Model 2600 Family
Programming Guide

November 16, 2009

Sensoray Co., Inc.
7313 SW Tech Center Dr., Tigard, Oregon 97223
voice: 503.684.8005, fax: 503.684.8164, e-mail: sales@sensoray.com
www.sensoray.com

Table of Contents

Table of Contents

Table of Contents

Chapter 1: Introduction

1.1 Scope

This document describes the contents and use of the distribution media that is supplied with boards belonging to the Sensoray model 2600 product family.

1.2 Description

The 2600 family middleware is an executable software module that will interface one or more Sensoray Model 2601 Main Modules (MMs) to a client application program of your design. A rich set of middleware API functions provides access to all resources on each MM, including its four asynchronous communication ports and I/O module gateway, as well as to all I/O modules that are connected to the MMs. Any number of MMs may be concurrently interfaced by the middleware, limited only by system resources.

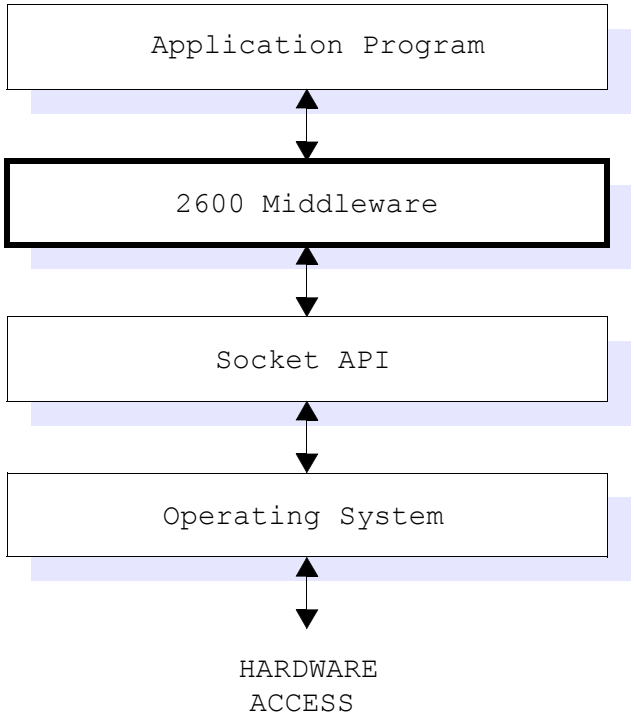
Two versions of the executable middleware are supplied in the distribution media: one for Windows and one for Linux.

1.2.1 Block Diagram

The middleware consists of a library file that serves as an interface between the application program and Ethernet network. The Windows version is implemented as a dynamic link library, `S2600.DLL`. The Linux version is a static library, `lib2600.a`.

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

Figure 1: Block diagram of the software hierarchy.



Chapter 2: Installation

2.1 Executable Software Components

Because the middleware is dependent on a network socket API, a suitable socket interface must be installed and properly configured. In addition, the middleware must be correctly installed on a 2600 client system as described in the following sub-sections.

2.1.1 Windows

Dynamic link library file `S2600.DLL` must be located in either (1) the directory containing the application that uses it, or (2) in one of the directories in the operating system's DLL search path (e.g., "`C:\WINDOWS\SYSTEM`").

2.1.2 Linux

Library file `lib2600.a` must be located in the linker's library search path. You can either (1) locate the library in one of the linker's default search path directories, or (2) 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 explicitly specify the library path:

```
gcc -g -o clientapp clientapp.o -L. -l2600
```

In this case, the "`-L.`" indicates that the current directory is to be searched for library files, and the "`-l2600`" requests linking of the `lib2600.a` library file.

2.1.3 Other

Source files are included in the SDK to enable you to port it to another operating system or cpu. Refer to the linux directory for a reference design that can serve as a basis for porting.

2.2 Application SDK Components

Distribution media for the Model 2600 family includes source-code files and demo applications that are designed to accelerate the development of your application program:

2.2.1 Windows

`win2600.c`	Functions used for dynamically linking to `S2600.DLL`. Compile and link this into any C/C++ application that calls functions in `S2600.DLL`.
`win2600.h`	Windows-specific. Include this in all C/C++ application modules that call functions in `S2600.DLL`.
`app2600.h`	Generic declarations. This file is included in `win2600.h`.
`s26app.h`	Windows-specific declarations. This file is included in `app2600.h`.
`s2600.bas`	Declarations required for Visual Basic applications. Include this file in any VB project that calls functions in `S2600.DLL`. Note: this is not compatible with VB.NET.

2.2.2 Linux

`app2600.h`	Generic declarations. Include this in all C/C++ application modules that call functions in `lib2600.a`.
`s26app.h`	Linux-specific declarations. This file is included in `app2600.h`.

Chapter 3: Fundamentals of Usage

3.1 Board Addressing

3.1.1 MM Handles

Each Model 2601 board—which is also referred to as a *main module*, or simply *MM*—is assigned a reference number called a *handle*. A handle is the logical address of a MM. Many of the middleware functions include the MM handle as an argument so that the function calls will be directed to a specific MM. The first MM is assigned the handle value 0. MM handles are numbered sequentially up to the value `N-1`, where `N` is the number of MMs in the system.

MM handles are not OS-allocated handles in the traditional sense, but rather are integer values that are assigned by the application program. When a MM is first declared to the middleware by the application program, any valid, unused handle may be specified for that MM. Once a handle has been assigned to a MM, it must not be used by any other MM.

3.1.2 IP Address

In addition to the MM handle, which is the logical address for a MM, each MM also has a physical address. The physical address is the Internet Protocol (IP) address at which the MM resides. A MM's physical address must always be specified to the middleware in dotted decimal form (e.g., "`192.168.3.35`").

3.1.2.1 Setting the MM's IP Address

A Windows utility program, `cfg2601.exe`, is supplied on the distribution media. This program enables you to examine and change a MM's IP address. Follow these steps to program the MM's IP address:

1. Turn off power to the target MM.
2. Attach a null modem cable from the MM's COM4 connector to any available comport on a PC.
3. Execute the utility program by typing "`CFG2601 x`" where `x` is the comport being used on the PC. For example, type "`cfg2601 2`" if COM2 is being used on the PC.
4. Wait until the program informs you that it is waiting for the 2601 to be reset.
5. Apply power to the MM.
6. Using the program's menu system, you may examine and change the MM's IP address.

It is strongly recommended that you assign IP addresses that are specifically reserved for private networks, such as `10.X.X.X` or `192.168.X.X`, to the MMs in your system.

3.2 Thread-Safety

With few exceptions, all middleware functions are thread-safe. Applications should be designed such that the thread-unsafe functions will not be re-entered while in use by other threads or processes. This is usually not difficult to achieve in practice as unsafe functions are associated with middleware initialization and shutdown.

3.3 Programming Examples

The C programming language has been used to code all programming examples. In most cases the programming examples can be easily adapted to other languages.

Many of the examples specify symbolic constants that are defined in `App2600.h`, which can be found on the distribution media.

3.3.1 Data Types

Data values passed to or received from library functions belong to a small set of fundamental data types. All custom data types employed by the API 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

3.4 Library Linking

3.4.1 Windows

An application that calls functions in `S2600.DLL` must first link to the DLL, and when terminated, an application must unlink from the DLL so that resources used by the DLL will be released. The means by which DLL linking and unlinking is implemented depends on your development environment.

- **Visual Basic**: VB applications do not require calls to `S26_DLLOpen()` or `S26_DLLClose()` because they automatically link when any DLL function is first called, and automatically unlink when the application terminates. Instead, VB applications must explicitly call `S26_DriverOpen()` and `S26_DriverClose()` when starting and terminating, respectively.
- **C/C++**: applications must call `S26_DLLOpen()` to link to the DLL before calling any of its functions, and `S26_DLLClose()` when the application terminates. Note that these two functions are not part of the DLL; they are provided in the `Win2600.c` module on the distribution media.
- **Other**: If you are using a development tool that does not perform automatic DLL linking, you must create functions equivalent to `S26_DLLOpen()` and `S26_DLLClose()` as shown in the `Win2600.c` module on the distribution media.

3.4.2 Linux

An application that calls functions in `lib2600.a` must be statically linked to the library when the application is built.

For example, suppose you created a simple C-language program named `app.c`, which you have compiled to produce object file `app.o`. In addition, you have previously located `lib2600.a` in your project directory. You can now execute the following command line to link the library and produce the `app` executable.

```
gcc -g -o app app.o -L. -l2600
```

3.5 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 **`S26_DriverOpen()`** to initialize the middleware. This should always be the first middleware function executed by a client application program. *Windows only:* this is called automatically if you call **`S26_DLLOpen()`**.
2. For each MM, call **`S26_OpenBoard()`** to enable communication with the target MM.
3. For each MM, call **`S26_ResetNetwork()`** to initialize the target MM and verify that it is detected, fault-free and ready to communicate. If more than one Ethernet client will be communicating with the target MM, this function should be called only once by a designated “master” client ; all other clients should wait until the master has called this function, and then they are free to communicate with the MM.
4. For each MM, call **`S26_RegisterAllIoms()`** to detect and register all I/O modules (IOMs) that are connected to the MM.
5. To guarantee proper cleanup upon application termination, call **`S26_DriverClose()`** once. *Windows only:* **`S26_DriverClose()`** is called automatically if you call **`S26_DLLClose()`**.

Chapter 4: Initialization and Status Functions

4.1 Overview

The functions described in this chapter are used to open, initialize and close the middleware library and all Main Modules in the 2600 system.

4.2 Middleware Initialization and Shutdown

4.2.1 S26_DriverOpen()

Function: Initializes the middleware.

Prototype: `u32 S26_DriverOpen( u32 NumMMs );`

Parameter	Type	Description
`NumMMs`	`u32`	Number of MMs (2601 modules) in the system.

Returns: u32 containing an error code. One of the following values is returned:

Value	Description
`DRVERR_NONE`	No errors were detected; middleware is open.
`DRVERR_MALLOC`	The version number of the socket API is incompatible with the middleware, or TCP/IP is not properly configured on the Ethernet client.
`DRVERR_NETWORKOPEN`	There was a problem when the network interface was opened. Any of the following conditions can cause this error: 1. The version number of the socket API is incompatible with the middleware. 2. TCP/IP is not properly configured on the Ethernet client. 3. The socket driver can't support the number of sockets required for communicating with the number of MMs in the system.
`DRVERR_CRITICALSECTION`	There was a problem creating internal semaphores.
`DRVERR_REOPEN`	The application attempted to open the API again before closing it.

Errors can often be resolved by reconfiguring your network settings. In Windows, you can do this by changing the TCP/IP settings through the network control panel.

Notes: This function allocates memory for and initializes the MM middleware. `S26_DriverOpen()` must be successfully invoked before any other middleware functions are called. Each Ethernet client must call this function exactly once. Multi-threaded applications must invoke this function one time before any other middleware functions are called by any of the application's threads.

Example: See section 4.4.4.

4.2.2 S26_DriverClose()

Function: Closes the middleware.

Prototype: `void S26_DriverClose();`

Returns: None.

Notes: If the prior call to `S26_DriverOpen()` was successful, this function must be called before the application closes to ensure that the middleware shuts down gracefully and properly releases all resources. If an error code was returned by `S26_DriverOpen()`, however, the application should not call `S26_DriverClose()`. This must be the last middleware function called by the application.

Example: See section 4.4.4.

4.2.3 S26_DriverVersion()

Function: Returns a middleware version string.

Prototype:
```
const char * S26_DriverVersion( void );
```

Returns: Pointer to the middlware's version string (e.g., "1.0.10").

Notes: This function is only available in middleware version 1.0.10 or higher.

Example:
```
// Fetch and display middleware version string.
printf( "%s", S26_DriverVersion() );
```

4.3 MM Initialization and Shutdown

4.3.1 S26_BoardOpen()

Function: Enables communications between an application and MM.

Prototype:
```
u32 S26_BoardOpen( u32 hbd, char *ClientAdrs, char *MMAdrs  );
```

Parameter	Type	Description
`hbd`	`u32`	MM handle. Use any value between 0 and N-1, where N is the number of MMs in the system. Do not use a value that has already been used for another MM.
`ClientAdrs`	`char*`	Pointer to a null-terminated string that specifies the Ethernet client's IP address in dotted decimal format. In the case of a multi-homed client, which is a client that has two or more network interfaces (NICs), specify the IP address of the NIC that is to be used. This should be set to zero if the client has only one NIC; this will cause the middleware to use the default NIC for communicating with the MM.
`MMAdrs`	`char*`	Pointer to a null-terminated string that specifies the MM's IP address in dotted decimal format. This is the address at which the MM is programmed to respond.

Returns: u32 consisting of a set of active-high error bit flags. All flags will contain zero if the board was successfully opened. If the board could not be opened, at least one of the flags will be asserted:

Symbolic Name	Description
`ERR_BADHANDLE`	An invalid MM handle was specified.
`ERR_BINDSOCKET`	The MM's network sockets could not be bound to the client's IP address. Some operating systems (e.g., Windows 98) do not support multiple NICs. In such systems, you must specify zero as the address for your NIC.
`ERR_CREATESOCKET`	One or more of the MM's network sockets could not be created.

Notes: `S26_BoardOpen()` registers a MM with the middleware so that communication between the application program and the MM will be enabled. Each MM must be registered before calling any other functions that reference the MM. In the context of this function, "opening" the MM is synonymous with registering the MM.

After opening the MM, the application may use the handle in all other functions that require a board handle.

Do not register the same MM at two different handles. This can result in unpredictable behavior and may cause your system to become unstable.

Example:

```
// Declare MM number 0, client is not multi-homed.
char MMAdrs[] = "10.10.10.1";
u32 errflags = S26_BoardOpen( 0, 0, MMAdrs );
if ( errflags )
{
    // .... Handle error ....
}
```

Example:

```
// Declare MM number 0, client is multi-homed.  Note that some operating
// systems do not support more than one network card.
char ClientAdrs[] = "192.168.10.1";
char MMAdrs[] = "10.10.10.1";
u32 errflags = S26_BoardOpen( 0, ClientAdrs, MMAdrs );
if ( errflags )
{
    // .... Handle error ....
}
```

4.3.2 S26_BoardClose()

Function: Unregisters a MM with the middleware.

Prototype: `void S26_BoardClose( u32 hbd );`

Parameter	Type	Description
hbd	u32	MM handle.

Returns: None.

Notes: This function unregisters a MM that has been previously registered by `S26_BoardOpen()`. Each MM that has been registered by `S26_BoardOpen()` must be unregistered when it is no longer needed by an application. All open MMs are automatically closed by `S26_DriverClose()`, so it is not necessary to explicitly call `S26_BoardClose()` when shutting down your application.

`S26_BoardClose()` severs the middleware's communication link between the application program and the MM, and frees the MM's board handle. Once freed, the board handle is available for assignment to the same MM or to any other MM.

All IOMs that have been registered for the target MM are unregistered. This can be useful if you will be connecting IOMs to or disconnecting IOMs from the MM while the application is running.

`S26_BoardClose()` does not alter the state of the MM. The MM's communication watchdog interval remains in effect, and the gateway and comports continue any autonomous operations that are already in progress. Since all communications will be severed between the client and the MM, the application should ensure that no gateway or comport transactions are in progress when `S26_BoardClose()` is called.

Example:

```
// Close MM number 0.
S26_BoardClose( 0 );
```

4.4 Status and Control

4.4.1 S26_ResetNetwork()

Function: Resets a MM and all connected IOMs and synchronizes communications between the client and the MM.

Prototype: `u32 S26_ResetNetwork( u32 hbd );`

Parameter	Type	Description
hbd	u32	MM handle.

Returns: u32 value that indicates whether the reset operation was successful. Returns a non-zero value if successful, or zero if the reset operation failed.

Notes: This function attempts to reset the specified MM and all of its connected IOMs, then it verifies that the MM has undergone a reset by checking to see if the MM's `HRST` flag is set. When the MM reset is confirmed, the `HRST` flag is cleared and the function returns a non-zero value to indicate that the MM is ready to communicate with the Ethernet client. If the MM does not respond, or it fails any part of the synchronization sequence, a zero value is returned to indicate the problem.

`S26_ResetNetwork()` should be called after opening the MM and before calling any of the gateway or comport transaction functions. In addition, this function should be called to resynchronize the client to the MM if the MM experiences an unexpected reset operation resulting from a communication watchdog time-out.

Assuming operation on a private LAN, a delay of up to seven seconds can elapse before this function returns, although the typical delay is much shorter. A delay of up to four seconds can occur if `S26_ResetNetwork()` is called while the MM is already undergoing a reset in response to a network communication watchdog time-out. The maximum delay will result if the MM is not reachable.

Example: See section 4.4.4.

4.4.2 S26_ResetIom()

Function: Executes an IOM module reset.

Prototype: `u32 S26_ResetIom( u32 hbd, IOMPORT IomPort, u32 msec, u32 retries );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`msec`	`u32`	Maximum time, in milliseconds, to allow for the MM to reply.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Notes: The target IOM will immediately undergo a reboot and it is unregistered with the middleware. The calling thread is blocked while the reboot is in progress, and while communication is being established with the target module following the module reset. If communication with the target module is successfully restored, the target module is re-registered with the MM and the target's RST flag is cleared.

It is strongly recommended that no other transactions be in progress for the target IOM's MM while this call is active. Other threads may resume transaction processing after the reset operation is finished and communication has been restored with the target module.

