Ethernet Industrial I/O Modules
API and Programming Guide

Model 24xx Family | Rev.A | August 2010

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 97223

Table of Contents

Table of Contents

Chapter 1: Introduction

1.1 Scope

This document describes the application programming interface (API) for Sensoray's Model 24xx product family of Ethernet industrial I/O modules.

1.2 Description

The API is a middleware library that will interface one or more Sensoray Model 24xx modules (e.g., Model 2410 48 Channel Digital I/O, Model 2426 Multi-Function I/O, etc.) to an application program of your design. A rich set of API functions provides access to all resources on the various types of modules found in the 24xx family. The API supports any arbitrary number of modules and any combination of module types, limited only by system resources.

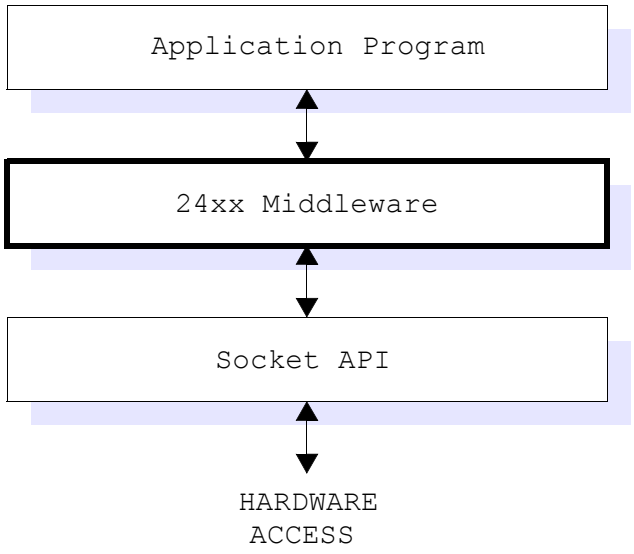
Linux and Windows libraries are supplied in the SDK distribution media.

1.2.1 Software Hierarchy

The middleware consists of an executable that serves as an interface between the application program and Ethernet network. The Windows version is implemented as a dynamic link library, `S24xx.DLL`. The Linux version is a static library, `lib24xx.a`.

Figure 1 illustrates the relationships between the middleware and related software components.

Figure 1: Software hierarchy.



Chapter 2: Installation

2.1 Executable Software Components

The middleware is dependent on a network API, so a suitable socket interface must be installed and properly configured. In addition, the middleware must be installed on a 24xx client system as described below.

2.1.1 Windows

Dynamic link library file `S24xx.DLL` must be located in either the directory containing the application that uses it, or in one of the directories in the operating system's DLL search path (e.g., "`C:\WINDOWS\SYSTEM\SYSTEM32`").

2.1.2 Linux

Library file `lib24xx.a` must be located in the linker's library search path. You can locate the library in one of the linker's default search path directories or, alternatively, you may explicitly specify the path of the library when invoking the linker. As an example of the latter, you could locate the library in your application project's directory and use a command like this to specify the library path:

```
gcc -g -o clientapp clientapp.o -L. -l24xx
```

In this case, the "`-L.`" indicates that the current directory is to be searched for library files, and the "`-l24xx`" requests linking of the `lib24xx.a` library file.

2.2 Application SDK Components

Distribution media for the Model 24xx family includes the API libraries, documentation, sample applications and other source code files:

`s24xx.dll`	Windows API dynamic link library.
`s24xx.lib`	Windows import library.
`s24xx.a`	Linux API static library.
`s24xx.h`	API declarations. Include this in all C/C++ application modules that call API library functions.
`stypes.h`	API types. This is required by `S24xx`.h.

Chapter 3: Fundamentals

3.1 Board Addressing

3.1.1 Board Handles

Every Model 24xx I/O module is assigned a reference number called a *session handle*. Many of the API functions include this handle as an argument.

3.1.2 IP Address and Port

Each module must be configured by assigning it a unique network address and, if desired, unique port numbers. The network address is the Internet Protocol (IP) address at which the module resides, and the port numbers specify the Telnet and HTTP ports to use.

Every I/O module has an identical factory-configured IP address and port number. The IP address is set to `192.168.24.xx`, where `xx` is the last two digits of the model number. For example, Model 7410 Digital I/O modules are set to `192.168.24.10`. All boards are factory configured to use port 23 for Telnet and port 80 for HTTP. If the default address or port numbers are incompatible with your network, or if the module's IP address conflicts with another host, it will be necessary to change the module's network settings. If you will not be operating the I/O module on a public network, we recommend that you assign IP addresses that are specifically reserved for private networks, such as `10.x.x.x` or `192.168.x.x`.

3.1.2.1 Configuring the Network Interface

1. **Connect the module to your network** with an Ethernet patch cable, Category-5 or higher. Use a crossover cable if you are connecting the module directly to a computer, otherwise use a standard patch cable. Note: this network need not be the one the module will operate on; it will only be used to configure the module.
2. **Apply 24VDC power** to the module.
3. **Switch to Configuration mode.** Hold down the module's CONFIG pushbutton while you press and then release the RST pushbutton. The blue LED under the CONFIG pushbutton will light when the module is in Configuration mode. If multiple modules are connected to your network, ensure that only one module is in Configuration mode.
4. **Assign a temporary IP address.** The temporary address will only be used during configuration. It is recommended that this temporary address not be the same as the permanent address you will be assigning to the module later. Choose a temporary address that is unique and reachable on your configuration network.

   Run ARP with this command line:

   ```
   arp -s <temp_addr> 08-00-56-FF-FF-FF
   ```

   Example:
   ```
   arp -s 192.168.1.25 08-00-56-FF-FF-FF
   ```

   Windows

   You can do this in either of the following ways:

   - Navigate to Start | Run, then type the command into the dialog box and click OK.
   - Open a console window, then type the command at the shell prompt followed by Enter.

   Linux

   Open a shell, then type the command at the shell prompt followed by Enter.
5. **Open this URL** from a web browser:

   ```
   http://<temp_addr>/config.htm
   ```

   Example:
   ```
   http://192.168.1.25/config.htm
   ```

   The module's Configuration web page should appear in your browser window.
6. **Program the permanent network settings.** In the designated field on the Configuration web page, enter the permanent IP

address you chose earlier. If necessary, also enter a new netmask and gateway address. Click the Submit button and wait for the page to reload.

7. **Reset the module.** Press and release the RST pushbutton. Your permanent network settings are now in effect.

3.2 Programming Examples

The C programming language has been used for all programming examples.

3.2.1 Constants

Many of the examples specify symbolic constants that are defined in `s24xx.h`, which can be found on the distribution media.

3.2.2 Data Types

In most cases, data values passed to or received from library functions belong to a small set of fundamental data types. All of these data types are listed in Table 1. Data types are referenced by their C-language type names, as shown in the left column of the table.

Table 1: Data types used by library functions

Type Name	Description
`u8`	8-bit unsigned integer
`s16/u16`	16-bit signed/unsigned integer
`s32/u32`	32-bit signed/unsigned integer
`BOOL`	32-bit integer (0=false, other=true)
`HSESSION`	void pointer (session handle)
`HEVCAP`	void pointer (event notification system handle)

A few functions make use of structures that are composites of the fundamental types. All structures are defined in header file `s24xx.h`.

3.3 Required Function Calls

Some library functions are used universally in all applications, while others, depending on application requirements, may or may not be used. All applications must, as a minimum, perform the following steps:

1. Call **`S24xx_ApiOpen()`** to initialize the API. This should always be the first API function executed by a client application.
2. For each I/O module, call **`s24xx_SessionOpen()`** to open a communication session with it.
3. To guarantee proper cleanup when your application terminates, call **`S24xx_SessionClose()`** for each previously opened session, and then call **`s24xx_ApiClose()`** after all sessions have been closed.

Chapter 4: Sessions and Transactions

4.1 Overview

Most API functions involve transactions between the client and an I/O module over a telnet session. When the application program invokes a transaction by calling an API function, the API internally executes a four-step process:

- Translate the API function and its arguments to an equivalent shell command.
- Send the command to the I/O module via telnet.
- Receive the reply from the I/O module via telnet.
- Translate the reply to the form expected by the application program.

