slot includes a settling time because the ADC input has just switched. The seven subsequent slots do not require settling time (and thus settling times are set to 0 µs) because the input has not switched.

The ADC's internal timing is used to pace conversions, by selecting the continuous triggering mode and assigning appropriate settling times. Slot 0 is assigned a long settling time, which effectively determines the sampling period and also provides a necessary settling delay. Slot 8 has a shorter settling time because it only needs to delay long enough for signal settling.

Note: Error checking has been omitted for clarity, but should always be included in robust application code.

```
// Settling times in microseconds:
#define TSETTLE0 10000  // Delay after switching to AIN0 (tweak this to adjust sampling rate).
#define TSETTLE1 20     // Delay after switching ADC to AIN1.

// Average 8 samples and convert to volts (assumes +/-10V measurement range)
#define VOLTS(SUM)  ((SUM) * 10.0 / (8.0 * 32768.0))

int slot;
int adcbuf[16];  // Sample buffer -- always set size=16 (even if fewer samples needed)
```

```
// Adc timeslot attributes.
struct SLOTATTR {
  uint chan;      // analog input channel
  uint tsettle;   // settling time in microseconds
} attr[16] = {                                          // During each burst:
  {0, TSETTLE0},                                        //  switch to AIN0, delay, then digitize
  {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0},  //  digitize AIN0 7x without delay
  {1, TSETTLE1},                                        //  switch to AIN1, delay, then digitize
  {1, 0}, {1, 0}, {1, 0}, {1, 0}, {1, 0}, {1, 0}, {1, 0}}; //  digitize AIN1 7x without delay

// Configure all 16 timeslots.
for (slot = 0; slot < 16; slot++)
  S826_AdcSlotConfigWrite(board, slot, attr[slot].chan, attr[slot].tsettle, S826_ADC_GAIN_1);

// Configure adc system and start it running.
S826_AdcSlotlistWrite(board, 0xFFFF, S826_BITWRITE);  // Enable all 16 timeslots.
S826_AdcTrigModeWrite(board, 0);                      // Select continuous triggering mode.
S826_AdcEnableWrite(board, 1);                        // Start conversions.

while (1)    // Repeatedly fetch and display oversampled data:
{
  int sum[] = {0, 0};                                 // Reset sample accumulators (one per AIN).
  uint slotlist = 0xFFFF;                             // Wait for ADC burst
  S826_AdcRead(board, adcbuf, NULL, &slotlist, 1000); //   and read samples from 16 slots.

  for (slot = 0; slot < 16; slot++)                   // Sum the 8 samples from each AIN
    sum[slot >> 3] += (short)(adcbuf[slot] & 0xFFFF); //   while masking off sample meta-data.

  // Compute and report measured voltages.
  printf("AIN0=%3.3fV, AIN1=%3.3fV\n", VOLTS(sum[0]), VOLTS(sum[1]));
}
```

5.4.5 Polled operation

In event-driven applications, the S826_AdcRead function is allowed to block until ADC samples are available. However, blocking necessarily involves context switching, which introduces overhead. To avoid this overhead, blocking must be disabled. This is done by setting argument tmax=0, which allows the ADC subsystem to be polled without blocking.

The following code employs polling to acquire data from all 16 AINs. Each call to S826_AdcRead will return immediately, with slotlist bits indicating the AINs that have new samples waiting in adcbuf. If desired, individual samples may be processed when they arrive in adcbuf (each time S826_AdcRead executes), or the program can wait for all 16 AINs and then process them en masse.

In this example a 7 µs settling delay is allowed before each conversion begins. The ADC conversion time is ≤3 µs, so the total time per AIN is ≤10 µs and the burst time for all AINs is ≤160 µs. Note that individual settling times may need adjustment depending on factors such as source impedance and required accuracy.

```
#define TSETTLE    7        // Settling delay after switching AIN (adjust as necessary).
#define SLOTFLAGS  0xFFFF   // Timeslot flags: use all 16 timeslots.

int i;
int adcbuf[16];  // Sample buffer -- always set size=16 (even if fewer samples needed)

// Configure all timeslots: 1 slot per AIN; constant settling time for all slots.
for (i = 0; i < 16; i++)
  S826_AdcSlotConfigWrite(board, i, i, TSETTLE, S826_ADC_GAIN_1);
```

```
// Configure adc system and start it running.
S826_AdcSlotlistWrite(board, SLOTFLAGS, S826_BITWRITE); // Enable all 16 timeslots.
S826_AdcTrigModeWrite(board, 0);                        // Select free-running mode.
S826_AdcEnableWrite(board, 1);                          // Start conversions.

while (1)    // Repeat forever:
{
  uint remaining = SLOTFLAGS;      // timeslots that have not yet been read
  do {
    uint slotlist = remaining;     // Read all available remaining timeslots.
    int errcode = S826_AdcRead(board, adcbuf, NULL, &slotlist, 0); // note: tmax=0
    remaining &= ~slotlist;

    // OPTIONAL: NEWLY ARRIVED SAMPLES MAY BE PROCESSED WHILE WAITING FOR REMAINING SAMPLES

  } while (errcode == S826_ERR_NOTREADY);

  if (errcode != S826_ERR_OK)
    break; // error

  // TODO: PROCESS ALL UNPROCESSED SAMPLES IN ADCBUF[]
}
```

Chapter 6: Analog outputs

6.1 Introduction

The 826 board has eight 16-bit, single-ended digital-to-analog converter (DAC) channels. Each channel can be independently configured to generate output voltages across one of four output voltage ranges: 0 to +5V (default upon reset), 0 to +10V, -5 to +5V, and -10 to +10V.

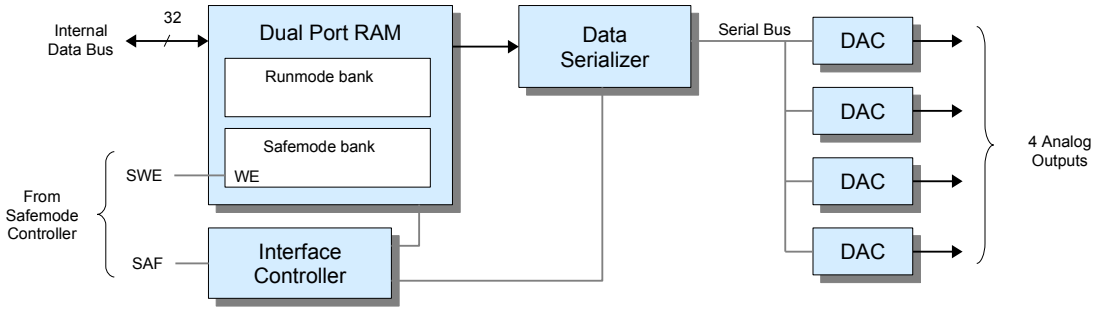
Each DAC channel has a setpoint and configuration register. A channel's output voltage is programmed by writing to its setpoint register. Before programming the setpoint, the DAC must be configured for the desired output range by writing to its configuration register.

Setpoint and configuration values are serially transmitted to the DAC devices over a high speed serial bus (Figure 4). The interface includes a dual-port RAM that allows the host to write setpoint and configuration values to the board while the serial bus is busy. If multiple untransmitted values are pending in the RAM, the controller will employ round-robin arbitration to ensure all pending values are transmitted in a fair and timely fashion. The data serializer requires 1.04 µs to transmit each setpoint or configuration value.

The board has two identical DAC interfaces as shown in Figure 4. Each interface manages a group of four DAC channels. One interface manages channels 0 to 3 and the other interface manages channels 4 to 7. The two interfaces operate concurrently, thus making it possible to simultaneously write to two DACs that reside in different four-channel groups. For example, DAC channels 0 and 4 can be written to simultaneously.

The host may write setpoint and configuration values to the board at any time. If a new value is written to the RAM for a particular channel before that channel's previously written value has been transmitted, the previous RAM value will be overwritten and only the new value will be transmitted. Consequently, the host is allowed to write setpoint data at a rate that exceeds the serial bus bandwidth, though doing so will result in dropped samples.

Figure 4: Analog output interface (1 of 2)



6.1.1 Safemode

The dual-port RAM has two memory banks, one for normal operation ("runmode") and another for "safemode" operation. The runmode bank stores the setpoint and configuration values that are used during normal operation; these values may be changed at any time as required by the application. The safemode bank contains alternate fail-safe settings that are typically programmed once during program initialization (or left at their default settings). Setpoint and configuration values can be written to the runmode bank at any time, but the safemode bank can only be written when SWE = '1'

The safemode signal (SAF) determines which RAM bank is being used by the interface controller ('1' = safemode, '0' = runmode). When SAF changes state, the appropriate RAM bank is selected and all of the DAC channels are reprogrammed to the settings stored in that bank. See "Safemode controller" for more information about the fail-safe system.

6.1.2 Reset state

Upon reset, all DAC outputs and all channels in both RAM banks are programmed to 0V with the output range set to 0 to +5V.

6.2 Connector J1

DAC output signals are available on the analog I/O connector J1, which is shared with the ADC system. See Section 5.2 for the connector pinout.

Each DAC channel has a single-ended output signal that is referenced to the board's power supply common. The output and common signals are available on the analog I/O connector. DAC output signals are sensed at the connector; no external sense inputs are available on the connector.

6.3 API functions

6.3.1 S826_DacRangeWrite

The S826_DacRangeWrite function programs the voltage range of an analog output channel.

```
int S826_DacRangeWrite(
  uint board,    // board identifier
  uint chan,     // channel number
  uint range,    // output range
  uint safemode  // RAM bank select
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
DAC channel number in the range 0 to 7.

range
Enumerated value that specifies the output voltage range:

range	Analog Output Range	Notes
0	0 to +5V	Default upon reset
1	0 to +10V	
2	-5 to +5V	
3	-10 to +10V	

safemode
Specifies the RAM bank to be written: '0' = runmode, '1' = safemode.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function configures a channel's output voltage range and programs the setpoint to zero volts. It can be called at any time, though it is typically called once per channel during program initialization for the runmode bank and, if necessary, once per channel for the safemode bank as well (if default safemode values are not suitable).

The SWE bit must be set (see S826_SafeWrenWrite) to allow writing to the safemode bank. This function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1) when writing to the safemode bank.

6.3.2 S826_DacDataWrite

The S826_DacDataWrite function programs the output voltage of a DAC channel.

```
int S826_DacDataWrite(
  uint board,     // board identifier
  uint chan,      // channel number
  uint setpoint,  // output level
  uint safemode   // RAM bank select
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
DAC channel number in the range 0 to 7.

setpoint
Analog output level. This is a value ranging from 0x0000 to 0xFFFF. The resulting output voltage depends on the channel's previously programmed output range.

Analog output range	setpoint range
0 to +5V	0x0000 to 0xFFFF
0 to +10V	0x0000 to 0xFFFF
-5V to +5V	0x0000 to 0xFFFF
-10V to +10V	0x0000 to 0xFFFF

safemode
Specifies the RAM bank to be written: '0' = runmode, '1' = safemode.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function programs a channel's runmode or safemode output voltage level. If the output range will also be changed, the range should be changed first, before calling this function. This function is frequently used to change a runmode setpoint whenever an output level change is required. It can also be used to change a safemode setpoint; this is typically done once per channel when the application starts, after programming the channel's safemode output range.

The SWE bit must be set (see S826_SafeWrenWrite) to allow writing to the safemode bank. This function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1) when writing to the safemode bank.

6.3.3 S826_DacRead

The S826_DacRead function returns the output range and setpoint of an analog output channel.

```
int S826_DacRead(
  uint board,     // board identifier
  uint chan,      // channel number
  uint *range,    // output range
  uint *setpoint, // output level
  uint safemode   // RAM bank select
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
DAC channel number in the range 0 to 7.

range
Pointer to a buffer that will receive the output range code as described in Section 5.3.1.

setpoint
Pointer to a buffer that will receive the output level as described in Section 5.3.2.

safemode
Specifies the RAM bank to be written: '0' = runmode, '1' = safemode.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function can be used to determine whether previously written configuration data has been sent to the DAC device. The value S826_ERR_NOTREADY will be returned if the range or setpoint is not yet active (i.e., one or both values have not yet been transmitted to the DAC device).

6.4 Application notes

6.4.1 Specifying output in Volts

The following function will set a DAC output to the specified voltage (note: the function does not check for illegal voltage values).

```
int SetDacOutput(uint board, uint chan, uint range, double volts)
{
  uint setpoint;
  switch (range) {  // conversion is based on dac output range:
    case S826_DAC_SPAN_0_5:   setpoint = (uint)(volts * 0xFFFF /  5); break;          // 0 to +5V
    case S826_DAC_SPAN_0_10:  setpoint = (uint)(volts * 0xFFFF / 10); break;          // 0 to +10V
    case S826_DAC_SPAN_5_5:   setpoint = (uint)(volts * 0xFFFF / 10) + 0x8000; break; // -5V to +5V
    case S826_DAC_SPAN_10_10: setpoint = (uint)(volts * 0xFFFF / 20) + 0x8000; break; // -10V to
+10V
  }
  return S826_DacDataWrite(board, chan, setpoint, 0);  // program DAC output
}
```

If the DAC output range is known, it can be explicitly specified like this:

```
// Program board0, dac0 output to -7.35 V
int errcode = SetDacOutput(0, 0, S826_DAC_SPAN_10_10, -7.35);
```

If the DAC output range is not known, call S826_DacRead first to determine the range:

```
// Program board0, dac0 output to +0.573 V
uint range, setpoint;
int errcode = S826_DacRead(0, 0, &range, &setpoint, 0);  // get range
if (errcode == S826_ERR_OK)
  errcode = SetDacOutput(0, 0, range, 0.573);
```

6.4.2 Setpoint readback

The following function will return a DAC's programmed output voltage. Note that the returned volts may not exactly match the value specified in the last SetDacOutput call because the output voltage can only be programmed to one of 64K discrete values.

```
// Read analog output voltage ------------------------

int GetDacOutput(uint board, uint chan, double *volts)
{
  uint raw;
  uint range;
  int errcode = S826_DacRead(board, chan, &range, &raw, 0);  // Get DAC output range & setpoint.
  switch (range) {                                           // Convert raw dacval to volts:
    case S826_DAC_SPAN_0_5:   *volts = setpoint * (5.0 / 0xFFFF); break;             // 0 to +5V
    case S826_DAC_SPAN_0_10:  *volts = setpoint * (10.0 / 0xFFFF); break;            // 0 to +10V
    case S826_DAC_SPAN_5_5:   *volts = (setpoint - 0x8000) * (10.0 / 0xFFFF); break; // -5V to +5V
    case S826_DAC_SPAN_10_10: *volts = (setpoint - 0x8000) * (20.0 / 0xFFFF); break; // -10V to +10V
  }
  return errcode;
}
```