Use `S26_ResetNetwork()` instead of `S26_Sched2600_Reset()` in cases where more than one module is to be reset or the MM must be reset.

Example:
```
// Reset the IOM connected to MM number 0, IOM port 6.
S26_ResetIom( 0, 6, 1 );
```

4.4.3 S26_RegisterAllIoms()

Function: Detects and registers all IOMs connected to a MM.

Prototype: `u32 S26_RegisterAllIoms(u32 hbd,u32 msec,u16 *nIoms, u16 *types, u8 *stat,u32 retries);`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`msec`	`u32`	Maximum time, in milliseconds, to allow for the MM to reply.
`nIoms`	`u16 *`	Pointer a 16-bit application buffer that will receive the number of detected IOMs. Set to zero if the detected IOM count is not needed.
`types`	`u16 *`	Pointer to a 16*16-bit array that will receive a list of the detected IOM types. `types[i]` will receive the model number of the IOM that is connected to IOM port number `i`. Set to zero if the list of detected IOM types is not needed.
`stat`	`u8 *`	Pointer to a 16-byte buffer that will receive the status bytes from all detected IOMs. Set to zero if you are not interested in receiving IOM status info.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Notes: `S26_RegisterAllIoms()` attempts to detect the presence of all IOMs that are connected to the specified MM. Each detected IOM is then queried to determine its type (i.e., model number), and its RST and CERR (but not any IOM-specific) status flags are reset to zero. Finally, the IOM types are registered to enable type-specific I/O operations to be scheduled and executed.

If no errors are detected, `nIoms` receives the number of detected IOMs, `types[]` receives a list of all detected IOM types, and `stat[]` receives the status flags returned from all detected IOMs. `nIoms` will have a value from 0 to 15, while each element of `types[]` will contain the IOM model number connected to the corresponding IOM port, or zero if no IOM is present at the port. Set any of these pointer arguments to zero if the corresponding values are not needed by the application.

This function must be called before any I/O operations are transacted with IOMs. It should be called after opening the MM and before calling any of the gateway transaction functions. In addition, this function should be called if the MM experiences an unexpected reset operation resulting from a communication watchdog time-out.

Example: See section 4.4.4.

4.4.4 Programming Examples

4.4.4.1 Windows

```
int main()
{
    u32 faults;
    char  MMAdrs[] = "10.10.10.1"
    int   NumIoms;
    u16  IomList[16];
    u8  IomStatus[16];

    // Open the MM middleware.
    if ( ( faults = S26_DLLOpen() ) != 0 )
    {
        //
        // .... Handle error ....
        //

        return faults;
    }

    // Open MM number 0 and process any errors.
```

```
    if ( ( faults = S26_BoardOpen( 0, 0, MMAdrs ) ) != 0 )
    {
        //
        // .... Handle error ....
        //

        S26_DLLClose();
        return faults;
    }

    // Reset MM number 0 and all of its connected I/O modules.
    S26_ResetNetwork( 0 );

    // Detect and register all IOMs connected to MM number 0.
    S26_RegisterAllIoms( 0, 1000, &NumIoms, IomList, IomStatus, 1 );
    printf( "%d IOMs were detected.\n" );

    //
    // .... Do all I/O operations and run the application's main function ....
    //

    // Close the 2600 system middleware.  S26_BoardClose() is called implicitly.
    S26_DLLClose();
    return 0;
}
```

4.4.4.2 Linux

```
int main()
{
    u32 faults;
    char  MMAdrs[] = "10.10.10.1"
    int   NumIoms;
    u16  IomList[16];
    u8  IomStatus[16];

    // Open the MM middleware.
    if ( ( faults = S26_DriverOpen() ) != 0 )
    {
        //
        // .... Handle error ....
        //

        return faults;
    }

    // Open MM number 0 and process any errors.
    if ( ( faults = S26_BoardOpen( 0, 0, MMAdrs ) ) != 0 )
    {
        //
        // .... Handle error ....
        //

        S26_DriverClose();
        return faults;
    }

    // Reset MM number 0 and all of its connected I/O modules.
    S26_ResetNetwork( 0 );

    // Detect and register all IOMs connected to MM number 0.
    S26_RegisterAllIoms( 0, 1000, &NumIoms, IomList, IomStatus, 1 );
    printf( "%d IOMs were detected.\n" );
```

```
    //
    // .... Do all I/O operations and run the application's main function ....
    //

    // Close the 2600 system middleware.  S26_BoardClose() is called implicitly.
    S26_DriverClose();
    return 0;
}
```

Chapter 5: Transactions

5.1 Overview

The majority of middleware functions are associated with gateway and comport transactions.

A *comport transaction* consists of sending to a MM a single Ethernet packet that contains a single comport command, and then receiving and parsing the resulting Ethernet response packet.

A *gateway transaction* consists of sending to a MM a single Ethernet packet containing one or more IOM action commands, and then receiving and parsing the resulting Ethernet response packet. Gateway transaction functions are designed to insulate the application programmer from the cumbersome details of network programming and packet parsing when conversing with the IOM gateway.

Aside from the programming simplifications, the gateway functions also help to optimize I/O system performance. By grouping multiple IOM actions into a single transaction, your application will realize higher throughput and lower communication latency. Because high throughput and low latency are hallmarks of the 2600 system, an extensive set of functions are provided for controlling and scheduling IOM actions.

5.2 Gateway Transaction Process

Gateway transactions are implemented using a three-step process:

1. **Begin a new transaction**. Every transaction begins with a call to `S26_SchedOpen()`, which returns a handle to an empty "transaction object."
2. **Schedule the actions**. Once a transaction object has been obtained, zero or more IOM actions may be scheduled into the transaction by means of the numerous action scheduling functions. For example, your application could call `S26_Sched2610_SetOutputs()` to program the 48 digital I/Os on a model 2610 digital I/O module, and then it could call `S26_Sched2608_GetAins()` to fetch the 16 digitized analog inputs from a model 2608 analog I/O module. It is important to understand that these functions only *schedule* the actions for later execution; the actions are not actually executed when the action scheduling functions are called. Note that it is not required for actions to be scheduled into a transaction; it is permissible to simply create the transaction object without scheduling any actions into it.
3. **Execute the transaction**. After all desired actions have been scheduled, a call to `S26_SchedExecute()` causes all of the scheduled actions to execute in a single transaction. Actions are executed in the same order they were scheduled. When `S26_SchedExecute()` returns, the 48 digital I/Os will have switched to their new states, all digitized analog input data will be stored in an application buffer and, since it is no longer needed, the transaction object is released. If no actions were scheduled into the transaction then the transaction object is simply released; in this case, no communication with the MM will take place.

Here is some sample code that illustrates this process. Note that error checking, which should always be performed in robust applications, is not shown here:

```
u8 douts[6] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB };  // Desired DIO states.
double ains[16];  // Analog input values will be put here.

// Obtain a transaction object for MM number 0.
void *x = S26_SchedOpen( 0, 1 );

// Schedule some I/O operations into the transaction object.
S26_Sched2608_GetCalData( x, 1, 0 );      // Get 2608's calibration info.
S26_Sched2608_GetAins( x, 1, ains, 0 );   // Get 2608's analog input data.
S26_Sched2610_SetOutputs( x, 2, douts );  // Set 2610's digital output states.

// Execute the transaction and release the transaction object.
S26_SchedExecute( x, 1000, 0 );
```

5.2.1 Thread Safety

All of the gateway transaction functions are thread safe, and it is permissible for multiple transactions to exist (and be in different states) at the same time. For example, your application could be partitioned into multiple threads (e.g., analog I/O thread, digital I/O thread, serial communication thread) such that each thread asynchronously begins, schedules actions into, and executes its own private gateway transactions. To guarantee thread safety in such a case, each transaction should be started, scheduled and executed only by the thread that "owns" that transaction. In general, a transaction object should not be shared by multiple threads.

5.3 Transaction Buffers

Each MM has fifteen internal transaction buffers which are kept in a pool. This buffer pool is shared by all comport and gateway transactions. A transaction buffer is dynamically allocated from the pool when a MM transaction begins (i.e., upon receipt of a packet from an Ethernet client), and when the transaction is finished (i.e., a response packet has been sent to the client), the buffer is returned to the pool. Every client-side transaction is associated with a dedicated transaction buffer on the MM.

A maximum of fifteen transactions—in any combination of comport and/or gateway transactions—may be in progress at the same time on one MM. For example, it is possible for a MM to process transactions on all four of its comports while simultaneously processing up to eleven gateway transactions. This means that a single Ethernet client may run multiple threads and/or processes in which each thread or process concurrently executes simultaneous transactions with a single MM.

Multiple simultaneous MM transactions may involve more than one Ethernet client. For example, it is permissible for two different Ethernet clients to simultaneously execute gateway transactions on a single MM. Each of the fifteen possible simultaneous MM transactions may be invoked by any arbitrary Ethernet client. The MM supports up to four Ethernet clients.

An error will occur on the MM if its transaction buffer pool is empty when a transaction begins. This can happen if a packet is received from a client while the maximum possible number of simultaneous MM transactions are already in progress. In such cases, the newest transaction will be dropped by the MM and no response will be sent to the client.

5.4 Blocking Behavior

Gateway transactions may be managed by either blocking or non-blocking functions. In contrast, all comport transaction functions are blocking operations.

In the case of a gateway transaction, execution of the calling thread is blocked by `S26_SchedExecute()` until a response packet is received from the target MM. This works well if your application has one or more dedicated gateway transaction threads because other threads can run while the transacting threads are blocked. There may be situations, however, in which it is impractical to employ separate transaction threads and, furthermore, other application processing must procede while transactions are in progress. To support these cases, several middleware functions have been provided to enable non-blocking gateway transactions.

To execute a non-blocking gateway transaction, call `S26_SchedExecuteStart()` instead of `S26_SchedExecute()`. This will initiate the transaction (i.e., send the Ethernet command packet to the MM) but will return immediately without waiting for a response packet.

At any convenient time after calling `S26_SchedExecuteStart()`, you may call `S26_SchedExecuteIsResponded()` to determine whether a response packet has arrived. If no response has arrived, the application may continue on with other tasks, calling `S26_SchedExecuteIsResponded()` again at any later times to poll as needed. If and when all other tasks have been completed, the application can continue to poll in this manner or it may call `S26_SchedExecuteWait()` to block until a response packet has arrived.

After the application determines that a response packet has been received, it must call `S26_SchedExecuteFinish()` to process the response packet and release the transaction object's resources.

5.5 Errors

With the exception of `S26_SchedOpen()`, all gateway and comport transaction functions return an enumerated error code. These error codes are referenced by their symbolic names as defined in `app2601.h`. Error codes occupy the most significant three bytes of a u32 value. Some error types return extended information in the error code's least significant byte.

Transaction error codes have the following meanings:

Symbolic Name	Com	Description	Error Code LSB
GWERR_BADVALUE	Yes	An illegal argument value was specified. For example: * MM handle is greater than or equal to the number of declared MM's. * IOM port number is outside the range 0 to 15, or 0xFF for the gateway. * A channel number does not exist on the target IOM. * A numerical value exceeds permitted limits.	Illegal value's position in the function argument list.
GWERR_IOMCLOSED	No	An attempt was made to schedule an IOM action for an IOM that is not open. This can happen if the application attempts to communicate with an IOM that has not been registered by S26_RegisterAllIoms().	Iom port number of the closed module.
GWERR_IOMERROR	No	One or more IOM communication error flags (CERR) are asserted. See the IomStatus[] array, which is populated by S26_SchedExecute(), for details.	Iom port number of the module for which CERR is first detected.
GWERR_IOMNORESPOND	No	Error(s) were detected in an MRsp within the gateway response packet. The application should assume that all of the associated IOM's scheduled actions, as well as all later actions that were scheduled for this and any other IOMs, failed to execute properly. This error can happen if: * MRsp module identifier field does not contain the expected value, or * MRsp payload length differs from that specified by the length field, or * MRsp length field does not match the expected value.	Iom port number associated with the MRsp.
GWERR_IOMRESET	Yes	The reset flag (RST) is asserted on the MM or one or more IOMs. If the port number indicates an IOM (i.e., port number is in the range 0x00 to 0x0F) then you may analyze the contents of the IomStatus[] array, which is populated by S26_SchedExecute(), for details. If the port number is 0xFF then the RST flag is asserted on the MM.	Iom port number of the module for which RST is first detected.
GWERR_IOMSPECIFIC	No	One or more IOM-specific status flags are asserted. See the IomStatus[] array, which is populated by S26_SchedExecute(), for details.	Iom port number of the module for which this is first detected.
GWERR_IOMTYPE	No	An attempt was made to schedule an IOM action that is not supported by the registered IOM type (e.g., scheduling a digital I/O action for an analog I/O module).	Iom port number associated with the scheduling error.
GWERR_MMCLOSED	Yes	An attempt was made to communicate with a MM that is not open. The application must first call S26_BoardOpen() to open the MM for communication.	0
GWERR_MMNORESPOND	Yes	The MM failed to respond, causing the client to time-out the transaction. See section 5.5 for a discussion of transaction time-outs.	0
GWERR_PACKETSEND	Yes	The socket driver failed to transmit the gateway command packet.	0
GWERR_TOOLARGE	Yes	This can happen in two situations: * The command packet's size or the expected response packet's size exceeds the maximum UDP payload size supported by the MM (1KB). * Too many MCmds are present in the gateway command packet. The middleware permits a maximum of 100 MCmds per command packet. If this error is raised, try redistributing the transaction's actions among multiple transactions.	0
GWERR_XACTALLOC	Yes	Allocation failure. GWERR_XACTALLOC will be asserted by any gateway transaction function that requires a transaction object in its argument list, but is instead passed a null (zero value) transaction object. This can happen if more than eight comport and/or gateway transactions exist at the same time. See section 5.3 for more information.	0

In the above table, the "Com" column indicates whether the error type is applicable to comport transactions. Note that all of the error types are applicable to gateway transactions.

5.5.1 Gateway Error Propagation

When any gateway transaction error has been detected, construction of the transaction's command packet is terminated and all subsequent gateway transaction functions will fail and return the last error value. Because of this "error propagation" behavior, it is usually unnecessary to check for transaction errors after each gateway transaction function is called. Instead, all transaction errors can be caught when `S26_SchedExecute()` returns.

Error propagation is extended to include `S26_SchedOpen()`, which returns zero if it fails to create a new transaction object. Instead of checking for errors after calling `S26_SchedOpen()`, the application is permitted to schedule actions into the "void transaction" and then execute the transaction as if it had been successfully created. At any point during action scheduling into or upon execution of a void transaction, all scheduling and execution functions will return `GWERR_XACTALLOC` to indicate the transaction was not successfully created. For obvious reasons, no physical transaction will occur and no actions will be invoked.

5.5.2 Scope of Errors

Each transaction keeps track of its own errors. When a transaction error is detected, it is known only to the transaction in which it occurs. Transactions are not aware of errors that have occurred in other transactions. Transaction errors are "cleared" automatically when `S26_SchedExecute()` returns, because the associated transaction object is released. The underlying cause of a transaction error, however, may still be pending after the transaction is finished. For example, an IOM's reset flag will remain asserted until explicitly reset by the client, even though the resulting `GWERR_IOMRESET` transaction error "disappears" when the transaction is finished.

5.5.3 Error Handling

In the cases of both comport and gateway transaction errors, the application's error handler should first determine if any errors were detected; this can be quickly done by testing the error code for a zero value. If the value is not zero (`GWERR_NONE`) then the indicated error must be processed by the error handler.

It is permissible to process the error code with a "switch" statement because the error codes are enumerated values. If a switch statement is employed, however, the least significant byte of the error code must first be masked because it may contain additional information about the error. Refer to the sample applications for examples of error handling.

Important: For each successful call to `S26_SchedOpen()` there must be a corresponding call to `S26_SchedExecute()`, even if gateway errors are detected before `S26_SchedExecute()` is called. This ensures that resources allocated by `S26_SchedOpen()` will be released, thereby preventing memory leaks and other potential problems.

5.6 Time-outs

Some of the gateway transaction functions and all of the comport transaction functions include a "`msec`" argument that specifies the maximum number of milliseconds to wait for the MM to respond before declaring a time-out error (i.e., `GWERR_MMNORESPOND`). When calling these functions, the application must specify an appropriate milliseconds value. The choice of the milliseconds value depends on several factors:

- Network traffic. High network traffic, caused by activities such as video multicasting, can interfere with the timely delivery of packets to and from the MM. To prevent this, it is best to dedicate a private LAN for the 2600 I/O system.
- Router hops. Routers can lead to unpredictable latencies, especially when the other networks through which 2600 packets flow have widely varying network traffic. A good policy is to eliminate routers from the 2600 communication path. If a 2600 client requires the services of a router, it is best to install two network interfaces in the client: one for the private 2600 network and the other for the external network.
- CPU loading. A heavily loaded client-side CPU may introduce communication latency if it becomes compute-bound. The solution to this problem is to reduce CPU loading or employ a faster CPU.
- Process priorities. Other network-related processes may "trump" the 2600 middleware's network access requests if process priorities are not set appropriately. The process that communicates with the 2600 system is usually classified as a "real-time" process, and as such it should have relatively high priority. Note: high priority does not always guarantee real-time behavior, especially with non real-time operating systems such as Windows.
- MM response time. Typically, only one transaction is in progress for a particular comport at any given time. As a result, comport transaction times depend mostly on packet sizes and are therefore relatively predictable. Gateway transactions, on the other hand, are less predictable. This is because a gateway transaction time depends not only on its command and response

packet sizes, but also on how many other gateway transactions are already in progress. "Simultaneous" gateway transactions are queued by the MM and executed in the order in which their command packets are received at the MM.