Transaction functions are designed to insulate the application programmer from the cumbersome details of network programming and packet parsing.

4.1.1 Blocking Behavior

All transaction functions are blocking functions, which means that calls to those functions will not return until the transaction (i.e., the above four-step process) has completed.

4.1.2 Thread Safety

All transaction functions are thread safe, so it is permissible for multiple API calls to be in progress at the same time on a single session. For example, an application may be partitioned into multiple threads (e.g., analog I/O thread, digital I/O thread, serial communication thread) such that each thread asynchronously invokes its own private transactions over a common session. In most cases, a thread will be blocked and its transaction will be held off if another transaction is already in progress on the same session.

4.2 Concurrent Transactions

Each I/O module supports up to three simultaneous telnet sessions and, as a result, up to three overlapped transactions may be in progress at the same time on an I/O module.

An Ethernet client may run multiple threads and/or processes in which each thread or process concurrently executes simultaneous transactions with an I/O module, with each transaction running on a unique session. Simultaneous transactions may also involve more than one Ethernet client. For example, it is permissible for two or three different Ethernet clients to simultaneously execute transactions on one module. Each of the three possible simultaneous transactions may be invoked by any arbitrary Ethernet client.

For best performance, multi-threaded applications should communicate over dedicated sessions whenever possible (i.e., one thread per session). This is because transactions on different sessions can be overlapped, whereas transactions that share a common session are executed serially.

4.3 Errors

Various errors can occur when interacting with I/O modules over a network. When an error is detected during a session transaction, it is only known to the session in which it occurs. Sessions are not aware of errors in other sessions.

When an error is detected, the currently executing API function is terminated and, if the error is classified as “fatal,” the error is permanently logged to the session. Subsequent transaction attempts on the same session will fail if a fatal error has been logged. Each session’s error log is retained across multiple transaction attempts, effectively propagating a fatal error across any number of API calls, client threads, and client network interfaces.

4.3.1 Error Passing Mechanism

Most API functions have an argument named `err`, which is a pointer to an error code that is allocated by the calling thread. Each thread typically sets its error code to `ERR_NONE` (zero) before calling an API function to indicate no errors are pending. Every subsequent call to an API function may change the value of the error code. If the error code is not `ERR_NONE` when an API function is called, the function will be aborted and the error code will not be modified. Because of this “error propagation” behavior, it may not be necessary to check for errors at the end of every transaction. Instead, the application may be designed to catch errors at the

end of a sequence of transactions, but only if it doesn't need to know exactly where in the sequence the error occurred. This is done by setting the error code to `ERR_NONE` once before calling a sequence of API functions and then checking the error code after the entire sequence has executed to determine if any errors occurred during the sequence.

This error propagation paradigm includes `s24xx_SessionOpen()`, which will set an appropriate error code and return NULL if it fails to create a new session. Instead of checking for errors after calling `s24xx_SessionOpen()`, the application may continue onward and attempt transactions as if the session had been successfully opened. All such attempts will fail and leave the error code unchanged.

4.3.2 Error Handling

Most API functions return a boolean that indicates whether the operation completed successfully. False (zero) is returned if the operation failed, otherwise True (non-zero) is returned. These functions return False if an error occurs during function execution or if a previously detected error is pending when the function is called.

Programming languages typically define False as zero, but True has no universally accepted definition. Consequently, in the case of API functions that return a boolean value, that value should generally be compared to False when deciding if an error occurred. For example, in VB.NET this is the recommended practice:

```
If s24xx_SomeFunction() = False Then
   ' handle error ...
Else
   ' do this if no error ...
Endif
```

In addition, most functions include in their argument lists a pointer to the caller's error code. When a function returns False (thus indicating an error has occurred), the error type can be determined by inspecting the error code. The application can then take corrective action based on the type of error that was detected.

It is not possible to restore communication on a session that has logged a fatal error; the session must be closed and, if communication is to be resumed, a new session must be opened.

4.3.3 Error Codes

Symbolic Name	Description
`ERR_NONE`	No errors.
	Initialization Errors
`ERR_MALLOC`	Internal memory allocation failed.
`ERR_SOCKETCREATE`	Failed to create socket.
`ERR_SESSIONCONNECT`	Failed to open a TCP connection t the I/O module's telnet server.
`ERR_TOOMANYSESSIONS`	The I/O module's telnet server is already running the maximum number of sessions. Most I/O modules are limited to three concurrent sessions.
`ERR_OPENSHELL`	Failed remote shell login.
`ERR_THREADCREATE`	Failed to create internal thread.
`ERR_CRITSECTCREATE`	Failed to create internal critical section.
`ERR_CREATEMUTEX`	Failed to create internal mutex.
	Fatal (unrecoverable) Run-time Errors
`ERR_INVALIDSESSION`	Session handle is zero. This can happen if your application unsuccessfully attempted to open a session and then continues on as if the session had opened.
`ERR_SOCKET`	Socket error.
`ERR_CONNCLOSED`	The session's TCP connection closed unexpectedly. This can happen if an I/O module's watchdog times out due to network inactivity.
`ERR_NETWORKWRITE`	The socket failed to transmit to the I/O module.

Symbolic Name	Description
ERR_TIMEOUT	Timed out waiting for a reply from the I/O module.
	Non-fatal (recoverable) Run-time Errors
ERR_COMPORTATTACH	Failed to attach session to remote comport. This can happen if the remote port is already attached to another telnet session.
ERR_SHELLCOMMAND	Invalid shell command. For example: * Invalid channel number, which does not exist on the target I/O module. * A numerical value exceeds permitted limits. * Transaction is not supported by the I/O module type (e.g., performing a digital I/O action on an analog I/O module).
ERR_RSPSIZE	Invalid reply received from I/O module.
ERR_CMDARG	Invalid command argument.
ERR_BUFWOULDOVERFLOW	Operation would overflow receive buffer.
ERR_COMPORTWASATTACHED	The session is attached to a remote comport and an attempt was made to execute a non-comport transaction.
ERR_COMPORTUNATTACHED	The session is not attached to a remote comport and an attempt was made to execute a comport transaction.

Chapter 5: Module-Independent Functions

5.1 Overview

The API functions discussed in this chapter are common to all I/O module types.

5.2 API Initialization and Shutdown

5.2.1 s24xx_ApiOpen()

Function: Open and initialize the API.

Prototype: `BOOL s24xx_ApiOpen( void );`

Returns: Returns a non-zero value if successful, or zero if the operation failed. This can fail if the version number of the socket API is incompatible with middleware, or if TCP/IP is not properly configured on the client.

Notes: `s24xx_ApiOpen()` must be successfully invoked before any other middleware functions are called. Each client process must call this function exactly once. A multi-threaded application must invoke this once before any other API functions are called by any of the application's threads.

Example: *See section 5.4.1.*

5.2.2 S24xx_ApiClose()

Function: Close the API.

Prototype: `void s24xx_ApiClose( void );`

Returns: None.

Notes: If an earlier call to `s24xx_ApiOpen()` was successful, this function must be called before the application closes to ensure that the API shuts down gracefully and properly releases all resources. If an error code was returned by `s24xx_ApiOpen()`, however, the application should not call `s24xx_ApiClose()`. This must be the last API function called by the application.

Example: *See section 5.4.1.*

5.3 Session Initialization and Shutdown

5.3.1 s24xx_SessionOpen()

Function: Open a communication session with an I/O module.

Prototype: `BOOL s24xx_SessionOpen( HSESSION *sess, u32 *err, u16 model, const char *addr, u16 port, u32 ms );`

Argument	Description
`sess`	Pointer to storage that will receive the session handle. The target storage will be set to zero if a session failed to open.
`err`	Pointer to error code. See Section 4.3.1 for details.
`model`	Model number of the I/O module (e.g., 2410, 2426, etc.)
`addr`	Pointer to a null-terminated string that specifies the I/O module's IP address in dotted decimal format.
`port`	Telnet port number used by the I/O board. Use the standard telnet port number (23) unless the board has been reconfigured to use a non-standard port number.
`ms`	Maximum amount of time to wait (in milliseconds) for the session to be established. This can be relatively short for dedicated LANs, but longer times may be needed if the I/O board is remotely located.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: After opening a session for an I/O module, the application may use the session handle in all other functions that require it.