```
// Example usage: Read board0, dac0 programmed output voltage.
double volts;
if (GetDacOutput(0, 0, &volts) == S826_ERR_OK)
  printf("dac0 output is set to %f volts", volts);
else
  printf("error reading dac0");
```

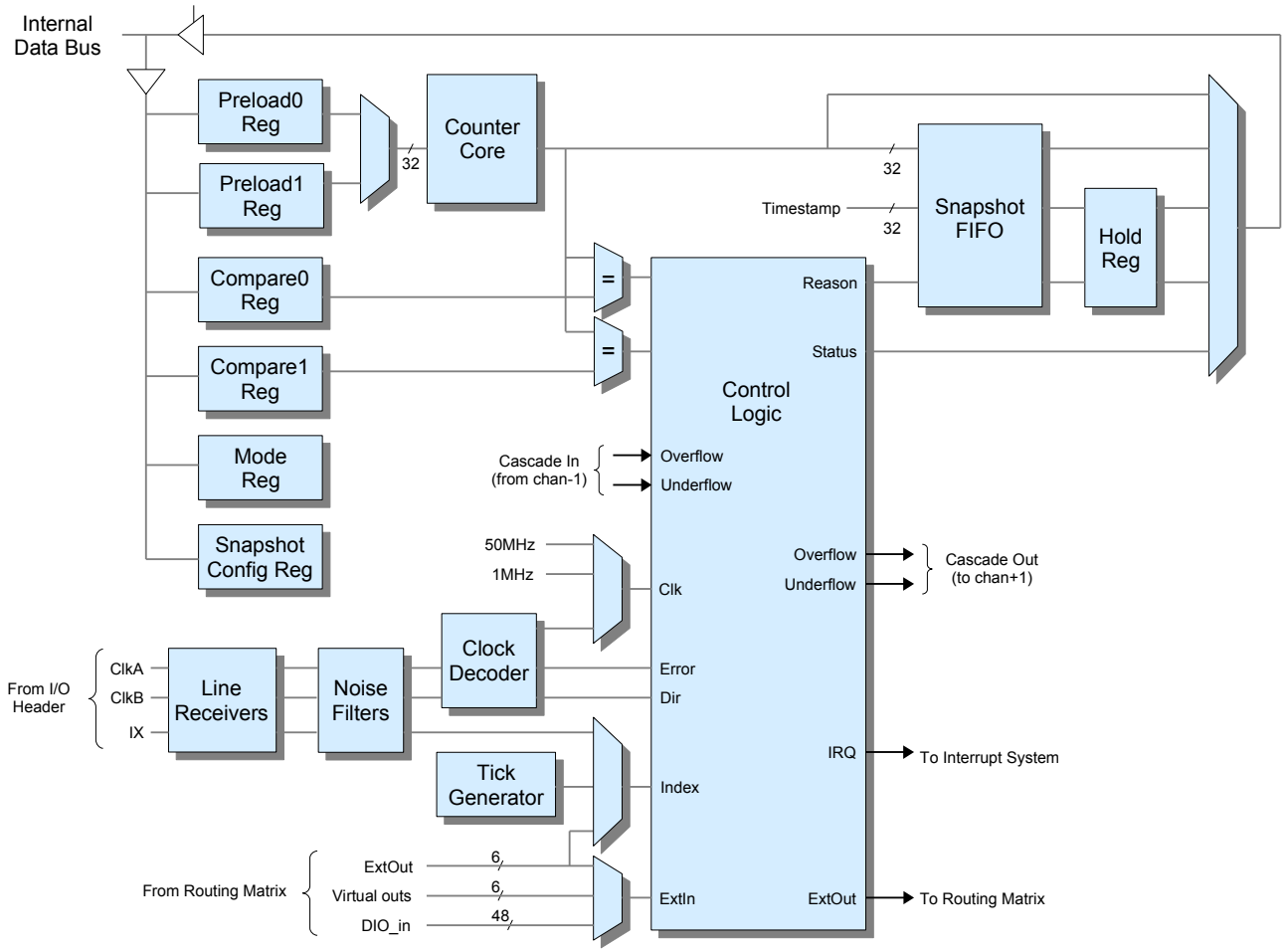
Chapter 7: Counters

7.1 Introduction

The model 826 board has six identical 32-bit programmable counter channels, numbered 0 to 5. Each counter channel can implement a complete solution for a variety of common applications, including incremental encoder interface, event counter, timer, pulse generator, PWM generator, pulse width measurement, period measurement, and frequency measurement. See “Application connections” for tips on connecting external signals to counter channels, and “Application notes” for programming strategies.

As shown in Figure 5, each channel can connect to as many as five external signals. Three input signals (ClkA, ClkB, and IX) are accessible through dedicated header pins. Two additional signals (input ExtIn and output ExtOut) can be routed to general-purpose digital I/O pins if physical access to these signals is needed.

Figure 5: Counter channel (1 of 6)



7.1.1 ClkA, ClkB and IX signals

A channel will accept either either single-phase or quadrature-encoded clock signals on its external ClkA and ClkB inputs. The maximum count rate is 25MHz regardless of external clock type. Single-phase clocks having exactly 50 percent duty rate can be counted at up to 25MHz; the maximum count frequency must be derated for other duty rates (see Specifications for details). Quadrature clocks up to 25 MHz (x1 multiplier), 12.5 MHz (x2), and 6.25 MHz (x4) are supported, with suitable derating for deviations from 90 degree clock phasing.

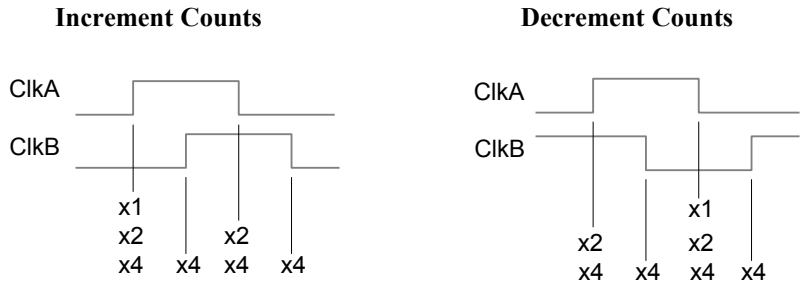
The ClkA, ClkB, and IX inputs employ differential RS-422 line receivers for buffering and noise immunity. The line receivers employ hysteresis and bias to guarantee valid output logic levels under abnormal input conditions (e.g., the output is '1' when inputs are shorted or open). The ClkA/B and IX inputs are compatible with differential RS-422 signal pairs as well as single-ended TTL and 5V CMOS signals.

Termination resistors are not provided for the RS-422 line receivers. External resistors must be supplied if termination resistors are required.

7.1.2 Quadrature clock decoder

When connected to quadrature-encoded clock signals, the ClkA and ClkB signals are processed by a quadrature decoder circuit to produce count direction, count enable, and quadrature error signals. Figure 6 shows the clock events that affect counting for each of the possible clock multipliers.

Figure 6: Quadrature Decoding



For example, with a x1 (“times 1”) multiplier, the counts will only change when ClkB is low; the core will count up on the rising edge of ClkA and down on the falling edge of ClkA.

If the decoder detects an encoding error, it will generate a special error snapshot (see Snapshots below) and set an internal error flag. This will happen when ClkA and ClkB transition within 20 ns of each other (which should never occur in normal operation). This can be caused by various conditions, including signal noise on ClkA/ClkB or excessively high input clock frequency. The error snapshot serves as a warning that the counter may no longer contain an accurate counts value.

7.1.3 ExtIn signal

Some applications may require an additional external input (ExtIn) signal. If needed, this signal can be routed from any general-purpose digital I/O (DIO) channel, any virtual digital output channel, or the ExtOut signal of any counter channel (see Section 7.3.12 for details). When ExtIn is routed from a DIO channel, the signal appearing on the DIO channel's connector pin must conform to the timing constraints of the DIO subsystem as explained in “Pin timing”.

The ExtIn signal can be used to trigger snapshots. Also, the IM field in the Mode register can configure ExtIn to function as a count enable, preload enable, or neither. See S826_CounterModeWrite for details.

7.1.4 ExtOut signal

Some counter applications (e.g., pulse or PWM generator) require a physical output from the counter. In such cases, the counter's ExtOut signal must be routed to a general-purpose DIO channel (see Section 8.3.12 for details). When so routed, the DIO channel will act as a dedicated counter output, while the DIO input and edge detection functions continue to operate normally.

A counter's ExtOut signal will be asserted only when the channel is running. When the channel is halted, ExtOut is held at the inactive state, as defined by its programmed polarity.

When routed to a DIO, the ExtOut signal timing is constrained by the DIO subsystem as explained in “Pin timing”. In particular, the ExtOut signal may be delayed up to 20 ns before it appears on the DIO connector pin.

It may be necessary to connect an external pull-up resistor to the associated DIO when outputting high frequencies or fast pulses. This is due to the DIO's rising-edge slew rate, which is limited by an internal pull-up resistor. Without an external pull-up, very short off-times may not be possible because there is not enough time for the signal to slew to a high logic level. See “Pin timing” for more information.

7.1.5 Snapshots

A “snapshot” consists of three values that are simultaneously sampled in response to a trigger: the counts contained in the counter core, the timestamp (which indicates the time the snapshot was captured), and a set of “reason” flags that indicate the type of event (or types of events, if multiple events occurred at the same time) that triggered the snapshot.

Snapshots are stored in the Snapshot FIFO. Snapshots are written to the FIFO as they occur, whereas the host may read them from the FIFO at any convenient time. The FIFO stores up to 16 snapshots. When the FIFO is full, a subsequent trigger will cause a new snapshot to be stored in the FIFO and the oldest snapshot in the FIFO will be deleted.

The Hold register caches a snapshot while it is being read by the host. The snapshot timestamp and reason code are copied to the Hold register when the snapshot counts are read. The host will then read the cached timestamp and reason code from the Hold register, thus ensuring the three snapshot values will remain correlated if a FIFO overflow occurs while the snapshot is being read.

A snapshot may be captured in response to various types of hardware and software triggers. All of the hardware trigger types can be individually enabled through the snapshot configuration register. Snapshots may be triggered when:

- The S826_CounterSnapshot function is called (i.e., a “soft” snapshot).
- The counter reaches the value stored in a compare register. The snapshot counts will always equal the value stored in the compare register. The snapshot occurs when the counts transition to the compare register value.
- Transitions occur on the Index input.
- Transitions occur on the ExtIn input.
- The counter transitions to zero counts. The snapshot counts will always equal zero.
- A quadrature clock encoding error is detected. Once this happens, no further error-triggered snapshots can be captured until the error is cleared. The resulting error snapshot enables the application to determine the counts at the moment the error occurred. See S826_CounterSnapshotRead for further information.

Multiple counters can use the same snapshot trigger source or sources. For example, two or more counters could be configured to capture snapshots in response to a signal from a common DIO channel, thus enabling an external signal to simultaneously trigger snapshots on all affected counters. Similarly, a virtual digital output channel could be used as a common trigger, thereby allowing software to simultaneously trigger snapshots on the affected counters.

7.1.6 Preloading

The counter core can be “preloaded” (parallel-loaded) from the Preload0 and Preload1 registers in response to various hardware and software preload triggers. All of the hardware triggers can be individually enabled through the mode register. Preloads may be triggered:

- When the S826_CounterPreload function is called (i.e., a “soft” preload).
- When the channel state switches from halted to running.
- When the counter reaches zero counts.
- When the counter matches a compare register.
- When transitions occur on the Index input.

- While the Index input is held at its active level. This has the effect of holding the counter core at the preload value while Index is asserted.

The mode register's BP bit specifies whether both preload registers will be used or only Preload0. Preload0 is active (selected) by default when the channel state switches from halted to running. When a preload occurs, the core is first preloaded from the preload register and then the active register selector will change.

The preload mechanism behaves as shown in the following table. For example, if both preload registers are being used (BP=1) and a preload occurs because the counts reached zero, the core will be preloaded from the active preload register and then the other preload register will be activated.

Preload Behavior

Mode Register BP bit	Preload Trigger Type	Counter Core Loads From	Activated After Preload
0	Any	Preload0	Preload0
1	Zero Counts Reached	Active preload	Alternate preload
	Any except Zero Counts Reached	Preload0	Preload1

Preload triggers are prioritized. If two simultaneous preload triggers occur, the one with the highest priority is considered to be the cause of the preload and the preload mechanism will behave accordingly.

Some types of preload triggers are enabled and disabled by ExtIn when ExtIn is configured as a preload permissive (see “ExtIn signal”). These triggers are disabled when ExtIn is negated and enabled with ExtIn is asserted.

Preload Trigger Type	Priority	Enabled/Disabled by ExtIn (when ExtIn is configured as preload permissive)
Channel switched to running state	7 (highest)	No
Zero counts reached	6	No
Compare1 match	5	Yes
Compare0 match	4	Yes
Index rising edge	3	Yes
Index falling edge	2	Yes
Index active level	1	Yes
Soft preload	0 (lowest)	Yes

7.1.7 Tick generator

Each counter has access to a shared tick generator that produces periodic pulses at intervals ranging from one microsecond to ten seconds in decade steps. The pulses can be internally routed to a counter's Index input to implement periodic sampling applications such as frequency and speed measurement.

7.1.8 Cascading

Limited cascading of counters is supported. Each channel receives overflow and underflow signals from the adjacent lower channel (channel 0 receives from channel 5). A channel may be cascaded onto the adjacent lower channel by setting K=4 in its mode register (see S826_CounterModeWrite). Note that when two channels are cascaded, only the count function is extended; the snapshot and preload mechanisms will continue to operate independently. Each cascade is limited to two counters.

7.1.9 Status LEDs

Each counter channel is associated with a status LED that indicates clock pulses are detected on that channel. The LED flashes at a constant rate while the clock signal is toggling. The LEDs are labeled E0-E5, corresponding to counter channels 0-5 respectively.

7.1.10 Reset state

Upon board reset, all counter channels are set to the “halted” state (see S826_CounterStateWrite for details). In addition, a board reset will also zero the Mode, Preload, and Compare registers.

7.1.11 Timing

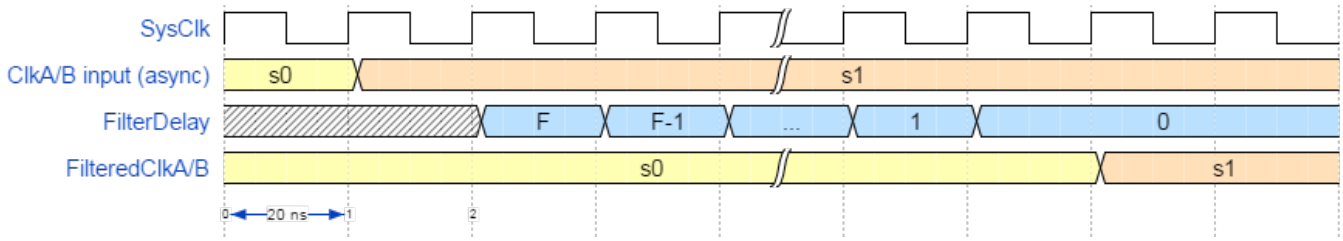
7.1.11.1 Input sampling and noise filters

All external counter inputs (ClkA, ClkB and IX) are synchronously sampled and then filtered. The filters can be used to remove glitches or compensate for slow edges on input signals, or they may be disabled if not needed.

The inputs are sampled using an internal 50 MHz clock (SysClk) with 3 ns minimum setup time. Consequently, an edge on any external counter signal will will be sampled within 20 ns after it occurs. For example, in Figure 7, the ClkA/B edge is sampled at SysClk 2. Pulses shorter than 20 ns may not be captured. The input setup time

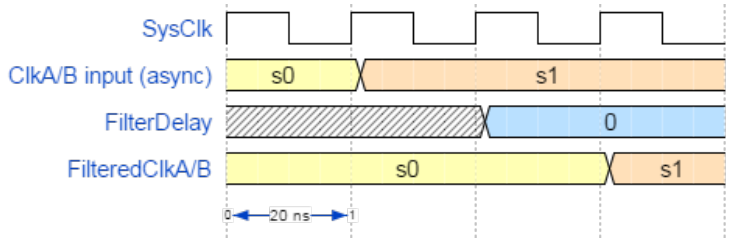
Every digital filter has a configurable time F, which is programmed by calling S826_CounterFilterWrite. A filter's input signal will appear on its output when the input has remained stable for time F. The filter delays sampled edges by 20*(F+1) ns. Figure 7 shows filter timing in the general case. Note that as F increases, the maximum input frequency that the filter will pass – and therefore the maximum count rate – decreases.

Figure 7: ClkA, ClkB or IX input sampling and filtering with filter time F > 0



The filter can be disabled by setting F=0 (see Figure 8), which reduces the filter delay to 20 ns, the minimum possible delay.

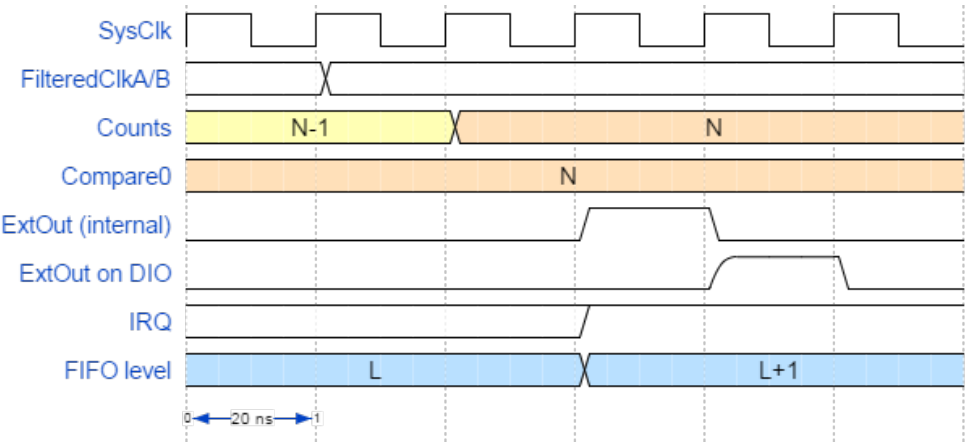
Figure 8: ClkA, ClkB or IX input sampling with filter disabled (F = 0)



7.1.11.2 Counting and output timing

The filtered ClkA and ClkB signals are decoded and used to control the counter (see Figure 9). The counts change 20 ns after a filtered clock edge. At 40 ns after the filtered edge, a snapshot is captured (to the event FIFO) and ExtOut begins to issue a 20 ns output pulse (when configured to do so). When the ExtOut signal is routed to a DIO (via programmable signal router), it is always delayed 20 ns on its way to the DIO connector pin.

Figure 9: Timing diagram of a ClkA / ClkB / IX input with filter disabled (F = 0)



In Figure 9, ExtOut has been configured to output a pulse when the counts match the value in the Compare0 register, and the rise time of the DIO output has been shortened with an external resistor. In this case, the DIO requires an external pull-up resistor because its internal pull-up, acting alone, would cause the output rise time to exceed the 20 ns pulse width (see External pull-up resistors).

Summary of counter timing parameters

Parameter		Min. (ns)	Max. (ns)
ClkA/ClkB to SysClk	Setup time	3	
	Hold time	0	
ClkA/ClkB to:	Counts change	40	60
	Snapshot	60	80
	ExtOut (internal)	60	80
	ExtOut on DIO pin	80	100

7.2 Connectors J4/J5

Two 26-pin headers, J4 and J5, bring out ClkA, ClkB, IX and power connections for the board's six counter channels. J4 is used for counters 0-2, and J5 for counters 3-5.

J4 and J5 Pinouts

J4: Counters 0-2			
Pin	**Name**	**Function**	**Chan**
1	+A0	Differential A clock inputs	0
2	-A0		
3	GND	Power supply return	
4	+B0	Differential B clock inputs	
5	-B0		
6	+5V	+5V power output	
7	+I0	Differential IX inputs	
8	-I0		
9	GND	Power supply return	
10	+A1	Differential A clock inputs	1
11	-A1		
12	+5V	+5V power output	
13	+B1	Differential B clock inputs	
14	-B1		
15	GND	Power supply return	
16	+I1	Differential IX inputs	
17	-I1		
18	+5V	+5V power output	
19	+A2	Differential A clock inputs	2
20	-A2		
21	GND	Power supply return	
22	+B2	Differential B clock inputs	
23	-B2		
24	+5V	+5V power output	
25	+I2	Differential IX inputs	
26	-I2		

J5: Counters 3-5			
Pin	**Name**	**Function**	**Chan**
1	+A3	Differential A clock inputs	3
2	-A3		
3	GND	Power supply return	
4	+B3	Differential B clock inputs	
5	-B3		
6	+5V	+5V power output	
7	+I3	Differential IX inputs	
8	-I3		
9	GND	Power supply return	
10	+A4	Differential A clock inputs	4
11	-A4		
12	+5V	+5V power output	
13	+B4	Differential B clock inputs	
14	-B4		
15	GND	Power supply return	
16	+I4	Differential IX inputs	
17	-I4		
18	+5V	+5V power output	
19	+A5	Differential A clock inputs	5
20	-A5		
21	GND	Power supply return	
22	+B5	Differential B clock inputs	
23	-B5		
24	+5V	+5V power output	
25	+I5	Differential IX inputs	
26	-I5		

7.2.1 Counter signals

Each counter channel has six input signals on its header connector: A+, A-, B+, B-, X+, and X-. The header also has power and ground pins that can be used to supply operating power to external devices such as incremental encoders. The following table details the header pins that are available for each channel and their recommended usage.

External signal connections to a counter channel

Function	Pin Name	Signal Type	Clock Source		
			Quadrature	Single-phase	Internal
ClkA	A+	RS-422	ClockA+	Clock+	NC
		TTL/CMOS	ClockA	Clock	NC
	A-	RS-422	ClockA-	Clock-	NC
		TTL/CMOS	NC	NC	NC
ClkB	B+	RS-422	ClockB+	NC	NC
		TTL/CMOS	ClockB	NC	NC
	B-	RS-422	ClockB-	NC	NC
		TTL/CMOS	NC	NC	NC
			NC		
			NC	Internal / None	
IX	X+	RS-422	NC	NC	
		TTL/CMOS	IX	NC	
	X-	RS-422	IX-	NC	
		TTL/CMOS	NC	NC	
ExtIn	Note 1		Auxiliary counter input. The behavior of this signal is configurable.		
ExtOut	Note 1		Counter output. The behavior of this signal is programmable.		
Device Power	GND	POWER	5V power supply return and ground reference for all logic signals.		
	+5V	POWER	+5VDC power. This can be used to power external devices such as incremental encoders. The total 5V current for all external loads on DIO and counter connectors must not exceed 400mA. Note that these pins do not have independent short circuit protection.		

Notes

1. **If used, this signal must connect to a DIO channel through the board's DIO signal routing matrix. Refer to the DIO documentation for details of the routing matrix and electrical characteristics of the DIO channels.**

7.2.2 Application connections

Typical counter connections for common applications

Application	Signals				
	ClkA	ClkB	IX	ExtIn	ExtOut
Encoder Interface	Phase A clock	Phase B clock	Snapshot trigger | Preload trigger | NC	Count enable | NC	NC
Event Counter	Event clock	NC	Snapshot trigger | Preload trigger | NC	Count enable | NC	NC
Pulse Generator	NC	NC	Pulse trigger | NC	Pulse trigger | NC	Pulse output
PWM Generator	NC	NC	Output enable | NC	NC	PWM output
Pulse Width Measurement	NC	NC	Signal to measure	NC	NC
Period Measurement	NC	NC	Signal to measure	NC	NC
Frequency Measurement	NC	NC	External time gate | NC	NC	NC

7.3 API functions

7.3.1 S826_CounterSnapshotRead

The S826_CounterSnapshotRead function reads a snapshot from a counter channel.

```
int S826_CounterSnapshotRead(
  uint board,     // board identifier
  uint chan,      // channel number
  uint *counts,   // pointer to counts buffer
  uint *tstamp,   // pointer to timestamp buffer
  uint *reason,   // pointer to reason buffer
  uint tmax       // maximum time to wait
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

counts
Pointer to a buffer that will receive the latched counts value. Set to NULL to ignore counts.

tstamp
Pointer to a buffer that will receive the timestamp. Set to NULL to ignore timestamp.

reason
Pointer to a buffer that will receive the reason flags, which indicate what caused the snapshot. When a flag bit = '1', the snapshot was caused by that event type. More than one event may have occurred at the same time, so multiple bits may be set.

bit	Description
8	Quadrature error
7	Soft snapshot (caused by calling S826_CounterSnapshot)
6	ExtIn rising edge
5	ExtIn falling edge
4	Index rising edge
3	Index falling edge
2	Zero counts reached
1	Compare1 match reached (note: snapshot counts always equal value in Compare1 register)
0	Compare0 match reached (note: snapshot counts always equal value in Compare0 register)

Set to NULL to ignore the reason.

tmax
Maximum time, in microseconds, to wait for data. See Event-driven applications for details.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function reads a previously acquired snapshot from the snapshot FIFO, consisting of the counts at a particular moment in time and the associated timestamp and reason flags. Each snapshot can be read only one time; when a snapshot is read, it is removed from the FIFO and cannot be read again.

Snapshots may be acquired in response to asynchronous events. In such cases, snapshots may occur at any time. However, if snapshots are not automatically acquired, or for some other reason it is necessary to invoke a snapshot under program control, the application program must call S826_CounterSnapshot to capture a new snapshot before calling this function to read the snapshot.

The reason flags indicate the type of event that caused the snapshot. If two or more simultaneous events would each cause a snapshot, all of the associated reason flags will be set.

Quadrature-encoded clocks are monitored for encoding errors as described in Quadrature clock decoder. When an encoding error is detected, a snapshot is captured (with reason bit 8 set) and an internal error flag is set. While the error flag remains set, subsequent encoding errors will be ignored and will not trigger new snapshots (though other types of triggers will continue to cause snapshots). The error flag is automatically cleared when this function reads the error-triggered snapshot, or when the channel is halted, or when the snapshot FIFO becomes empty. The latter case ensures that the error flag will be cleared if the error-triggered snapshot is lost due to FIFO overflow.

This function can operate in either blocking or non-blocking mode, depending on the value of tmax. If tmax is zero, the function will return immediately. If tmax is greater than zero, the calling thread will block until a snapshot is available or tmax elapses. The function will return either S826_ERR_OK or S826_ERR_FIFOOVERFLOW if a snapshot was read, or S826_ERR_NOTREADY if no snapshot is available. S826_ERR_ FIFOOVERFLOW indicates that a snapshot was successfully read, but the channel's snapshot FIFO overflowed (one or more snapshots were lost); the FIFO overflow status will be automatically cleared when the function returns.