A `msec` value should be chosen that is at least as long as the worst-case transaction time after allowing for all of the above factors. On the other hand, the value should be sufficiently short to ensure timely detection of a gateway transaction failure. The programming examples in this manual use a somewhat arbitrary value of 1000 milliseconds. In most cases, this is far more than enough time for a typical private LAN that imposes no routers between the client and the MM, yet it ensures that a transaction time-out will be detected within one second.

In addition to communication latencies, transaction time-outs can also be caused by dropped packets or situations in which multiple Ethernet clients are attempting to run too many simultaneous transactions.

5.7 Retries

A *transaction retry* is performed by re-sending a transaction's command packet and waiting for its response packet to arrive, or a response timeout, whichever comes first. If the MM did not previously receive the command packet, it will execute the commands, and both cache and transmit the response packet. If the MM recognizes the command packet as being a duplicate of a previously executed command packet, it will drop the packet (i.e., not execute the commands) and instead send the corresponding response packet that was previously sent and cached. This retry mechanism relieves the client application of the responsibility for communication error correction, and makes possible recovery from certain types of errors that would otherwise be unrecoverable (e.g., reading data from a comport).

All gateway and comport transactions include a `retries` value that is specified in one of the middleware function calls associated with the transaction.

When `retries` is set to a positive number, the middleware will automatically retry the transaction if it doesn't receive a reply from the MM within the transaction's specified time-out interval. Transaction retries will repeat until a reply is received from the MM or the specified number of retries have been attempted. If the maximum number of allowed retries have been attempted and there is still no response from the MM, the transaction will fail with `GWERR_MMNORESPOND` returned.

Retries are disabled when `retries` is set to zero. In this case, the transaction will fail with `GWERR_MMNORESPOND` upon the first MM response time-out.

The worst-case transaction time equals the time-out interval times `retries`. This is the total elapsed time the application will wait for a transaction to complete in the event of a MM communication failure.

A `retries` value should be chosen based on your network error rate, which in turn depends on whether collisions are possible (e.g., your installation uses hubs instead of switches), cable lengths, electrical noise, and other factors. The programming examples in this manual use a value of `1`, which is sufficient for most private LAN environments.

Chapter 6: Gateway Transaction Control

6.1 Overview

The functions in this section are used to initialize and execute gateway transactions.

6.2 Transaction Control Functions

6.2.1 S26_SchedOpen()

Function: Begins a new gateway transaction.

Prototype: `void *S26_SchedOpen( u32 hbd, u32 retries );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Handle to a new gateway transaction, or zero if the transaction could not be created.

Notes: `S26_SchedOpen()` starts the construction of a new gateway transaction. The new transaction will be empty (i.e., it will have no scheduled IOM actions). After successfully calling this function, IOM actions can be scheduled into the transaction, and when all desired actions have been scheduled, the transaction may be executed.

Except for `GWERR_MMCLOSED`, all transaction errors are negated in the new transaction. The `GWERR_MMCLOSED` error will be asserted if the target MM is closed when the transaction is created. If the MM is open, `GWERR_MMCLOSED` will be negated and it will be possible to schedule actions into and execute the transaction.

Important: To prevent resource leaks and other potential problems, `S26_SchedExecute()` or `S26_SchedExecuteFinish()` must be called for each transaction that is successfully started by `S26_SchedOpen()`. This must be done even if gateway errors were generated while scheduling IOM actions into the transaction and the errors are detected before the transaction is executed. There is no need, however, to call either `S26_SchedExecute()` or `S26_SchedExecuteFinish()` if `S26_SchedOpen()` fails to create a new transaction, although there is no harm in doing so.

Example: See the example in section 6.2.2.

6.2.2 S26_SchedExecute()

Function: Executes a transaction.

Prototype: `u32 S26_SchedExecute( XACT x, u32 msec, u8 *IomStatus );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`msec`	`u32`	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.
`IomStatus`	`u8 *`	Pointer to a 16-byte buffer that will receive the status bytes from all IOMs. Set to zero if you are not interested in receiving IOM status info.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Notes: `S26_SchedExecute()` sends the transaction's gateway command packet to the MM and waits for the MM to reply with a gateway response packet. The calling thread is blocked until either a response packet is received or the

time-out interval has elapsed, whichever occurs first. When it is received, the response packet is checked for errors and, if no errors are detected, all of the embedded IOM responses are extracted from the response packet and copied to their target application buffers.

This function is the equivalent of calling, in sequence, `S26_SchedExecuteStart()`, `S26_SchedExecuteWait()` and `S26_SchedExecuteFinish()`.

Important: The specified transaction object will no longer exist and the transaction handle will no longer be valid when this function returns. After calling this function, do not attempt to use the transaction handle again in calls to action scheduling functions.

If `IomStatus` is non-zero (i.e., it points to a 16-byte application buffer), this buffer will be populated with the status bytes received from all 16 IOMs. One status byte is populated for each IOM; for example, `IomStatus[14]` contains the status byte for the IOM that is connected to the MM's IOM port number 14. A status byte will be set to zero in cases where no IOM is connected to the port or no actions have been scheduled for the IOM.

Multiple status bytes will be received from an IOM if two or more module commands (MCmds) are addressed to the same IOM within a single transaction. This can happen for various reasons:

- Two actions scheduled for an IOM are separated by an action that is scheduled for a different IOM. In this case the new MCmd is implicitly forced by the application program, because a new MCmd is required whenever an action is scheduled for an IOM that differs from the previous action's IOM.
- A new MCmd is automatically forced by an action that would have overflowed the IOM's response buffer. This causes the IOM response buffer to be flushed before the new action response is generated.
- A new MCmd is automatically forced by an action that would have exceeded the maximum legal MCmd size.
- A new MCmd is automatically forced if status bits that are masked off in the current MCmd are regarded as relevant by a scheduled action. For example, `S26_Sched2600_ClearStatus()` may mask the `RST` status bit so that it will not generate an error, but most other actions, such as `S26_Sched2600_IomGetProductID()`, treat the `RST` bit as relevant. Consequently, a new MCmd will be automatically started between sequential calls to `S26_Sched2600_ClearStatus()` and `S26_Sched2600_IomGetProductID()`.

The status byte that is written to `IomStatus[]` for each IOM is formed by logically or'ing together the status bytes, after masking non-relevant bits, that are received from each of the IOM's MCmds. As a result, each of the application's IOM status bytes is the consolidation of all relevant status information from all actions with the associated IOM.

Example:

```
// Execute some I/O operations on MM number 0, and its connected IOMs.

u8 status[16];     // All IOM status bytes will be put here.
u32 gwerr;         // Transaction error code will be put here.

// Create a new transaction for MM number 0.
void *x = S26_SchedOpen( 0, 1 );

//
// ToDo: Schedule the desired I/O operations into the transaction ...
//

// Execute the transaction.  Report if errors were encountered.
if ( ( gwerr = S26_SchedExecute( x, 1000, status ) ) != 0 )
    printf( "Transaction error: %d\n", gwerr );
```

6.2.3 S26_SchedExecuteStart()

Function: Starts a transaction execution.

Prototype: `u32 S26_SchedExecuteStart( XACT x );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Notes: `S26_SchedExecuteStart()` sends the transaction's gateway command packet to the MM and then returns immediately. It can be called in place of `S26_SchedExecute()` by threads that must not block while transactions are in progress.

After calling this function, `S26_SchedExecuteIsResponded()` can be called in a non-blocking polling loop to determine when the gateway response packet has been received and is ready for processing. Alternately, `S26_SchedExecuteWait()` can be called to block the calling thread until the response packet is received. When the response packet has been received, `S26_SchedExecuteFinish()` must be called to complete the transaction.

Example: See section 6.2.6.

6.2.4 S26_SchedExecuteIsResponded()

Function: Determines whether a gateway response packet has been received.

Prototype: `u32 S26_SchedExecuteIsResponded( XACT x, u32 msec );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
msec	u32	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.

Returns: u32 containing an error code. One of the following values is returned:

Value	Description
`RESP_READY`	A response packet has been successfully received.
`RESP_BUSY`	No response has been received, and no errors have been detected.
all other values	Error code as described in section 5.5.

Notes: This function indicates the availability of a received gateway response packet for the specified transaction. It may be called by threads that must not block while transactions are in progress. The specified transaction should already be executing as a result of a prior call to `S26_SchedExecuteStart()`.

Example: See section 6.2.6.

6.2.5 S26_SchedExecuteWait()

Function: Waits for a gateway response packet to be received.

Prototype: `u32 S26_SchedExecuteWait( XACT x, u32 msec );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
msec	u32	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Notes: This function waits for a received gateway response packet or a time-out, whichever occurs first. The calling thread will block until a packet is received or the receive times out. The specified transaction should already be executing as a result of a prior call to `S26_SchedExecuteStart()`.

Example: See section 6.2.6.

6.2.6 S26_SchedExecuteFinish()

Function: Processes a received gateway response packet.

Prototype: `u32 S26_SchedExecuteFinish( XACT x, u8 *IomStatus );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomStatus`	`u8 *`	Pointer to a 16-byte buffer that will receive the status bytes from all IOMs. Set to zero if you are not interested in receiving IOM status info.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Notes: This function assumes that a gateway response packet has already been received for the specified transaction. An error code will be returned if a received packet is not available for processing. The response packet is checked for errors and, if no errors are detected, all of the embedded IOM responses are extracted from the response packet and copied to their target application buffers.

The specified transaction object will no longer exist and the transaction handle will no longer be valid when this function returns. After calling this function, do not attempt to use the transaction handle again in calls to action scheduling functions.

Example:

```
// Do some I/O operations on MM number 0 and its connected IOMs in a non-blocking way.

u8 status[16];     // All IOM status bytes will be put here.
u32 gwerr;         // Transaction error code will be put here.

// Create a new transaction for MM number 0.
void *x = S26_SchedOpen( 0, 1 );

//
// ToDo: Schedule the desired I/O operations into the transaction ...
//

// Start the transaction executing.
S26_SchedExecuteStart( x );

// Do some other things while the transaction executes.
while ( !S26_SchedExecuteIsResponded( x, 1000 ) )
{
    //
    // ToDo: Do some other things ...
    //

    if ( no_more_things_to_do )
    {
        // Wait for transaction response packet, then exit the loop.
        S26_SchedExecuteWait( x, 1000 )
        break;
    }
}

// Process the transaction response packet.  Report if errors were encountered.
if ( ( gwerr = S26_SchedExecuteFinish( x, status ) ) != 0 )
    printf( "Transaction error: %d\n", gwerr );
```

Chapter 7: Gateway Action Scheduling

7.1 Overview

This chapter details all of the functions that are used to schedule I/O actions on IOMs and the gateway. Except where noted, all of these functions assume that a gateway transaction has been previously opened and is ready to schedule IOM actions.

7.1.1 Performance Benchmarks

Timing benchmarks are specified in all of the scheduling function descriptions. These benchmark times are nominal values that may be used to estimate gateway transaction times. Benchmarks are conservatively rated and, as a result, applications will tend to exhibit better performance than indicated by the benchmarks.

Each published benchmark specifies the time required for execution of a specific I/O action. The total gateway transaction time may be estimated by summing the action execution benchmark times, and then adding overhead for protocol stack transit, packet time on the wire, etc. Since the overhead time is influenced by elements that are not under control of the middleware, the latencies associated with these overhead items are not specified in this document.

Important: Benchmark times are conservatively rated, but they are *not* worst-case values.

7.1.2 Returned IOM Status

A status byte, consisting of a set of bit flags, is returned by an IOM when it executes a scheduled IOM action. Two of the bit flags, `STATUS_RST` and `STATUS_CERR`, are common to all IOM types. If these flags are asserted by any IOM, a transaction error of type `GWERR_IOMRESET` or `GWERR_IOMERROR`, respectively, will be generated.

In addition to the common bit flags, some IOM types support type-specific bit flags. These special flags are described in the "Type-Specific Errors" subsection near the beginning of each IOM reference section. If any of the type-specific bit flags are asserted by any IOM, a transaction error of type `GWERR_IOMSPECIFIC` will be generated.

7.1.3 Argument Lifetime

Most scheduling functions have arguments that specify data that is to be exchanged with a target module. Examples of this include the value to be written to an analog output channel (outgoing data), or a pointer to a buffer that will receive analog input data (incoming data). Each such argument has a life expectancy that depends on whether the associated data is outgoing or incoming.

7.1.3.1 Outgoing Data

All scheduling functions copy outgoing data to private internal storage before returning. Consequently, outgoing data is no longer needed and thus may be permitted to change value or go out of scope after the scheduling function returns. For example, this code is legal because `states` is copied by the scheduling function:

```
// Program solid state relay control outputs on a model 2652 IOM.
u8 states = 0x55;                          // Desired SSR output states.
void *x = S26_SchedOpen( 0, 1 );             // Obtain a transaction object.
S26_Sched2652_SetOutputs( x, 9, &states );  // Schedule the action.
states = 0xAA;                             // IT'S OK TO CHANGE THE VALUE NOW !!!
S26_SchedExecute( x, 1000, 0 );              // Execute the transaction.
```

7.1.3.2 Incoming Data

The scheduling functions handle incoming data by scheduling a callback for each incoming data argument. All scheduled callbacks will execute when the associated transaction executes. Accordingly, buffers that will receive incoming data must remain in scope until the transaction is completed. This code example illustrates a violation of this requirement:

```
// Fetch solid state relay inputs from a model 2652 IOM.
void ScheduleReadSSR( void *x )
{
```

```
    u8 states;                                  // Buffer that will receive SSR states.
    S26_Sched2652_GetInputs( x, 9, &states );   // Schedule the action.
}

void *x = S26_SchedOpen( 0, 1 );                // Obtain a transaction object.
ScheduleReadSSR( x );                           // states NO LONGER EXISTS UPON RETURN !!!
S26_SchedExecute( x, 1000, 0 );                 // Execute the transaction.  ERROR!
```

In the above example, `states` has been designated as the buffer that will receive incoming data. Unfortunately, `states` will cease to exist when `SchedultReadSSR()` returns, resulting in an error when the transaction executes.

7.2 Common IOM Actions

The functions in this section are used to schedule common IOM actions that apply to all IOM types, and in most cases, to the gateway as well. Note that these functions only *schedule* actions into a transaction; they do not cause the actions to be immediately executed. Usage of these functions is not dependent on the target IOM being any particular type, nor is it necessary for the IOM type to be registered for the referenced IOM port.

7.2.1 Type-Specific Errors

In addition to the common bit flags (`STATUS_RST` and `STATUS_CERR`), some IOM types have type-specific bit flags. These special flags may be asserted upon execution of any of the actions listed in this section. If any of these flags are asserted by any IOM, a transaction error of type `GWERR_IOMSPECIFIC` will be generated.

7.2.2 S26_Sched2600_ClearStatus()

Function: Schedules the resetting of one or more status bits for an IOM or the gateway.

Prototype: `u32 S26_Sched2600_ClearStatus( XACT x, IOMPORT IomPort, u8 BitMask );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected, or `0xFF` if the gateway is the target.
`BitMask`	`u8`	Specifies the status bits that are to be reset to zero.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms for IOMs, or 0.1 ms for the gateway.

Notes: This function schedules a `ResetFlags` action, which will reset to zero the specified bit flags in the target IOM's status byte. Refer to the `ResetFlags` action in the Model 2600 Family Instruction Manual for more information.

Example:

```
// Clear the RST flag on the IOM connected to MM number 0, IOM port 6.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2600_ClearStatus( x, 6, STATUS_RST );
S26_SchedExecute( x, 1000, 0 );
```

7.2.3 S26_Sched2600_GetAddress()

Function: Schedules the fetching of an IOM's address shunt settings.

Prototype: `u32 S26_Sched2600_GetAddress( XACT x, IOMPORT IomPort, u8 *adrs );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`adrs`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the address.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Notes: Some IOM types have provision for installing address shunts. These shunts enable the system integrator to specify an address for the module, with a value in the range 0 to 15 decimal. This function can be used to read the address shunts from IOMs that include this hardware feature.

Example:
```
// Fetch the address shunt settings from MM number 0, IOM port 6.
u8 shunts;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2600_GetAddress( x, 6, &shunts );
S26_SchedExecute( x, 1000, 0 );
printf( "Shunts = %d\n", shunts );
```

7.2.4 S26_Sched2600_GetFirmwareVersion()

Function: Schedules the fetching of the firmware version number from an IOM or the gateway.

Prototype: `u32 S26_Sched2600_GetFirmwareVersion( XACT x, IOMPORT IomPort, u16 *Version );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected, or `0xFF` if the gateway is the target.
`Version`	`u16 *`	Pointer to a 16-bit application buffer that will receive the target IOM's version number as two decimal values. The first byte is the major version number. The second byte is the minor version number.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms for IOMs, or 0.1 ms for the gateway.

Example:
```
// Fetch the IOM firmware version number from MM number 0, IOM port 6.
u8 vers[2];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2600_GetFirmwareVersion( x, 6, &vers );
S26_SchedExecute( x, 1000, 0 );
printf( "IOM version number = %d.%d\n", vers[0], vers[1] );
```

Example:
```
// Fetch the firmware version number from MM number 0.
u8 vers[2];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2600_GetFirmwareVersion( x, MODID_GATEWAY, &vers );
S26_SchedExecute( x, 1000, 0 );
printf( "MM version number = %d.%d\n", vers[0], vers[1] );
```

7.2.5 S26_Sched2600_IomGetProductID()

Function: Schedules the fetching of the model number from an IOM or the gateway.