Example: *See section 5.4.1.*

5.3.2 s24xx_SessionClose()

Function: Terminate a communication session with an I/O module.

Prototype: `void s24xx_BoardClose( HSESSION sess );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.

Returns: None.

Notes: Each session that has been established by `s24xx_SessionOpen()` must be closed when it is no longer needed by an application. `s24xx_SessionClose()` severs the communication link between the application program and the I/O module. The session handle is no longer valid after this call.

`s24xx_SessionClose()` does not alter the state of the module and the module will continue any autonomous operations already in progress. Since all communications will be severed between the client and the module, the application should ensure that all I/O interfaces are in appropriate states when `s24xx_SessionClose()` is called.

Example: *See section 5.4.1.*

5.4 Error Functions

5.4.1 s24xx_ErrorText()

Function: Return an error description string.

Prototype: `const char * s24xx_ErrorText( u32 err );`

Argument	Description
`err`	Error code. See Section 4.3.3 for list of error codes.

Returns: Pointer to a text string that describes the error code contained in `err`.

Example:

```
// A simple application program.
int main( void )
{
    int      rtnval = 1;
    u32      err = ERR_NONE;
    char     ipaddr[] = "192.168.24.10"
    HSESSION sess;

    // Open the API.
    if ( !s24xx_ApiOpen() )
        printf( "Failed to open API\n" );
    else {
        // Open a session on a Model 2410 I/O module.
        if ( !s24xx_SessionOpen( &sess, &err, 2410, ipaddr, 23, 1000 ) ) {
            printf( "Error: %s\n", s24xx_ErrorText(err) );
            rtnval = err;
        } else {

            // ... perform I/O operations as required by the application ...

            s24xx_SessionClose( sess );  // Close the session and API.
        }
        s24xx_ApiClose();
    }
    return rtnval;
}
```

5.5 Status and Control

5.5.1 s24xx_SetTimeout()

Function: Configure a session's network watchdog timer.

Prototype: `BOOL s24xx_SetTimeout( SESSION sess, u32 *err, u32 count, u32 units, u32 action );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`count`	Time interval specified in seconds or milliseconds. Set to zero to disable watchdog (not recommended for production software).
`units`	Time units: `UNITS_MILLISECONDS` or `UNITS_SECONDS`
`action`	Action to be taken upon time-out: `ACTION_NONE` – terminate the session, but don't reset any I/O interfaces `ACTION_RESET` – terminate the session and reset all I/O interfaces

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Each session employs a timer to detect the absence of communications with clients. If no communication is received from a client within the specified time interval, the network watchdog will time-out and the specified action will be taken. Upon time-out, the session is automatically closed. This behavior ensures that the server session will be freed for other uses and all I/O will be (optionally) reset in the event the client shuts down abnormally.

By default, the network watchdog timer is set to five minutes when a session is opened, and `action` is set to `ACTION_NONE`. If these defaults suit the application then there is no need to call this function.

Upon executing this function, the new time interval is effective immediately and the watchdog timer is restarted.

The time interval may be set to zero to disable the watchdog timer. This should be avoided except during application development, as it could make it impossible to open new sessions if the client fails to properly close previous sessions.

Example:

```
// Set the watchdog interval to 3.5 seconds, with no I/O reset upon time-out.
u32 err = ERR_NONE;
if ( !s24xx_SetTimeout( sess, &err, 3500, UNITS_MILLISECONDS, ACTION_NONE ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

5.5.2 s24xx_ResetIo()

Function: Reset all I/O interfaces to their default power-up condition.

Prototype: `BOOL s24xx_ResetIo( HSESSION sess, u32 *err );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function resets all I/O interfaces on the target module. For example, digital and analog outputs will be set to their default power-up states. The module will not reboot, and the session used to invoke this function will remain open. Typically, this function is only used during application development.

Example:

```
// Reset all I/O interfaces.
```

```
u32 err = ERR_NONE;
if ( !s24xx_ResetIo( sess, &err ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

5.5.3 s24xx_GetVersionInfo()

Function: Read a module's firmware version information.

Prototype: `BOOL s24xx_GetVersionInfo( HSESSION sess, u32 *err, char *buf, u32 len, BOOL *secondary );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`buf`	Pointer to buffer that will receive version number string. Set to NULL if string is not needed.
`len`	Size of buffer that will receive the version number string. Set to zero if not needed.
`secondary`	Pointer to buffer that receives indication that secondary firmware is executing. Set to NULL if not needed. If not NULL, the buffer will be set to False if the factory-installed base firmware is running, or True if upgraded firmware is running.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Fetch and display firmware version info.
char buf[100];
BOOL secondary;
u32 err = ERR_NONE;
if ( !s24xx_GetVersionInfo( sess, &err, buf, sizeof(buf), &secondary ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "Firmware version: %s (%s)\n", buf, secondary ? "secondary" : "primary" );
```

5.6 Timestamp Functions

Every I/O module maintains an independent clock that is used to timestamp data returned by various API functions. The clock starts at zero upon module boot-up or reset and then increments every microsecond. The 32-bit clock overflows (i.e., restarts at zero) approximately every 71.5 minutes.

5.6.1 s24xx_ReadTimestamp()

Function: Read module's system time.

Prototype: `BOOL s24xx_ReadTimestamp( SESSION sess, u32 *err, u32 *timestamp );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`timestamp`	Pointer to buffer that will receive the module's system time.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Read and display the current timestamp.
u32 err = ERR_NONE;
u32 systime;
if ( !s24xx_ReadTimestamp( sess, &err, &timestamp ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "Timestamp = %d microseconds\n", timestamp );
```

5.6.2 s24xx_WriteTimestamp()

Function: Set module's system time.

Prototype: `BOOL s24xx_WriteTimestamp( SESSION sess, u32 *err, u32 systime );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`timestamp`	Module's new system time, in microseconds.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function forces a module's system clock to a desired initial time.

Example:

```
// Force the system clock to zero.
u32 err = ERR_NONE;
if ( !s24xx_WriteTimestamp( sess, &err, 0 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

Chapter 6: Model 2410 Digital I/O Module

6.1 Overview

The API functions in this chapter are used to monitor and control Model 2410 48-channel digital I/O modules. They are applicable only to Model 2410 I/O modules. Any attempt to call them for other I/O module types will result in a `ERR_SHELLCOMMAND` transaction error.

Several of these API functions convey information for all 48 DIO channels through an array of three 16-bit words, with each bit representing one DIO channel. In such cases, the first word (array index 0) represents DIO channels 0-15 (lsb-msb), the second word represents channels 16-31, and the third word represents channels 32-47.

6.2 Digital I/O Functions

6.2.1 s2410_SetDebounceTime()

Function: Program the debounce time interval of one digital input channel.

Prototype: `BOOL s2410_SetDebounceInterval( SESSION sess, u32 *err, u8 chan, u8 msec );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`chan`	DIO channel number: 0 to 47.
`msec`	Debounce time interval in milliseconds: 0 to 255.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Physical input states are sampled periodically at one millisecond intervals and passed through a debounce filter. A digital input is regarded to be in a particular state only after it has held steady in that state for its debounce interval.

For example, consider the case of a digital input channel that has a 30 millisecond debounce interval. If the channel has been in the inactive state for a long time and then it switches to the active state, `s2410_ReadDin()` will not indicate the new (active) state until 30 milliseconds after the physical input became active. If the input goes active and then switches to inactive before the 30 milliseconds has elapsed, `s2410_ReadDin()` will never indicate that the input is active.

Upon boot-up, all digital inputs are configured to have a ten millisecond debounce interval by default.

Example:

```
// Configure channel 3 for a 50 millisecond debounce interval.
u32 err = ERR_NONE;
if ( !s2410_SetDebounceTime( sess, &err, 3, 50 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.2.2 s2410_ReadDin()

Function: Read the debounced physical states of all DIO channels.

Prototype: `BOOL s2410_ReadDin( SESSION sess, u32 *err, u16 *states );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`states`	Pointer to an array that will receive the state values. Each bit has the following meaning: 1 - active state (driven low by on-board driver or off-board signal). 0 - inactive state (pulled high).