When this function is blocking, it will immediately return S826_ERR_CANCELLED if S826_CounterWaitCancel is called by another thread, or S826_ERR_BOARDCLOSED if S826_SystemClose is called. In either case, the counts, tstamp and reason values will be invalid.

7.3.2 S826_CounterWaitCancel

The S826_CounterWaitCancel function cancels a blocking wait on a counter channel.

```
int S826_CounterWaitCancel(
  uint board,    // board identifier
  uint chan      // channel number
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5, for which waiting is to be canceled.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function cancels blocking on a counter channel so that another thread, which is blocked by S826_CounterSnapshotRead while waiting for a snapshot, will return immediately with S826_ERR_CANCELLED.

7.3.3 S826_CounterCompareWrite

The S826_CounterCompareWrite function writes to a compare register.

```
int S826_CounterCompareWrite(
  uint board,    // board identifier
  uint chan,     // channel number
  uint regid,    // register identifier
  uint counts    // counts value
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

regid
Selects Compare register: 1 = Compare1 register, 0 = Compare0 register.

counts
Value to be written to the Compare register.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.4 S826_CounterCompareRead

The S826_CounterCompareRead function returns the contents of a compare register.

```
int S826_CounterCompareRead(
  uint board,    // board identifier
  uint chan,     // channel number
  uint regid,    // register identifier
  uint *counts   // counts value
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

regid
Selects Compare register: 1 = Compare1 register, 0 = Compare0 register.

counts
Pointer to a buffer that will receive the contents of the selected Compare register.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.5 S826_CounterSnapshot

The S826_CounterSnapshot function invokes an immediate snapshot.

```
int S826_CounterSnapshot(
  uint board,    // board identifier
  uint chan      // channel number
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function forces a snapshot to be captured immediately. It is typically used to capture a snapshot under program control prior to reading a counter. This is one of two methods of reading the core's instantaneous counts; the other method is S826_CounterRead. This function may be called at any time when the counter channel is running.

7.3.6 S826_CounterRead

The S826_CounterRead function reads the counter core's instantaneous counts.

```
int S826_CounterRead(
  uint board,    // board identifier
  uint chan,     // channel number
  uint *counts   // counts value
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

counts
Pointer to a buffer that will receive the instantaneous counts from the counter core.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function directly reads and returns the value currently stored in the counter core. If a timestamp is also needed, the application can call this function and also call S826_TimestampRead, or it may trigger a soft snapshot and then read the snapshot.

7.3.7 S826_CounterPreloadWrite

The S826_CounterPreloadWrite function writes to a preload register.

```
int S826_CounterPreloadWrite(
  uint board,   // board identifier
  uint chan,    // channel number
  uint reg,     // register identifier
  uint counts   // counts value
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

reg
Selects the preload register to write to: 0 = Preload0 register, 1 = Preload1 register.

counts
The 32-bit value to be written to the selected Preload register.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

The counts value is immediately written to the selected preload register when this function executes. The counter core is not affected until the next core preload occurs.

7.3.8 S826_CounterPreloadRead

The S826_CounterPreloadRead function reads a preload register.

```
int S826_CounterPreloadRead(
  uint board,   // board identifier
  uint chan,    // channel number
  uint reg,     // register identifier
  uint *counts  // counts value
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

reg
Selects the preload register to write to: 0 = Preload0 register, 1 = Preload1 register.

counts
Pointer to a buffer that will receive the counts from the selected Preload register.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.9 S826_CounterPreload

The S826_CounterPreload function manually invokes a core preload from the Preload0 register.

```
int S826_CounterPreload(
  uint board,   // board identifier
  uint chan,    // channel number
  uint level,   // preload signal level
  uint sticky   // make level "sticky"
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

level
Effective preload signal level: 1 = invoke preload, 0 =don't invoke preload. This is ignored if sticky=0.

sticky
Persistence: 1 = maintain level until reprogrammed, 0 = strobe trigger (level is ignored).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

If sticky is false, the level will be ignored. In this case, the channel's software preload command will be strobed active and then immediately return to its inactive state, thus causing Preload0 to be copied to the counter core.

If sticky is true, the specified level will be continuously applied to the channel's software preload command until changed by a subsequent call to this function. Typically, sticky will only be asserted if the counter is operating as a Pulse Generator, and only when output retriggering is enabled. When sticky and level are both '1', the counter will continuously preload from Preload0, thus causing the pulse output to go active and remain active until the function is called again to set the level to '0'; at that time, the counter will be allowed to resume counting.

7.3.10 S826_CounterStateWrite

The S826_CounterStateWrite function starts or stops channel operation.

```
int S826_CounterStateWrite(
  uint board,   // board identifier
  uint chan,    // channel number
  uint state    // channel state
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

state
Channel state: 0 = halted, 1 = running.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function is used to start and stop counter channel operation, thus placing the channel in either the “running” or “halted” state. A channel's operating mode should be configured before switching it to the running state; this is done by calling S826_CounterModeWrite. A channel will operate as specified by the mode register while it is in the running state.

In the halted state:

- Counting and preloading are not allowed.
- Counter core is reset to zero counts.
- Snapshot register is marked empty.
- Output is held inactive.
- Quadrature error is reset.

The mode, preload, compare, and snapshot configuration registers are not affected when a channel transitions between halted and running states. This function has no effect if used to start a channel that is already running, or to stop a channel that is already halted.

This function can be used to zero the counter core by calling it once to halt the channel (and zero the counts) and again to start the channel running. Another way to zero the counts is to zero to the Preload0 register (by calling S826_CounterPreloadWrite) and then transfer the preload value to the core (via S826_CounterPreload).

7.3.11 S826_CounterStatusRead

The S826_CounterStatusRead function returns information about a counter channel's status.

```
int S826_CounterStatusRead(
  uint board,    // board identifier
  uint chan,     // channel number
  uint *status   // channel status info
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

status
Pointer to a buffer that will receive the status:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	RUN	0	ST	0	0	0	0	0	0	0	0	0	0	PLS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	Description
RUN	Channel enable: '1' = running, '0' = halted. This is controlled by S826_CounterStateWrite.
ST	Sticky preload command. This is controlled by S826_CounterPreload.
PLS	Preload selector. This indicates the active preload register: '1' = Preload1, '0' = Preload0. PLS can be '1' only if mode register bit BP=1 (see S826_CounterModeWrite).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.12 S826_CounterExtInRoutingWrite

The S826_CounterExtInRoutingWrite function selects the signal source for a counter channel's ExtIn input.

```
int S826_CounterExtInRoutingWrite(
  uint board,    // board identifier
  uint chan,     // counter channel number
  uint route     // routing configuration
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

route
Signal to be connected to ExtIn (this is ignored if mode register field IM=0):
0 to 47 = DIO channel 0 to 47;
48 to 53 = counter channel 0 to 5 ExtOut;
54 to 59 = virtual digital output channel 0 to 5.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function will route the counter channel's ExtIn input to a general-purpose digital I/O (DIO) channel so that an external signal (applied to the DIO channel's I/O pin) can control the channel's ExtIn input. Alternatively, the ExtIn input may internally routed to any counter channel's ExtOut output, or to any virtual digital output channel.

7.3.13 S826_CounterExtInRoutingRead

The S826_CounterExtInRoutingRead function returns a counter channel's ExtIn signal routing configuration.

```
int S826_CounterExtInRoutingRead(
  uint board,    // board identifier
  uint chan,     // counter channel number
  uint *route    // routing configuration
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

route
Pointer to buffer that will receive the ExtIn routing configuration. The format of the returned value is identical to the route argument used in the S826_CounterExtInRoutingWrite function.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.14 S826_CounterSnapshotConfigWrite

The S826_CounterSnapshotConfigWrite function programs the snapshot configuration register.

```
int S826_CounterSnapshotConfigWrite(
  uint board,    // board identifier
  uint chan,     // counter channel number
  uint cfg,      // snapshot configuration flags
  uint mode      // 0=write, 1=clear bits, 2=set bits
);
```

Parameters

board

826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan

Counter channel number in the range 0 to 5.

cfg

Flags with 'E' prefix determine the types of events that will capture snapshots: '1' = enable capturing, '0' = disable capturing. Each flag with 'R' prefix determines whether the corresponding 'E' flag will be automatically cleared upon snapshot capture, thus preventing subsequent events of that type from invoking snapshots.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	RER	REF	RXR	RXF	RZ	RM1	RMO	0	0	0	0	0	0	0	0	0	EER	EEF	EXR	EXF	EZ	EM1	EM0

Flag	Snapshot trigger event
ER	ExtIn leading edge
EF	ExtIn falling edge
XR	Index leading edge
XF	Index falling edge
Z	Zero counts reached
M1	Compare1 register matches counter core
M0	Compare0 register matches counter core

mode

Write mode for cfg: 0 = write, 1 = clear bits, 2 = set bits (see “Atomic read-modify-write”).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function programs the snapshot configuration register. A snapshot will be captured when an 'E' flag is programmed to '1' and the corresponding event occurs. Upon snapshot capture, the associated 'E' flag will be automatically cleared if the corresponding 'R' flag is set.

Independent of these flags, snapshots may also be captured upon quadrature clock error or in response to calls to S826_CounterSnapshot.

7.3.15 S826_CounterSnapshotConfigRead

The S826_CounterSnapshotConfigRead function reads the snapshot configuration register.

```
int S826_CounterSnapshotConfigRead(
  uint board,    // board identifier
  uint chan,     // counter channel number
  uint *cfg      // configuration flags
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

cfg
Pointer to buffer that will receive the configuration flags. The format of the returned value is identical to the events value used in the S826_CounterSnapshotConfigWrite function. Note that 'E' flags (flags with 'E' name prefix) may be automatically reset in response to captured snapshots.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.16 S826_CounterFilterWrite

The S826_CounterFilterWrite function configures the IX, CLKA, and CLKB noise filters.

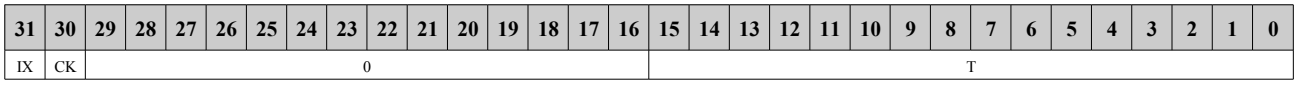
```
int S826_CounterFilterWrite(
  uint board,    // board identifier
  uint chan,     // counter channel number
  uint cfg       // filter configuration
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

cfg
Filter configuration:



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IX	CK	0														T															

Field	Description
IX	Enable IX filter: '1' = enable, '0' = disable.
CK	Enable CLKA/CLKB filters: '1' = enable, '0' = disable.
T	Filter time interval specified as multiple of 20ns, common to IX, CLKA and CLKB input filters.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

Each of the IX, CLKA, and CLKB input signals has a dedicated noise filter. This function enables and disables the filters and programs the filter time interval. The CLKA and CLKB filters are enabled or disabled as a group, whereas the IX filter is independently enabled or disabled. The filter time interval, T, is common to all enabled filters.

A filter's output will not change state until its input has held a constant state for T * 20ns. The maximum T value is 65535, corresponding to a filter time of 1.3107ms. When a filter is enabled, it will delay its input signal by T * 20ns. Note that when the noise filter is enabled for the CLKA/CLKB inputs (CK = '1'), the counter's maximum clock frequency is reduced; the maximum frequency reduction is inversely proportional to T.

7.3.17 S826_CounterFilterRead

The S826_CounterFilterRead function reads the configuration of the IX, CLKA, and CLKB noise filters.

```
int S826_CounterFilterRead(
  uint board,    // board identifier
  uint chan,     // counter channel number
  uint *cfg      // filter configuration
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

cfg
Pointer to buffer that will receive the configuration. The format of the returned value is identical to the cfg value used in the S826_CounterFilterWrite function.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.18 S826_CounterModeWrite

The S826_CounterModeWrite function configures a counter channel's operating mode.

```
int S826_CounterModeWrite(
  uint board,    // board identifier
  uint chan,     // channel number
  uint mode      // operating mode
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

mode
Channel operating mode (see “Application notes” for examples):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	IP	IM		0	0	0	TP	NR	UD	BP	OM			OP	TP						TE		TD		K			XS			

IP	ExtIn signal polarity. This is ignored if IM=0.
0	Normal polarity.
1	Inverted polarity.

IM	ExtIn mode. If IM>0 then ExtIn is routed to a DIO pin as specified by S826_CounterExtInRoutingWrite.
0	ExtIn input is not used as a permissive. This is typically used when ExtIn will only be used to trigger snapshots.
1	ExtIn input is a count enable permissive. This is typically used with incremental encoders and event counting, with ExtIn acting as a count enable.
2	ExtIn input is a preload enable permissive. This is typically used when operating as a pulse generator, with ExtIn used as a pulse trigger.

TP bit	Preload enables. Determines event(s) that will cause a preload register to be copied to the counter. Each bit enables preloads for one type of event: '1' = enable, '0' = disable.
24	Preload upon start (when channel switches from halted to running)
16	Preload upon Index active level.
15	Preload upon Index leading edge.
14	Preload upon Index falling edge.
13	Preload upon zero counts reached.
12	Preload when Compare1 matches the counter core.
11	Preload when Compare0 matches the counter core.

NR	Preload permissive. Typically used for "one-shot" pulse generator applications.
0	Allow preloading any time. This can be used for applications such as a retriggerable one-shot.
1	Allow preloading only when counts equal zero. This can be used for applications such as a non-retriggerable one-shot. This applies to both hardware- and software-induced (via S826_CounterPreload function) preloads.

UD	Count direction
0	Normal count direction.
1	Reverse count direction.

BP	Preload toggle enable.
0	Use only Preload0. The Preload1 register will not be used.
1	Use both preload registers. This is typically used for PWM output.

OM	ExtOut mode. Determines when the channel's ExtOut output signal is active. The ExtOut signal may be routed to a DIO pin by calling S826_DioOutputSourceWrite.
0	Output always inactive.
1	Output pulses active upon Compare register snapshot.
2	Output active when Preload1 register is selected. This is typically used for PWM output.
3	Output active when the counts are not zero; useful for pulse generator applications.
4	Output active when the counts are zero; useful for watchdog timer applications.

OP	ExtOut signal polarity.
0	Normal polarity
1	Inverted

TE	Count enable trigger. Determines event(s) that will cause counting to be enabled.
0	Enable counting at start-up (i.e., when S826_CounterStateWrite is called to switch to running mode).
1	Enable counting upon Index leading edge.
2	Enable counting upon preload.
3	reserved -- do not use.

TD	Count disable trigger. Determines event(s) that will cause counting to be disabled.
0	Never disable counting.
1	Disable counting upon Index falling edge.
2	Disable counting upon zero counts reached.
3	reserved -- do not use

K	Clock mode.
0	External single-phase clock, increment on ClkA rising edge.
1	External single-phase clock, increment on ClkA falling edge.
2	Internal 1 MHz clock, increment every microsecond.
3	Internal 50 MHz clock, increment every 40 nanoseconds.
4	Link to adjacent channel's overflow/underflow outputs to this channel's overflow/underflow inputs (see "Cascading"). The adjacent channel's overflow/underflow output signals will cause this channel to increment/decrement. For channel 0, channel 5 is the adjacent channel; for all other channels N, the adjacent channel is N-1.
5	External quadrature clock, x1 multiplier.
6	External quadrature clock, x2 multiplier.
7	External quadrature clock, x4 multiplier.

XS	Index source.
0	External IX input, normal polarity.
1	External IX input, inverted.
2-7	ExtOut from counter channel 0-5 (e.g., 3 = ExtOut from channel 1).
8-15	Internal tick generator: 8 = 0.1 Hz, 9 = 1 Hz, 10 = 10 Hz, 11 = 100 Hz, 12 = 1 KHz, 13 = 10 KHz, 14 = 100 KHz, 15 = 1 MHz.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.3.19 S826_CounterModeRead

The S826_**Counter**ModeRead function returns a counter channel's operating mode.

```
int S826_CounterModeRead(
  uint board,    // board identifier
  uint chan,     // channel number
  uint *mode     // operating mode
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chan
Counter channel number in the range 0 to 5.

mode
Buffer that will receive the channel operating mode, as defined in "S826_CounterModeWrite".

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

7.4 Application notes

7.4.1 Hardware interrupts

An interrupt request (IRQ) is generated when a counter event (such as Compare register match) is detected and stored in the counter's FIFO (i.e., a "snapshot" is captured). This IRQ is managed by the API function S826_CounterSnapshotRead, which enables/disables counter interrupts and handles IRQs as required so that software developers need not be concerned with the complexities of interrupt programming.

To wait for the next IRQ, simply call S826_CounterSnapshotRead. The function will return when an event is available, and will block and allow other threads to run while the event FIFO is empty. See the following sections for examples of how this works.

7.4.2 Time delay

A counter can be used as a delay timer to allow a software task to be executed after a specific amount of time has elapsed. To implement this, preload the counter with the desired delay time and configure it to count down and capture a snapshot when it reaches zero counts.

The following function creates a delay timer that has 1 µs resolution (because it uses the 1 MHz internal clock). For higher resolution (20 ns), use the 50 MHz internal clock. Note that error checking has been simplified for clarity (it is recommended to always perform error checking in production software).

```
// Create a delay timer and start it running ---------------

int CreateDelay(uint board, uint ctr, uint delay)  // delay specified in microseconds
{
  // Configure the delay timer:
  S826_CounterModeWrite(board, ctr,                // Configure counter0 mode:
    S826_CM_K_1MHZ |                               //   clock source = 1 MHz internal
    S826_CM_UD_REVERSE |                           //   count down
    S826_CM_PX_START |                             //   preload when counter enabled
    S826_CM_TD_ZERO);                              //   stop counting when counts==0
  S826_CounterPreloadWrite(board, ctr, 0, delay);  // Delay time in microseconds.
  S826_CounterSnapshotConfigWrite(board, ctr,      // Configure snapshots:
      S826_SSRMASK_ZERO |                          //  capture snapshot when counts==0
      (S826_SSRMASK_ZERO << 16),                   //  disable snapshots when counts==0
      S826_BITWRITE);
  return S826_CounterStateWrite(board, ctr, 1);    // Start the timer running.
}
```

To wait for the timer's interrupt, call S826_CounterSnapshotRead with a non-zero tmax (must be longer than the delay time). S826_CounterSnapshotRead will return upon snapshot interrupt and you can then perform the desired task.

```
int WaitForDelay(uint board, uint ctr)
{
  return S826_CounterSnapshotRead(board, ctr,   // Block while waiting for timer:
    NULL, NULL, NULL,                           //  ignore snapshot counts, timestamp and reason
    S826_WAIT_INFINITE);                        //  never timeout
}
```

For example, this code will delay for 0.5 seconds while other threads are allowed to run, using counter0 on board number 0:

```
CreateDelay(0, 0, 500000);          // Create 0.5 second delay timer and start it running.
WaitForDelay(0, 0);                 // Wait for delay time to elapse.
printf("Delay time has elapsed!");  // Perform the delayed task.
```

In the above example, the calling thread simply waits for WaitForDelay to return. If desired, it can do some work while the delay timer is running:

```
CreateDelay(0, 0, 500000);          // Create 0.5 second delay timer and start it running.
                                    // If desired, do some work here while timer runs.
WaitForDelay(0, 0);                 // Wait for delay time to elapse.
printf("Delay time has elapsed!");  // Perform the delayed task.
```

7.4.3 Periodic timer

A counter can be used as a periodic timer, which is useful for periodically executing software or triggering ADC conversions. To implement this, configure the counter so that it repeatedly counts down to zero and then preloads (see examples below). The preload value and clock frequency determine the period. The following functions create a timer having 1 µs resolution because they use the 1 MHz internal clock; for higher resolution (20 ns), use the 50 MHz internal clock.

7.4.3.1 Hardware timer

The following function will cause a counter to periodically pulse its ExtOut output. The pulses can be internally routed to the ADC and used to trigger conversion bursts. If a physical output from the board is needed, ExtOut may be internally routed to a DIO for external use, but note that the output pulses are always 20 ns wide regardless of the input clock frequency (for longer pulses, configure the counter to operate as a PWM generator).