Prototype: `u32 S26_Sched2600_IomGetProductID( XACT x, IOMPORT IomPort, u16 *ProductID );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected, or `0xFF` if the gateway is the target.
`ProductID`	`u16 *`	Pointer to a 16-bit application buffer that is to receive the product identifier. The product identifier is always expressed as a decimal value.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms for IOMs, or 0.1 ms for the gateway.

Notes: The returned value indicates the model number of the target module. For example, the decimal value 2652 is returned by the Model 2652 Solid State Relay IOM. The value 2601 is returned when the gateway is the target module.

Example:

```
// Fetch the IOM model number from MM number 0, IOM port 6.
u16 modelnum;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2600_IomGetProductID( x, 6, &modelnum );
S26_SchedExecute( x, 1000, 0 );
printf( "Model number = %d\n", modelnum );
```

7.2.6 S26_Sched2600_Nop()

Function: Schedules a "no-operation" action for an IOM or the gateway.

Prototype: `u32 S26_Sched2600_Nop( XACT x, IOMPORT IomPort );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected, or `0xFF` if the gateway is the target.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.7 ms for IOMs, or 0.1 ms for the gateway.

Notes: `S26_Sched2600_Nop()` may be used to acquire IOM status when no other actions are required.

Example:

```
// Fetch IOM status from MM number 0, IOM port 6.
u8 status[16];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2600_Nop( x, 6 );
S26_SchedExecute( x, 1000, status );
printf( "IOM_6 status = %d\n", status[6] );
```

7.3 Model 2601 Gateway

The functions in this section are used to schedule gateway actions on a MM. These functions are applicable only to Model 2601 MMs. Any attempt to call these functions for IOMs will result in a `GWERR_IOMTYPE` transaction error. Note that these functions only *schedule* actions into a transaction; they do not cause the actions to be immediately executed

7.3.1 Type-Specific Errors

The gateway employs only the `STATUS_RST` flag. It does not have a `STATUS_CERR` flag, nor does it have any type-specific flags. If the gateway's `STATUS_RST` flag is asserted, a transaction error of type `GWERR_IOMRESET` will be generated and the error code's least significant byte (which indicates the module in which the error was detected) will contain `MODID_GATEWAY`.

7.3.2 S26_Sched2601_GetInterlocks()

Function: Schedules the fetching of the MM's power interlock status.

Prototype: `u32 S26_Sched2601_GetInterlocks( XACT x, u8 *LockFlags );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`LockFlags`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the interlock power state flags. Each bit is associated with an interlock channel number. For example, bit 4 is associated with interlock channel 4. A bit flag is set to one to indicate that the interlock input has applied power, or zero to indicate that the interlock input has no applied power.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.1 ms.

Notes: The MM includes two connectors for interlock power distribution. One connector receives power from up to six interlock contacts, while the other connector serves as a daisy-chain to distribute the interlock power to IOMs. Each interlock signal is called an *interlock channel*.

Every interlock channel occupies one circuit in each of the MM's interlock power connectors. In addition, each channel is routed to a metering circuit that enables the MM to monitor the channel's voltage level. If a channel's interlock contact is closed, the interlock will supply voltage to the input connector, which in turn will convey the voltage to the output connector and metering circuit.

Example:

```
// Fetch the interlock status from MM number 0.
u8 flags;
u8 mask;
int i;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2601_GetInterlocks( x, &flags );
S26_SchedExecute( x, 1000, 0 );
for ( i = 0, mask = 1; i < 6; i++, mask <<= 1 )
    printf( "Power %d is %s\n", i, ( ( flags & mask ) ? "on" : "off" );
```

7.3.3 S26_Sched2601_GetLinkStatus()

Function: Schedules the fetching of the status of the gateway's sixteen IOM ports.

Prototype: `u32 S26_Sched2601_GetLinkStatus( XACT x, u16 *LinkFlags );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`LinkFlags`	`u16 *`	Pointer to a 16-bit application buffer that is to receive the link status flags. Each bit is associated with a single port. For example, bit 4 is associated with port 4. A bit flag is set to one to indicate active link (IOM connected), or zero to indicate inactive link.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.1 ms.

Notes: The gateway automatically maintains a list of active IOM ports, called the Active Port List (APL). This function returns a snapshot of the APL to the client.

Example:

```
// Fetch the link status from MM number 0.
u16 flags;
u16 mask;
int i;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2601_GetLinkStatus( x, &flags );
S26_SchedExecute( x, 1000, 0 );
for ( i = 0, mask = 1; i < 16; i++, mask <<= 1 )
```

```
{
    if ( flags & mask )
        printf( "Module detected at IOM port %d\n", i );
}
```

7.3.4 S26_Sched2601_SetWatchdog()

Function: Schedules the programming of the gateway's communication watchdog interval.

Prototype: `u32 S26_Sched2601_SetWatchdog( XACT x, u8 NumTenthSeconds );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
NumTenthSeconds	u8	MM communication watchdog interval, expressed in 100 millisecond increments. For example, the value 25 specifies a 2.5 second watchdog interval. Specify *zero* to disable the communication watchdog.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 0.1 ms.

Notes: The MM employs a timer to detect the absence of communications between the Ethernet client and the gateway. If no communications are received from the client within the specified time-out interval, the communication timer will time-out and the MM will execute a hardware reset. This behavior ensures that all I/O will be turned off in the event the client shuts down abnormally.

The watchdog interval defaults to 10 seconds (NumTenthSeconds = 100) in response to a MM reset. If the default interval is suitable for the application, no SetWatchdog action need be issued to the MM.

Upon receipt of a SetWatchdog action, the new watchdog interval is effective immediately and the watchdog timer is reset so that it will time out when the new interval elapses.

Example:
```
// Set the communication watchdog interval on MM number 0 to 3.5 seconds.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2601_SetWatchdog( x, 35 );
S26_SchedExecute( x, 1000, 0 );
```

7.4 Model 2608 Analog IOM

The functions in this section are used to schedule IOM actions for Model 2608 Analog IOMs. These functions are applicable only to Model 2608 IOMs. Any attempt to call these functions for other IOM types will result in a GWERR_IOMTYPE transaction error. Note that these functions only *schedule* IOM actions into a transaction; they do not cause the actions to be immediately executed.

7.4.1 Type-Specific Errors

In addition to the common IOM status bit flags (STATUS_RST and STATUS_CERR), this IOM type supports the following type-specific flag. If this flag is asserted, a transaction error of type GWERR_IOMSPECIFIC will be generated:

Symbolic Name	Description
STATUS_2608_CALERR	A problem was encountered when this IOM was initialized and, as a result, the calibration values stored in the module's EEPROM are not being used. Instead, default values are being used which may affect the accuracy of analog inputs and outputs. This can be caused by any of the following: * A communication fault occurred when fetching values from the IOM's EEPROM. * The EEPROM checksum is invalid. * One or more calibration values stored in the EEPROM exceed tolerance limits.

7.4.2 Analog Input Types

Several analog input types are supported by the middleware. Depending on the input type declared for an analog input channel, the middleware will automatically configure the channel's gain as required. For example, declaring any of thermocouple input types

will cause the corresponding channel to be programmed for the 100 millivolt measurement range. The following table shows the relevant attributes for all supported input types. The enumerated input type names are defined in the `App2600.h` header file.

Enumerated Type	Resolution	Data Units	Description
`RAW_LG_TYPE`	1 count	ADC counts	Corrected ADC counts, 10V range. This is the default type.
`RAW_HG_TYPE`	1 count	ADC counts	Corrected ADC counts, 100mV range.
`V_10_TYPE`	320 µV	Volts	Measured voltage, 10V range.
`V_001_TYPE`	3.2 µV	Volts	Measured voltage, 100mV range.
`TC_B_TYPE`	0.457 °C @ 800 °C	°C or °F	B thermocouple.
`TC_C_TYPE`	0.168 °C @ 800 °C	°C or °F	C thermocouple.
`TC_E_TYPE`	0.0478 °C @ 100 °C	°C or °F	E thermocouple.
`TC_J_TYPE`	0.0593 °C @ 100 °C	°C or °F	J thermocouple.
`TC_K_TYPE`	0.0762 °C @ 100 °C	°C or °F	K thermocouple.
`TC_N_TYPE`	0.107 °C @ 100 °C	°C or °F	N thermocouple.
`TC_R_TYPE`	0.267 °C @ 800 °C	°C or °F	R thermocouple.
`TC_S_TYPE`	0.291 °C @ 800 °C	°C or °F	S thermocouple.
`TC_T_TYPE`	0.0696 °C @ 100 °C	°C or °F	T thermocouple.

7.4.3 Calibration

Calibration is achieved by storing values in the 2608 module's EEPROM. Values may be stored in the EEPROM by calling `S26_2608_WriteEeprom()`, and stored values may be retrieved by calling `S26_Sched2608_ReadEeprom()`.

Various EEPROM locations are reserved for calibration values as described in section 7.4.4. All calibration values are multi-byte values that are stored in little-endian byte order.

7.4.4 Reserved EEPROM Locations

As shown in the following table, the first 176 EEPROM locations are reserved for calibration and configuration data:

Address	Data Type	Description
`0`	`u8`	Number of analog output channels present on the 2608 module. This is factory programmed and should never be changed.
`1 to 11`	`--`	Reserved for future use.
`12`	`u32`	Exact voltage of 10V reference standard, times 1e6.
`16`	`u32`	Exact voltage of 100mV reference standard, times 1e6.
`20+6*CHAN`	`s16`	Raw binary value that would be programmed onto analog output channel `CHAN` in order to produce exactly zero volts out. This typically has a value between -15 and +15.
`22+6*CHAN`	`u32`	Scalar, times 1e6, that is applied to values programmed onto analog output channel `CHAN` to compensate DAC full scale error. The scalar value is typically within ±5% of 1.0.
`68+2*CHAN`	`s16`	Raw binary value that must be subtracted from on-board reference temperature sensor CHAN to compensate its offset temperature.
`84`	`u8`	Checksum of all stored bytes from address 0 through 83.
`85 to 175`	`--`	Reserved for future use.
`176 to 255`	`--`	Available for application use.

7.4.5 S26_Sched2608_SetTempUnits()

Function: Schedules the setting of temperature units for thermocouple data returned from a model 2608 IOM.

Prototype: `u32 S26_Sched2608_SetTempUnits( XACT x, IOMPORT IomPort, int DegreesF );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`DegreesF`	`int`	Set to 0 for degrees C, or to 1 for degrees F.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.1 ms.

Notes: This sets the units for all analog input channels that are configured for thermocouple interface. Temperature units default to degrees C upon module reset. After calling this function, no delay is required before fetching analog input data.

Example:

```
// Set temperature units to degrees F on the 2608 at MM number 0, IOM port 1.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_SetTempUnits( x, 1, 1 );
S26_SchedExecute( x, 1000, 0 );
```

7.4.6 S26_Sched2608_GetAins()

Function: Schedules the fetching of all analog input values from a model 2608 IOM.

Prototype: `u32 S26_Sched2608_GetAins( XACT x, IOMPORT IomPort, double *data, BOOL Integrated );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`data`	`double *`	Pointer to a 16-element application buffer that is to receive the analog input data values.
`Integrated`	`int`	Set to 0 to fetch the "snapshot" values, or set to 1 to fetch the "integrated" values.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 4.1 ms.

Notes: **Important**: The application must call `S26_Sched2608_GetCalData()` at least once before calling `S26_Sched2608_GetAins()`. See section 7.4.9 for details.

Sixteen double-precision values are fetched when the scheduled action executes, so the application's data buffer must be large enough to receive all sixteen data values. One data value is fetched for each of the 2608's analog input channels. This action does not cause the analog input channels to be digitized; it simply fetches the previously digitized values. Excluding communication latency, the fetched values range in age from 0 to 2 milliseconds for snapshot values, or from 0 to 16 (or 20, if the power line frequency has been declared to be 50 Hz) milliseconds for integrated values.

For each analog input channel, the fetched data value is represented in units that are appropriate for the declared input type that was specified in a prior call to `S26_Sched2608_SetAinTypes()`. See section 7.4.2 for details. Thermocouple channels will be returned in either degrees C (default) or F, depending on the units selected in any prior call to `S26_Sched2608_SetTempUnits()`.

When the 2608 module is reset, or when the client calls `S26_Sched2608_SetAinTypes()`, the client must wait at least 32 milliseconds before calling `S26_Sched2608_GetAins()`. This delay ensures that the digitizer will have enough time to acquire valid data for all analog input channels before the data is passed to the client.

Example:

```
// Do a calibrate and read snapshot data from the 2608 at MM number 0, IOM port 1.
double ain[16];
```

```
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_GetCalData( x, 1, 0 );
S26_Sched2608_GetAins( x, 1, ain, 0 );
S26_SchedExecute( x, 1000, 0 );
```

7.4.7 S26_Sched2608_GetAinTypes()

Function: Schedules the fetching of all programmed analog input types from a model 2608 IOM.

Prototype: `u32 S26_Sched2608_GetAinTypes( XACT x, IOMPORT IomPort, u8 *types );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
types	u8 *	Pointer to a 16-byte application buffer that is to receive the programmed, enumerated analog input types for all analog input channels.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 0.9 ms.

Notes: The fetched values are the enumerated input types that were registered with the middleware when they were programmed, after validation against the gain flags that are returned from the 2608 in response to a OP_AIO_GETINPUTRANGES action.

There are two possible reasons for differences between the fetched and previously programmed values: (1) an illegal type was specified when the input types were programmed, or (2) the 2608 IOM was unexpectedly reset. In either case, all sixteen of the IOM's registered analog input types will be reset to their default values (RAW_LG_TYPE) when this scheduled action is executed.

Example:

```
// Get the analog input types from the 2608 at MM number 0, IOM port 1.
u8 aintypes[16];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_GetAinTypes( x, 1, aintypes );
S26_SchedExecute( x, 1000, 0 );
```

7.4.8 S26_Sched2608_GetAout()

Function: Schedules the fetching of one analog output setpoint from a model 2608 IOM.

Prototype: `u32 S26_Sched2608_GetAout( XACT x, IOMPORT IomPort, u8 chan, double *volts );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	Analog output channel number. Valid channel numbers range from 0 to 3 on model 2608-4, or from 0 to 7 on model 2608-8. There are no analog output channels on model 2608-0.
volts	double *	Pointer to a double-precision value that is to receive the fetched setpoint value. The fetched value will be expressed as a voltage, with a value in the range from -10.0 to +10.0.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 0.9 ms.

Notes: In most cases the fetched setpoint value will equal the last programmed value. The exception to this is if the 2608 IOM was unexpectedly reset, in which case the fetched setpoint will be reset to zero.

Example:

```
// Get the analog output channel 2 setpoint from the 2608 at MM number 0, IOM port 1.
double setpoint;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_GetAout( x, 1, 2, setpoint );
S26_SchedExecute( x, 1000, 0 );
```

7.4.9 S26_Sched2608_GetCalData()

Function: Schedules the fetching of calibration data from a model 2608 IOM.

Prototype: `u32 S26_Sched2608_GetCalData( XACT x, IOMPORT IomPort, short *caldata );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`caldata`	`short *`	Pointer to a 12*16-bit application buffer that is to receive the calibration data. Set to zero if the application is not interested in receiving calibration data (this is the case for most applications).

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 3.3 ms.

Notes: `S26_Sched2608_GetCalData()` schedules the fetching of calibration data that is used internally by the middleware. The middleware uses the calibration data to perform offset and gain corrections for voltage measurements and for reference-junction compensation for thermocouple measurements. In most applications it is sufficient to call this function with the `caldata` argument set to zero because the typical application has no need for direct access to calibration data.

`S26_Sched2608_GetCalData()` must be called at least once before calling `S26_Sched2608_GetAins()`. In addition, `S26_Sched2608_GetCalData()` should be called as needed to minimize errors due to circuit warm-up, ambient temperature drift and thermal transients. The individual situation dictates when and how often this function should be called, but as general rules-of-thumb:

- If `S26_Sched2608_GetAins()` is called infrequently, call `S26_Sched2608_GetCalData()` just before each call to `S26_Sched2608_GetAins()`.
- If `S26_Sched2608_GetAins()` is called frequently, call `S26_Sched2608_GetCalData()` periodically. The rate at which these periodic calls are made depends mostly on the 2608's environment. Higher rates are required where the 2608 is subjected to sudden temperature changes; in such cases once-per-second is a suitable rate. In more stable environments, once per ten seconds may be adequate. Of course, if time permits it is also permissible to simply call `S26_Sched2608_GetCalData()` just before each call to `S26_Sched2608_GetAins()`.

Example: See the example in section 7.4.6.

7.4.10 S26_Sched2608_ReadEeprom()

Function: Schedules the fetching of one data byte from the EEPROM on a model 2608 IOM.

Prototype: `u32 S26_Sched2608_ReadEeprom( XACT x, IOMPORT IomPort, u8 address, u8 *value );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`address`	`u8`	EEPROM memory address, in range `0x00` to `0xFF`.
`value`	`u8 *`	Pointer to application u8 buffer that is to receive the EEPROM data byte.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Notes: EEPROM addresses `0x00` through `0xAF` are reserved for use by the middleware. Addresses `0xB0` through `0xFF` are available for general application use.