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Every DIO channel includes a monitoring circuit that enables that channel's physical state to be read. This function acquires a snapshot of the physical state of each channel without regard for whether the channel is driven by its on-board output driver or by an externally generated signal.

Physical states are sampled in parallel (all 48 channels are sampled simultaneously) at one millisecond intervals. Sample data are passed through a debounce filter which may cause latency or, in the case of rapidly changing physical states, undetected state changes. Consequently, the values returned in `states[]` may not accurately reflect the instantaneous physical states of channels that have changed within the debounce interval.

Example:

```
// Read and display the debounced input states of all DIO channels.
u16 states[3];
u32 err = ERR_NONE;
if ( !s2410_ReadDin( sess, &err, states ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "input states: %04x %04x %04x\n", states[2], states[1], states[0] );
```

6.2.3 s2410_ReadDout()

Function: Read the programmed output states of all DIO channels.

Prototype: `BOOL s2410_ReadDout( SESSION sess, u32 *err, u16 *states );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`states`	Pointer to an array that will receive the state values. Each bit has the following meaning: 1 - active state (driven low by on-board driver). 0 - inactive state (pulled high by on-board driver or driven low by off-board signal).

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function reads the programmed states of all DIO output drivers. Note that these states may differ from those returned by `s2410_ReadDin()` because channels can be driven by off-board signals as well as on-board drivers. Also, in the case of channels operating in PWM output mode, the instantaneous driver states are determined by each channel's PWM generator.

Example:

```
// Get all DIO output states.
u16 states[3];
u32 err = ERR_NONE;
```

```
if ( !s2410_ReadDout( sess, &err, states ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "driver states: %04x %04x %04x\n", states[2], states[1], states[0] );
```

6.2.4 s2410_WriteDout()

Function: Programs the output states of all DIO channels.

Prototype: `BOOL s2410_WriteDout( SESSION sess, u32 *err, u16 *states );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`states`	Pointer to array that contains the desired output states. Each bit has the following meaning: 1 - active state (driven low by on-board driver). 0 - inactive state (pulled high by on-board driver or driven low by off-board signal).

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function programs the states of all DIO output drivers of channels that are operating in the Standard output mode. Note that this has no effect on channels operating in the PWM output mode, as their drivers are autonomously controlled by PWM generators.

Example:

```
// Program all DIO output states.
u32 err = ERR_NONE;
u16 states[] = { 0x0123, 0x4567, &0x89AB };    // desired DIO states
if ( !2410_WriteDout( sess, &err, states ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.2.5 s2410_SetDoutMode()

Function: Program the output operating mode of one DIO channel.

Prototype: `BOOL s2410_SetDoutMode( SESSION sess, u32 *err, u8 chan, u32 mode );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`chan`	Channel number in the range 0 to 47.
`mode`	Channel operating mode: `DOUT2410_MODE_STANDARD` or `DOUT2410_MODE_PWM`.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Each channel may be independently configured to operate in either Standard mode or PWM mode. When operating in Standard mode, a channel's output state can be manually programmed by calling `s2410_WriteDout()`. In PWM mode, however, the state is automatically controlled by the I/O module, with duty cycle and frequency programmed by `s2410_WritePwm()`.

Example: *See section 6.2.6.*

6.2.6 s2410_WritePwm()

Function: Program the PWM ratio for one DIO channel.

Prototype: `BOOL s2410_WritePwm( SESSION sess, u32 *err, u8 chan, u16 ontime, u16 offtime );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`chan`	Channel number in the range 0 to 47.
`ontime`	PWM on time in milliseconds. Range: 0 to 65535.
`offtime`	PWM off time in milliseconds. Range: 0 to 65535.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function applies to channels operating in PWM mode; it has no affect on channels operating in Standard mode. The `ontime` and `offtime` arguments specify the amount of time that the DIO is to be in the active and inactive states, respectively. If `ontime` is zero and `offtime` is non-zero then the DIO output will always be inactive. Similarly, if `offtime` is zero and `ontime` is non-zero then the output will always be active. The output state is indeterminate if both `ontime` and `offtime` are set to zero.

The designated DIO channel will switch to the active state and remain active until `ontime` has elapsed, then it will switch to the inactive state and remain in that state until `offtime` has elapsed. This sequence will repeat with the same duty cycle and frequency until one of these events occurs:

- The `ontime` and/or `offtime` is changed by calling `s2410_WritePwm()`.
- The channel's operating mode is switched from PWM to Standard. The operating mode can be switched under software control by calling `s2410_SetDoutMode()` or `s24xx_ResetIo()`, and it may also be automatically switched in response to a module hardware reset.

Example:

```
// Configure DIO channel 5 for PWM mode: on for 20 ms, off for 30 ms.
u32 err = ERR_NONE;
s2410_SetDoutMode( sess, &err, 5, DOUT2410_MODE_PWM );
s2410_SetWritePwm( sess, &err, 5, 20, 30 );
if ( err != ERR_NONE )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.3 Utility Functions

6.3.1 s2410_SetLedBrightness()

Function: Set the brightness level for all DIO status LEDs.

Prototype: `BOOL s2410_SetLedBrightness( SESSION sess, u32 *err, uint intensity );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`intensity`	Brightness level: 0 (always off) to 16 (maximum intensity).

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This can be used to set LED brightness to a comfortable level or to decrease power consumption. DIO status LEDs can be completely disabled by setting `intensity` to 0. Upon boot-up, the LED intensity defaults to 16 (maximum intensity).

Example:

```
// Set LED brightness to 1 (dim, but visible).
u32 err = ERR_NONE;
if ( !s2410_SetLedBrightness( sess, &err, 1 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.4 Event Capture Functions

6.4.1 Overview

The module's 48 digital inputs are sampled by the module once per millisecond. After passing through debounce filters, the channels are monitored for state changes. Model 2410 implements an event capture system that enables the module to automatically record occurrences of debounced state changes.

The application program can access records of captured events by polling the module at convenient times, or by activating an asynchronous notification system that will callback into the application program whenever an event is captured. Application programs may employ either polling or callbacks to handle captured events, but both methods should not be used simultaneously.

Several API functions issue a command to the event capture system and return when the module acknowledges receipt of the command. Since the capture system is a synchronous state machine (SM), all received commands are enqueued and then executed synchronously in the order they were received at the next SM clock. When asynchronous notification is active, the application's callback function will be called when each command is executed so that the application can synchronize to the SM. In polled mode, however, there is no way to know exactly when a command is executed.

Some of the event capture functions convey boolean flags for the 48 DIO channels through an array of three 16-bit words, with each bit representing one DIO channel. In such cases, the first word (array index 0) represents channels 0-15 (lsb-msb), the second word represents channels 16-31, and the third word represents channels 32-47.

6.4.2 s2410_ReadCapFlags()

Function: Read event capture flags from all DIO channels.

Prototype: `BOOL s2410_ReadCapFlags( SESSION sess, u32 *err, u16 *flags );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`flags`	Pointer to three-word array that will receive the event flags. Each bit has the following meaning: 1 - event captured. 0 - no event captured.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function reads, and then immediately clears, the event capture flags from all 48 digital input channels. It can be used to poll for captured events at convenient times. Alternatively, `s2410_AsyncCapBegin()` may be called to enable asynchronous event notification and thus make polling unnecessary. As a general rule, an application can receive notification of captured events either by polling or by means of asynchronous event notification, but these two methods should not be mixed. `s2410_ReadCapFlags()` should not be called while asynchronous event notification is active.

Event flags will accumulate until `s2410_ReadCapFlags()` is called. When the function is called, all channels that experienced events since the previous call will be indicated by logic ones in `flags[]`. All event flags are synchronously cleared to zero when they are read from the board. If a channel's event flag is set, and then another event is detected on that channel before `s2410_ReadCapFlags()` is called, there will be no indication that two events have occurred on the channel.

Example:

```
// Poll and display the event capture flags.
u32 err = ERR_NONE;
```

```
u16 flags[3];
if ( !s2410_ReadCapFlags( sess, &err, flags ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "Events: %04x %04x %04x\n", flags[2], flags[1], flags[0] );
```

6.4.3 s2410_AsyncCapBegin()

Function: Enable asynchronous notifications for capture system events.

Prototype: `HEVCAP s2410_AsyncCapBegin( const char *addr, u32 *err, u16 port, EVENT_CBK callback, u32 msec );`

Argument	Description
`addr`	Pointer to a null-terminated string that specifies the I/O module's IP address.
`err`	Pointer to error code. See Section 4.3.1 for details.
`port`	Telnet port number used by the I/O board. Use the standard telnet port number (23) unless the board has been reconfigured to use a non-standard port number.
`callback`	Pointer to the application's callback function. See section 6.4.3.1 for details.
`msec`	Maximum amount of time to wait (in milliseconds) for the session to be established. This can be relatively short for dedicated LANs, but longer times may be needed if the I/O board is remotely located.

Returns: Handle to asynchronous notification system. This will be non-zero if the operation was successful, otherwise NULL is returned and `err` will contain the associated error code.

Notes: `s2410_AsyncCapBegin()` activates a notification system that will asynchronously call an application function (a "callback") in response to various events that occur on the module. A new, private session is opened on the module to support the notification system. Consequently, a free session must be available on the module when this function is called.

When the notification system is activated, the callback function will be called once with message type `CAPMSG_ATTACH` (see section 6.4.3.1 for message types) to inform the application that the notification system has activated.

Example:

```
// Activate asynchronous event notifications.
u32 err = ERR_NONE;
if ( !s2410_AsyncCapBegin( "192.168.24.10", &err, 23, callback, 5000 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.4.3.1 Callback Function

Function: Application callback that handles asynchronous notifications from the event capture system.

Prototype: `void callback( char msgtype, const u16 *val );`

Argument	Description
`msgtype`	Message type code.
`val`	Pointer to additional information that depends on the message type.

Notes: The callback function executes in the context of a private thread. Each time it is called, it receives a message type code that indicates the nature of the notification. Messages are received in the order they were produced so that the application can synchronize to the capture system state machine.

Depending on the message type, additional information may be passed to the callback through `val`, which is a pointer to the information. In many cases, the information will be an array of three 16-bit words, where each bit is a boolean that is associated with one DIO channel.

Message Type Code	Description
`CAPMSG_ATTACH`	The async notification system has been activated. This is always the first message received by the callback. It can be used to initialize the application's event handling system, if desired. `val` is not used and should be ignored.
`CAPMSG_DETACH`	The async notification system has shut down. This is always the last message received by the callback. The notification system will shut down if the application calls `s2410_AsyncCapEnd()` or if the connection closes unexpectedly. `val` is not used and should be ignored.
`CAPMSG_POLARITY`	Capture polarities have changed as a result of an earlier call to `s2410_WriteCapPolarity()`. `val` points to an array of three words that indicate the new polarities that are now in effect.
`CAPMSG_CONTINUOUS`	Continuous capturing has been enabled for an arbitrary set of DIO channels in response to an earlier call to `s2410_WriteCapContinuous()`. `val` points to an array of three words that indicate the channels for which continuous capturing has been enabled.
`CAPMSG_ONESHOT`	One-shot capturing has been enabled for an arbitrary set of DIO channels in response to an earlier call to `s2410_WriteCapOneShot()`. `val` points to an array of three words that indicate the channels for which one-shot capturing has been enabled.
`CAPMSG_DISABLE`	Capturing has been disabled for an arbitrary set of DIO channels in response to an earlier call to `s2410_WriteCapDisable()`. `val` points to an array of three words that indicate the channels for which capturing has been disabled.
`CAPMSG_EVENT`	Events have been captured on one or more DIO channels, or the timer started in an earlier call to `s2410_AsyncCapTimer()` has timed out. `val` points to an array of three words that contain event flags for all 48 DIO channels. If one or more flags are set then the corresponding channels captured edge events. If no flags are set then the timer timed out before any events were detected.
`CAPMSG_TIMER`	A timer has started in response to an earlier call to `s2410_AsyncCapTimer()`. `val` points to a single word that contains the time interval, in milliseconds, that remain until time-out.
`CAPMSG_ERROR`	The connection closed unexpectedly. One more callback will follow this one, with message type set to `CAPMSG_DETACH` to indicate that the notification system has shut down.

Example:

```
// A simple callback function that reports capture system events.
void callback( char msgtype, const u16 *val )
{
    switch ( msgtype )
    {
    case CAPMSG_ATTACH:
        printf( "first callback\n" );
        break;
    case CAPMSG_DETACH:
        printf( "final callback\n" );
        break;
    case CAPMSG_TIMER:
        printf( "started timer: %d msec\n", val[0] );
        break;
    case CAPMSG_POLARITY:
        printf( "polarities set: %04x %04x %04x\n", val[2], val[1], val[0] );
        break;
    case CAPMSG_CONTINUOUS:
        printf( "cont cap enabled: %04x %04x %04x\n", val[2], val[1], val[0] );
        break;
```

```
        case CAPMSG_ONESHOT:
            printf( "one-shot cap enabled: %04x %04x %04x\n", val[2], val[1], val[0] );
            break;
        case CAPMSG_DISABLE:
            printf( "cap disabled: %04x %04x %04x\n", val[2], val[1], val[0] );
            break;
        case CAPMSG_EVENT:
            if ( val[0] | val[1] | val[2] )
                printf( "captured events: %04x %04x %04x\n", val[2], val[1], val[0] );
            else
                printf( "time-out -- no events captured" );
            break;
        case CAPMSG_ERROR:
            printf( "connection closed unexpectedly\n" );
            break;
        default:
            printf( "unknown error\n" );
            break;
        }
}
```

6.4.4 s2410_AsyncCapEnd()

Function: Terminate asynchronous notification messages.

Prototype: `BOOL s2410_AsyncCapEnd( HEVCAP hevcap, u32 &err );`

Argument	Description
`hevcap`	Handle to async event notification system obtained from `s24xx_AsyncCapBegin()`.
`err`	Pointer to error code. See Section 4.3.1 for details.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function instructs the asynchronous event notification system to terminate operations. In response, the application's callback function will be called one final time, with message type `CAPMSG_DETACH`, to inform the application that the notification system has shut down.

`s2410_AsyncCapEnd()` should only be called if asynchronous event notifications have been enabled by a previous call to `s2410_AsyncCapBegin()`.

Example:

```
// Shut down the asynchronous event notification system.
u32 err = ERR_NONE;
if ( !s2410_AsyncCapEnd( hevcap, &err ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.4.5 s2410_WriteCapPolarity()

Function: Select the polarities of events to be captured on all DIO channels.

Prototype: `BOOL s2410_WriteCapPolarity( SESSION sess, u32 *err, u16 *flags );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`flags`	Pointer to array that contains capture polarity flags for all 48 digital input channels. Each bit has the following meaning: 1 - enables capture of inactive-to-active (physical high-to-low) edges. 0 - enables capture of active-to-inactive (physical low-to-high) edges.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This selects the edge transition type that is to be captured on each of the 48 DIO channels. For any given channel, only one edge may be selected for capture at a time.

If asynchronous event notification is enabled, the application's callback function will be called with message type `CAPMSG_POLARITY` when the specified polarities go into effect.

Example:

```
// Set capture polarities: chan0=inactive-to-active, all others=active-to-inactive.
u32 err = ERR_NONE;
u16 flags[3] = { 0, 0, 1 };
if ( !s2410_WriteCapPolarity( sess, &err, flags ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.4.6 s2410_WriteCapContinuous()

Function: Enable continuous event capture on an arbitrary set of DIO channels.

Prototype: `BOOL s2410_WriteCapContinuous( SESSION sess, u32 *err, u16 *flags );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`flags`	Pointer to array that contains boolean flags for all digital input channels. Each bit has the following meaning: 1 - enable continuous capture on the channel. 0 - don't modify the channel's capture configuration.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function enables continuous capture for an arbitrary set of digital input channels. Any channel that is configured for continuous capture will remain enabled when an event is captured on that channel. This is in contrast to one-shot capture mode, in which event capturing is automatically disabled when an event is captured.