```
// Configure a counter as a 20 ns pulse generator and start it running.
int CreateHwTimer(uint board, uint chan, uint period) // period in microseconds
{
  S826_CounterModeWrite(board, chan,                // Configure counter mode:
    S826_CM_K_1MHZ |                                //   clock source = 1 MHz internal
    S826_CM_PX_START | S826_CM_PX_ZERO |            //   preload @start and counts==0
    S826_CM_UD_REVERSE |                            //   count down
    S826_CM_OM_NOTZERO);                            //   ExtOut = (counts!=0)
  S826_CounterPreloadWrite(board, chan, 0, period); // Set period in microseconds.
  return S826_CounterStateWrite(board, chan, 1);    // Start the timer running.
}
```

7.4.3.2 Software timer

This function will configure a counter to capture snapshots at regular intervals and start it running:

```
// Configure a counter as a periodic software timer and start it running.
int CreateTimer(uint board, uint chan, uint period)   // period in microseconds
{
  S826_CounterModeWrite(board, chan,                // Configure counter mode:
    S826_CM_K_1MHZ |                                //   clock source = 1 MHz internal
    S826_CM_PX_START | S826_CM_PX_ZERO |             //   preload @start and counts==0
    S826_CM_UD_REVERSE);                            //   count down
  S826_CounterPreloadWrite(board, chan, 0, period); // Set period in microseconds.
  S826_CounterSnapshotConfigWrite(board, chan,      // Snapshot when counts==0.
      S826_SSRMASK_ZERO, S826_BITWRITE);
  return S826_CounterStateWrite(board, chan, 1);    // Start the timer running.
}
```

While the timer is running, a thread can call the following function to wait for the next snapshot. Other threads are allowed to run while the calling thread waits for S826_CounterSnapshotRead, resulting in efficient, event-driven operation.

```
// Wait for the next timer period. Returns 0=ok, else=826 error code.
int WaitForTimer(uint board, uint chan)
{
  return S826_CounterSnapshotRead(board, chan,      // Block until next period.
      NULL, NULL, NULL,       // ignore snapshot values
      S826_WAIT_INFINITE);    // no time limit on wait
}
```

7.4.4 Call a function periodically

To call a function periodically, first create a timer as shown above. For example, the following code creates a 10 millisecond timer using counter0:

```
CreateTimer(0, 0, 10000);   // Create 10 ms (10000 us) timer and start it running.
```

Now run the following loop. Note that other threads can run while this thread is waiting for the next timer period.

```
// Repeat while no errors detected:
while (!WaitForTimer(0, 0)) {  // Block until next timer period.
  PeriodicFunction();          // Execute the periodic function.
}
```

Alternatively, if you need to perform other processing while waiting for the next period and cannot use another thread to do so, you can poll the counter by calling S826_CounterSnapshotRead with a zero (or other finite) wait time:

```
while (1) { // Repeat forever:
  uint errcode = S826_CounterSnapshotRead( // Poll to see if period has elapsed (don't block).
    0, 0, NULL, NULL, NULL, 0);
  if (errcode == S826_ERR_OK)              // If it's a new period
    PeriodicFunction();                    //  execute the periodic function.
  else if (errcode != S826_ERR_NOTREADY)   // Else if fatal error detected
    break;                                 //  exit the polling loop.
  else                                     // Else
    DoSomeOtherStuff();                    //  do other processing.
}
```

7.4.5 Frequency counter

The frequency of a digital signal can be precisely measured with one counter. To do this, connect the external signal to the counter's ClkA+ input and route one of the internal tick generator signals to the counter's Index input. The selected tick signal is used as a counting gate and it also determines the sampling rate and measurement resolution. The sampling rate is equal to the tick frequency, whereas resolution is inversely proportional to the tick frequency. The following function configures a counter for frequency measurement:

```
// Configure a frequency counter and start it running.
int CreateFrequencyCounter(uint board, uint chan, uint ticksel)
{
  if ((ticksel < S826_CM_XS_R1HZ) || (ticksel > S826_CM_XS_1MHZ))
    return S826_ERR_S826_ERR_VALUE;               // Abort if invalid tick selector.
  S826_CounterModeWrite(board, chan,              // Configure counter:
    S826_CM_K_ARISE |                             //   clock = ClkA (external digital signal)
    ticksel |                                     //   route tick generator to Index input
    S826_CM_PX_IXRISE);                           //   preload upon tick rising edge
  S826_CounterPreloadWrite(board, chan, 0, 0);    // Reset counts upon tick rising edge.
  S826_CounterSnapshotConfigWrite(board, chan,    // Acquire counts upon tick rising edge.
      S826_SSRMASK_IXRISE, S826_BITWRITE);
  return S826_CounterStateWrite(board, chan, 1);  // Start the frequency counter running.
}
```

While the frequency counter is running, a thread can call the following function to wait for the next sample. Other threads are allowed to run while it waits for the next sample to arrive.

```
// Receive the next frequency sample.
int ReadFrequency(uint board, uint chan, uint *sample)
{
  return S826_CounterSnapshotRead(board, chan,      // Block until next sample
      sample, NULL, NULL,                           //   and then receive accumulated counts.
      S826_WAIT_INFINITE);
}
```

For example, the following code will measure frequency 10 times per second using counter0 on board 0, with a resolution of 10 Hz. Note that the first sample is discarded as it may not be valid.

```
// Measure frequency 10 times per second.
uint counts;
CreateFrequencyCounter(0, 0, S826_CM_XS_10HZ);  // Create frequency counter and start it running.
ReadFrequency(0, 0, NULL);                      // Discard first sample.
while (!ReadFrequency(0, 0, &counts)) {         // Wait for next sample; if no errors then
  printf("Frequency = %d Hz\n", counts * 10);   //   display measured frequency.
}
```

7.4.6 Period and pulse width measurement

A counter can be used to measure the period of a digital signal applied to its IX+ input. It does this by counting internal clocks during one complete cycle of the input signal. At the end of each cycle, the accumulated counts are captured in a snapshot and the counter is zeroed to start the next measurement. Since a new measurement begins at the end of each input cycle, the sampling rate will equal the signal's input frequency.

Signal period is indicated by the snapshot counts, so a higher clock frequency will result in a greater number of counts and higher resolution. In the example functions below, set hires=0 to select the 1 MHz internal clock (1 µs resolution for periods up to approximately 71.6 minutes) or set hires=1 to select the 50 MHz internal clock (20 ns resolution for periods up to approximately 1.43 minutes).

```
// Create a period timer and start it running.
int CreatePeriodTimer(uint board, uint ctr, int hires)
{
  S826_CounterModeWrite(board, ctr,             // Configure the period counter:
    (hires ?                                    //   select internal clock based on resolution:
      S826_CM_K_50MHZ :                         //     for high res (20 ns) use 50 MHz
      S826_CM_K_1MHZ) |                         //     for low res (1 us) use 1 MHz
    S826_CM_UD_NORMAL |                         //   count up
    S826_CM_XS_EXTNORMAL |                      //   connect index input to IX (external signal)
    S826_CM_PX_IXRISE);                         //   preload upon signal rising edge

  S826_CounterPreloadWrite(board, ctr, 0, 0);   // Preload 0 at start of each cycle.
  S826_CounterSnapshotConfigWrite(board, ctr,   // Snapshot upon signal rising edge.
    S826_SSRMASK_IXRISE, S826_BITWRITE);
  return S826_CounterStateWrite(board, ctr, 1); // Enable period measurement.
}
```

In a nearly identical manner, a counter can be used to measure the width of digital pulses applied to the IX+ input. To do this, capture snapshots at the falling edge of the signal (at end of pulse vs. end of cycle):

```
// Create a pulse timer and start it running.
int CreatePulseTimer(uint board, uint ctr, int hires)
{
  S826_CounterModeWrite(board, ctr,             // Configure the period counter:
    (hires ?                                    //   select internal clock based on resolution:
      S826_CM_K_50MHZ :                         //     for high res (20 ns) use 50 MHz
      S826_CM_K_1MHZ) |                         //     for low res (1 us) use 1 MHz
    S826_CM_UD_NORMAL |                         //   count up
    S826_CM_XS_EXTNORMAL |                      //   connect index input to IX (external signal)
    S826_CM_PX_IXRISE);                         //   preload upon signal rising edge

  S826_CounterPreloadWrite(board, ctr, 0, 0);   // Preload 0 at start of each cycle.
  S826_CounterSnapshotConfigWrite(board, ctr,   // Snapshot upon signal falling edge.
    S826_SSRMASK_IXFALL, S826_BITWRITE);
  return S826_CounterStateWrite(board, ctr, 1); // Enable period measurement.
}
```

The following function will wait for the counter to complete a measurement and then return the measured time interval. Other threads are allowed to run while the counter is measuring.

```
// Read measured time interval in seconds.
int ReadTimer(uint board, uint ctr, double *period, int hires)
{
  uint counts;
  int errcode = S826_CounterSnapshotRead(board, ctr,  // Block until next sample available.
     &counts, NULL, NULL, S826_WAIT_INFINITE);
  *period = counts * (hires ? 2e-8 : 1e-6);
  return errcode;
}
```

The first sample should always be discarded because it may not cover a complete cycle (or pulse, if measuring pulse width):

```
// Example: Monitor period of external digital signal.
double period;     // period in seconds
int hires = TRUE;
CreatePeriodTimer(0, 0, hires);    // Create high-resolution (20 ns) period timer.
ReadTimer(0, 0, &period, hires);   // Discard the first sample (may not be a complete cycle).
while (1) {
  ReadTimer(0, 0, &period, hires);
  printf("Signal period (in seconds) = %f", period);
}
```

7.4.7 Routing a counter output to DIO pins

The following function will route a counter's ExtOut signal to a DIO pin, which is useful when a physical output is needed (e.g., PWMs, one-shots).

```
int RouteCounterOutput(uint board, uint ctr, uint dio)
{
  uint data[2];      // dio routing mask
  if ((dio >= S826_NUM_DIO) || (ctr >= S826_NUM_COUNT))
    return S826_ERR_VALUE;  // bad channel number
  if ((dio & 7) != ctr)
    return S826_ERR_VALUE;  // counter output can't be routed to dio

  // Route counter output to DIO pin:
  S826_SafeWrenWrite(board, S826_SAFEN_SWE);        // Enable writes to DIO signal router.
  S826_DioOutputSourceRead(board, data);            // Route counter output to DIO
  data[dio > 23] |= (1 << (dio % 24));              //   without altering other routes.
  S826_DioOutputSourceWrite(board, data);
  return S826_SafeWrenWrite(board, S826_SAFEN_SWD); // Disable writes to DIO signal router.
}
```

Each DIO is associated with a specific counter (see S826_DioOutputSourceWrite for details) and can only be routed to that counter's ExtOut. For example, counter3 can be routed to dio 3, 11, 19, 27, 35 or 43, or to any combination of these. This example shows how to route counter3's ExtOut signal to both dio3 and dio11:

```
RouteCounterOutput(0, 3, 3);   // On board number 0, route counter3's ExtOut signal to dio3
RouteCounterOutput(0, 3, 11);  //   and also to dio11.
```

Note that RouteCounterOutput is not thread-safe because it uses an unprotected read-modify-write (RMW) sequence to configure the DIO signal router. If multiple threads will be calling this function then a mutex should be used to ensure thread safety.

7.4.8 One-shot operation

A counter can be used to implement a "one-shot" that outputs a pulse when triggered. The trigger can be an external signal (applied to the counter's IX+ input) or another counter's ExtOut signal. The one-shot can also be triggered by software by invoking a counter preload. The output pulse appears on the counter's ExtOut signal, which can be routed to a DIO pin and/or another counter's Index input, if desired.

The following function will configure a counter as a one-shot, with the pulse width (ontime) specified in microseconds. It can be used to implement a retriggerable or non-retriggerable one-shot.

```
// Configure a counter as as a one-shot -----------

// trig specifies the trigger source: 0 = external (IX input); 2-7 = counter[trig-2] ExtOut.
int CreateOneShot(uint board, uint ctr, uint trig, uint ontime)
{
  S826_CounterModeWrite(board, ctr,                // Configure counter:
    S826_CM_K_1MHZ |                               //   clock = internal 1 MHz
    S826_CM_UD_REVERSE |                           //   count down
    trig |                                         //   route trig to Index input
    S826_CM_PX_IXRISE |                            //   preload @ trig rising edge
    S826_CM_TE_PRELOAD |                           //   enable upon preload
    S826_CM_TD_ZERO |                              //   disable when counts reach 0
    S826_CM_OM_NOTZERO);                           //   ExtOut = (counts!=0)
  S826_CounterPreloadWrite(board, ctr, 0, ontime); // Pulse width in microseconds.
  return S826_CounterStateWrite(board, ctr, 1);    // Enable the 1-shot.
}
```

The following examples show how to create a 700 µs externally-triggered one shot using counter0, with output on dio0:

```
// Implement a retriggerable one-shot.
RouteCounterOutput(0, 0, 0);                  // Route one-shot (counter0) output to dio0.
CreateOneShot(0, 0, 0, 700);                  // Create 700 us retriggerable one-shot.

// Implement a non-retriggerable one-shot.
RouteCounterOutput(0, 0, 0);                  // Route one-shot (counter0) output to dio0.
CreateOneShot(0, 0, S826_CM_NR_NORETRIG, 700); // Create 700 us non-retriggerable one-shot.
```

7.4.9 Jamming counts into a counter

In some situations an application may need to programmatically load a particular counts value into a counter. This is done by writing the value to the counter's preload0 register and then invoking a soft counter preload, as shown below:

```
// Jam a value into a counter.
void JamCounts(uint board, uint ctr, uint value)
{
  S826_CounterPreloadWrite(board, ctr, 0, value);   // Write value to preload0 register.
  S826_CounterPreload(board, ctr, 1, 0);            // Copy preload0 to counter.
}

JamCounts(0, 3, 1234);  // Example: Jam 1234 into counter3 of board0 (see section 7.4.9).
```

7.4.10 PWM generator

The following functions will configure a PWM generator and start it running. The PWM period is defined by the ontime and offtime arguments, which are specified in microseconds. The PWM output appears on the counter's ExtOut signal, which can be routed to a DIO pin if desired.

```
int CreatePWM(uint board, uint ctr, uint ontime, uint offtime)
{
  S826_CounterModeWrite(board, ctr,        // Configure counter for PWM:
    S826_CM_K_1MHZ |                       //   clock = internal 1 MHz
    S826_CM_UD_REVERSE |                   //   count down
    S826_CM_PX_START | S826_CM_PX_ZERO |   //   preload @startup and counts==0
    S826_CM_BP_BOTH |                      //   use both preloads (toggle)
    S826_CM_OM_PRELOAD);                   //   assert ExtOut during preload0 interval
  SetPWM(board, ctr, ontime, offtime);     // Program initial on/off times.
}

int StartPWM(uint board, uint ctr)
{
  return S826_CounterStateWrite(board, ctr, 1);      // Start the PWM generator.
}

int SetPWM(uint board, uint ctr, uint ontime, uint offtime)
{
  S826_CounterPreloadWrite(board, ctr, 0, ontime);   // On time in us.
  S826_CounterPreloadWrite(board, ctr, 1, offtime);  // Off time in us.
}

// Example: Use counter0 & dio0 as PWM generator; ontime=900, offtime=500 microseconds.
CreatePWM(0, 0, 0, 900, 500);    // Configure counter0 as PWM.
RouteCounterOutput(0, 0, 0);     // Route counter0 output to dio0 (see Section 7.4.7).
StartPWM(0, 0);                  // Start the PWM running.
```

Synchronized PWM generators can be implemented by configuring the PWM channels as hardware triggered one-shots (pulse generators). Use an additional channel to generate a common trigger for the PWM channels; this channel should be configured to generate periodic output pulses at the desired PWM frequency. The duty cycle of each PWM channel is controlled by adjusting its output pulse width.

7.4.11 Incremental encoders

7.4.11.1 Basic operation

The flexible counter architecture allows for many options, but basic operation works as follows. First configure and enable the counter channel to which the encoder is connected:

```
S826_CounterModeWrite(0, 0, S826_CM_K_QUADX4);  // Configure counter0 as incremental encoder
interface.
S826_CounterStateWrite(0, 0, 1);                // Start tracking encoder position.
```

To read the instantaneous encoder position without invoking a snapshot:

```
uint counts;
S826_CounterRead(0, 0, &counts);          // Read current encoder counts.
printf("Encoder counts = %d\n", counts);  // Display encoder counts.
```

When reading instantaneous counts you may need to know when the counts were sampled. You could rely on your software and operating system to sample the counts at precise times, but there's an easier and more accurate way: trigger a snapshot (via software) and then read the counts and sample time, which is accurate to within 1 µs:

```
uint counts;      // encoder counts when the snapshot was captured
uint timestamp;   // time the snapshot was captured

S826_CounterSnapshot(0, 0);               // Trigger snapshot on counter 0.
S826_CounterSnapshotRead(0, 0,            // Read the snapshot:
    &counts, &timestamp, NULL,            //  receive the snapshot info here
    0);                                   //  no need to wait for snapshot; it's already been
captured
printf("Counts = %d at time = %d\n", counts, timestamp);
```

Encoder counts can be changed to an arbitrary value at any time. This is typically done when the encoder is at a known reference position (e.g., at startup or whenever mechanical registration is required), but not at other times as it would disrupt position tracking.

```
JamCounts(0, 0, 12345);   // Force encoder counts (counter0) to 12345 (see section 7.4.9).
```

7.4.11.2 Using interrupts

The following code employs hardware interrupts to block the calling thread until the encoder counts reach a particular value. Other threads are allowed to run while the calling thread blocks. A snapshot is captured when the target count is reached. The snapshot generates an interrupt request, which in turn causes S826_CounterSnapshotRead to return. This example ignores the snapshot counts (which will always equal the target value), the timestamp, and the reason code (which will always indicate a Compare0 match event).

```
// Wait for counts to reach 5000. Allow other threads to run while waiting.

S826_CounterCompareWrite(0, 0, 0,         // Set Compare0 register to target value:
    5000);                                //  5000 counts (for this example)
S826_CounterSnapshotConfigWrite(0, 0,     // Enable snapshots:
    S826_SSRMASK_MATCH0,                  //  when counts==Compare0
    S826_BITWRITE);                       //  disable all other snapshot triggers
S826_CounterSnapshotRead(0, 0,            // Wait for counter to reach target counts:
    NULL, NULL, NULL,                     //  ignore snapshot counts, timestamp and reason
    S826_WAIT_INFINITE);                  //  disable function timeout
printf("Counter reached target counts");
```

In some cases you may want to wait for two different count values at the same time. The following example shows how to wait for the encoder counts to reach an upper or lower threshold (whichever occurs first). Furthermore, it will only wait for a limited amount of time. To set this up, the target values are programmed into Compare registers and then snapshots are enabled for matches to both Compare registers. To set a limit on how long to wait, a time limit value is specified by tmax when calling S826_CounterSnapshotRead.

Since snapshots can now be caused by different events, it's necessary to know what triggered a snapshot in order to decide how to handle it; this is indicated by the snapshot's reason flags. Alternatively, you could use the snapshot counts value, which will always equal the target value that triggered the snapshot.

```
// Wait for counts to reach 3000 or 4000, or for 10 seconds to elapse, whichever comes first.

uint counts;      // encoder counts when the snapshot was captured
uint timestamp;   // time the snapshot was captured
uint reason;      // event(s) that caused the snapshot
uint errcode;     // API error code

S826_CounterCompareWrite(0, 0, 0, 3000);  // Set Compare0 register to low limit (3000 counts)
S826_CounterCompareWrite(0, 0, 1, 4000);  // Set Compare1 register to high limit (4000 counts)
S826_CounterSnapshotConfigWrite(0, 0,     // Enable snapshots:
    S826_SSRMASK_MATCH0                   //  when counts==low limit
    | S826_SSRMASK_MATCH1,                //  or when counts==high limit
    S826_BITWRITE);                       //  disable all other snapshot triggers
errcode = S826_CounterSnapshotRead(0, 0,  // Wait for a snapshot:
    &counts, &timestamp, &reason,         //  receive the snapshot info here
    10000000);                            //  timeout if wait exceeds 10 seconds (10000000 us)

switch (errcode) {                        // Decode and handle the snapshot:
  case S826_ERR_NOTREADY:                 //  snapshot not available (wait timed out), so
    S826_CounterRead(0, 0, &counts);      //   read counts manually
    printf("Timeout -- counter didn't hit limits within 10 seconds; current counts = %d", counts);
    break;
  case S826_ERR_OK:
    if (reason & S826_SSRMASK_MATCH0)
      printf("Counter reached upper threshold at timestamp %d", timestamp);
    if (reason & S826_SSRMASK_MATCH1)
      printf("Counter reached lower threshold at timestamp %d", timestamp);
    break;
  default:
    printf("Error detected: errcode=%d", errcode);
}
```

7.4.11.3 Measuring speed

Speed measurement depends on the ability to sample displacement (distance traveled) at known times. Counter snapshots are ideally suited for this task because they include both displacement and time information. To set this up, configure the counter to track encoder position and arrange for it to periodically capture snapshots. Each time a new snapshot arrives, the count/timestamp pairs from the new and previous snapshots can be used to precisely compute speed.

An easy way to do this is to select the internal tick generator as the IX signal source, and configure the counter to capture snapshots on IX rising edges. The following code shows how to do this:

```
// Measure and display speed at 10 samples/second -----

#define DISP_PER_CNT 0.005  // Displacement per encoder count (e.g., 0.005 meters)

uint tstamp[2];       // Previous and current timestamps.
uint counts[2];       // Previous and current encoder counts.
double cnts_per_us;   // Encoder counts per microsecond.
double speed;         // Speed in user units (e.g., meters/second).