Example:

```
// Read EEPROM byte at address 0xB0 from the 2608 at MM number 0, IOM port 1.
u8 eeval;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_ReadEeprom( x, 1, 0xB0, &eeval );
S26_SchedExecute( x, 1000, 0 );
```

7.4.11 S26_Sched2608_SetAinTypes()

Function: Schedules the programming of all analog input types on a model 2608 IOM.

Prototype: `u32 S26_Sched2608_SetAinTypes( XACT x, IOMPORT IomPort, const u8 *types );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`types`	`u8 *`	Pointer to a 16-byte application buffer that contains the enumerated analog input types for all analog input channels. See section 7.4.2 for details.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Notes: `S26_Sched2608_SetAinTypes()` informs the middleware as to what sensor types and input ranges are being used on the target IOM's analog input channels. These type declarations are used by the middleware to convert digitized input values to engineering units when `S26_Sched2608_GetAins()` executes. In addition to registering the input types with the middleware, `S26_Sched2608_SetAinTypes()` schedules the programming of the input ranges for all analog input channels, as appropriate for the declared types.

Example:

```
// Specify the input types.
u8 SensorTypes[16] = {
    V_10_TYPE, V_10_TYPE, V_10_TYPE, V_10_TYPE,       // Chan 0-7: ±10V range.
    V_10_TYPE, V_10_TYPE, V_10_TYPE, V_10_TYPE,
    V_001_TYPE, V_001_TYPE, V_001_TYPE, V_001_TYPE,   // Chan 8-11: ±100mV range.
    TC_K_TYPE, TC_K_TYPE, TC_K_TYPE, TC_K_TYPE        // Chan 9-15: K thermocouples.
};

// Program the input types for the model 2608 at MM number 0, IOM port 1.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_SetAinTypes( x, 1, SensorTypes );
S26_SchedExecute( x, 1000, 0 );
```

7.4.12 S26_Sched2608_SetAout()

Function: Schedules the programming of one analog output setpoint on a model 2608 IOM.

Prototype: `u32 S26_Sched2608_SetAout( XACT x, IOMPORT IomPort, u8 chan, double volts );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	Analog output channel number. Valid channel numbers range from 0 to 3 on model 2608-4, or from 0 to 7 on model 2608-8. There are no analog output channels on model 2608-0.
`volts`	`double`	The desired output voltage: from -10.0 to +10.0.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:
```
// Program analog output channel 2 to 5.35V on the 2608 at MM number 0, IOM port 1.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_SetAout( x, 1, 2, 5.35 );
S26_SchedExecute( x, 1000, 0 );
```

7.4.13 S26_Sched2608_SetLineFreq()

Function: Schedules the declaration of power line frequency to a model 2608 IOM.

Prototype: `u32 S26_Sched2608_SetLineFreq( XACT x, IOMPORT IomPort, u8 freq );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`freq`	`u8`	Enumerated line frequency: 0 = 60 Hz (default), 1 = 50 Hz.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.7 ms.

Notes: The integration period for the "integrated" adc values defaults to 16 milliseconds to help reject power line noise. This action enables the application to change the integration period to 20 milliseconds in cases where the power line frequency is 50 Hz.

Example:
```
// Declare 50 Hz line frequency to the 2608 at MM number 0, IOM port 1.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2608_SetLineFreq( x, 1, 1 );
S26_SchedExecute( x, 1000, 0 );
```

7.4.14 S26_2608_WriteEeprom()

Function: Writes one data byte to the EEPROM on a model 2608 IOM.

Prototype: `u32 S26_2608_WriteEeprom(u32 hbd,IOMPORT IomPort,u32 msec,u8 addr,u8 val,u32 retries);`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`msec`	`u32`	Maximum time, in milliseconds, to wait for a response before declaring a time-out.
`addr`	`u8`	EEPROM memory address, in range `0x00` to `0xFF`.
`val`	`u8`	Data value that is to be written to the EEPROM.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Error code as described in section 5.5. Zero is returned if the operation was successful.

Benchmark: 0.8 ms.

Notes: EEPROM addresses `0x00` through `0xAF` are reserved for use by the middleware; applications should not write to any locations in this reserved address range. Addresses `0xB0` through `0xFF` are available for application use.

Note that this function *executes* an EEPROM write, rather than just scheduling one.

Example:
```
// Write 0x05 to EEPROM address 0xB0 on the 2608 at MM number 0, IOM port 1.
S26_2608_WriteEeprom( 0, 1, 1000, 0xB0, 0x05, 1 );
```

7.5 Model 2610 Digital IOM

The functions in this section are used to schedule IOM actions for Model 2610 48-channel Digital IOMs. These functions are applicable only to Model 2610 IOMs. Any attempt to call these functions for other IOM types will result in a `GWERR_IOMTYPE` transaction error. Note that these functions only *schedule* IOM actions into a transaction; they do not cause the actions to be immediately executed.

See section 6.2.2 for programming examples that show how to use these functions. For additional information on the IOM actions that are invoked by these functions, see the Model 2600 Family Instruction Manual.

7.5.1 Type-Specific Errors

In addition to the common IOM status bit flags (`STATUS_RST` and `STATUS_CERR`), this IOM type supports the following type-specific flag. If this flag is asserted, a transaction error of type `GWERR_IOMSPECIFIC` will be generated:

Symbolic Name	Description
`STATUS_2610_STRM`	An error was detected in the serial data stream that is used to control the DIO output drivers. This flag can be cleared by invoking a `ClearStatus` action.

7.5.2 S26_Sched2610_GetInputs()

Function: Schedules the fetching of all DIO input states from a model 2610 IOM.

Prototype:
```
u32 S26_Sched2610_GetInputs( XACT x, IOMPORT IomPort, u8 *states );
```

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to 6-byte application buffer that is to receive the input states of the 48 digital I/O channels. The first byte receives channels 0 (lsb) to 7 (msb), the second byte receives channels 8 (lsb) to 15 (msb), and so on.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 1.3 ms.

Notes: The fetched state values represent the debounced, physical states of all DIO channels. Because the inputs are sampled every 2 milliseconds, and the debounce period is 10 milliseconds, the returned state values will all have an age ranging from 10 to 12 milliseconds, plus any network communication latency. The physical states of *all* DIO channels are returned, regardless of their respective operating modes.

Example:
```
// Get all DIO input states from the 2610 at MM number 0, IOM port 2.
u8 dins[6];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_GetInputs( x, 2, dins );
S26_SchedExecute( x, 1000, 0 );
```

7.5.3 S26_Sched2610_GetModes()

Function: Schedules the fetching of the operating modes for DIO channels 0 to 7 from a model 2610 IOM.

Prototype: `u32 S26_Sched2610_GetModes( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`modes`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the channel mode info. The byte indicates the operating modes for DIO channels 0 to 7. Each bit is associated with a channel number. For example, bit 4 is associated with channel 4. A bit is set to one when operating in the PWM mode, or to zero in the Standard mode.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:

```
// Get DIO channel 0-7 operating modes from the 2610 at MM number 0, IOM port 2.
u8 modes;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_GetModes( x, 2, &modes );
S26_SchedExecute( x, 1000, 0 );
```

7.5.4 S26_Sched2610_GetModes32()

Function: Schedules the fetching of the operating modes for DIO channels 0 to 31 from a model 2610 IOM.

Prototype: `u32 S26_Sched2610_GetModes32( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`modes`	`u8 *`	Pointer to a 4-byte application buffer that is to receive the channel mode info. The bytes indicates the operating modes for DIO channels 0 to 31. Each bit is associated with a channel number. For example, bit 4 of `modes[0]` is associated with channel 4. A bit is set to one when operating in the PWM mode, or to zero in the Standard mode.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 1.1 ms.

Notes: This function is compatible with 7410 firmware version 1.02 or higher. Earlier firmware versions support only eight pwm channels.

Example:

```
// Get DIO channel 0-31 operating modes from the 2610 at MM number 0, IOM port 2.
u8 modes[4];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_GetModes32( x, 2, modes );
S26_SchedExecute( x, 1000, 0 );
```

7.5.5 S26_Sched2610_GetOutputs()

Function: Schedules the fetching of all 48 DIO programmed output states from a model 2610 IOM.

Prototype: `u32 S26_Sched2610_GetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to 6-byte application buffer that is to receive the output states of the 48 digital I/O channels. The first byte receives channels 0 (lsb) to 7 (msb), the second byte receives channels 8 (lsb) to 15 (msb), and so on.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 1.3 ms.

Notes: This function fetches the programmed output driver states of all DIO channels. Note that the programmed output driver states may not correspond exactly to the physical channel states because some channels may be driven by external signal sources. In the case of channels that have been configured for the PWM mode, this function returns indeterminate state values.

Example:
```
// Get all DIO output states from the 2610 at MM number 0, IOM port 2.
u8 douts[6];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_GetOutputs( x, 2, douts );
S26_SchedExecute( x, 1000, 0 );
```

7.5.6 S26_Sched2610_GetPwmRatio()

Function: Schedules the fetching of the PWM ratio for one DIO channel from a model 2610 IOM.

Prototype: `u32 S26_Sched2610_GetPwmRatio( XACT x, IOMPORT IomPort, u8 chan, u8 *OnTime, u8 *OffTime );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The DIO channel number that is to be queried. Legal values range from 0 to 7 for 7410 firmware version 1.01 and below, or 0 to 31 for firmware versions 1.02 and higher.
`OnTime`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the programmed PWM on time expressed in 2 msec increments.
`OffTime`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the programmed PWM off time expressed in 2 msec increments.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:
```
// Fetch the PWM ratio for DIO channel 5 on the 2610 at MM number 0, IOM port 2.
u8 ontime;
u8 offtime;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_GetPwmRatio( x, 2, 5, &ontime, &offtime );
S26_SchedExecute( x, 1000, 0 );
```

7.5.7 S26_Sched2610_SetModes()

Function: Schedules the programming of the operating modes for DIO channels 0 to 7 on a model 2610 IOM.

Prototype: `u32 S26_Sched2610_SetModes( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`modes`	`u8 *`	Pointer to a 1-byte application buffer that contains the channel mode flags. The byte indicates the operating modes for DIO channels 0 to 7. Each bit is associated with a channel number. For example, bit 4 is associated with channel 4. Set a bit to one to operate in the PWM mode, or to z*ero* to operate in the Standard mode.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:

```
// Set DIO channel 0-7 operating modes on the 2610 at MM number 0, IOM port 2.
// Channels 0 to 5: Standard mode, channels 6 and 7: PWM mode.
u8 modes = 0xC0;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_SetModes( x, 2, &modes );
S26_SchedExecute( x, 1000, 0 );
```

7.5.8 S26_Sched2610_SetModes32()

Function: Schedules the programming of the operating modes for DIO channels 0 to 31 on a model 2610 IOM.

Prototype: `u32 S26_Sched2610_SetModes32( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`modes`	`u8 *`	Pointer to a 4-byte application buffer that contains the channel mode flags. The bytes indicates the operating modes for DIO channels 0 to 31. Each bit is associated with a channel number. For example, bit 4 of `modes[0]` is associated with channel 4. Set a bit to logic one for PWM mode, or to z*ero* for Standard mode.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 1.1 ms.

Notes: This function is compatible with 7410 firmware version 1.02 or higher. Earlier firmware versions support only eight pwm channels.

Example:

```
// Set DIO channel 0-31 operating modes on the 2610 at MM number 0, IOM port 2.
// Channels 6 and 7: PWM mode, all other channels: Standard mode.
u8 modes[4] = { 0xC0, 0x00, 0x00, 0x00 };
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_SetModes32( x, 2, modes );
S26_SchedExecute( x, 1000, 0 );
```

7.5.9 S26_Sched2610_SetOutputs()

Function: Schedules the programming of all 48 DIO output states on a model 2610 IOM.

Prototype: `u32 S26_Sched2610_SetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to 6-byte application buffer that contains the desired output states of the 48 digital I/O channels. The first byte contains channels 0 (lsb) to 7 (msb), the second byte contains channels 8 (lsb) to 15 (msb), and so on.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 1.3 ms.

Example:

```
// Program all DIO output states on the 2610 at MM number 0, IOM port 2.
u8 douts[6] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xAB };  // Desired DIO states.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_SetOutputs( x, 2, douts );
S26_SchedExecute( x, 1000, 0 );
```

7.5.10 S26_Sched2610_SetPwmRatio()

Function: Schedules the programming of the PWM ratio for one DIO channel on a model 2610 IOM.

Prototype: `u32 S26_Sched2610_SetPwmRatio( XACT x, IOMPORT IomPort, u8 chan, u8 OnTime, u8 OffTime );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The DIO channel number that is to be programmed. Legal values range from 0 to 7 for 7410 firmware version 1.01 and below, or 0 to 31 for firmware versions 1.02 and higher.
`OnTime`	`u8`	PWM on time, expressed in 2 msec increments, to be programmed.
`OffTime`	`u8`	PWM off time, expressed in 2 msec increments, to be programmed.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Set the PWM ratio for DIO channel 5 on the 2610 at MM number 0, IOM port 2.
// PWM ratio = on for 20 msec, off for 30 msec.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2610_SetPwmRatio( x, 2, 5, 10, 15 );
S26_SchedExecute( x, 1000, 0 );
```

7.6 Model 2612 Analog IOM

The functions in this section are used to schedule IOM actions for Model 2612 Analog IOMs. These functions are applicable only to Model 2612 IOMs. Any attempt to call these functions for other IOM types will result in a `GWERR_IOMTYPE` transaction error. Note that some of these functions only *schedule* IOM actions into a transaction; they do not cause the actions to be immediately executed.

7.6.1 Type-Specific Errors

In addition to the common IOM status bit flags (`STATUS_RST` and `STATUS_CERR`), this IOM type supports the following type-specific flags. If any of these flags are asserted, a transaction error of type `GWERR_IOMSPECIFIC` will be generated:

Symbolic Name	Description
`STATUS_2612_OVERFLOW`	Differential input voltage exceeds ADC positive input limit.
`STATUS_2612_UNDERFLOW`	Differential input voltage exceeds ADC negative input limit.
`STATUS_2612_EEPROM`	EEPROM read/write error. This flag will be asserted if the 2412's internal EEPROM checksum is invalid. This can happen if `S26_2612_RestoreCalibrations()` is called before `S26_2612_RestoreCalibrations()` has been called at least once for the target channel.

7.6.2 Analog Input Modes

Input mode registers allow to set timing parameters of the analog-digital conversion for each channel. The enumerated `oversample ratio settings`, `speed multiplier` and `reference voltages` type names are defined in the `App2600.h` header file.

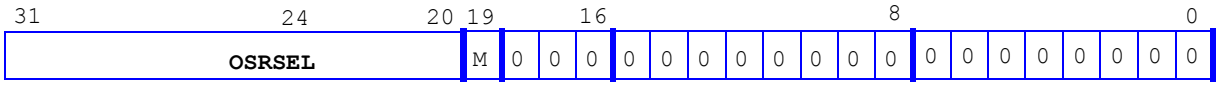
7.6.3 S26_Sched2612_SetMode()

Schedules the programming of the measurement mode for one channel on model 2612 IOM.

Prototype: `u32 S26_Sched2612_SetMode( XACT x, IOMPORT IomPort, u8 chan, u32 mode );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The channel number that is to be programmed. Legal values range from 0 to 3.
`mode`	`u32`	One of the OSR symbolic constants. To use the speed multiplier mode, the MODE_2X symbolic constant must be ORed with the OSR.

The `mode` parameter contains the OSR (oversample ratio) selector and speed multiplier enable bit:



The `OSRSEL` field selects the oversample ratio, which in turn configures a number of other behavioral attributes. Any of the following values may be specified:

OSRSEL	OSR	RMS Noise (µV)	Convert Rate (Hz)	First Notch (Hz)	–3dB Point (Hz)	Effective Bits
1	64	23	3515.6	28125	1696	17.0
2	128	4.5	1757.8	14062.5	848	20.1
3	256	2.8	878.9	7031.3	424	20.8
4	512	2.0	439.5	3515.6	212	21.3
5	1024	1.4	219.7	1757.8	106	21.8
6	2048	1.1	109.9	878.9	53	22.1
7	4096	0.72	54.9	439.5	26.5	22.7
8	8192	0.53	27.5	219.7	13.2	23.2
9	16384	0.35	13.75	109.9	6.6	23.8
15	32768	0.28	6.875	54.9	3.3	24.1

The `M` bit enables the speed multiplier when set to logic one, and it disables the multiplier when set to zero. When enabled, the speed multiplier doubles the conversion rate at the expense of one additional cycle of latency.

The most significant nibble must be set to 0xA. All other bits, which are reserved for future use, must be set to zero.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Notes: This function, as well as `S26_Sched2612_SetVoltages()`, should be called before calibrating or acquiring digitized data from an analog input channel. Digitized data may be fetched from the 2612 immediately after calling this function; no delay is required.

Example:

```
// Set measurement mode on the 2612 at MM number 0, IOM port 10, channel 2.
// Set the OSR to 32768 and enable the speed multiplier.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2612_SetMode( x, 10, 2, OSR_32768 | MODE_2X );
S26_SchedExecute( x, 1000, 0 );
```

7.6.4 S26_Sched2612_SetVoltages()

Schedules the programming of all power output channels on a model 2612 IOM.

Prototype: `u32 S26_Sched2612_SetVoltages( XACT x, IOMPORT IomPort, u8 volts );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`volts`	`u8`	A collection of four, two-bit fields. Each field specifies the output voltage for a power output channel. Two-bit field for each channel has to be set to one of the REF_OUT_xV values. Bits 0,1 are responsible for channel 0, bits 2,3 - for channel 1, bits 4,5 - for channel 2 and bits 6,7 - for channel 3.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.3 ms.

Notes: This function establishes the power output voltages on all channels. These output voltages are typically used to supply excitation to sensors such as strain gauges.

This function, as well as `S26_Sched2612_SetMode()`, should be called before calibrating or acquiring digitized data from an analog input channel. Since a change in the output voltage will cause a step change in the associated input channel's applied voltage, there will be a delay of one conversion time before valid digitized data becomes available.

Example:

```
// Set reference voltage on the 2612 at MM number 0, IOM port 10:
// channel 0 to 2V, channel 1 to 1.25V, channel 2 to 5V and channel 3 to 3V.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2612_SetVoltages( x, 10,
    REF_OUT_2V |
    ( REF_OUT_1V << 2 ) |
    ( REF_OUT_5V << 4 ) |
    ( REF_OUT_3V << 6 )
);
S26_SchedExecute( x, 1000, 0 );
```

7.6.5 S26_Sched2612_GetValues()

Schedules the fetching of the digitized values of all analog input channels on model 2612 IOM.

Prototype: `u32 S26_Sched2612_GetValues( XACT x, IOMPORT IomPort, s32 *values, u8 *tstamp );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`values`	`s32 *`	Pointer to array of 4*32bit buffer to receive the values.
`tstamp`	`u8 *`	Pointer to array of 4*8bit buffer to receive the last sample numbers (timestamps).

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 6.2 ms.

Example:

```
// Get raw values on the 2612 at MM number 0, IOM port 10.
s32 values[4];
u8 tstamp[4];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2612_GetValues( x, 10, vbuff, tbuff );
S26_SchedExecute( x, 1000, 0 );
```

7.6.6 S26_Sched2612_RefreshData()

Schedules the fetching of the raw values of all channels to internal middleware buffers on a model 2612 IOM.

Prototype: `u32 S26_Sched2612_RefreshData( XACT x, IOMPORT IomPort );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.1 ms.

Notes: This function does the actual work of scheduling the transfer of measured raw data from the 2612 IOM into a middleware buffer. Once the raw data values have been transferred to the middleware buffer, the application program can then call `S26_2612_GetCalibratedValue()` to obtain corrected data in the desired engineering units.

This function must be called periodically to refresh internally buffered data. To avoid a timestamp overflow this period must be less than 256 sample periods. For example, if the sample rate is 55 Hz, the data must be refreshed at least every 4.6 (255 / 55) seconds.

Since all 2612-specific functiona are thread-safe, it is possible to call `S26_Sched2612_RefreshData()` from one thread to fetch raw data into the internal middleware buffer, while another thread calls `S26_2612_GetCalibratedValue()` to acquire corrected data for use by the application.

Example:

```
// Get buffered values on the 2612 at MM number 0, IOM port 10.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2612_RefreshData( x, 10 );
S26_SchedExecute( x, 1000, 0 );
```

7.6.7 S26_2612_RegisterZero()

Establishes the “zero offset” on one analog input channel on a model 2612 IOM.

Prototype: `u32 S26_2612_RegisterZero( u32 hbd, IOMPORT IomPort, u32 msec, u8 chan, u32 nsmp );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`msec`	`u32`	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.
`nsmp`	`u32`	Number of samples used to average calibration result.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: `(tsample + 0.4 ms) * nsmp`, where `tsample` is the target channel's sample period.

Notes: This function is part of the calibration process for an analog input channel. It calculates the "zero offset" value for an analog input channel and stores the value in an internal buffer. The target channel will be measured in rapid succession a number of times, as specified by `nsmp`. The resulting digitized values are averaged and then stored for later use. Later, when the application program samples the analog input, the offset value is used to offset-adjust the resulting digital data value.

When this function is called, an actual zero value reference signal must be applied to the measurement inputs of the target analog input channel, and the reference level must be held constant until the measurement is finished.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Register zero values on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_RegisterZero( 0, 10, 1000, 2, 100 );
```

7.6.8 S26_2612_RegisterSpan()

Measures and calculates the "positive full scale" value for one channel on a model 2612 IOM.

Prototype: `u32 S26_2612_RegisterSpan( u32 hbd, IOMPORT IomPort, u32 msec, u8 chan, u32 nsmp, double load );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`msec`	`u32`	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.
`nsmp`	`u32`	Number of samples used to average calibration result.
`load`	`double`	Actual value applied to the channel input in any user's units.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: `(tsample + 0.4 ms) * nsmp`, where `tsample` is the target channel's sample period.

Notes: This function calculates the "positive full scale" value and stores the value in an internal buffer. The actual positive full scale value must be applied to the corresponding channel's input before start of calibration and held until the function returns. This function is called as the final step in a two-step physical gauge calibration procedure.

When this function executes, `nsmp` measurements are taken, the average value is calculated and stored as the "positive full scale" value for one channel. The stored value will be used later to calculate corrected values.

A gauge load parameter, `load`, must be specified when this function is called. This value represents the difference between the load that is applied when `S26_2612_RegisterZero()` was called and the load that is applied when `S26_2612_RegisterSpan()` is called. For example, suppose the applied load is 2,000 pounds. The `load` parameter should be set to 2,000.0. After executing this command, `S26_Sched2612_GetCalibratedValues()` will return data from this channel in units of pounds. In this case, an applied load of 153.7 pounds, for example, would cause `S26_Sched2612_GetCalibratedValues()` to return the value 153.7.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:

```
// Set span to 2000 pounds on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_RegisterSpan( 0, 10, 1000, 2, 100, 2000.0 );
```

7.6.9 S26_2612_RegisterTare()

Measures and calculates the "permanent offset" (i.e., tare) value for one channel on a model 2612 IOM

Prototype: `u32 S26_2612_RegisterTare( u32 hbd, IOMPORT IomPort, u32 msec, u8 chan, u32 nsmp );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`msec`	`u32`	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.
`nsmp`	`u32`	Number of samples used to average calibration result.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: `(tsample + 0.4 ms) * nsmp`, where `tsample` is the target channel's sample period.

Notes: This function calculates the "permanent offset" (tare) value and stores the value in an internal buffer. The actual tare value must be applied to the corresponding channel's input before start of calibration and held until the function returns. Taring is accomplished by adjusting the data offset so that data returned by `S26_Sched2612_GetCalibratedValues()` will equal zero at the current load condition.

When this function executes, `nsmp` measurements are taken, the average value is calculated and stored as the "permanent offset" (tare) for one channel. The stored value will be used later to calculate corrected values.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:

```
// Register tare on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_RegisterTare( 0, 10, 1000, 2, 100 );
```

7.6.10 S26_2612_GetCalibratedValue()

Calculates and returns the corrected, measured value for one channel on a model 2612 IOM.

Prototype: `double S26_2612_GetCalibratedValue( u32 hbd, IOMPORT IomPort, u8 chan, u32 *sample );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.
`sample`	`u32 *`	Pointer to 32bit buffer to receive the sample counter value.

Returns: Calibrated value.

Benchmark: << 0.1 ms (no network transactions used).

Notes: This function converts the specified channel's raw digitized value, which was previously acquired by calling `S26_Sched2612_RefreshData()` , to a corrected value. The corrected value is computed by this function as follows:

```
corrected_value = (raw_value - offset) * scale - tare
```

The `offset`, `scale` and `tare` values, and consequently the engineering units that apply to the returned value, must have been previously established by calling `S26_2612_RegisterZero()`, `S26_2612_RegisterSpan()` and `S26_2612_RegisterTare()`, or set with `S26_2612_SetCalibrations()`.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Get calibrated value on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_GetCalibratedValue( 0, 10, 2, sample_pointer );
```

7.6.11 S26_2612_GetOffset()

Returns the `offset` value for one channel on a model 2612 IOM.

Prototype: `double S26_2612_GetOffset( u32 hbd, IOMPORT IomPort, u8 chan );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.

Returns: Offset value.

Benchmark: << 0.1 ms (no network transactions used).

Notes: This function returns the `offset` calibration parameter from a previously calibrated channel. The returned value can be used by `S26_2612_SetCalibrations()` to restore a channel calibration without having to perform a physical calibration.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Get offset value on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_GetOffset( 0, 10, 2 );
```

7.6.12 S26_2612_GetScale()

Returns the `scale` value for one channel on a model 2612 IOM.

Prototype: `double S26_2612_GetScale( u32 hbd, IOMPORT IomPort, u8 chan );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.

Returns: Scale value.

Benchmark: << 0.1 ms (no network transactions used).

Notes: This function returns the `scale` calibration parameter from a previously calibrated channel. The returned value can be used by `S26_2612_SetCalibrations()` to restore a channel calibration without having to perform a physical calibration.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Get scale value on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_GetScale( 0, 10, 2 );
```

7.6.13 S26_2612_GetTare()

Returns the `tare` value for one channel on model 2612 IOM.

Prototype: `double S26_2612_GetTare( u32 hbd, IOMPORT IomPort, u8 chan );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.

Returns: Tare value.

Benchmark: << 0.1 ms (no network transactions used).

Notes: This function returns the `tare` calibration parameter from a previously calibrated channel. The returned value can be used by `S26_2612_SetCalibrations()` to restore a channel calibration without having to perform a physical calibration.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Get tare value on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_GetTare( 0, 10, 2 );
```

7.6.14 S26_2612_SetCalibrations()

Ses the middleware `offset, scale and tare` values for one channel on a model 2612 IOM.

Prototype: `u32 S26_2612_SetCalibrations( u32 hbd, IOMPORT IomPort, u8 chan, double Offset, double Scale, double Tare );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.
`Offset`	`double`	The `offset` value
`Scale`	`double`	The `scale` value
`Tare`	`double`	The `tare` value

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: << 0.1 ms (no network transactions used).

Notes: This function establishes all of the calibration values for the specified channel without having to perform a physical calibration. This is useful in situations where a physical calibration need be performed only one time. For example, a physical calibration could be performed once and the calibration values could then be obtained by calling `S26_2612_GetOffset()`, `S26_2612_GetOffset()`, and `S26_2612_GetOffset()`. Later, after the

middleware has been closed and reopened, this function can be called to restore the calibration values. In many applications, it is useful to store and retrieve the calibration values from the 2412's internal EEPROM by calling `S26_2612_SaveCalibrations()` and `S26_2612_RestoreCalibrations()`.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Set all calibration values on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_SetCalibrations( 0, 10, 2, Offset, Scale, Tare );
```

7.6.15 S26_2612_SaveCalibrations()

Saves one channel's calibration values to internal EEPROM on model 2612 IOM.

Prototype: `u32 S26_2612_SaveCalibrations( u32 hbd, IOMPORT IomPort, u32 msec, u8 chan );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`msec`	`u32`	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 12 ms.

Notes: This function copies the calibration values from internal middleware buffers to the 2612 module's EEPROM. Later, the values can be retrieved from the module's EEPROM by calling `S26_2612_RestoreCalibrations()`.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Save calibrations on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_SaveCalibrations( 0, 10, 1000, 2 );
```

7.6.16 S26_2612_RestoreCalibrations()

Restores one channel's calibration values from a 2612 IOM's EEPROM.

Prototype: `u32 S26_2612_RestoreCalibrations( u32 hbd, IOMPORT IomPort, u32 msec, u8 chan );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`msec`	`u32`	Maximum time to wait for the gateway response packet, in milliseconds, before declaring a time-out.
`chan`	`u8`	The channel number that is to be registered. Legal values range from 0 to 3.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 7 ms.

Notes: This function copies the calibration values from the 2612 module's EEPROM to internal middleware buffers, thus activating the new calibration values. It is assumed that `S26_2612_SaveCalibrations()` was previously called to store the calibration values in the EEPROM.

Note that this function performs its action when called, unlike many other middlware functions that simply schedule future actions.

Example:
```
// Restore calibrations on the 2612 at MM number 0, IOM port 10, channel 2.
S26_2612_RestoreCalibrations( 0, 10, 1000, 2 );
```

7.7 Model 2620 Counter IOM

The functions in this section are used to schedule IOM actions for Model 2620 4-channel Counter IOMs. These functions are applicable only to Model 2620 IOMs. Any attempt to call these functions for other IOM types will result in a `GWERR_IOMTYPE` transaction error. Note that these functions only *schedule* IOM actions into a transaction; they do not cause the actions to be immediately executed.

7.7.1 Type-Specific Errors

This IOM type has no type-specific IOM status flags.

7.7.2 S26_Sched2620_GetCounts()

Function: Schedules the fetching of the latched counts from one counter channel on a model 2620 IOM.

Prototype:
```
u32 S26_Sched2620_GetCounts( XACT x, IOMPORT IomPort, u8 chan, u32 *value, u16 *tstamp );
```

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The counter channel number that is to be accessed. Legal values range from 0 to 3.
`value`	`u32 *`	Pointer to a 32-bit application buffer that is to receive the counts.
`tstamp`	`u16 *`	Pointer to a 16-bit application buffer that is to receive the time stamp. Specify zero if you do not need the time stamp value.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 1.1 ms when `tstamp` is zero, 1.6 ms when `tstamp` is non-zero.

Notes: The fetched `value` will be the value contained in the target counter's data latch at the moment the action executes on the IOM. If tstamp is non-zero, the time stamp value will also be fetched; the fetched time stamp value will be the value contained in the target counter's time stamp latch at the moment the action executes on the IOM.

The time stamp value should be fetched only if it is needed as extra communication overhead is required to fetch this value.

Example:
```
// Get latched counts from counter 3 on the 2620 at MM number 0, IOM port 12.
u32 counts;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_GetCounts( x, 12, 3, &counts, 0 );
S26_SchedExecute( x, 1000, 0 );
```

Example:
```
// Get counts and timestamp from counter 3 on the 2620 at MM number 0, IOM port 12.
u32 counts;
u16  tstamp;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_GetCounts( x, 12, 3, &counts, &tstamp );
S26_SchedExecute( x, 1000, 0 );
```

7.7.3 S26_Sched2620_GetStatus()

Function: Schedules the fetching of the status of one counter channel from a model 2620 IOM.

Prototype: `u32 S26_Sched2620_GetStatus( XACT x, IOMPORT IomPort, u8 chan, u16 *status );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The counter channel number that is to be accessed. Legal values range from 0 to 3.
`status`	`u16 *`	Pointer to a 16-bit application buffer that is to receive the status info.

The returned `status` value has the following format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	QDE	LAT	GO	LOD	EXT	UF	OF	ZER

QDE: Quadrature decoder error. This bit is automatically reset by this GetStatus action.

LAT: Counter core was latched. This bit is automatically reset by a GetCounts action.

GO: Counter was enabled by a trigger.

LOD: Counter was pre-loaded. This bit is automatically reset by this GetStatus action.

EXT: Counter extension bit 32.

UF: Counter underflowed. This bit is automatically reset by this GetStatus action.

OF: Counter overflowed. This bit is automatically reset by this GetStatus action.

ZER: Counter value is now zero.

All other bits are reserved for future use.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Notes: The format of the fetched status word is described in the Model 2600 Family Instruction Manual.

Example:

```
// Get status info from counter 3 on the 2620 at MM number 0, IOM port 12.
u16 status;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_GetStatus( x, 12, 3, &status );
S26_SchedExecute( x, 1000, 0 );
```

7.7.4 S26_Sched2620_SetControlReg()

Function: Triggers a data transfer action for one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetControlReg( XACT x, IOMPORT IomPort, u8 chan, u8 DataVal );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The counter channel number that is to be accessed. Legal values range from 0 to 3.
`DataVal`	`u8`	Data transfer action to be triggered: `CTC_TRIG_PRELOAD` (1) - Preload counter core. `CTC_TRIG_LATCH` (2) - Latch counter core.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Notes: Use `CTC_TRIG_LATCH` to manually transfer the core's counts to the data latch just before reading the counter. This must be done, for example, when a channel is configured to operate with a quadrature encoder. This should not be

done, however, if you have configured a channel so that it's core is automatically transferred to the latch in response to an event (e.g., active index input).

Use `CTC_TRIG_PRELOAD` to manually transfer the Preload0 register into the counter core.