The boolean bits in `flags[]` specify the channels that are to be affected. A boolean True enables continuous capture on the associated channel, while False will leave the channel's capture mode unchanged. When this function enables continuous capture on a channel, the channel's previous capture mode (i.e., disabled or one-shot) is no longer in effect.

If asynchronous event notification is enabled, the application's callback function will be called with message type `CAPMSG_CONTINUOUS` when the specified channels become enabled for continuous capture.

Example:

```
// Enable continuous capture on channels 0, 1, and 47.
u32 err = ERR_NONE;
u16 flags[3] = { 0x8000, 0, 3 };
if ( !s2410_WriteCapContinuous( sess, &err, flags ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.4.7 s2410_WriteCapOneshot()

Function: Enable one-shot event capture on an arbitrary set of DIO channels.

Prototype: `BOOL s2410_WriteCapOneShot( SESSION sess, u32 *err, u16 *flags );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`flags`	Pointer to array that contains boolean flags for all digital input channels. Each bit has the following meaning: 1 - enable one-shot capture on the channel. 0 - don't modify the channel's capture configuration.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function enables one-shot capture for any arbitrary set of digital input channels. When a channel is configured for one-shot capture, only one event can be captured on that channel. Upon the first event capture for that channel, capturing will be disabled and no subsequent events will be captured. This is in contrast to continuous capture mode, in which channels remain enabled for capturing after captures have occurred.

The boolean bits in `flags[]` specify the channels that are to be affected. A boolean True enables one-shot capture on the associated channel, while False will leave the channel's capture mode unchanged. When this function enables one-shot capture on a channel, the channel's previous capture mode (i.e., disabled or continuous) is no longer in effect.

If asynchronous event notification is enabled, the application's callback function will be called with message type `CAPMSG_ONESHOT` when the specified channels become enabled for one-shot capture.

Example:

```
// Enable one-shot capture on channel 8.
u32 err = ERR_NONE;
u16 flags[3] = { 0x8000, 0, 3 };
if ( !s2410_WriteCapContinuous( sess, &err, flags ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.4.8 s2410_WriteCapDisable()

Function: Disable event capture on an arbitrary set of DIO channels.

Prototype: `BOOL s2410_WriteCapDisable( SESSION sess, u32 *err, u16 *flags );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`flags`	Pointer to array that contains boolean flags for all digital input channels. Each bit has the following meaning: 1 - disable capture on the channel. 0 - don't modify the channel's capture configuration.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function disables event capturing on an arbitrary set of digital input channels.

The boolean bits in `flags[]` specify the channels that are to be affected. A boolean True disables capture on the associated channel, while False will leave the channel's capture mode unchanged. When this function disables capturing on a channel, the channel's previous capture mode (i.e., one-shot or continuous) is no longer in effect.

If asynchronous event notification is enabled, the application's callback function will be called with message type `CAPMSG_DISABLED` when the specified channels become disabled for capture.

Example:

```
// Disable event capturing on channels 0 through 15.
u32 err = ERR_NONE;
u16 flags[3] = { 0, 0, 0xFFFF };
if ( !s2410_WriteCapDisable( sess, &err, flags ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

6.4.9 s2410_WriteCapTimer()

Function: Start a timer that will notify the client if no events are captured within a time interval.

Prototype: `BOOL s2410_WriteCapTimer( SESSION sess, u32 *err, u16 msec );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`msec`	Time interval, in milliseconds, to wait for an event before issuing a client notification. Set to zero to disable the timer.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This API function starts a timer that monitors the elapsed time between successive event captures. If no events are captured within the specified time interval, the application's notification callback will be called with message type `CAPMSG_EVENT`, with no capture flags asserted. This can be useful in cases where the application expects an event to be captured within a certain time frame but no events occur during that time.

The timer is cancelled if an event is captured within the specified time interval. If the timer is started and an event is captured before the time interval expires, the timer will no longer be active.

The application's callback function will be called with message type `CAPMSG_TIMER` when the timer is started or cancelled by this function in order to inform the application of the change in the timer's operational status.

`s2410_WriteCapTimer()` should not be called if the application employs polling to detect captured events. It should only be called when asynchronous event notifications have been enabled by a previous call to `s2410_AsyncCapBegin()`.

Example:

```
// Send a notification if no events are captured within the next 5 seconds.
u32 err = ERR_NONE;
if ( !s2410_WriteCapTimer( sess, &err, 5000 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

Chapter 7: Model 2426 Multi-Function I/O Module

7.1 Overview

The API functions in this chapter can be used to monitor and control Model 2426 multi-function I/O modules. These functions are only applicable to Model 2426 I/O modules; attempting to call them for other I/O module types will result in a `ERR_SHELLCOMMAND` transaction error.

7.2 Digital I/O Functions

7.2.1 s2426_SetDebounceTime()

Function: Program the debounce time interval of one digital input channel.

Prototype: `BOOL s2426_SetDebounceInterval( SESSION sess, u32 *err, u8 chan, u8 msec );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`chan`	Channel number in the range 0 to 7.
`msec`	Debounce time interval in milliseconds: 0 to 255.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Physical input states are sampled periodically at one millisecond intervals and passed through a debounce filter. A digital input is regarded to be in a particular state only after it has held steady in that state for its debounce interval.

For example, consider the case of a digital input channel that has a 30 millisecond debounce interval. If the channel has been in the inactive state for a long time, and then switches to the active state, `s2426_ReadDin()` will not indicate the new (active) state until 30 milliseconds after the physical input switched active. If the input goes active and then switches to inactive before the 30 milliseconds has elapsed, `s2426_ReadDin()` will never indicate that the input is active.

Upon boot-up, all digital inputs are configured to have a ten millisecond debounce interval by default.

Example:

```
// Configure channel 3 for a 50 millisecond debounce interval.
u32 err = ERR_NONE;
if ( !s2426_SetDebounceTime( sess, &err, 3, 50 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.2.2 s2426_ReadDin()

Function: Read the debounced physical states of the eight digital input channels.

Prototype: `BOOL s2426_ReadDin( SESSION sess, u32 *err, u8 *states, u32 *timestamp );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`states`	Pointer to an application buffer that will receive the physical states of the digital input channels. Each bit is associated with one channel. For example, bit 7 is associated with channel 7. Logic one indicates the associated channel is in the active state (driven low), while logic zero indicates inactive state (pulled high).
`timestamp`	Pointer to buffer that will receive the timestamp. The timestamp is a snapshot of the I/O module's system timer at the moment `counts` is sampled. Set to NULL if the timestamp is not needed. See section 5.6 for more information about timestamps.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Each digital input channel includes a monitoring circuit that enables the on-board processor to determine the physical state of the channel. This function acquires a snapshot of the physical state of each channel, no matter whether the channel is driven by its own output driver or by an externally generated signal.

Physical states are sampled periodically at one millisecond intervals and passed through a debounce filter. Consequently, `states` may not accurately reflect the state of a channel that has changed its physical state within the debounce interval.

Example:

```
// Read digital input states.
u8 states;
u32 err = ERR_NONE;
if ( !s2426_ReadDin( sess, &err, &states ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.2.3 s2426_ReadDout()

Function: Read the programmed states of all sixteen digital output channels.

Prototype: `BOOL s2426_GetOutputs( SESSION sess, u32 *err, u16 *states );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`states`	Pointer to an application buffer that will receive the programmed states of the digital output channels. Each bit is associated with one channel. For example, bit 7 is associated with channel 7. Logic one indicates the associated channel is programmed to the active state, while logic zero indicates inactive state.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Get all programmed digital output states.
u32 err = ERR_NONE;
u16 states;
if ( !s2426_ReadDout( sess, &err, &states ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.2.4 s2426_SetDoutMode()

Function: Program the operating mode of one digital output channel.

Prototype: `BOOL s2426_SetDoutMode( SESSION sess, u32 *err, u8 chan, u32 mode );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`chan`	Channel number in the range 0 to 15.

`mode`	Channel operating mode:
	`DOUT2426_MODE_STANDARD`
	`DOUT2426_MODE_PWM`

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function configures the operating mode of one digital output channel. Each channel can operate in either the Standard mode or PWM mode. In Standard mode, the channel state can be manually programmed by calling `s2426_WriteDout()`. The state is automatically controlled by the I/O module in PWM mode, with duty cycle and frequency programmed by calling `s2426_WritePwm()`.