// Configure counter0 as encoder interface; automatically snapshot every 0.1 seconds.
S826_CounterModeWrite(0, 0,  // Configure counter mode:
  S826_CM_K_QUADX4 |         //   clock source = external quadrature x4
  S826_CM_XS_10HZ);          //   route internal 10 Hz clock to IX
S826_CounterSnapshotConfigWrite(0, 0, 0x10, 0); // Enable snapshots upon IX^.
S826_CounterStateWrite(0, 0, 1);                // Start tracking encoder position.
```

```
// Wait for first snapshot (captures starting displacement).
S826_CounterSnapshotRead(0, 0, counts, tstamp, NULL, S826_WAIT_INFINITE);

while (1) {  // Repeat every 0.1 seconds:
  // Wait for next displacement snapshot, then compute and display speed.
  S826_CounterSnapshotRead(0, 0, &counts[1], &tstamp[1], NULL, S826_WAIT_INFINITE);
  cnts_per_us = (double)(counts[1] - counts[0]) / (tstamp[1] - tstamp[0]);
  speed = DISP_PER_CNT * 1000000 * cnts_per_us;  // convert to meters/second
  printf("%f\n", speed);   // Display speed.
  counts[0] = counts[1];   // Prepare for next snapshot.
  tstamp[0] = tstamp[1];
}
```

7.4.11.4 Precise homing

To achieve high-accuracy homing, it is necessary to sample the "offset" counts without delay when a pulse edge is issued by the home position sensor. The delay time becomes increasingly important as homing speed increases, and is critically important in high-performance systems that seek home at high speeds.

The following example shows how to precisely determine the home position at any traversal speed by sampling the counts within 20 ns of the home sensor's rising edge. The sensor is connected to the counter's IX+ terminal, which is internally routed to the index input. The counter is effectively "armed" to enable sampling when the next sensor rising edge is detected. When the edge is detected, the counter will be "disarmed" automatically so that subsequent sensor edges won't cause sampling. Other threads are allowed to run while the counter waits for the sensor edge.

```
uint offset;   // counts offset at home position

S826_CounterSnapshotConfigWrite(0, 0,  // Set up snapshot triggering for counter0 (on board0):
  S826_SSRMASK_IXRISE |                //   arm for snapshot upon IX rising edge
  (S826_SSRMASK_IXRISE << 16),         //   automatically disarm upon first IX rising edge
  S826_BITSET);                        //   leave other triggering rules intact

S826_CounterSnapshotRead(0, 0,         // Block while waiting for IX rising edge.
  &offset,                             //   store exact home position counts here
  NULL, NULL, S826_INFINITE_WAIT);
```

The offset counts can now be used to correct all subsequent position samples:

```
uint position = sample - offset;
```

Alternatively, if the axis is briefly held motionless, the current position can be corrected and jammed into the counter so that subsequent samples will not need correction:

```
uint curposn;
S826_CounterRead(0, 0, &curposn);       // Sample current position.
void JamCounts(0, 0, curposn - offset); // Correct the position and jam it into counter.
```

7.4.11.5 Output pulse at specific position

A counter can output a pulse on a DIO when the counts reach a particular value. The pulse duration will be 20 ns regardless of how long the counts remain at the target value. Consequently, an external pull-up must be added to the DIO to speed up its rising edge (see Using external pull-up resistors for details) and thereby ensure proper pulse formation.

The following code will generate a 20 ns output pulse when the counts reach 5000, using counter0 and dio0 on board number 0. A snapshot is captured when the pulse is issued; this is required to produce the pulse but the snapshot data is not used in this example.

```
// Output a 20 nanosecond pulse when counts reach 5000 ----------

// Configure counter0 as encoder interface; output 20ns pulse when counts reach 5000.
S826_CounterModeWrite(0, 0,                  // Configure counter mode:
  S826_CM_K_QUADX4 |                         //   clock = external quadrature x4
  S826_CM_OM_MATCH);                         //   pulse ExtOut upon snapshot.
S826_CounterCompareWrite(0, 0, 0, 5000);     // Load target counts into compare0 reg.
S826_CounterSnapshotConfigWrite(0, 0, 1, 2); // Snapshot when counts reach target.
RouteCounterOutput(0, 0, 0);                 // Route counter0 output to dio0.
S826_CounterStateWrite(0, 0, 1);             // Start tracking encoder position.
```

If desired, a second counter can be used to stretch the pulse to any desired width. To do this, configure the second counter as a one-shot (using the encoder counter's ExtOut as a preload trigger) and program its preload counts to the desired pulse width. In this case the one-shot's (vs. encoder counter's) ExtOut must be routed to the DIO. Note that if rise time is critical, an external pull-up may still be required.

The following code uses a second counter to stretch the pulse width: it will output a 500 µs pulse on dio0 when 5000 counts is reached.

```
// Output a 500 microsecond pulse when counts reach 5000 ----------

// Configure counter0 as one-shot, used to stretch output pulses from counter1.
CreateOneShot(0, 0, 3, 500);                 // Create 500 us 1-shot, trig=ExtOut1.
RouteCounterOutput(0, 0, 0);                 // Route one-shot output to dio0.

// Configure counter1 as encoder interface; output 20ns pulse upon compare0 match.
S826_CounterModeWrite(0, 1,                  // Configure counter mode:
  S826_CM_K_QUADX4 |                         //   clock = external quadrature x4
  S826_CM_OM_MATCH);                         //   pulse ExtOut upon snapshot.
S826_CounterCompareWrite(0, 0, 0, 5000);     // Load target counts into compare0 reg.
S826_CounterSnapshotConfigWrite(0, 0, 1, 2); // Snapshot when counts reach target.
S826_CounterStateWrite(0, 1, 1);             // Start tracking encoder position.
```

7.4.11.6 Output pulse every N encoder pulses

The following code will generate a 20 ns output pulse every 20K encoder counts, using counter0 and dio0 on board number 0. A snapshot is captured each time an output pulse is issued, which is used to set up the next target counts. Note that the pulse duration will always be 20 ns no matter how long the counts remain at the target value. Consequently, an external pull-up must be added to the DIO to speed up its rising edge as explained in Using external pull-up resistors.

```
// Output a 20 nanosecond pulse every 20000 encoder counts ----------

#define PULSE_INTERVAL 20000     // output a DIO pulse every 20,000 encoder counts

uint targ = PULSE_INTERVAL;      // Generate pulse when encoder counts reach this.

// Configure counter0 as encoder interface; output 20ns pulse upon compare0 match.
S826_CounterModeWrite(0, 0,                  // Configure counter mode:
  S826_CM_K_QUADX4 |                         //   clock = external quadrature x4
  S826_CM_OM_MATCH);                         //   pulse ExtOut when counts reach target
S826_CounterSnapshotConfigWrite(0, 0, 1, 2); // Enable snapshots upon counts match.
```

```
RouteCounterOutput(0, 0, 0);                // Route counter0 output to dio0.
S826_CounterStateWrite(0, 0, 1);            // Start tracking encoder position.

do {  // Repeat forever …
  S826_CounterCompareWrite(0, 0, 0, targ);  // Program target counts and wait for match.
  int errcode = S826_CounterSnapshotRead(0, 0, NULL, NULL, NULL, S826_WAIT_INFINITE);
  targ += PULSE_INTERVAL;                   // Bump target counts.

  // TODO: PERFORM ANY OTHER REQUIRED PROCESSING

} while (errcode == S826_ERR_OK);
```

The following code builds on the previous example by using a second counter to stretch the pulse width: it will output a 500 µs pulse on dio0 every 20K encoder counts. In this case the output pulse is much longer than 20 ns, however, if rise time is critical, an external pull-up may still be necessary.

```
// Output a 500 microsecond pulse every 20000 encoder counts ----------

#define PULSE_INTERVAL 20000        // output a DIO pulse every 20,000 encoder counts

uint targ = PULSE_INTERVAL;         // generate pulse when encoder counts reach this

// Configure counter0 as one-shot, used to stretch output pulses from counter1.
CreateOneShot(0, 0, 3, 500);                // Create 500 us 1-shot, trig=ExtOut1.
RouteCounterOutput(0, 0, 0);                // Route one-shot output to dio0.

// Configure counter1 as encoder interface; output 20ns pulse upon compare0 match.
S826_CounterModeWrite(0, 1,                 // Configure counter mode:
  S826_CM_K_QUADX4 |                        //   clock = external quadrature x4
  S826_CM_OM_MATCH);                        //   pulse ExtOut when counts reach target
S826_CounterSnapshotConfigWrite(0, 1, 1, 2); // Enable snapshots upon compare0 match.
S826_CounterStateWrite(0, 1, 1);            // Start tracking encoder position.

do {  // Repeat forever …
  S826_CounterCompareWrite(0, 1, 0, targ);  // Program target counts and wait for match.
  int errcode = S826_CounterSnapshotRead(0, 1, NULL, NULL, NULL, S826_WAIT_INFINITE);
  targ += PULSE_INTERVAL;                   // Bump target counts.

  // TODO: PERFORM ANY OTHER REQUIRED PROCESSING

} while (errcode == S826_ERR_OK);
```

7.4.11.7 Interfacing touch-trigger probes

Touch-trigger probes (such as those made by Renishaw, Tormach, Marposs, etc.) are commonly used in CNC machine tools and coordinate measuring machines (CMMs). In a typical application, three or more incremental encoders monitor the position of a probe as it moves relative to a workpiece. The probe will output a digital signal when its stylus contacts the workpiece. When this happens, the stylus location must be determined by sampling all of the encoder counters.

At lower traverse speeds and in cases where moderate accuracy is acceptable, a probe-triggered interrupt service routine (ISR) can be used to sequentially sample the counters. This is feasible because ISR overhead and ISR-induced measurement errors (due to trigger latency and sampling skew) are negligible in such cases. However, these issues become significant when high accuracy and throughput is required. Consequently, in high-performance machines a hardware mechanism is needed that will simultaneously sample all counters with very low trigger latency.

Model 826 directly supports high-performance probe measurement by providing the ability to simultaneously sample up to six encoder counters with sub-microsecond trigger latency. In the following example, the probe's trigger signal is connected to dio17, and consecutive counters (starting with counter0) are assigned to the encoders of a 3-axis probe. The counters' high-speed FIFOs will simultaneously capture all encoder positions within 20 ns of receiving a probe trigger.

First, define a data structure for receiving touch point coordinates:

```
typedef struct POINT {  // Probe coordinates (extensible to higher dimensions)
  uint x;
  uint y;
  uint z;
} POINT;
```

This function will set up the probe interface and start it running:

```
CreateProbeInterface(uint trig, uint dio)
{
  for (int i = 0; i < 3; i++) {  // For each axis, assign a counter and configure it:
    S826_CounterExtInRoutingWrite(0, i, dio);       // Route dio to counter's ExtIn input.
    S826_CounterSnapshotConfigWrite(0, i, trig, 0); // Capture counts upon these events.
    S826_CounterModeWrite(0, i, S826_CM_K_QUADX4);  // Counter clock = external quadrature x4.
    S826_CounterStateWrite(0, i, 1);                // Start tracking encoder position.
  }
}
```

The following function will wait for a probe trigger. When a touch is detected, it will populate the POINT structure with the sampled encoder counts and return a code (trig) that indicates what caused the counts to be sampled (trig may be NULL if not needed).

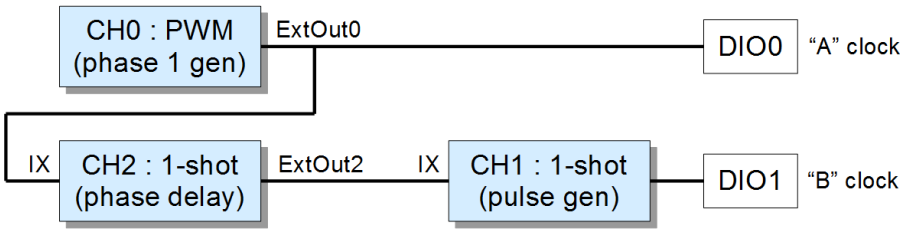
```
int InputTouch(POINT *p, uint *trig)
{
  // Wait for a snapshot, then get X coordinate and snapshot's trigger source.
  int errcode = S826_CounterSnapshotRead(0, 0, &p->x, NULL, trig, S826_WAIT_INFINITE);

  // Get Y and Z coordinates; no need to wait because they've already been triggered.
  if (errcode == S826_ERR_OK) {
    errcode = S826_CounterSnapshotRead(0, 1, &p->y, NULL, NULL, 0);
    if (errcode == S826_ERR_OK)
      errcode = S826_CounterSnapshotRead(0, 2, &p->z, NULL, NULL, 0);
  }
  return errcode;
}
```

Now put it all together. This code will instantiate the probe interface and then repeatedly process touch events:

```
// Implement a high-performance touch probe interface --------

#define PROBE_DIO   17      // The probe signal is connected to this DIO (0-47)
#define PROBE_EDGE  0x20    // Trigger on this probe edge (0x20=falling, 0x40=rising)

POINT point;  // touch coordinates

CreateProbeInterface(PROBE_EDGE, PROBE_DIO); // Establish encoder interface.

// Process triggers from the touch probe:
while (InputTouch(&point, NULL) == S826_ERR_OK)  // Wait for next touch and get coordinates.
  ProcessTouchPoint(coord);                      // TODO: PROCESS TOUCH (e.g., add to point list).
```

7.4.12 Quadrature clock generator

A quadrature clock generator can be implemented with three counters and two DIOs as shown in the example below. Except for DIO load connections, no external wiring is required (counter and DIO interconnects are established by the board's programmable signal router).



Counter0 is a PWM running at 50% duty cycle, which directly generates the quadrature "A" output clock on dio0. Counter2 is used to hold off the "B" generator after "A" rises, to create the 90° phase angle between "A" and "B" clocks. Counter1 is a one-shot that generates the "B" clock on dio1; it does this by issuing a pulse when triggered by Counter2.

This quadrature generator has 1 µs resolution because the counters are using the board's internal 1 MHz clock. If higher resolution is needed, the functions can be modified to use the internal 50 MHz clock.

```
// Generate A/B quadrature clocks ---------

#define PERIOD 600          // Output period in microseconds.
#define HP ((PERIOD) / 2)   // Half-period.
#define QP ((PERIOD) / 4)   // Quarter-period.

CreatePWM(0, 0, HP, HP);      // Generate "A" clock with counter0.
CreateOneShot(0, 2, 2, QP);   // Generate phase shift with counter2.
CreateOneShot(0, 1, 4, HP);   // Generate "B" clock with counter1.
RouteCounterOutput(0, 0, 0);  // Route "A" clock (ExtOut0) to dio0.
RouteCounterOutput(0, 1, 1);  // Route "B" clock (ExtOut1) to dio1.
StartPWM(0, 0);               // Start the quadrature generator.
```

7.4.13 Pulse burst generator

Many applications (e.g., radar, laser) require a fixed number of precisely-timed pulses. This example shows how to generate a burst of pulses with selectable resolution of 20 ns or 1 µs. It uses two counters (counter0 and counter1) to generate the pulse burst and one DIO (dio0) to output the pulses. The DIO must be externally connected to counter1's clkA+ input to allow it to control the pulse count. A snapshot is used to signal burst completion, which allows other threads to run while the burst is in progress. A pulse burst can be triggered by software (by calling OutputBurst) or by an external signal applied to counter1's IX+ input (optional).

This function will configure the hardware as a pulse burst generator:

```
// Create a pulse burst generator.
//  res - timing resolution: set to 1 (microsecond) or 20 (nanoseconds).
void CreateBurstGenerator(uint board, int res)
{
  S826_CounterModeWrite(board, 0,           // Config counter0 as PWM generator:
    ((res == 1) ?                           //   select clock based on resolution:
      S826_CM_K_1MHZ :                      //     1 us -> internal 1 MHz
      S826_CM_K_50MHZ) |                    //     20 ns -> internal 50 MHz
    S826_CM_UD_REVERSE |                    //   count down
    S826_CM_BP_BOTH |                       //   use both preloads (toggle)
    S826_CM_XS_EXTOUT(1) |                  //   connect index input to ExtOut1 (pulse gate)
    S826_CM_PX_IXRISE | S826_CM_PX_ZERO |   //   preload @ index^ and counts=0
    S826_CM_TE_IXRISE |                     //   enable counting upon index^
    S826_CM_TD_IXFALL |                     //   disable counting upon index falling edge
    S826_CM_OM_PRELOAD);                    //   assert ExtOut during preload0 interval

  S826_CounterModeWrite(board, 1,           // Config counter1 as pulse counter (PWM enable):
    S826_CM_K_ARISE |                       //   clock = external clkA (wire from ExtOut0)
    S826_CM_UD_REVERSE |                    //   count down
    S826_CM_TE_PRELOAD |                    //   enable counting upon preload
    S826_CM_PX_IXRISE |                     //   preload @ index^ (allows hardware triggering)
    S826_CM_TD_ZERO |                       //   disable counting upon counts=0
    S826_CM_OM_NOTZERO);                    //   assert ExtOut while counts!=0

  S826_CounterSnapshotConfigWrite(board, 1, // Assert IRQ when a burst has completed.
    S826_SSRMASK_ZERO,
    S826_BITWRITE);

  RouteCounterOutput(board, 0, 0);              // Route PWM output to dio0.
  S826_CounterStateWrite(board, 0, 1);           // Enable PWM generator.
  S826_CounterStateWrite(board, 1, 1);           // Enable the pulse counter.
}
```

After configuring the hardware, call this function to set up the next burst:

```
// Specify attributes of next pulse burst.
//  npulses - number of pulses in burst.
//  period  - pulse period in res units (e.g., when res=20, period=5 -> 5*20ns = 100ns).
//  width   - pulse width in res units (e.g, when res=1, width=4 -> 4*1us = 4us).
void ConfigBurst(uint board, int npulses, uint period, uint width)
{
  S826_CounterPreloadWrite(board, 0, 0, width);   // Set pulse width (in res units).
  S826_CounterPreloadWrite(board, 0, 1,           // Set pulse period (in res units).
    period - width);
  S826_CounterPreloadWrite(board, 0, 0, npulses); // Set pulse count.
  S826_CounterStateWrite(board, 0, 1);            // Enable PWM generator.
  S826_CounterStateWrite(board, 1, 1);            // Enable the pulse counter.
}
```

When the burst generator is ready to operate, call the following function to start the output burst. The function will return when the burst is finished; other threads are allowed to run while the burst is in progress.