Example:

```
// Latch channel 3 counter core on the 2620 at MM number 0, IOM port 12.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetControlReg( x, 12, 3, CTC_TRIG_LATCH );
S26_SchedExecute( x, 1000, 0 );
```

7.7.5 S26_Sched2620_SetCommonControl()

Function: Schedules the programming of the common control register for all counter channels on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetCommonControl( XACT x, IOMPORT IomPort, u16 gperiod, u8 tstamp );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
gperiod	u16	Gate period, in milliseconds, for the time gate generator. This is the gate time applied to all channels operating as frequency counters that use the internal time gate generator. Any even value from 2 to 32766 may be specified, resulting in gate times from 2 milliseconds to 32.766 seconds.
tstamp	u8	Timestamp resolution. May be set to one of the following values: 0 = 1 microsecond. 1 = 10 microseconds. 2 = 100 microseconds. 3 = 1 millisecond.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Set gate period to 1 second, and timestamp resolution to 10 microseconds
// on the 2620 at MM number 0, IOM port 12.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetCommonControl( x, 12, 1000, 1 );
S26_SchedExecute( x, 1000, 0 );
```

7.7.6 S26_Sched2620_SetModeEncoder()

Function: Schedules the programming of the operating mode of one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetModeEncoder( XACT x, IOMPORT IomPort, u8 chan, u16 xp, u16 pl, u16 m );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.

Parameter	Type	Description
chan	u8	The counter channel number that is to be accessed. Legal values range from 0 to 3.
xp	u16	Index pin polarity: 1 = active low, 0 = active high.
pl	u16	Preload upon index leading edge: 0 = disable, 1 = enable.
m	u16	Clock mode. Set to one of these values: 0 - quadrature x1, clock on A rising edge, B sets direction. 1 - quadrature x1, clock on A falling edge, B sets direction. 2 - quadrature x2, clock on both A edges, B sets direction. 3 - quadrature x4, clock on all A and B edges. 4 - mono, clock on A rising edge, B sets direction. 5 - mono, clock on A falling edge, B sets direction. 6 - mono, clock on both A edges, B sets direction.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 1.5 ms.

Notes: This function configures a counter channel so that it will interface to either a quadrature-encoded or a single-phase clock source, with optional index input for preload triggering.

Example:

```
// Configure counter 3 as an encoder interface on the 2620 at MM number 0, IOM port 12.
// Assumes: quadrature encoder, x4 clock multiplier, no index-triggered preloads.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetModeEncoder( x, 12, 3, 0, 0, 3 );   // Set mode.
S26_SchedExecute( x, 1000, 0 );
```

7.7.7 S26_Sched2620_SetModeFreqMeas()

Function: Schedules the programming of the operating mode of one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetModeFreqMeas( XACT x, IOMPORT IomPort, u8 chan, u16 igate );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The counter channel number that is to be accessed. Legal values range from 0 to 3.
igate	u16	Time gate signal source: 0 = external signal on index input, 1 = internal gate generator.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 2.0 ms.

Notes: This function configures a counter channel so that it will measure the frequency of an external digital signal applied to the `ClkA` input.

A periodic time gate signal is required. Note that it is the *period* of the gate signal (vs. its time in the active state) that determines the sampling time for frequency measurement. The gate time is defined as the time between consecutive rising edges of the gate signal.

The gate signal may be derived from an external signal that is applied to the index input pin or from the internal time gate generator that is shared by all counter channels. When using the internal time gate generator, the gate generator should be configured before calling this function; see `S26_Sched2620_SetCommonControl()` for details. When using an external time gate generator, the index polarity defaults to active high so that sample intervals begin in gate (index) rising edges.

The channel's preload registers are automatically configured by this function. The preload registers should not be modified while frequency measurement mode is in effect.

Example:

```
// Configure counter 3 as a frequency counter on the 2620 at MM number 0, IOM port 12.
// Assumes: using previously configured internal time gate generator.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetModeFreqMeas( x, 12, 3, 1 );   // Set mode.
S26_SchedExecute( x, 1000, 0 );
```

7.7.8 S26_Sched2620_SetModePeriodMeas()

Function: Schedules the programming of the operating mode of one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetModePeriodMeas( XACT x, IOMPORT IomPort, u8 chan, u16 ActLowX );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The counter channel number that is to be accessed. Legal values range from 0 to 3.
ActLowX	u16	Index pin polarity: 1 = active low, 0 = active high. This doesn't matter unless one signal edge has more jitter than the other edge.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 2.0 ms.

Notes: This function configures a counter channel so that it will measure the period of an external digital waveform applied to the index input.

Example:

```
// Configure counter 3 as a frequency counter on the 2620 at MM number 0, IOM port 12.
// Assumes: both signal edges have similar jitter, so polarity is don't care.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetModePeriodMeas( x, 12, 3, 0 );   // Set mode.
S26_SchedExecute( x, 1000, 0 );
```

7.7.9 S26_Sched2620_SetModePulseGen()

Function: Schedules the programming of the operating mode of one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetModePulseGen(XACT x, IOMPORT IomPort, u8 chan, u16 xp, u16 pl, u16 op);`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The counter channel number that is to be accessed. Legal values range from 0 to 3.
xp	u16	Index pin polarity: 1 = active low, 0 = active high. This is a "don't care" if pl is set to zero.
pl	u16	Hardware triggered by index input: 0 = disable, 1 = enable. Note that a pulse can always be triggered by software.
op	u16	Output pin polarity: 1 = active low, 0 = active high.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 1.5 ms.

Notes: This function configures a counter channel so that it will generate a single output pulse in response to a hardware or software trigger. The duration of the output pulse is determined by the value stored in the Preload0 register.

Example:

```
// Configure counter 3 as a pulse generator on the 2620 at MM number 0, IOM port 12.
// Assumes: active low output pulse, hardware triggered by active low signal.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetModePulseGen( x, 12, 3, 1, 1, 3 );   // Set mode.
S26_SchedExecute( x, 1000, 0 );
```

7.7.10 S26_Sched2620_SetModePulseMeas()

Function: Schedules the programming of the operating mode of one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetModePulseMeas( XACT x, IOMPORT IomPort, u8 chan, u16 ActLowX );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The counter channel number that is to be accessed. Legal values range from 0 to 3.
ActLowX	u16	Index pin polarity: 1 = active low, 0 = active high.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 1.5 ms.

Notes: This function configures a counter channel so that it will measure the width of pulses applied to the index input. When a measurement is completed, the result is latched and the next measurement begins automatically. The most recently acquired measurement value may be read from the latch at any time.

The channel's preload registers are automatically configured by this function. The preload registers should not be modified while pulse width measurement mode is in effect.

Example:

```
// Configure counter 3 for pulse width measurement on the 2620 at MM number 0, IOM port 12.
// Assumes: active high pulse is being measured.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetModePulseMeas( x, 12, 3, 0 );   // Set mode.
S26_SchedExecute( x, 1000, 0 );
```

7.7.11 S26_Sched2620_SetModePwmGen()

Function: Schedules the programming of the operating mode of one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetModePwmGen( XACT x, IOMPORT IomPort, u8 chan, u16 op );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The counter channel number that is to be accessed. Legal values range from 0 to 3.
op	u16	Output pin polarity: 1 = active low, 0 = active high.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 1.5 ms.

Notes: S26_Sched2620_SetModePwmGen() configures a counter channel so that it will toggle its output signal at periodic intervals, with programmable period and duty cycle. This can be used to generate a continuous train of output pulses in which both the pulse width and time gap between pulses is programmable.

Pulse width and gap times are determined by the values stored in the Preload registers. Preload1 specifies the duration of the pulse, and Preload0 specifies the time interval between pulses. Preload values are related to time as follows: `value = 10 * t - 1`, where t is specified in microseconds. For example, the value 99 corresponds to 10 microseconds.

The application should program the initial pulse width and gap times into the preload registers before calling this function. After calling `S26_Sched2620_SetModePwmGen()`, the pulse width and/or gap times may be changed at any time by programming new values into the associated preload registers.

Example:

```
// Configure counter 3 for pwm generation on the 2620 at MM number 0, IOM port 12.
// Settings: active high output pin, 2KHz @ 2% duty cycle.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetPreload( x, 12, 3, 1, 99 );     // 10.0 us pulse width.
S26_Sched2620_SetPreload( x, 12, 3, 0, 4899 );   // 490.0 us gap time.
S26_Sched2620_SetModePwmGen( x, 12, 3, 0 );      // Set mode.
S26_SchedExecute( x, 1000, 0 );
```

7.7.12 S26_Sched2620_SetMode()

Function: Schedules the programming of the operating mode of one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetMode( XACT x, IOMPORT IomPort, u8 chan, u16 mode );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The counter channel number that is to be accessed. Legal values range from 0 to 3.
`mode`	`u16`	Value to be written to the mode register.

The `mode` value is a collection of bit flags:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RUN	OM1	OM0	XP	PL1	PL0	LAT	CET	OP	M2	M1	M0	CD1	CD0	PLM	XC

RUN: Enable channel operations.

0 - (default upon module reset) Halt channel, force core to zero (including bit 32), force status bits to their default states, reset trigger latches. Preload and latch registers are not modified. After writing to the mode register with this bit cleared, it is necessary to write to it again with this bit set to start the channel running.

1 - Run or continue to run in the specified mode.

OM: Output pin's mode (2-bit field):

0 - Counter bit 31.

1 - Counter bit 32 (toggles at zero counts).

2 - Active when counts are zero.

3 - Active during counter under/overflow.

XP: Index input polarity:

0 - Active high.

1 - Active low.

PL: Preload trigger (2-bit field):

0 - Preload on soft trigger only.

1 - Preload on index leading edge or soft trigger.

2 - Preload on zero counts reached or soft trigger.

3 - Reserved.

LAT: Latch trigger:

0 - Latch on soft trigger only.

1 - Latch on index leading edge or soft trigger.

CET: Count enable trigger:

0 - Enable upon configuration (no trig needed).

1 - Enable on index leading edge.

OP: Output pin's polarity:

0 - Active high.

1 - Active low.

M: Mode (3-bit field). Modes 0-3 use quadrature-encoded two-phase clock, modes 4-6 use single-phase clock, and mode 7 uses the internal clock:

0 - quad x1, clock on rising A.

1 - quad x1, clock on falling A.

2 - quad x2, clock on either edge of A.

3 - quad x4, clock on either edge of A or B.

4 - mono, clock on rising A, B controls count direction.

5 - mono, clock on falling A, B controls count direction.

6 - mono, clock on either edge A, B controls count direction.

7 - internal clock (10MHz), A is the gate (enables counting while asserted), B controls count direction.

CD: Count disable trigger:

0 - Never disabled by any trigger.

1 - Disable on index trailing edge (if enabled).

2 - Disable when zero counts reached.

PLM: Select preload register:

0 - Only preload register 0.

1 - Use both preload registers.

XC: Index source:

0 - External Index pin.

1 - Internal free-running gate generator.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Notes: This function may be used to establish any arbitrary operating mode for a counter channel. It is provided so that applications can tailor the counter operating mode in cases where the other mode setting functions, such as `S26_Sched2620_SetModeFreqMeas()`, do not provide sufficient control over counter operating parameters.

The new mode should be invoked by calling `S26_Sched2620_SetMode()` with the `RUN` flag negated so that the counter will halt while the mode is being changed; this guarantees that the channel will be properly initialized regardless of the physical state of its I/O pins. `S26_Sched2620_SetMode()` should then be called again with `RUN` asserted to enable the counter channel to run in the new mode.

Example:

```
// Set counter 3 operating mode to 0x0001 on the 2620 at MM number 0, IOM port 12.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetMode( x, 12, 3, 0x0001 );   // Halt channel and set mode.
S26_Sched2620_SetMode( x, 12, 3, 0x1001 );   // Run in the new mode.
S26_SchedExecute( x, 1000, 0 );
```

7.7.13 S26_Sched2620_SetPreload()

Function: Schedules the programming of a preload register for one counter channel on a model 2620 IOM.

Prototype: `u32 S26_Sched2620_SetPreload( XACT x, IOMPORT IomPort, u8 chan, u8 reg, u32 value );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The counter channel number that is to be accessed. Legal values range from 0 to 3.
reg	u8	Selects the preload register that is to be written to: 0 or 1.
value	u32	32-bit value to be written to the preload register.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 1.1 ms.

Example:

```
// Set counter 3's preload0 reg to 0x0001 on the 2620 at MM number 0, IOM port 12.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2620_SetPreload( x, 12, 3, 0, 0x0001 );
S26_SchedExecute( x, 1000, 0 );
```

7.8 Model 2650 Relay IOM

The functions in this section are used to schedule IOM actions for Model 2650 8-channel Relay IOMs. These functions are applicable only to Model 2650 IOMs. Any attempt to call these functions for other IOM types will result in a GWERR_IOMTYPE transaction error. Note that these functions only *schedule* IOM actions into a transaction; they do not cause the actions to be immediately executed.

7.8.1 Type-Specific Errors

In addition to the common IOM status bit flags (STATUS_RST and STATUS_CERR), this IOM type supports the following type-specific flags. If any of these flags are asserted, a transaction error of type GWERR_IOMSPECIFIC will be generated:

Symbolic Name	Description
STATUS_2650_DRVR	One or more relay coil drivers failed to go to the commanded state. This may be caused by a driver fault, a shorted relay coil or a serial data stream problem. This flag can be cleared by invoking a ClearStatus action.
STATUS_2650_STRM	An error was detected in the serial data stream that is used to control the relay drivers and monitor driver states. This flag can be cleared by invoking a ClearStatus action.

7.8.2 S26_Sched2650_GetInputs()

Function: Schedules the fetching of the measured states of all eight relay coil drivers on a model 2650 IOM.

Prototype: `u32 S26_Sched2650_GetInputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
states	u8 *	Pointer to a 1-byte application buffer that is to receive the measured states of the relay drivers. Each bit is associated with one relay channel. For example, bit 7 is associated with relay channel 7. Any bit set to one indicates the associated channel is set to the active state; any bit set to zero indicates the channel is set to the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Notes: Each relay channel includes a monitoring circuit that enables the on-board processor to determine the physical state of the relay coil driver. This scheduled action will fetch the monitored physical state of each coil driver, even if the relay is not present or its coil winding has opened.

Coil driver states are acquired periodically at two millisecond intervals. Consequently, `states` may not accurately reflect the state of a coil driver that has changed its physical state within the last two milliseconds.

Example:

```
// Get all relay driver coil states from the 2650 at MM number 0, IOM port 9.
u8 states;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2650_GetInputs( x, 9, &states );
S26_SchedExecute( x, 1000, 0 );
```

7.8.3 S26_Sched2650_GetOutputs()

Function: Schedules the fetching of the programmed states of all eight relays on a model 2650 IOM.

Prototype: `u32 S26_Sched2650_GetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the programmed states of the relay drivers. Each bit is associated with one relay channel. For example, bit 7 is associated with relay channel 7. Any bit set to one indicates the associated channel is set to the active state; any bit set to zero indicates the channel is set to the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:

```
// Get all programmed relay driver states from the 2650 at MM number 0, IOM port 9.
u8 states;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2650_GetOutputs( x, 9, &states );
S26_SchedExecute( x, 1000, 0 );
```

7.8.4 S26_Sched2650_SetOutputs()

Function: Schedules the programming of all eight relays on a model 2650 IOM.

Prototype: `u32 S26_Sched2650_SetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to a 1-byte application buffer that contains the desired states of the relay drivers. Each bit is associated with one relay channel. For example, bit 7 is associated with relay channel 7. Any bit set to one indicates the associated channel is set to the active state; any bit set to zero indicates the channel is set to the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:
```
// Program all relay driver states on the 2650 at MM number 0, IOM port 9.
u8 states = 0x5A;     // The desired relay states.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2650_SetOutputs( x, 9, &states );
S26_SchedExecute( x, 1000, 0 );
```

7.9 Model 2652 Solid-State Relay IOM

The functions in this section are used to schedule IOM actions for Model 2652 8-channel Solid-State Relay IOMs. These functions are applicable only to Model 2652 IOMs. Any attempt to call these functions for other IOM types will result in a `GWERR_IOMTYPE` transaction error. Note that these functions only *schedule* IOM actions into a transaction; they do not cause the actions to be immediately executed.

7.9.1 Type-Specific Errors

In addition to the common IOM status bit flags (`STATUS_RST` and `STATUS_CERR`), this IOM type supports the following type-specific flags. If any of these flags are asserted, a transaction error of type `GWERR_IOMSPECIFIC` will be generated:

Symbolic Name	Description
`STATUS_2652_STRM`	An error was detected in the serial data stream that is used to control the SSR output drivers. This flag can be cleared by invoking a `ClearStatus` action.

7.9.2 S26_Sched2652_GetInputs()

Function: Schedules the fetching of the physical states of all eight SSR channels on a model 2652 IOM.

Prototype:
```
u32 S26_Sched2652_GetInputs( XACT x, IOMPORT IomPort, u8 *states );
```

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the physical states of the SSR channels. 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 in the active state; any bit set to zero indicates the channel is in the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Notes: Each SSR channel includes a monitoring circuit that enables the on-board processor to determine the physical state of the channel. This scheduled action will fetch the monitored physical state of each channel, no matter whether the channel is driven by its own output driver or by an external signal through an input SSR.

Physical states are sampled periodically at two millisecond intervals and passed through a 10 millisecond debounce filter. Consequently, `states` may not accurately reflect the state of a channel that has changed its physical state within the last twelve milliseconds.

Example:
```
// Get all physical SSR states from the 2652 at MM number 0, IOM port 9.
u8 states;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2652_GetInputs( x, 9, &states );
S26_SchedExecute( x, 1000, 0 );
```

7.9.3 S26_Sched2652_GetModes()

Function: Schedules the fetching of the operating modes for all SSR channels on a model 2652 IOM.

Prototype: `u32 S26_Sched2652_GetModes( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
modes	u8 *	Pointer to a 1-byte application buffer that is to receive the channel mode info. The byte indicates the operating modes for SSR channels 0 to 7. Each bit is associated with a channel number. For example, bit 4 is associated with channel 4. A bit is set to one when operating in the PWM mode, or to zero in the Standard mode.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:

```
// Get all SSR channel operating modes from the 2652 at MM number 0, IOM port 2.
u8 modes;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2652_GetModes( x, 2, &modes );
S26_SchedExecute( x, 1000, 0 );
```

7.9.4 S26_Sched2652_GetOutputs()

Function: Schedules the fetching of the programmed states of all eight SSR drivers on a model 2652 IOM.

Prototype: `u32 S26_Sched2652_GetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
states	u8 *	Pointer to a 1-byte application buffer that is to receive the programmed states of the SSR output drivers. Each bit is associated with one SSR channel. For example, bit 7 is associated with channel 7. Any bit set to one indicates the associated channel is programmed to the active state; any bit set to zero indicates the channel is programmed to the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:

```
// Get all programmed SSR driver states from the 2652 at MM number 0, IOM port 9.
u8 states;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2652_GetOutputs( x, 9, &states );
S26_SchedExecute( x, 1000, 0 );
```

7.9.5 S26_Sched2652_GetPwmRatio()

Function: Schedules the fetching of the PWM ratio for one SSR channel from a model 2652 IOM.

Prototype: `u32 S26_Sched2652_GetPwmRatio( XACT x, IOMPORT IomPort, u8 chan, u8 *OnTime, u8 *OffTime );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.

Parameter	Type	Description
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The SSR channel number that is to be queried. Legal values range from 0 to 7.
OnTime	u8 *	Pointer to a 1-byte application buffer that is to receive the programmed PWM on time expressed in 2 msec increments.
OffTime	u8 *	Pointer to a 1-byte application buffer that is to receive the programmed PWM off time expressed in 2 msec increments.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Fetch the PWM ratio for SSR channel 5 on the 2652 at MM number 0, IOM port 2.
u8 ontime;
u8 offtime;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2652_GetPwmRatio( x, 2, 5, &ontime, &offtime );
S26_SchedExecute( x, 1000, 0 );
```

7.9.6 S26_Sched2652_SetModes()

Function: Schedules the programming of the operating modes for all SSR channels on a model 2652 IOM.