Example:

```
// Configure channel 2 for PWM operation.
u32 err = ERR_NONE;
if ( !s2426_SetDoutMode( sess, &err, 2, DOUT2426_MODE_PWM ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.2.5 s2426_WriteDout()

Function: Program all digital output channels.

Prototype: `BOOL s2426_WriteDout( SESSION sess, u32 *err, u16 states );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`states`	Desired output states. Each bit is associated with one channel. For example, bit 7 is associated with channel 7. Any bit set to one indicates the associated channel is to be set to the active state (driven low); zero indicates the channel is to be set to the inactive state (pulled high).

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Program digital outputs to 0x5A17.
u32 err = ERR_NONE;
if ( !s2426_WriteDout( sess, &err, 0x5A17 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.2.6 s2426_WritePwm()

Function: Program the PWM ratio for one digital output channel.

Prototype: `BOOL s2426_WritePwm( SESSION sess, u32 *err, u8 chan, u16 ontime, u16 offtime );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`chan`	Channel number in the range 0 to 15.
`ontime`	PWM on time in milliseconds.
`offtime`	PWM off time in milliseconds.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function applies to channels operating in PWM mode; it has no affect on channels operating in Standard mode. The `ontime` and `offtime` arguments specify the amount of time that the channel is to be in the active and inactive states, respectively. If `ontime` is zero and `offtime` is non-zero then the output will always be inactive. Similarly, if `offtime` is zero and `ontime` is non-zero then the output will always be active. The output state is indeterminate if both `ontime` and `offtime` are set to zero.

The designated digital output channel will switch to the active state and remain active until `ontime` has elapsed, then it will switch to the inactive state and remain in that state until `offtime` has elapsed. This sequence will repeat with the same duty cycle and frequency until one of these events occurs:

- The `ontime` and/or `offtime` is changed by calling `s2426_WritePwm()`.
- The channel's operating mode is switched from PWM to Standard. The operating mode can be switched under software control by calling `s2426_SetDoutMode()` or `s24xx_ResetIo()`, and it may also be automatically switched in response to a module hardware reset.

Example:

```
// Set the PWM ratio for channel 5: on for 20 ms, off for 30 ms.
u32 err = ERR_NONE;
if ( !s2426_WritePwm( sess, &err, 5, 20, 30 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.3 Analog I/O Functions

7.3.1 s2426_WriteAout()

Function: Program the analog output voltage level.

Prototype: `BOOL s2426_WriteAout( SESSION sess, u32 *err, s16 setpoint, BOOL correct );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`setpoint`	Desired output level: -32768 (-10V) to 32767 (+10V).
`correct`	Set to True to apply calibration correction, or False to write the setpoint directly to the analog output interface without modification.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Program analog output to -5V.
u32 err = ERR_NONE;
if ( !s2426_WriteAout( sess, &err, -16384, TRUE ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.3.2 s2426_ReadAout()

Function: Read the programmed analog output level.

Prototype: `BOOL s2426_ReadAout( SESSION sess, u32 *err, s16 *setpoint );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`setpoint`	Pointer to buffer that will receive the current analog output level.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Read and display the programmed analog output level.
u32 err = ERR_NONE;
s16 setpoint;
if ( !s2426_ReadAout( sess, &err, &setpoint ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "Aout: %f Volts\n", (double)setpoint / 3276.8 );
```

7.3.3 s2426_ReadAdc()

Function: Read all eight digital input channels.

Prototype: `BOOL s2426_ReadAdc( SESSION sess, u32 *err, S2426_ADC_SAMPLE *samples, BOOL timestamps );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`samples`	Pointer to an array of eight `S2426_ADC_SAMPLE` structures. These will be filled with digitized sample data and, if desired, timestamps for those samples. Channels 0 through 5 measure external signals, while channels 6 and 7 measure on-board reference standards.
`timestamps`	Set to True to receive timestamps with digitized sample data.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function returns sample data from the module's eight analog input channels.

Example:

```
// Read and display the six external analog input channels.
int i;
u32 err = ERR_NONE;
S2426_ADC_SAMPLE samp[8];  // allocate space for 8 chans
if ( !s2426_ReadAdc( sess, &err, &samp, TRUE ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else {
    for ( i = 0; i < 6; i++ )
        printf( "chan %d: %f at t=%d\n", i, samp[i].volts, samp[i].timestamp );
}
```

7.4 Encoder Functions

7.4.1 s2426_ReadEncoderCounts()

Function: Read the encoder counter.

Prototype: `BOOL s2426_ReadEncoderCounts( SESSION sess, u32 *err, u32 *counts, u32 *timestamp );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`counts`	Pointer to buffer that is to receive the counts.
`timestamp`	Pointer to buffer that will receive the timestamp. The timestamp is a snapshot of the I/O module's system timer at the moment `counts` is sampled. Set to NULL if the timestamp is not needed. See section 5.6 for more information about timestamps.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Upon exit, `counts` will contain a snapshot of the encoder counter.

Example:
```
// Get current counts from counter.
u32 err = ERR_NONE;
u32 counts;
if ( !s2426_ReadEncoderCounts( sess, &err, &counts, NULL ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

Example:
```
// Get counts and timestamp.
u32 err = ERR_NONE;
u32 counts;
u32 tstamp;
if ( !s2426_ReadEncoderCounts( sess, 12, 3, &counts, &tstamp ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "counts=%d at t=%d microseconds\n", counts, tstamp );
```

7.4.2 s2426_WriteEncoderMode()

Function: Program the encoder interface operating mode.

Prototype: `BOOL s2426_WriteEncoderMode( SESSION sess, u32 *err, u32 clock, u32 preload );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`clock`	Counter clock mode. Specify one of these values: `NONQUADRATURE` – single clock phase (e.g., tachometer) `QUADRATURE_X1` – quadrature clock (e.g., encoder), count at clock frequency `QUADRATURE_X2` – quadrature clock, count at two times the clock frequency `QUADRATURE_X4` – quadrature clock, count at four times the clock frequency
`preload`	Specifies whether an active edge on the index input will cause the counter to be parallel loaded from the preload register. Specify one of these values: `ENC_PRELOAD_DISABLE` – index active edge does not cause counts change `ENC_PRELOAD_ENABLE` – index active edge causes counts to change to preload value

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Set count rate to 4x the encoder frequency, with no preload upon index active edge.
u32 err = ERR_NONE;
if ( !s2426_WriteEncoderMode( sess, &err, QUADRATURE_X4, ENC_PRELOAD_DISABLE ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.4.3 s2426_WriteEncoderPreload()

Function: Store a value in the encoder interface preload register.

Prototype: `BOOL s2426_WriteEncoderPreload( SESSION sess, u32 *err, u32 preload );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`preload`	Value to be written to the preload register.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Set preload register to 100.
u32 err = ERR_NONE;
if ( !s2426_WriteEncoderPreload( sess, &err, 100 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.4.4 s2426_ReadEncoderPreload()

Function: Return the contents of the encoder interface preload register.

Prototype: `BOOL s2426_ReadEncoderPreload( SESSION sess, u32 *err, u32 *preload );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`preload`	Pointer to buffer that will receive preload value.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Example:

```
// Read preload register.
u32 err = ERR_NONE;
u32 preload;
if ( !s2426_ReadEncoderPreload( sess, &err, &preload ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "Preload counts: %d\n", preload );
```

7.5 Comport Functions

7.5.1 s2426_ComportOpen()

Function: Configure the comport and attach it to the specified session.

Prototype: `BOOL s2426_ComportOpen( HSESSION sess, u32 *err, u32 br, u32 parity, u32 databits, u32 stopbits );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`br`	Baud rate. Specify any standard value from 110 to 115200.

`parity`	Parity type. Specify one of these values:
	`COMPORT_PARITY_NONE`
	`COMPORT_PARITY_ODD`
	`COMPORT_PARITY_EVEN`

`databits`	Number of data bits per character: 5 to 8.
`stopbits`	Number of stop bits per character: 1 or 2.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: The comport must be attached to a session to enable the client to send or receive data over the module's serial communication interface. If the comport is already attached to another session, this function will fail and the error code will be set to `ERR_COMPORTATTACHED`.