```
void OutputBurst(uint board)
{
  S826_CounterPreload(board, 1, 1, 0);     // Trigger a pulse burst.
  S826_CounterSnapshotRead(board, 1, NULL, // Block until burst completes.
    NULL, NULL, S826_WAIT_INFINITE);
}
```

This example shows how to use the above functions to generate a burst of pulses:

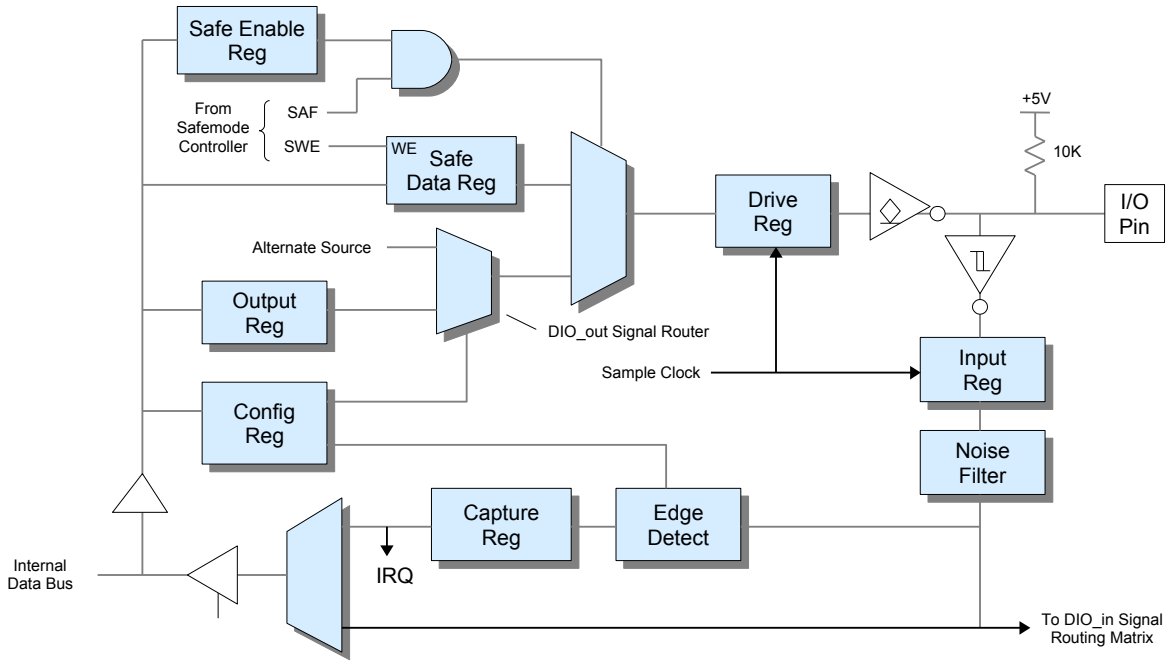
```
CreatePulseGenerator(0, 20); // Create a pulse-burst generator w/20 ns resolution.
ConfigBurst(0, 99, 25, 3);   // Burst attributes: 99 pulses @ 2.0 MHz, pulse width = 60 ns.
OutputBurst(0);              // Start burst and wait for it to complete.
```

Chapter 8: Digital I/O

8.1 Introduction

The 826 board has 48 general-purpose digital I/O (DIO) channels. On the output side, each channel has a one-bit, writable output register, a synchronous drive register, and an inverting open-drain buffer that drives the channel's I/O pin. The open-drain buffer enables the pin to be driven low by the channel's output buffer or by an external signal. The pin, when not driven, is pulled up to +5V by a 10 kohm resistor. The output is high impedance when the board is unpowered so as to prevent unintended activation of an external solid state relay, if one is connected. The pin's physical state is sampled by an input register and then processed by a noise/debounce filter. The filter output is monitored by an edge detector and can also be directly read by the host computer.

Figure 10: DIO channel (1 of 48)



DIO signals are active-high on the local bus and active-low on the I/O pin. Writing a '1' to the output register causes the I/O pin to be driven low, whereas writing '0' allows the pin to be internally pulled up or driven high or low by an external circuit. Logic '0' must be written to the output register if the pin will be driven high by an external circuit; this will prevent high currents that could potentially damage the pin's output buffer.

The value read from a DIO channel indicates the filtered, sampled physical state of the I/O pin. If no external signals are driving the pin then the read value will equal the value stored in the drive register. The read value will differ from the drive register value if the drive register contains '0' while an external circuit drives the pin low.

Most of the host-accessible DIO registers support masked write operations so as to implement the atomic bit set and clear functions required for high performance, thread-safe operation.

8.1.1 Signal router

Each DIO channel connects to the board's internal signal routing matrix as shown in Figure 10. The matrix can be programmed to route the DIO connector pin to or from another interface (e.g., counter, watchdog, analog input system) so that the pin will act as a physical input or output for that interface.

The channel's DIO_out signal router consists of a data selector that can route either the DIO output register or an alternate source to the I/O pin. The pin will function as a general-purpose digital output when the DIO output register is selected. If the alternate source is selected, the pin state will be controlled by the alternate signal, but all DIO input functions (read, edge detection) will continue to operate normally. Each DIO is associated with a specific alternate source as explained in S826_DioOutputSourceWrite.

The DIO_in routing matrix connects the sampled DIO pin signal to other interfaces. The ADC trigger input and the six counter ExtIn inputs are connected to the DIO_in matrix so that any of these signals can be sourced from a DIO pin. When a DIO signal is routed to another interface via the DIO_in matrix, all of the DIO channel's input and output functions will continue to operate normally.

8.1.2 Safemode

Safemode is activated when the SAF signal (see Figure 10) is asserted. When operating in safemode, the DIO pin state is determined by the Safe Enable and Safe Data registers: when Safe Enable equals '1', the pin will be driven to the fail-safe value stored in Safe Data; when Safe Enable equals '0' the pin will output its normal runmode signal.

Upon board reset, the Safe Enable register is set to '1' so that the DIO pin will exhibit fail-safe behavior by default (i.e., it will output the contents of the Safe Data register when SAF is asserted). Fail-safe operation can be disabled for a DIO pin by programming its Safe Enable register to '0'.

The Safe Data register is typically programmed once (or left at its default value) by the application when it starts, though it may be changed at any time if required by the application. It can be written to only when the SWE bit is set to '1'. See “Safemode controller” for more information about safemode.

8.1.3 Edge capture

Every DIO channel includes an edge detection circuit and a capture flag register. When edge capturing is enabled, a channel's capture flag will be set when an edge is detected on its I/O pin. Each channel may be programmed to capture rising edges, falling edges, or both edges, or capturing may be disabled.

The API allows capture flags to be monitored by polling or, if the application is event driven, the calling thread can block while it waits for captured events. When blocking on edge capture events, the calling thread can specify a set of capture flags to wait for, and it can wait for either all of the events or any one event in the set.

When read by the host, capture flags are reset but remain enabled to capture future events. Edge events that occur on a channel while its capture flag is set will be lost. An input signal must hold for at least 20 ns after a transition for the transition to be reliably detected.

8.1.4 Pin timing

The DIO subsystem is a fully synchronous system that is controlled by a 50 MHz sampling clock. The DIO pin drivers are updated and pin receivers are sampled once per cycle. As a result, outputs cannot change faster than the cycle time and inputs cannot be sampled faster than the cycle time.

Output registers are organized as two 24-channel groups. When these registers are written (via S826_DioOutputWrite), channels 0-23 will change simultaneously and channels 24-47 will also change simultaneously, but these two 24-channel groups are not guaranteed to change output states at the same time. Also, a DIO pin does not change output state immediately when its signal source (DIO output register or counter ExtOut) changes; it will be delayed for 20 ns due to the sampling clock.

When used as inputs, all 48 DIO channels are sampled simultaneously every 20 ns. As a result, the received signal on a DIO pin may be delayed up to 20 ns en route to its destination (e.g., DIO edge detector or read data, counter ExtIn input, or ADC trigger input), and input signal pulses shorter than 20 ns may not be recognized. The host reads pin states (via S826_DioInputRead) as two 24-channel groups (channels 0-23 and 24-47) in two separate read cycles. Consequently, channels within each group are guaranteed to be sampled simultaneously, but the two groups are not guaranteed to be associated with the same sample clock.

8.1.4.1 External pull-up resistors

In some cases it may be necessary to connect an external pull-up resistor to a DIO that is operating as an output. This is required if the DIO's native rise-time (200 ns) is too slow. Examples of this include outputting high frequencies or short positive pulses from a counter, generating highly accurate pulse widths, or when fast edges are required by a remote line receiver.

The following table shows nominal rise times for unloaded DIO pins. Lower resistance values may be needed to compensate external circuit capacitance. To avoid excessive DIO output current, do not use an external pull-up resistance less than 220 ohms.

External Pull-up Resistor	DIO Rise Time
None	200 ns
1.2K ohm	20 ns
680 ohm	10 ns

8.1.4.2 Noise filter

Each DIO channel input circuit includes a noise filter that can be used to filter glitches (see S826_DioFilterWrite). A filter's output will change state only when its input has held constant for time T, the filter time interval. Consequently, when a DIO channel's filter is enabled, the sampled input signal will be delayed for an additional T * 20 ns en route to its destination, and input signal pulses shorter than T * 20 ns will not be recognized.

8.1.5 Reset state

DIO channels are forced to the following condition upon board reset:

- Output and Safe Data registers programmed to zero.
- Safe Enable registers programmed to all 1's.
- Output register is selected as I/O pin data source.
- Event capture disabled.

8.2 Connectors J2/J3

J2 Pinout - DIO channels 24-47

Pin	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	Even
DIO Channel	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	30	30	29	28	27	26	25	24	+5V	GND

J3 Pinout - DIO channels 0-23

Pin	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45	47	49	Even
DIO Channel	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	+5V	GND

All even pin numbers on J2 and J3 connect to the power supply return. Odd pin numbers 1-47 are the active-low DIO channel I/O pins. Pin 49 is a +5V power output for low power loads such as solid state relay racks. The total +5V output current for all external circuitry (including DIO and counter connectors) is limited to 400 mA. Note that these pins do not have independent short circuit protection.

8.3 API functions

The DIO functions use individual bits to convey information about the DIO channels, wherein each bit represents the information for one channel. In such cases, the information for the 48 DIO channels is organized as an array of two bit quadlets (32-bit values), with each quadlet containing the information for 24 DIO channels:

The quadlet at array[0] is associated with DIO channels 0 to 23:

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIO	-	-	-	-	-	-	-	-	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The quadlet at array[1] is associated with DIO channels 24 to 47:

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIO	-	-	-	-	-	-	-	-	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24

Typically, the DIO functions expect a pointer to a two-quadlet array, which must have been previously allocated by the program.

Although the DIO functions read or write values for all 48 DIO channels, the physical read or write operation is performed in steps; channels 0 to 23 first, as a group, and then channels 24 to 47 as another group. As a result, reads and writes do not sample or update all channels simultaneously. In general, channels 0 to 23 are sampled or updated concurrently as a group, and channels 24 to 47 are sampled or updated concurrently as a separate group.

8.3.1 S826_DioOutputWrite

The S826_DioOutputWrite function programs the DIO output registers.

```
int S826_DioOutputWrite(
  uint board,     // board identifier
  uint data[2],   // pointer to DIO data
  uint mode       // 0=write, 1=clear bits, 2=set bits
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to data array (see Section 8.3).

mode
Write mode for data: 0 = write, 1 = clear bits, 2 = set bits (see “Atomic read-modify-write”).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

In mode zero, this function will unconditionally write new values to all DIO output registers. In modes one and two, the function will selectively clear or set any arbitrary combination of output registers; data bits that contain logic '1' indicate DIO output registers that are to be modified, while all other output registers will remain unchanged. Modes one and two can be used to ensure thread-safe operation as they atomically set or clear the specified bits.

8.3.2 S826_DioOutputRead

The S826_DioOutputRead function reads the programmed states of all DIO output registers.

```
int S826_DioOutputRead(
  uint board,    // board identifier
  uint data[2]   // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to a buffer (see Section 8.3) that will receive the output register states.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function returns the output register states. Note that the returned values may not be the same as the physical I/O pin states in the case of pins that are externally driven or routed to a counter channel's output signal.

8.3.3 S826_DioInputRead

The S826_DioInputRead function reads the physical states of all DIO channel I/O pins.

```
int S826_DioInputRead(
  uint board,    // board identifier
  uint data[2]   // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to a buffer (see Section 8.3) that will receive the physical I/O pin states.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function returns the sampled physical states of all DIO pins in the data buffer.

If this function is called immediately after calling S826_DioOutputWrite, the read data (sampled pin states) may not accurately reflect the previously written data. This happens because the DIO pins are driven by open-drain buffers with pull-up resistors. A pin will change state quickly when driven low, but when it is switched high, the state cannot change as quickly because additional time is required for the circuit capacitance to be charged through the pull-up resistor. The amount of time required for this depends on circuit capacitance; it can be shortened by decreasing the capacitance or by decreasing the pull-up resistance (by adding an external pull-up resistor).

8.3.4 S826_DioSafeWrite

The S826_DioSafeWrite function programs the DIO Safe registers.

```
int S826_DioSafeWrite(
  uint board,     // board identifier
  uint data[2],   // pointer to safemode data
  uint mode       // 0=write, 1=clear bits, 2=set bits
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to data array (see Section 8.3) to be programmed into the Safe registers.

mode
Write mode for data: 0 = write, 1 = clear bits, 2 = set bits (see "Atomic read-modify-write").

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

8.3.5 S826_DioSafeRead

The S826_DioSafeRead function returns the contents of the DIO Safe registers.

```
int S826_DioSafeRead(
  uint board,    // board identifier
  uint data[2]   // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to a buffer (see Section 8.3) that will receive the Safe register states.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

8.3.6 S826_DioSafeEnablesWrite

The S826_DioSafeEnablesWrite function programs the DIO Safe Enable registers.

```
int S826_DioSafeEnablesWrite(
  uint board,       // board identifier
  uint enables[2]   // pointer to safemode enables
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enables
Pointer to array of values (see Section 8.3) to be programmed into the Safe Enable registers.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

8.3.7 S826_DioSafeEnablesRead

The S826_DioSafeEnablesRead function returns the contents of the DIO Safe Enable registers.

```
int S826_DioSafeEnablesRead(
  uint board,      // board identifier
  uint enables[2]  // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enables
Pointer to a buffer (see Section 8.3) that will receive the Safe Enable register contents.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

8.3.8 S826_DioCapEnablesWrite

The S826_DioCapEnablessWrite function programs the edge sensitivity for DIO edge capturing.

```
int S826_DioCapEnablesWrite(
  uint board,       // board identifier
  uint rising[2],   // pointer to data buffer
  uint falling[2],  // pointer to data buffer
  uint mode         // write mode
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

rising
Pointer to a buffer (see Section 8.3) that specifies rising edge sensitivity: '1'=capture rising edges, '0'=don't capture rising edges.

falling
Pointer to a buffer (see Section 8.3) that specifies falling edge sensitivity: '1'=capture falling edges, '0'=don't capture falling edges.

mode
Write mode for rising/falling: 0 = write, 1 = clear bits, 2 = set bits (see “Atomic read-modify-write”).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function specifies the edges to be captured on DIO channels. It defines the edge sensitivity for each DIO channel in terms of the I/O pin voltage. For example, if falling edge sensitivity is enabled, edge capturing will occur when the I/O pin voltage transitions from 5V to 0V.

When mode is zero, all data flags are directly written to the capture enable registers. In modes one and two, the data flags determine which of the 48 DIO channels are to be affected; any flag that contains a logic '1' will cause the associated channel to be affected, while '0' will leave the channel unmodified.

After capturing has been enabled, it will remain enabled until disabled by this function or a board reset. Capturing is disabled on all channels following a board reset.

8.3.9 S826_DioCapEnablesRead

The S826_DioCapEnablesRead function returns the programmed edge sensitivity for DIO edge capturing.

```
int S826_DioCapEnablesRead(
  uint board,       // board identifier
  uint rising[2],   // pointer to data buffer
  uint falling[2]   // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

rising
Pointer to a buffer (see Section 8.3) that receives rising edge sensitivity: '1'=capture rising edges, '0'=don't capture rising edges.

falling
Pointer to a buffer (see Section 8.3) that receives falling edge sensitivity: '1'=capture falling edges, '0'=don't capture falling edges.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function receives information about the types of edge events that will be captured, which were previously programmed by calling This function returns the sampled physical states of all DIO pins in the data buffer..

8.3.10 S826_DioCapRead

The S826_DioCapRead function waits for edge events on one or more DIO channels.

```
int S826_DioCapRead(
  uint board,         // board identifier
  uint chanlist[2],   // pointer to channel flags
  uint waitall,       // logic operator: 1=and (all), 0=or (any)
  uint tmax           // maximum time to wait
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chanlist
Pointer to channel flag bits (see Section 8.3). Upon entry, set flags indicate channels of interest. Upon exit, set flags indicate channels with captured edges.

waitall
Logic operator to apply to channels of interest: 1 = AND (return when all channels have events), 0 = OR (return when any channel has an event). This is ignored if tmax = 0.

tmax
Maximum time, in microseconds, to wait for the events of interest. See “Event-driven applications“ for details.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function waits for edge events to be captured on an arbitrary set of DIO channels (the “channels of interest”) and then returns information about the events. Event capturing must have been previously enabled for channels of interest by calling S826_DioCapEnablesWrite.

Before calling the function, one or more chanlist bits must be set to identify the channels of interest. The function will modify chanlist to indicate channels of interest that have captured events, and it will reset the capture flag registers for all such indicated channels, thus re-enabling event capturing on those channels.

The function operates in either blocking or non-blocking mode depending on the value of tmax. If tmax is zero (non-blocking mode), the function will return immediately. If tmax is greater than zero (blocking mode), the calling thread will block until the capture criteria is satisfied or tmax elapses. In either case, some channels of interest may have captured events while others may not have; these will be indicated by chanlist when the function returns.

In blocking mode, the capture criteria is specified by waitall. Depending on waitall, the function will wait for events to be captured on either any, or all channels of interest. When waitall is true, the function will return when all channels of interest have captured events. When waitall is false, the function will return when any channel of interest has captured an event. The function will return S826_ERR_NOTREADY if tmax elapses before the capture criteria is satisfied.

In non-blocking mode, waitall is ignored and the function will never return S826_ERR_NOTREADY.

This function will return S826_ERR_CANCELLED if, while it is blocking, another thread calls S826_DioWaitCancel to cancel waits on any of the blocking channels. It will return immediately if the wait criteria is completely satisfied due to the wait cancellations, otherwise it will continue to block and return when all remaining wait criteria is satisfied. S826_ERR_BOARDCLOSED will be returned immediately if S826_SystemClose executes while this function is blocking. In either case, chanlist will be invalid when the function returns.

Thread-safe operation is guaranteed if the channels of interest for any given thread do not coincide with those of another thread. For example, thread safety is assured if a thread designates channels 1 and 3-5 as channels of interest while another thread designates channels 2 and 9.

8.3.11 S826_DioWaitCancel

The S826_DioWaitCancel function cancels a blocking wait on one or more DIO channels.

```
int S826_DioWaitCancel(
  uint board,          // board identifier
  uint chanlist[2]     // pointer to channel flags
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

chanlist
Pointer to DIO channel flag bits (see Section 8.3) that indicate channels for which waiting is to be cancelled.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function cancels blocking for an arbitrary set of DIO channels so that another thread, which is blocked by S826_DioCapRead while waiting for DIO edge events to be captured, will return immediately with S826_ERR_CANCELLED.

8.3.12 S826_DioOutputSourceWrite

The S826_DioOutputSourceWrite function assigns the signal sources for all DIO pins.

```
int S826_DioOutputSourceWrite(
  uint board,      // board identifier
  uint data[2]     // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to a buffer (see Section 8.3) that specifies signal sources: 0=DIO output register, 1=alternate source.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function specifies the signal source that will be used to drive each DIO pin. Each DIO pin may be driven by its output register or by its alternate signal source. Upon board reset, all DIO channels assume their default configuration so that all DIO pins are driven by the DIO output registers (vs. alternate sources).

A DIO pin's signal source is determined by the corresponding bit in the data buffer. When the bit is set to '1' the pin will be driven by the channel's alternate signal source; when set to '0' (default) the pin will behave as a standard digital output, driven by the channel's output register.

The data[0] quadlet selects the signal sources for DIO channels 0 to 23:

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIO	-	-	-	-	-	-	-	-	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Src	-	-	-	-	-	-	-	-	NMI	RST	C5	C4	C3	C2	C1	C0	NMI	RST	C5	C4	C3	C2	C1	C0	NMI	RST	C5	C4	C3	C2	C1	C0

The data[1] quadlet selects the signal sources for DIO channels 24 to 47:

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIO	-	-	-	-	-	-	-	-	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24
Src	-	-	-	-	-	-	-	-	NMI	RST	C5	C4	C3	C2	C1	C0	NMI	RST	C5	C4	C3	C2	C1	C0	NMI	RST	C5	C4	C3	C2	C1	C0

Every DIO channel is associated with one of eight alternate signal sources as shown in the above tables. The tables use the following abbreviations for alternate signal sources:

Symbol	Signal Source
C0	Counter 0 ExtOut
C1	Counter 1 ExtOut
C2	Counter 2 ExtOut
C3	Counter 3 ExtOut
C4	Counter 4 ExtOut
C5	Counter 5 ExtOut
RST	Watchdog RST (Timer2) output
NMI	Watchdog NMI (Timer1) output

Examples:

- When data[1] bit 2 is set, DIO26 will be driven by the ExtOut signal from counter channel 2.
- When data[0] bit 14 is set, DIO14 will be driven by the Watchdog RST output signal.

Each of the eight alternate signal sources is associated with six DIO channels. For example, the watchdog RST signal is associated with DIO channels, 6, 14, 22, 30, 38, and 46. Typically, an alternate source is either not used or it is routed to one of its associated DIO pins, though it may be simultaneously routed to any combination of its associated pins.

This function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

8.3.13 S826_DioOutputSourceRead

The S826_DioOutputSourceRead function reads the signal sources assigned to all DIO channels.

```
int S826_DioOutputSourceRead(
  uint board,    // board identifier
  uint data[2]   // pointer to data buffer
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Pointer to a buffer (see Section8.3) that will receive the signal source assignments for all DIO channels.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

8.3.14 S826_DioFilterWrite

The S826_DioFilterWrite function configures the DIO input noise filters.

```
int S826_DioFilterWrite(
  uint board,       // board identifier
  uint interval,    // filter interval (T)
  uint enables[2]   // filter enable flags
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

interval
Filter time interval in range 1 to 65535, specified as a multiple of 20ns. This is common to all DIO channels.

enables
Pointer to a buffer (see Section 8.3) that specifies filter enables for the DIO channels: '1'=enable, '0'=disable.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

Each DIO has an input noise filter that can be independently enabled or disabled. This function selectively enables and disables the individual filters and programs the filter time interval T, which is common to all DIO filters.

A filter's output will not change state until its input has held a constant state for T * 20ns. The maximum T value is 65535, corresponding to a filter time of 1.3107ms. When a DIO channel's noise filter is enabled, its input signal will be delayed by T * 20ns and input pulses shorter than T * 20ns will not be recognized.

8.3.15 S826_DioFilterRead

The S826_DioFilterRead function reads the configuration of the DIO input noise filters.

```
int S826_DioFilterRead(
  uint board,       // board identifier
  uint *interval,   // filter interval (T)
  uint enables[2]   // filter enable flags
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

interval
Pointer to a buffer that will receive the filter time interval as described in S826_DioFilterWrite.

enables
Pointer to a buffer (see Section 8.3) that will receive filter enable flags for the DIO channels: '1'=enable, '0'=disable.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

8.4 Application notes

8.4.1 Basic operation

To unconditionally program all DIO outputs, call S826_DioOutputWrite with the mode argument set to S826_BITWRITE (zero). For example, this code will turn on all DIOs:

```
uint dios[] = { // Specify DIOs that are to be turned on (driven to 0 V):
  0x00FFFFFF,   //   DIO 0-23
  0x00FFFFFF    //   DIO 24-47
};
S826_DioOutputWrite(0, dios, S826_BITWRITE); // Turn on all DIOs.
```

This example shows how to turn on DIOs 7, 13 and 38 and turn off all other DIOs:

```
uint dios[] = {         // Specify DIOs that are to be turned on:
  (1 << 7) + (1 << 13), //   DIOs 7 & 13 are in first 24-bit mask (DIOs 0-23),
  (1 << (38 - 24))      //   DIO 38 is in second 24-bit mask (DIOs 24-47).
};
S826_DioOutputWrite(0,  // Program all DIO outputs on board 0:
  dios,                 //   desired output states
  S826_BITWRITE);       //   unconditionally change all DIOs
```

To read the pin levels of all DIOs, call S826_DioInputRead. This example shows how to read and display all DIO pins on board 0:

```
int i;
uint pins[2];                // Buffer for pin states.
S826_DioInputRead(0, pins);  // Read all DIO pin states into buffer.
for (i = 0; i < 24; i++)     // Display states of DIOs 0-23.
  printf("dio%d = %d\n", i, (pins[0] >> i) & 1);
for (i = 24; i < 48; i++)    // Display states of DIOs 24-47.
  printf("dio%d = %d\n", i, (pins[1] >> (i - 24)) & 1);
```

8.4.2 Waiting for stable outputs

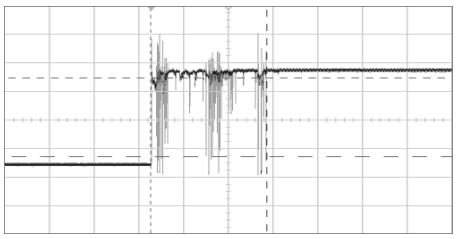
In some cases (e.g., when DIOs are driving circuits with high capacitance) it may be necessary to wait for DIO pins to transition to their programmed states before proceeding. This can be done by calling S826_DioOutputWrite and then polling via S826_DioInputRead until the programmed states are reached:

```
uint pins[2];          // pin states
uint dios[] = {3, 0};  // select DIO0 & DIO1
S826_DioOutputWrite(0, dios, S826_BITWRITE);   // Turn on DIO0 and DIO1 (turn off all other DIOs).
do {
  S826_DioInputRead(0, pins);                  // Wait for output signals to stabilize.
} while (memcmp(pins, dios, sizeof(pins)));
```

8.4.3 Debouncing inputs

DIOs are commonly used to monitor real-world signals that have glitches or other noise. For example, the electrical contacts of a mechanical switch may "bounce" when first connected, thus producing pulses that may be incorrectly interpreted by the application program as state changes. This problem can be usually be solved by implementing a software debounce algorithm, but there's an easier way: use the input filters provided by the DIO system.

To use the DIO input filters, simply specify a filter interval and designate the DIOs to be filtered. Choose a filter interval that is long enough to suppress the longest expected noise pulse, but not so long that valid state changes will be missed. The following example shows how to suppress pulses up to 200 µs wide on dio0 and dio2:



Oscilloscope trace showing voltage "bounce" when a switch turns on

```
// Debounce dio0 and dio2 ------------------

#define FILT_USEC 200                  // Desired filter
interval (max = 1.31 ms).
uint tfilt = FILT_USEC * 50;           // Interval specified as
multiple of 20ns.
uint enabs[] = {5, 0};                 // Enable filters on dio0 and dio2.
S826_DioFilterWrite(0, tfilt, enabs);  // Activate the DIO filters.
```

8.4.4 Setting and clearing DIOs

Sometimes it's necessary to set or clear particular DIOs without disturbing others. Typically, this is accomplished with a *read-modify-write* (RMW) sequence consisting of reading all DIOs into a buffer, modifying the buffer, and then writing the buffer back to the DIOs. This task is further complicated when multiple threads control the DIOs — a common practice in high-performance event-driven applications — because concurrent RMWs on different threads will conflict and cause DIO control errors. To avoid such conflicts, the RMW must be treated as a critical section (e.g., protected by a mutex or semaphore).

Fortunately, the 826 provides a fast, efficient way to set and clear DIOs without the complications of RMW. Simply call S826_DioOutputWrite with the mode argument set to S826_BITSET or S826_BITCLR, as shown in this example:

```
// Set and clear some DIOs without affecting other DIOs

uint maskA[] = {7, 0};  // bitmask for DIOs 0-2
uint maskB[] = {0, 1};  // bitmask for DIO 24
S826_DioOutputWrite(0, maskA, S826_BITSET);  // Set DIOs 0, 1 and 2.
S826_DioOutputWrite(0, maskB, S826_BITCLR);  // Clear DIO 24.
```

8.4.5 Edge detection

Event-driven application programs often must execute code in response to DIO input changes (e.g., when a button is pressed, a photoelectric sensor is activated, etc.). It's usually considered bad practice to detect these events via polling because polling wastes CPU time and degrades system responsiveness. However, the alternative — using interrupts — can complicate and prolong program development.

Model 826 provides the best of both worlds: it supports DIO interrupts for efficient event detection, and the API makes the interrupts easy to use. For example, the following function will return when a falling edge is detected on a particular DIO. It is a blocking function, meaning that other threads can run while it waits for the DIO edge.

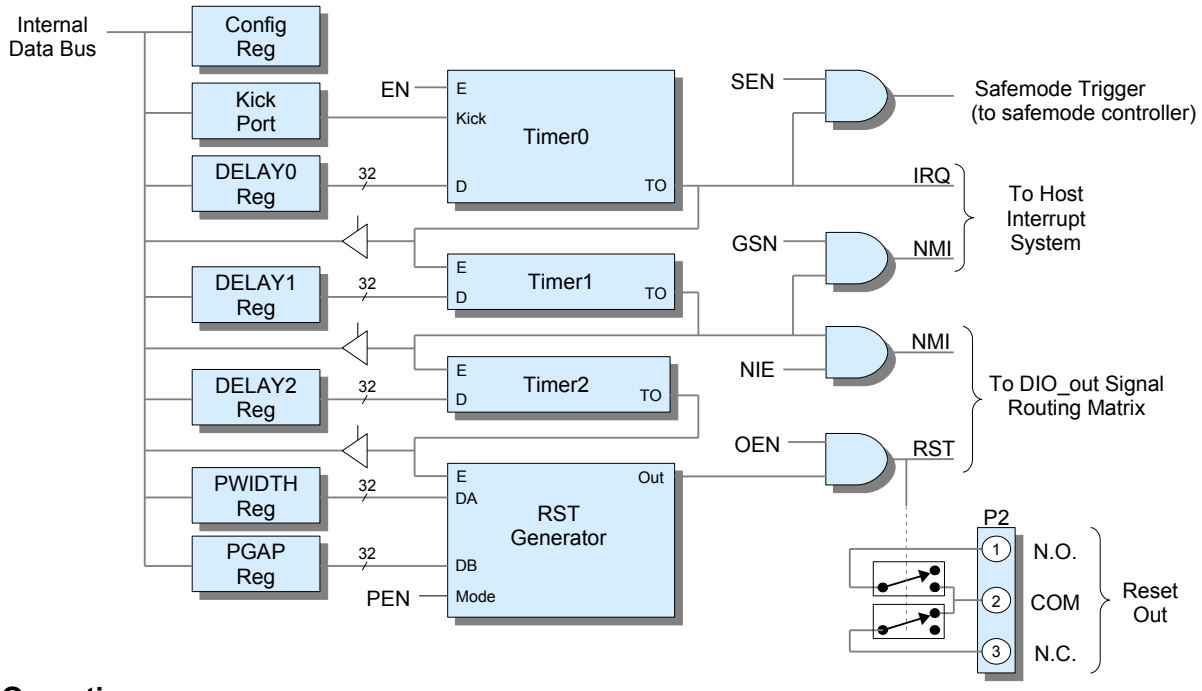
```
int WaitForDioFallingEdge(uint board, uint dio)
{
  uint rise[] = {0, 0};       // not interested in rising edge
  uint fall[] = {(dio < 24) << (dio % 24), (dio > 23) << (dio % 24)}; // interested in falling edge
  S826_DioCapEnablesWrite(board, rise, fall, S826_BITSET);   // Enable falling edge detection.
  return S826_DioCapRead(board, fall, 0, INFINITE_WAIT);     // Block until falling edge.
}

WaitForDioFallingEdge(0, 29);  // Example usage: wait for falling edge on dio29 of board0.
```

Chapter 9: Watchdog timer

9.1 Introduction

The model 826 board has a multistage watchdog timer that can activate the board's safemode system and generate service requests. The watchdog has three timer stages that generate timeout events in sequence according to user-defined timing. Each event is associated with an output signal. Typically, the event signals are utilized in such a way that successive events will generate service requests of progressively higher priority.

Figure 11: Watchdog system



9.1.1 Operation

When the watchdog system is disabled (default upon board reset), the three timers are halted and loaded from their associated DELAY registers. When the system becomes enabled (EN='1') by calling S826_WatchdogEnableWrite, initially only Timer0 is enabled so that it will count down towards zero while the other timers remain idle. During normal operation, the program regularly “kicks” the watchdog by writing to the Kick port, thus reloading Timer0 from DELAY0 before it can count down to zero.

If a fault condition prevents the program from kicking the watchdog, Timer0 will count down to zero and assert its timeout (TO) signal. After this event occurs, all subsequent kicks will be ignored. This event enables Timer1 and generates an interrupt request, and if SEN='1', it switches all control outputs to fail-safe states by triggering the safemode system. The program can wait for the interrupt by calling S826_WatchdogEventWait.

Timer1 asserts its TO signal upon counting down to zero. This event enables Timer2 and, if NIE='1', it asserts the NMI net of the DIO_out signal routing matrix, which in turn may route the net to a DIO pin (see S826_DioOutputSourceWrite). When GSN='1', a Timer1 event will cause the board to issue a PCI Express fatal error message; this can be used to generate a system non-maskable interrupt request if the system has been appropriately configured. Refer to your system documentation for information about generating NMI in response to a PCI Express fatal error message.

Timer2 asserts its TO signal upon counting down to zero, thus activating the RST signal generator. If OEN='1', the RST generator's output is routed to the RST net of the DIO_out signal routing matrix and to the board's Reset Out circuit. The RST generator can produce a continuous (non-pulsed) output or pulsed output. When generating a pulsed output, PWIDTH

determines the pulse duration and PGAP determines the gap time between pulses. The RST signal is not internally connected to the host computer's system reset input; if desired, this must be implemented by externally routing the selected DIO pin to the computer's reset input.

9.1.2 Reset Out circuit

Two solid state relays (SSRs) are provided for controlling external reset circuits. Both SSRs are energized when the watchdog RST generator is asserting its output signal. One SSR has normally open contacts and the other normally closed contacts. The SSRs are galvanically isolated from other board circuitry.

9.1.3 Initialization

Before enabling the watchdog, the program must initialize it by writing to the configuration register and the five timing control registers (DELAY0-DELAY2, PWIDTH, and PGAP). This is done by calling S826_WatchdogConfigWrite. The DELAY registers determine the time intervals of their three associated timers, whereas the PWIDTH and PGAP registers determine the timing of the RST output signal.

9.2 Connector P2

Connector P2 is a three-pin connector that interfaces external circuitry to the watchdog's Reset Out solid state relay. Refer to the block diagram in section 9.1 for connector pinout. These connectors (not included) will mate to P2:

- Molex 22-01-2037 (ramp only)
- Molex 22-01-3037 (ramp + alignment ribs)

9.3 API functions

9.3.1 S826_WatchdogConfigWrite

The S826_WatchdogConfigWrite function configures the watchdog system.

```
int S826_WatchdogConfigWrite(
  uint board,     // board identifier
  uint cfg,       // configuration flags
  uint timing[5]  // time intervals
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

cfg
Configuration flags: '1' = enable feature, '0' = disable feature.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	GSN	0	SEN	NIE	PEN	0	OEN

Flag	Function
GSN	Generate host system NMI upon Timer1 event.
SEN	Activate safemode upon Timer0 event.
NIE	Connect Timer1 event signal to the DIO_out routing matrix NMI net.
PEN	Enable RST output to pulse (vs. continuous active level).
OEN	Connect RST generator to the DIO_out routing matrix RST net.

timing
Pointer to array of five quadlets that define the watchdog's time intervals. Each quadlet is written to one of the watchdog timing control registers as shown below. All times are specified as multiples of 20 nanoseconds. For example, use the value 50,000,000 for a one-second time interval.

Quadlet	Register	Function
timing[0]	DELAY0	Timer0 interval. The program must kick the watchdog within this interval to prevent a watchdog timeout. This must be set to a non-zero value; set to 1 for shortest possible delay.
timing[1]	DELAY1	Timer1 interval. This specifies the elapsed time from Timer1 timeout to Timer2 timeout. This must be set to a non-zero value; set to 1 for shortest possible delay.
timing[2]	DELAY2	Timer2 interval. This specifies the elapsed time from Timer2 timeout to RST generator enable. This must be set to a non-zero value; set to 1 for shortest possible delay.
timing[3]	PWIDTH	RST pulse width. This is ignored if PEN='0'.
timing[4]	PGAP	Time gap between RST pulses. This is ignored if PEN='0'.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function programs the watchdog configuration register and timing control registers. To ensure reliable operation, it should be called only when the watchdog is disabled.

The function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

9.3.2 S826_WatchdogConfigRead

The S826_WatchdogConfigRead function reads the watchdog configuration.

```
int S826_WatchdogConfigRead(
  uint board,     // board identifier
  uint *cfg,      // configuration flags
  uint timing[5]  // time intervals
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

cfg
Pointer to buffer that will receive the watchdog configuration flags.

timing
Pointer to array of five quadlets that will receive the watchdog time intervals.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

See Section 9.3.1 for details about the returned cfg and timing values.

9.3.3 S826_WatchdogEnableWrite

The S826_WatchdogEnableWrite function enables or disables the watchdog system.

```
int S826_WatchdogEnableWrite(
  uint board,    // board identifier
  uint enable    // enable watchdog when true
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enable
Set to '1' to enable, or '0' to disable the watchdog. The watchdog is disabled by default upon board reset.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

9.3.4 S826_WatchdogEnableRead

The S826_WatchdogEnableRead function returns the enable status of the watchdog system.

```
int S826_WatchdogEnableRead(
  uint board,    // board identifier
  uint *enable   // enable status
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

enable
Buffer that will receive the watchdog system enable status: '1' = enabled, '0' = disabled.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

9.3.5 S826_WatchdogStatusRead

The S826_WatchdogStatusRead function reads the watchdog timeout status.

```
int S826_WatchdogStatusRead(
  uint board,       // board identifier
  uint *status      // watchdog status
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

status
Pointer to a quadlet buffer that will receive the watchdog status. Each status bit indicates the timeout status of one watchdog timer stage ('1' = timed out):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	TO2	TO1	TO0

Flag	Function
TO2	Timer2 timeout
TO1	Timer1 timeout
TO0	Timer0 timeout

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

9.3.6 S826_WatchdogKick

The S826_WatchdogKick function reads the watchdog timeout status.

```
int S826_WatchdogKick(
  uint board,      // board identifier
  uint data        // valid signature
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

data
Valid signature. This must be 0x5A55AA5A to kick the watchdog; any other value will fail to kick the watchdog, although no error will be returned.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

When the watchdog system is running, the program should call this function to prevent a watchdog timeout. The DELAY0 value determines how often this function must be called to prevent a timeout.

9.3.7 S826_WatchdogEventWait

The S826_WatchdogEventWait function waits for a watchdog event (timeout) on Timer1.

```
int S826_WatchdogEventWait(
  uint board,     // board identifier
  uint tmax       // maximum time to wait
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

tmax
Maximum time, in microseconds, to wait for data. See “Event-driven applications” for details.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function can operate in either blocking or non-blocking mode. If tmax is zero, the function will return immediately. If tmax is greater than zero, the calling thread will block until a watchdog event or tmax elapses. The function will return zero if a watchdog timeout event occurred, or S826_ERR_NOTREADY if the watchdog has not timed out.

When this function is blocking, it will return immediately with return code S826_ERR_CANCELLED if S826_WatchdogWaitCancel is called by another thread, or with S826_ERR_BOARDCLOSED if S826_SystemClose is called by another thread.

9.3.8 S826_WatchdogWaitCancel

The S826_WatchdogWaitCancel function cancels a blocking wait on watchdog Timer1.

```
int S826_WatchdogWaitCancel(
  uint board    // board identifier
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function cancels blocking on the watchdog so that another thread, which is blocked by S826_WatchdogEventWait while waiting for a watchdog timeout event, will return immediately with S826_ERR_CANCELLED.

9.4 Application notes

9.4.1 Kicking the watchdog

After the application program has started (enabled) the watchdog timer, it must regularly "kick" the watchdog to prevent a timeout. This is typically done by periodically executing a kick algorithm. The kick algorithm may be simple or complex, depending on the number of running threads and other factors. The simplest algorithm will simply kick the watchdog, unconditionally:

```
CreateTimer(0, 0, 100000);  // Execute this loop 10 times per second:
while (1) {
  S826_WatchdogKick(0, 0x5A55AA5A);  // Unconditionally kick the watchdog.
  WaitForTimer(0, 0);
}
```

Here's a slightly more complex version that monitors two other threads (threadA and threadB). When each monitored thread completes its task, it stores a special value in a reserved memory location. The special values, when OR'ed together, form the value required for a watchdog kick.

```
int a_kick;  // ThreadA stores 0x5A550000 here when it completes.
int b_kick;  // ThreadB stores 0x0000AA5A here when it completes.

CreateTimer(0, 0, 100000);  // Execute this loop 10 times per second:
while (1) {
  S826_WatchdogKick(0, a_kick | b_kick);  // Kick watchdog if both threads completed.
  a_kick = b_kick = 0;                    // Reset completion status.
  WaitForTimer(0, 0);
}
```

When watchdog timer0 times out, it may be necessary to notify a “system health” monitoring thread so it can take appropriate corrective action. This is easily done, as shown in the following code:

```
if (S826_WatchdogEventWait(0, INFINITE_WAIT) == S826_ERR_OK) {
  // TODO: Watchdog timer0 timed out, so take appropriate corrective action
}
```

9.4.2 Activating safemode with the watchdog

In many control applications, the analog and digital outputs must automatically switch to safe states if software fails to execute normally. This can be implemented by using watchdog Timer0 to activate safemode. To set this up, configure the watchdog and safemode systems during initialization (before I/O operations begin):

```
#define WD_MILLISECONDS 100  // Watchdog Timer0 will timeout if unkicked for this long
wdtiming[] = {WD_MILLISECONDS * 50000, 1, 1, 0, 0};

S826_SafeWrenWrite(0, S826_SAFEN_SWE);       // Write-enable watchdog/safemode settings.
S826_WatchdogConfigWrite(0, 0x10, wdtiming); // Set t0 interval; t0 triggers safemode.

// TODO: Program safemode states (see Section 10.3.1)

S826_WatchdogEnableWrite(0, 1);              // Start the watchdog running.
S826_SafeWrenWrite(0, S826_SAFEN_SWD);       // Write-protect watchdog/safemode settings.
```

The above code starts the watchdog timer, so the application program must now regularly "kick" the watchdog to prevent a timeout.