Example:

```
// Attach session to comport and set to 9600 baud, no parity, 8 data, 1 stop.
u32 err = ERR_NONE;
if ( !s2426_ComportOpen( sess, &err, 9600, COMPORT_PARITY_NONE, 8, 1 ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.5.2 s2426_ComportClose()

Function: Close comport and detach it from session.

Prototype: `BOOL s2426_ComportClose( HSESSION sess, u32 *err );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: This function flushes the target comport's serial transmitter and receiver queues and detaches the comport from the current session.

The comport must be attached to the session when this function is called. If the comport is closed, the function will return False and `err` will be set to `COMPORT_UNATTACHED`.

Example:

```
// Close comport.
u32 err = ERR_NONE;
if ( !s2426_ComportClose( sess, &err ) )
    printf( "Problem closing comport\n" );
```

7.5.3 s2426_ComportRead()

Function: Fetch data and line state events from the comport's serial receiver queue.

Prototype: `int s2426_ComportRead( HSESSION sess, u32 *err, void *buf, int len, BOOL wait );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`buf`	Pointer to a buffer that will receive the comport data.
`len`	Size of `buf` or maximum number of bytes to receive, whichever is smaller.
`wait`	Enable blocking operation. When True, the function will return when data is available or upon error. When False, the function will return immediately regardless of data availability.

Returns: One of these values:

Return Value	Description
`0 to len`	Number of received comport bytes that were copied to `buf`. Zero may indicate an error.
`-1`	Line state change notification, with event flags stored in `buf[0]`. Up to four event flags will be set to True to indicate the associated events were detected: `COMPORT_BREAK_ERROR`, `COMPORT_FRAMING_ERROR`, `COMPORT_PARITY_ERROR`, and `COMPORT_OVERRUN_ERROR`.

Notes: This function transfers received data and line state change notifications from the comport's serial receiver queue into `buf[]`. If stream data was read, a positive value will be returned that indicates the number of bytes copied to `buf`. Zero will be returned if the receiver queue is empty or an error occurred. If the underlying connection was closed, the function will return zero and `err` will be set to `ERR_CONNCLOSED`.

Line state change notifications indicate events that cannot be conveyed as stream data, such as parity errors and incoming line breaks. These notifications appear in their order of occurrence in the receive stream. When this function returns -1, event flags can be found in `buf[0]` that indicates the type of line state event (or events) that occurred, and no other data will be copied to `buf`.

The comport must be attached to the specified session when this function is called. If the comport is closed, the function will return zero and `err` will be set to `COMPORT_UNATTACHED`.

Example:

```
// Fetch and display received comport data.
char buf[256];
u32 err = ERR_NONE;
// Leave space at end of buf for string terminating NUL char.
int nchars = s2426_ComportRead( sess, &err, buf, sizeof(buf)-1, FALSE );
if ( nchars > 0 ) {
    buf[nchars] = 0;  // convert to C string
    printf( "data received: %s\n", buf );
} else if ( nchars < 0 ) {
    printf( "linestate change(s):\n" );
    if ( buf[0] & COMPORT_BREAK_ERROR   ) printf( "receive break\n" );
    if ( buf[0] & COMPORT_FRAMING_ERROR ) printf( "framing error\n" );
    if ( buf[0] & COMPORT_PARITY_ERROR  ) printf( "parity error\n" );
    if ( buf[0] & COMPORT_OVERRUN_ERROR ) printf( "overrun error\n" );
} else if ( err == ERR_NONE )
    printf( "receive queue is empty\n" );
else
    printf( "Error: %s\n", s24xx_ErrorText(err) );
```

7.5.4 s2426_ComportWrite()

Function: Enqueue data for transmission on the comport.

Prototype: `u32 s2426_ComportWrite( HSESSION sess, u32 *err, void *buf, int len, BOOL wait );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`buf`	Pointer to a buffer that contains data to be enqueued for transmission.
`len`	Number of bytes to be enqueued.
`wait`	Enable blocking operation. When True, this function will return when data has been enqueued or upon error. When False, the function will return immediately regardless of whether data was successfully enqueued.

Returns: Number of data bytes that were enqueued for transmission. If an error occurs, zero is returned and an error code is stored in `err`.

Notes: This function copies data from `buf` to the comport's transmit FIFO. The comport consumes byte data from the FIFO output and transmits the bytes onto the comport's physical interface. Data bytes are transmitted in the same order they were enqueued. Previously enqueued, but unsent data pending in the FIFO will be transmitted before new data. This function doesn't actually transmit data on the comport; it only enqueues data for later transmission.

If `wait` is False and the transmit FIFO would overflow as a result of enqueing all of the new data, only a portion of the data will be enqueued and the returned value will be less than `len`.

The comport must be attached to the specified session when this function is called. If the comport is closed, the function will return zero and `err` will be set to `COMPORT_UNATTACHED`.

Example:

```
// Send a text string to the comport.
char msg[] = "This is a test\n";
u32 err = ERR_NONE;
u32 len = sizeof(msg) - 1;  // exclude NUL char at end of string
u32 nsent = s2426_ComSend( sess, &err, msg, len, TRUE );
if ( nsent == len )
    printf( "sent string\n" );
else if ( err != ERR_NONE )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "Error: no data sent\n" );
```

7.5.5 s2426_ComportIoctl()

Function: Execute an I/O control operation on the comport.

Prototype: `BOOL s2426_ComportIoctl( HSESSION sess, u32 *err, u32 ioctl, void *val );`

Argument	Description
`sess`	Session handle obtained from `s24xx_SessionOpen()`.
`err`	Pointer to error code. See Section 4.3.1 for details.
`ioctl`	Operation type code. Specify one of these values: `BAUDRATE` Set/get baud rate `PARITY` Set/get parity type `DATABITS` Set/get character size in bits `STOPBITS` Set/get number of stop bits per character `TXBREAKON` Begin transmit break (`val` not used) `TXBREAKOFF` End transmit break (`val` not used) `RXFLUSH` Flush receive pipeline (`val` not used)
`val`	Pointer to storage for the operation's value. Before calling this function, set `val` to the desired new value, or to zero to keep the current comport setting. Upon return, `val` contains the current setting (in the case of `BAUDRATE`, `PARITY`, `DATABITS` and `STOPBITS`).

Returns: True if the operation was successful, otherwise False is returned and `err` will contain the associated error code.

Notes: Some operations (`BAUDRATE`, `PARITY`, `DATABITS`, and `STOPBITS`) make use of the `val` argument, while others (`TXBREAKON`, `TXBREAKOFF` and `RXFLUSH`) don't. For those that do, setting val to a non-zero value and then calling this function will cause the comport's control setting to change to the target value. For example, the comport's baud rate can be changed by storing the desired baud rate value in `val` and then calling this function with the operation type set to `BAUDRATE`.

The current setting will remain in effect if `val` contains zero when this function is called. This can be used to read a setting without making changes to it. The comport's current setting, whether new or unchanged, is returned in `val`. Upon returning, `val` indicates the actual setting in effect on the comport, which can differ from the desired setting if it is a non-standard value. For example, attempting to set the baud rate to 9625 will result in it being set to 9600.

`TXBREAKON` and `TXBREAKOFF` are used to begin and end a line break condition on the comport transmitter.

`RXFLUSH` may be used to "reset" the receive pipeline to compensate for a receive error (e.g., a parity, framing or overrun). When a receive error occurs, the contents of the receiver pipeline should be considered corrupt and therefore all queued data should be discarded.

The comport must be attached to the specified session when this function is called. If the comport is closed, the function will return zero and `err` will be set to `COMPORT_UNATTACHED`.

Example:

```
// Set baud rate to 9600.
u32 err = ERR_NONE;
u32 val = 9600;
if ( !s2426_ComportIoctl( sess, &err, BAUDRATE, &val ) )
    printf( "Error: %s\n", s24xx_ErrorText(err) );
else
    printf( "Baud rate was set to %d\n", val );
```