9.4.3 Output watchdog on a DIO

The outputs of watchdog Timer1 and Timer2 can be routed to select DIOs (see Section 8.3.12):

- Timer1 can be routed to DIO 7, 15, 23, 31, 39 and 47.
- Timer2 can be routed to DIO 6, 14, 22, 30, 38 and 46.

In the following example, DIO7 is connected to the output of Timer1 so that it will be turned on (driven low, to 0 V) when Timer1 times out. As always, Timer0 is used to time the kicks. Timer1 is assigned a minimum delay (DELAY1=1) so that it will timeout (and thereby activate dio7) one clock (20 ns) after timer0 times out.

```
#define WD_MILLISECONDS 100  // Watchdog Timer0 will timeout if unkicked for this long

wdtiming[] = {WD_MILLISECONDS * 50000, 1, 1, 0, 0};  // timing parameters
uint routing[] = {1 << 7, 0};                        // mask for dio7

S826_SafeWrenWrite(0, S826_SAFEN_SWE);   // Write-enable protected settings.
S826_WatchdogConfigWrite(0,              // Configure watchdog:
  S826_WD_NIE,                           //   connect timer1 to NMI net
  wdtiming);                             //   set timer0 & timer1 intervals
S826_DioOutputSourceWrite(0, routing);   // Route NMI net to dio7.
S826_WatchdogEnableWrite(0, 1);          // Start the watchdog (AND START KICKING!)
S826_SafeWrenWrite(0, S826_SAFEN_SWD);   // Write-disable protected settings.
```

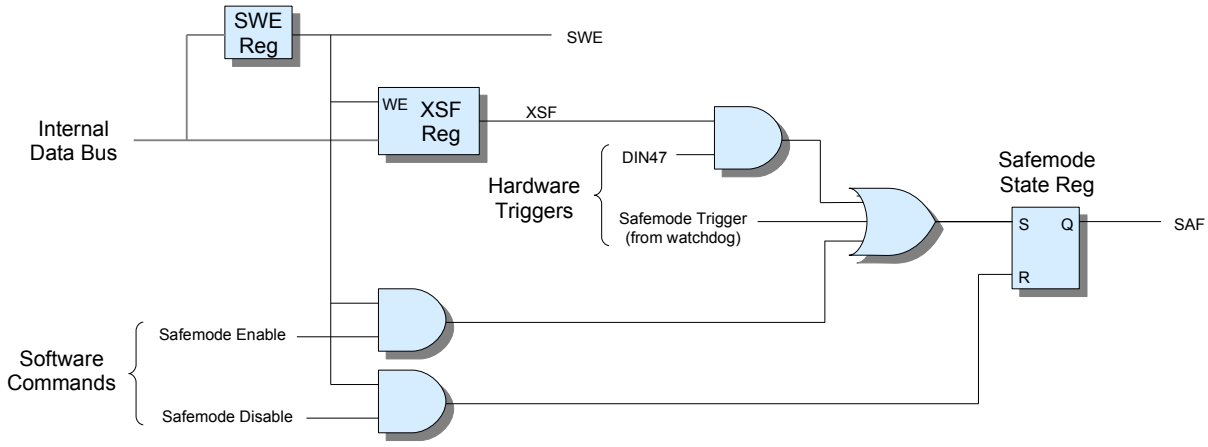
The above code starts the watchdog timer, so the application program must now regularly "kick" the watchdog to prevent a timeout.

Chapter 10: Safemode controller

10.1 Introduction

The 826 board features a fail-safe controller that forces analog and digital outputs to predetermined levels in response to hardware triggers. The controller works in concert with the watchdog timer and external devices such as emergency shutdown contacts to switch the board's outputs to fail-safe levels without software intervention.

Figure 12: Safemode Controller



The controller consists of configuration (XSF) and write protection control (SWE) registers, triggering logic, and a state register. When safemode is active (SAF = '1'), the board's analog and digital outputs are automatically switched to their fail-safe states. SAF can be set by the program and in response to hardware triggers, but only the program can reset SAF to turn off safemode. Upon power-up or board reset, SAF is reset.

If the watchdog is allowed to activate safemode (see S826_WatchdogConfigWrite), it will assert the Safemode Trigger signal upon Timer0 event, thus setting SAF. Once asserted, the trigger will remain asserted until the watchdog is disabled. Consequently, the program cannot reset SAF until the watchdog is disabled.

When XSF = '1', safemode can be triggered by an active-low signal applied to the DIO channel 47 connector pin (DIO47). When this happens, the program cannot reset SAF until DIO47 is negated or XSF is cleared.

Additional information about safemode can be found in Section 6.1.1 (analog outputs) and Section 8.1.2 (DIO outputs).

10.1.1 Write protection

The SWE register controls write protection for registers associated with the watchdog and safemode controller. All affected registers are write-protected when SWE = '0'; this is the default state of SWE at power-up and upon system reset. The SWE register state does not change when the board is opened or closed.

Before writing to protected registers, the program must set SWE (by calling S826_SafeWrenWrite) to allow writes to the registers. During initialization, the program will typically disable write protection, write all fail-safe states as required by the application, and then re-enable write protection to prevent modification of the registers due to subsequent wayward software execution.

Several of the API functions write to SWE protected registers. These functions can fail without notification if called while SWE = '0' (they will return S826_ERR_OK if no other errors are detected, but the protected register will not be written). If it is necessary to detect a failed write to a write-protected register, the program should read the register after writing to it

and compare the read and written values; a failed write is indicated when the read and written values are not equal. Each of the write functions has a corresponding read function that can be used to read back the programmed register state; these are not affected by the state of the SWE register.

These API functions write to SWE protected registers:

- S826_DacRangeWrite and S826_DacDataWrite (when safemode argument = '1')
- S826_DioOutputSourceWrite, S826_DioSafeWrite, and S826_DioSafeEnablesWrite
- S826_WatchdogConfigWrite, S826_WatchdogEnableWrite
- S826_SafeControlWrite
- S826_VirtualSafeWrite and S826_VirtualSafeEnablesWrite

10.2 API functions

10.2.1 S826_SafeControlWrite

The S826_SafeControlWrite function programs the board's fail-safe configuration and state.

```
int S826_SafeControlWrite(
  uint board,     // board identifier
  uint settings,  // safemode settings
  uint mode       // write mode
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

settings
Safemode configuration and state bits:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	XSF	0	SAF	0

Bit	Function
XSF	DIO47 trigger enable. Programmed to '0' upon reset. When '1', a low level (0 volts) on the DIO channel 47 header pin will set the SAF bit. When '0', DIO47 will not affect the SAF bit. When XSF=1, a thread can block on DIO47 falling edge events to receive notification when safemode is triggered. Alternatively,the program can poll SAF.
SAF	Safemode state. Programmed to '0' upon reset. '1' = safemode active, '0' = runmode active. This can be written by the application program. This bit can also be set by DIO47 when XSF=1, or by a timeout event on watchdog Timer0.

mode
Write mode: 0 = write, 1 = clear bits, 2 = set bits (see “Atomic read-modify-write”).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

This function should only be called when the SWE bit is set (see S826_SafeWrenWrite). The function will fail without notification (return S826_ERR_OK) if SWE=0 (see Section 10.1.1).

10.2.2 S826_SafeControlRead

The S826_SafeControlRead function returns the board's fail-safe configuration and state.

```
int S826_SafeControlRead(
  uint board,     // board identifier
  uint *settings  // safemode settings
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

settings
Pointer to buffer that will receive the safemode configuration and state as detailed in S826_SafeControlWrite.

mode
Write mode: 0 = write, 1 = clear bits, 2 = set bits (see “Atomic read-modify-write”).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

10.2.3 S826_SafeWrenWrite

The S826_SafeWrenWrite function enables or disables write protection for safemode-related registers.

```
int S826_SafeWrenWrite(
  uint board,     // board identifier
  uint wren       // write enable/disable
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

wren
1 = write protect (default upon board reset), 2 = write enable, other values have no effect. When writes are disabled (write protected), attempts to write to protected registers will without notification (return S826_ERR_OK).

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

Remarks

See “Write protection” for a list of registers that are write protected/enabled by this function.

10.2.4 S826_SafeWrenRead

The S826_SafeWrenRead function returns the board's fail-safe configuration and control settings.

```
int S826_SafeWrenRead(
  uint board,     // board identifier
  uint *wren      // write protection status
);
```

Parameters

board
826 board number. This must match the settings of the board's dip switches as described in section 2.2.

wren
Pointer to buffer that will receive the write protection status: 0 = write protected, 2 = write enabled.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is an error code.

10.3 Application notes

10.3.1 Programming safemode states

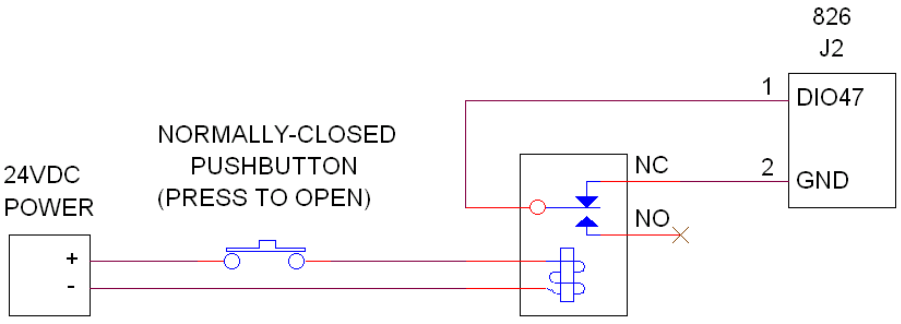
It is recommended to program the safemode data registers even if you will be using default values. This will serve to document fail-safe operation in your code and enable you to easily change safemode states if you need to. The following example shows how to do this for board number 0:

```
// Program fail-safe states for all analog and digital outputs --------------

int aout;                   // analog output channel number
uint SafeDio[] = {0, 0};    // fail-safe DIO states

S826_SafeWrenWrite(0, S826_SAFEN_SWE);                // Write-enable safemode data registers.
S826_DioSafeWrite(0, SafeDio, S826_BITWRITE);         // Program safemode DIO states.
for (aout = 0; aout < S826_NUM_DAC; aout++) {         // Program safemode analog output condition:
  S826_DacRangeWrite(0, aout, S826_DAC_SPAN_0_5, 1); //   output range
  S826_DacDataWrite(0, aout, 0, 1);                  //   output voltage
}
S826_SafeWrenWrite(0, S826_SAFEN_SWD);                // Protect safemode data registers.
```

10.3.2 Activating safemode with an E-stop

An external emergency stop (E-stop) switch can be used to force analog and digital outputs to fail-safe states. To implement this, you must convert the E-stop signal to active-low TTL levels and apply it to DIO47 so that DIO47 will be driven low when the E-stop button is pressed. The following schematic shows a robust, reliable way to do this for a 24V E-stop contact. This circuit ensures that safemode will be activated if the E-stop pushbutton is pressed, or 24VDC power is lost, or the relay coil opens.

Figure 13: A robust interface circuit for a 24V E-stop contact



Typically, the application program will configure the fail-safe system during initialization, before I/O operations begin. This is done by programming the analog and digital safemode states, and then "arming" the system by enabling DIO47 to trigger safemode operation. The following example illustrates this process:

```
// Configure and arm the fail-safe system ---------------

int i;

// The desired fail-safe output conditions (change as required):
uint safeDioEnables[2] = {0x00FFFFFF, 0x00FFFFFF};  // Switch all DIOs to fail-safe states.
uint safeDioData[2]    = {0, 0};                    // Safemode DIO states.
uint safeAoutRange[4]  = {0, 0, 0, 0};              // Safemode analog output ranges.
uint safeAoutLevel[4]  = {0, 0, 0, 0};              // Safemode analog output levels.

// Allow modifications to fail-safe settings.
S826_SafeWrenWrite(0, S826_SAFEN_SWE);

// Program analog output fail-safe conditions.
for (i = 0; i < S826_NUM_DAC; i++) {
  S826_DacRangeWrite(0, i, safeAoutRange[i], 1);
  S826_DacDataWrite(0, i, safeAoutLevel[i], 1);
}

// Program digital output fail-safe conditions.
S826_SafeEnablesWrite(0, safeDioEnables);
S826_DioSafeWrite(0, safeDioData, S826_BITWRITE);

// Allow the E-stop switch to activate fail-safe operation.
S826_SafeControlWrite(0, S826_CONFIG_XSF, S826_BITSET);

// Prevent errant software from modifying fail-safe settings.
S826_SafeWrenWrite(0, S826_SAFEN_SWD);
```

In some cases the application program must be signaled when the E-stop pushbutton is pressed so that it can execute relevant tasks (e.g., record the event to a log). The following example shows how to detect and handle an E-stop event.

```
// Detect and handle an E-stop pushbutton press ------------

WaitForDioFallingEdge(0, 47); // Wait for DIO47 falling edge.
printf("E-stop pushbutton was pressed!");
```

Chapter 11: Specifications

11.1 Timebase

Master oscillator	Frequency	50 MHz
	Stability	50 ppm
Timestamp generator	Resolution	32 bits
	Count rate	1 Mcounts/s
	Rollover period	2^{32} µs (approx. 71.6 minutes)

11.2 Digital I/O

Channels	Number/type	48 bi-directional
	Signal levels	TTL/5VCMOS
	Internal sampling rate	50 Ms/s
Output	Driver type	Open drain w/internal pull-up resistor to +5 VDC
	Internal pull-up resistance	10 kΩ, 5%
	Rise time (nominal)	No external pull-up: 200 ns 1.2 KΩ external pull-up: 20 ns 680 Ω external pull-up: 10 ns
	Low-level output current	24 mA max.
	Low-level output voltage	0.6 V max. @ 12 mA 0.8 V max. @ 24 mA
	Sink current when board unpowered	< 20 µA
	Fail-safe mode	Yes
Input	Receiver type	Schmitt trigger
	Voltage range [1]	0 to +5.5 V (-0.5 to +6.5 V absolute max.)
	High-level threshold voltage	2.2 V max.
	Low-level threshold voltage	0.6 V min.
	Input hysteresis	0.4 V min; 1.2 V max.
	Noise/debounce filter	Interval: 0 to 1.3107 ms in 20 ns steps, common to all channels. Enable/disable per channel.
	Input leakage [1]	50 µA @ 5 V, max.

11.3 Counters

Channels	Number	6
	Resolution	32 bits
	Count rate	0 to 50 Mcounts/s
Event capture (per channel)	Capture rate	50 MS/s max.
	Sample width	73 bits (counts + timestamp + trigger flags)
	FIFO depth	16 samples
	Triggering	9 trigger sources; supports multiple concurrent triggers
Clock inputs (ClkA or ClkB) AC characteristics	Frequency range, quadrature clocks [2]	0 to 12.5 MHz, corresponding to these count rates: x4: 0 to 50 Mcounts/s x2: 0 to 25 Mcounts/s x1: 0 to 12.5 Mcounts/s
	Frequency range, mono clock	0 to 25 MHz, corresponding to 0 to 25 Mcounts/s
	High-level time, minimum	Quadrature modes: 40 ns Mono modes: 20 ns
	Low-level time, minimum	Quadrature modes: 40 ns Mono modes: 20 ns
IX input AC characteristics	High-level time, minimum	20 ns
	Low-level time, minimum	20 ns
Inputs (CLK, IX) DC characteristics	Signal compatibility	RS-422 differential, TTL/5VCMOS single-ended
	RS-422 termination	External
	Ground-referenced voltage range [1]	-0.3 to +5.5 V (-4 to +8 V absolute max.)
	Differential voltage range	±5.8 V (±12 V absolute max.)
	Differential high-level threshold voltage	0.2 V max.
	Differential low-level threshold voltage	-0.2 V max.
	Input resistance [1]	12 kΩ typical; 7 kΩ min.
Input filter	Filter interval	0 to 1.3107 ms in 20 ns steps, common per channel. Enable/disable: independent IX, CLK pair.

11.4 Analog inputs

Channels	Number/type	16 differential
ADC	Resolution	16 bits
	Conversion time	3 µs max.
	Integral nonlinearity error	±1.5 LSB max. (±0.75 typical)
	Differential nonlinearity error	±1.25 LSB max. (±0.5 typical)
	Gain error	±40 LSB max. (±2 typical)
	Gain error temperature drift	±0.3 ppm/ºC (typical)
	Zero error	±0.8 mV (max)
	Zero temperature drift	±0.3 ppm/ºC (typical)
Input	Differential voltage measurement ranges	±1 V, ±2 V, ±5 V, ±10 V
	CMV	±11 V for specified accuracy
	Maximum input voltage [1] (at either terminal of differential input pair)	±20 V continuous, ±30 V peak (1 ms pulses, 10% duty cycle max.)
	CMRR	> 80 dB @ 1 kHz, > 65 dB @ 10 kHz
	Input impedance	>10 MΩ in parallel with 100 pF
	Settling time for multi-channel measurements	4µs max. @ < 1 kΩ input Z with no gain change
Triggering	Modes	Hardware: 48 external digital inputs, 6 internal counter outputs. Software: 6 virtual digital outputs. Untriggered (free-running).

11.5 Analog outputs

Channels	Number/type	8 single ended, with local (on-board) sense
DAC	Resolution	16 bits
	Conversion time [3]	1.04 µs
	Integral nonlinearity	±2 LSB max.
	Differential nonlinearity	±1 LSB max.
	Gain error	±20 LSB max. (±4 LSB typical)
	Gain temperature coefficient	±2 ppm/ºC (typical)
	Unipolar zero-scale error	5V unipolar range, 25 ºC: ±200 µV max. (±80 µV typical) 10V unipolar range, 25 ºC: ±300 µV max. (±100 µV typical) 5V unipolar range: ±400 µV max. (±140 µV typical) 10V unipolar range: ±600 µV max. (±150 µV typical)
	Voltage offset temperature coefficient	±2 µV/ ºC (typical)
	Bipolar zero error	±12 LSB max. (±2 LSB typical)
Output	Voltage ranges	0 to +5 V, 0 to +10 V, ±5 V, ±10 V
	Load current (maximum)	2 mA
	Fail-safe mode	Yes

11.6 Watchdog timer

Timer stages	Number	3
	Interval (per stage)	Programmable from 1 to 2^{32}-1 * 20 ns (20 ns to ~85.9 s)
	Output events	Stage 0: Fail-safe trigger, IRQ Stage 1: NMI out via digital output, PCIe Fatal Error Stage 2: Reset via digital output or onboard solid state relay (SSR)
Solid state relay (Reset out)	On-state resistance (I_L = 10mA)	20 ohms typical, 30 Ω max.
	Off-state leakage current	0.03 µA typical, 1.0 µA max.
	Applied voltage (maximum)	200 V
	Load current (maximum)	100 mA

11.7 Power and environmental

Power	Input power	350 mA (+12V) and 450 mA (+3.3V), nominal, with no loads
	5V outputs (on DIO/counter connectors)	+5 VDC ±5%, 400 mA max. (total for all external loads) Note: these do not have independent short circuit protection
Temperature	Operating	0 to 70ºC
	Storage	-40 to 70ºC
Mating Connectors (not included)	Counters	Sullins SFH210-PPPC-D13-ID-BK or equivalent (qty. 2)
	Analog I/O	Sullins SFH210-PPPC-D25-ID-BK or equivalent (qty. 1)
	Digital I/O	Sullins SFH210-PPPC-D25-ID-BK or equivalent (qty. 2)
	Watchdog Reset Out	Molex 22-01-2037 (ramp only), or Molex 22-01-3037 (ramp and alignment ribs)
Mechanical	Board dimensions	3.88 x 6.55 in.

Notes:

1. Applies when board is powered or unpowered.
2. Specified for 90º ClkA/ClkB phase angle; derate for deviations from 90º (consecutive ClkA/ClkB edges must be separated by 20 ns minimum).
3. Includes data transmission and analog conversion times for full-scale output step.