Prototype: `u32 S26_Sched2652_SetModes( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
modes	u8 *	Pointer to a 1-byte application buffer that contains the channel mode flags. The byte indicates the operating modes for SSR channels 0 to 7. Each bit is associated with a channel number. For example, bit 4 is associated with channel 4. Set a bit to one to operate in the PWM mode, or to zero to operate in the Standard mode.

Returns: Error code as described in section 5.5. GWERR_NONE is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:

```
// Set all SSR channel operating modes on the 2652 at MM number 0, IOM port 2.
// Channels 0 to 5: Standard mode, channels 6 and 7: PWM mode.
u8 modes = 0xC0;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2652_SetModes( x, 2, &modes );
S26_SchedExecute( x, 1000, 0 );
```

7.9.7 S26_Sched2652_SetOutputs()

Function: Schedules the programming of all eight SSR output drivers on a model 2652 IOM.

Prototype: `u32 S26_Sched2652_SetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
x	void *	Transaction handle obtained from S26_SchedOpen().
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
states	u8 *	Pointer to a 1-byte application buffer that contains the desired states of the SSR output drivers. Each bit is associated with one SSR 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; zero indicates the channel is to be set to the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.8 ms.

Example:
```
// Program all SSR driver states on the 2652 at MM number 0, IOM port 9.
u8 states = 0x5A;    // The desired relay states.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2652_SetOutputs( x, 9, &states );
S26_SchedExecute( x, 1000, 0 );
```

7.9.8 S26_Sched2652_SetPwmRatio()

Function: Schedules the programming of the PWM ratio for one SSR channel on a model 2652 IOM.

Prototype: `u32 S26_Sched2652_SetPwmRatio( XACT x, IOMPORT IomPort, u8 chan, u8 OnTime, u8 OffTime );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The SSR channel number that is to be programmed. Legal values range from 0 to 7.
`OnTime`	`u8`	PWM on time, expressed in 2 msec increments, to be programmed.
`OffTime`	`u8`	PWM off time, expressed in 2 msec increments, to be programmed.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:
```
// Set the PWM ratio for SSR channel 5 on the 2652 at MM number 0, IOM port 2.
// PWM ratio = on for 20 msec, off for 30 msec.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2652_SetPwmRatio( x, 2, 5, 10, 15 );
S26_SchedExecute( x, 1000, 0 );
```

7.10 Model 2653 Solid-State Relay IOM

The functions in this section are used to schedule IOM actions for Model 2653 16-channel Solid-State Relay IOMs. These functions are applicable only to Model 2653 IOMs. Any attempt to call these functions for other IOM types will result in a `GWERR_IOMTYPE` transaction error. Note that these functions only *schedule* IOM actions into a transaction; they do not cause the actions to be immediately executed.

7.10.1 Type-Specific Errors

In addition to the common IOM status bit flags (`STATUS_RST` and `STATUS_CERR`), this IOM type supports the following type-specific flags. If any of these flags are asserted, a transaction error of type `GWERR_IOMSPECIFIC` will be generated:

Symbolic Name	Description
`STATUS_2653_STRM`	An error was detected in the serial data stream that is used to control the SSR output drivers. This flag can be cleared by invoking a `ClearStatus` action.

7.10.2 S26_Sched2653_GetInputs()

Function: Schedules the fetching of the physical states of all SSR channels on a model 2653 IOM.

Prototype: `u32 S26_Sched2653_GetInputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to a 2-byte application buffer that is to receive the physical states of the SSR channels. The first byte corresponds to channels 0 (lsb) to 7 (msb), and the second byte to channels 8 (lsb) to 15 (msb). A logic one indicates the channel is in its active state; logic zero indicates the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Notes: Each SSR channel includes a monitoring circuit that enables the on-board processor to determine the physical state of the channel. This scheduled action will fetch the monitored physical state of each channel, no matter whether the channel is driven by its own output driver or by an external signal through an input SSR.

Physical states are sampled periodically at two millisecond intervals and passed through a 10 millisecond debounce filter. Consequently, `states` may not accurately reflect the state of a channel that has changed its physical state within the last twelve milliseconds.

Example:

```
// Get all physical SSR states from the 2653 at MM number 0, IOM port 9.
u8 states[2];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2653_GetInputs( x, 9, states );
S26_SchedExecute( x, 1000, 0 );
```

7.10.3 S26_Sched2653_GetModes()

Function: Schedules the fetching of the operating modes for all SSR channels on a model 2653 IOM.

Prototype: `u32 S26_Sched2653_GetModes( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`modes`	`u8 *`	Pointer to a 2-byte application buffer that is to receive the SSR channel mode info. The first byte corresponds to channels 0 (lsb) to 7 (msb), and the second byte to channels 8 (lsb) to 15 (msb). A logic one indicates the channel is operating in PWM mode; logic zero indicates Standard mode.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Get all SSR channel operating modes from the 2653 at MM number 0, IOM port 2.
u8 modes[2];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2653_GetModes( x, 2, modes );
S26_SchedExecute( x, 1000, 0 );
```

7.10.4 S26_Sched2653_GetOutputs()

Function: Schedules the fetching of the programmed states of all SSR drivers on a model 2653 IOM.

Prototype: `u32 S26_Sched2653_GetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`states`	`u8 *`	Pointer to a 2-byte application buffer that is to receive the programmed states of the SSR output drivers. The first byte corresponds to channels 0 (lsb) to 7 (msb), and the second byte to channels 8 (lsb) to 15 (msb). Any bit set to one indicates the associated channel is programmed to the active state; zero indicates the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Get all programmed SSR driver states from the 2653 at MM number 0, IOM port 9.
u8 states[2];
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2653_GetOutputs( x, 9, states );
S26_SchedExecute( x, 1000, 0 );
```

7.10.5 S26_Sched2653_GetPwmRatio()

Function: Schedules the fetching of the PWM ratio for one SSR channel on a model 2653 IOM.

Prototype: `u32 S26_Sched2653_GetPwmRatio( XACT x, IOMPORT IomPort, u8 chan, u8 *OnTime, u8 *OffTime );`

Parameter	Type	Description
`x`	`void *`	Transaction handle obtained from `S26_SchedOpen()`.
`IomPort`	`u8`	The IOM port number (on the MM) to which the target IOM is connected.
`chan`	`u8`	The SSR channel number that is to be queried. Legal values range from 0 to 7.
`OnTime`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the programmed PWM on time expressed in 2 msec increments.
`OffTime`	`u8 *`	Pointer to a 1-byte application buffer that is to receive the programmed PWM off time expressed in 2 msec increments.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Fetch the PWM ratio for SSR channel 5 on the 2653 at MM number 0, IOM port 2.
u8 ontime;
u8 offtime;
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2653_GetPwmRatio( x, 2, 5, &ontime, &offtime );
S26_SchedExecute( x, 1000, 0 );
```

7.10.6 S26_Sched2653_SetModes()

Function: Schedules the programming of the operating modes for all SSR channels on a model 2653 IOM.

Prototype: `u32 S26_Sched2653_SetModes( XACT x, IOMPORT IomPort, u8 *modes );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
modes	u8 *	Pointer to a 2-byte application buffer that contains the channel mode flags. The first byte corresponds to channels 0 (lsb) to 7 (msb), and the second byte to channels 8 (lsb) to 15 (msb). Set a bit to one to operate in the PWM mode, or to *zero* to operate in the Standard mode.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Set all SSR channel operating modes on the 2653 at MM number 0, IOM port 2.
// Channels 6 and 7: PWM mode; all other channels: Standard mode.
u8 modes[] = { 0x00, 0xC0 };
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2653_SetModes( x, 2, modes );
S26_SchedExecute( x, 1000, 0 );
```

7.10.7 S26_Sched2653_SetOutputs()

Function: Schedules the programming of all SSR output drivers on a model 2653 IOM.

Prototype: `u32 S26_Sched2653_SetOutputs( XACT x, IOMPORT IomPort, u8 *states );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
states	u8 *	Pointer to a 2-byte application buffer that contains the desired states of the SSR output drivers. The first byte corresponds to channels 0 (lsb) to 7 (msb), and the second byte to channels 8 (lsb) to 15 (msb). Logic one indicates the associated channel is to be set to the active state; zero indicates the channel is to be set to the inactive state.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Program all SSR driver states on the 2653 at MM number 0, IOM port 9.
u8 states[] = {0x12, 0x5A };    // The desired relay states.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2653_SetOutputs( x, 9, states );
S26_SchedExecute( x, 1000, 0 );
```

7.10.8 S26_Sched2653_SetPwmRatio()

Function: Schedules the programming of the PWM ratio for one SSR channel on a model 2653 IOM.

Prototype: `u32 S26_Sched2653_SetPwmRatio( XACT x, IOMPORT IomPort, u8 chan, u8 OnTime, u8 OffTime );`

Parameter	Type	Description
x	void *	Transaction handle obtained from `S26_SchedOpen()`.
IomPort	u8	The IOM port number (on the MM) to which the target IOM is connected.
chan	u8	The SSR channel number that is to be programmed. Legal values range from 0 to 7.
OnTime	u8	PWM on time, expressed in 2 msec increments, to be programmed.
OffTime	u8	PWM off time, expressed in 2 msec increments, to be programmed.

Returns: Error code as described in section 5.5. `GWERR_NONE` is returned if the operation was successful.

Benchmark: 0.9 ms.

Example:

```
// Set the PWM ratio for SSR channel 5 on the 2653 at MM number 0, IOM port 2.
// PWM ratio = on for 20 msec, off for 30 msec.
void *x = S26_SchedOpen( 0, 1 );
S26_Sched2653_SetPwmRatio( x, 2, 5, 10, 15 );
S26_SchedExecute( x, 1000, 0 );
```

Chapter 8: Comport Transaction Functions

8.1 Overview

This section describes the middleware functions that are used to configure and operate the MM's asynchronous serial communication ports. All of the programming examples reference constants that are defined in the header file `App2600.h`.

8.1.1 Return Values

All comport functions return a u32 value consisting of a three-byte error code and a status byte. The comport error types are a subset of the transaction error types described in section 5.5. The error code occupies the most significant three bytes of the returned value, and the status byte resides in the least significant byte. Zero is returned for the error type if the comport transaction was successful. The returned status byte, which is valid only if the error code is zero, contains a set of active-high bit flags:

Bit	Description
`COM_REJECTED`	A comport command was rejected by the MM. This flag has various meanings, depending on the command that was executed. This is automatically reset at the beginning of each command.
`COM_ISOPEN`	The comport is open (i.e., transmit and receive operations are enabled). This is set by `S26_ComOpen()` and reset by `S26_ComClose()`.
`COM_FRAMINGERROR`	The UART detected a framing error on a received character. This may be reset by calling `S26_ComClearFlags()` or `S26_ComFlush()`.
`COM_PARITYERROR`	The UART detected a parity error on a received character. This may be reset by calling `S26_ComClearFlags()` or `S26_ComFlush()`.
`COM_OVERFLOWERROR`	This indicates one of two conditions: 1. The UART receiver overflowed. This may be reset by calling `S26_ComClearFlags()` or `S26_ComFlush()`. 2. The receiver's ring buffer overflowed, or the UART receiver overflowed. To avoid this error, be sure to remove received data from the receiver buffer before it becomes full. This may be reset by calling `S26_ComClearFlags()` or `S26_ComFlush()`.

All comport status flags are passed through to the application exactly as they are received in the MM's response packet.

8.2 Configuration

8.2.1 S26_ComSetMode()

Function: Sets the operating mode for a comport.

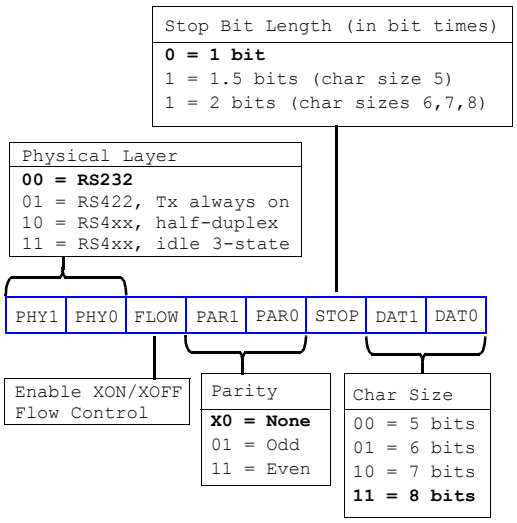
Prototype: `u32 S26_ComSetMode(u32 hbd,u8 dev,u16 cdiv,u8 mode,u8 leds,u32 msec,u32 retries);`

Parameter	Type	Description
hbd	u32	MM handle.
dev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
cdiv	u16	Clock divisor for the baud rate generator. May be set to any value between SIO_BR_300 and SIO_BR_115200.
mode	u8	Collection of enumerated values that set various operating attributes. The byte value is formed by logically or'ing one value from each of the following groups: **Parity**: SIO_PARITY_ODD, SIO_PARITY_EVEN, or SIO_PARITY_NONE. **Databits**: from SIO_DATABITS_5 to SIO_DATABITS_8. **Stopbits**: SIO_STOPBITS_1 or SIO_STOPBITS_2. **Flow control**: SIO_FLOWCTRL_OFF or SIO_FLOWCTRL_ON. **Interface type**: SIO_PHY_RS232, SIO_PHY_RS422_IDLEON, SIO_PHY_RS485 or SIO_PHY_RS422_IDLEOFF.
leds	u8	Specifies the events that will cause the comport status LED to light. May be any combination of the following: SIO_LED_TRANSMIT, SIO_LED_RECEIVE, SIO_LED_ERROR. The byte value is formed by logically or'ing the desired events.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

`cdiv` may be set to any of the following values in order to program standard baud rates. Bold type indicates the default setting after a module reset. Setting `cdiv` to any value not shown in the table will result in a non-standard baud rate.

Baud Rate	cdiv[0]
300	0x0300
600	0x0180
1200	0x00C0
2400	0x0060
4800	0x0030
9600	**0x0018**
19.2K	0x000C
38.4K	0x0006
57.6K	0x0004
115.2K	0x0002

The `mode` byte is shown below. Bold type indicates the default settings after a module reset.



The `leds` byte is shown to the right.

When set to logic one, each bit will cause the comport's status LED to light for approximately 100 milliseconds in response to the associated event.

Any combination of these bits may be specified. For example, the `XMT` and `RCV` bits may both be set, in which case the `LED` will light when characters are sent or received at the comport.

After a module reset, the `RCV` flag is set and all other flags are reset to zero.

7	6	5	4	3	2	1	0
0	0	0	0	XMT	RCV	ERR	0

XMT causes the LED to light when a character is transmitted.

RCV causes the LED to light when a character is received.

ERR causes the LED to light when a receiver break condition is detected or when an error (framing, overrun or parity) is detected.

Returns: Error/status value, as described in section 8.1.1.

Notes: Following a MM reset, `S26_ComSetMode()` should be called to configure each comport that will be used. A comport must be configured before opening it or attempting to send data to or receive data from its remote serial device.

The target comport must be closed when `S26_ComSetMode()` is called. If the comport is open, the command will be rejected and the status byte's `COM_REJECTED` flag will be set.

Example:

```
// Configure COM1 on MM number 0 for the following operating mode:
// 9600 baud, no parity, 8 data, one stop, no flow control, light LED upon receive.
u32 errstat = S26_ComSetMode( 0,
    LOGDEV_COM1,
    SIO_BR_9600,
    SIO_PHY_RS232 | SIO_PARITY_NONE | SIO_DATA_8 | SIO_STOP_1 | SIO_FLOW_OFF,
    SIO_LED_RECEIVE,
    1000,
    1 );
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else if ( errstat & COM_REJECTED )
    printf( "Error: cannot set mode while COM1 is open.\n" );
else
    printf( "Successfully configured COM1.\n" );
```

8.2.2 S26_ComSetBreakChar()

Function: Specifies the Break character for a comport.

Prototype: `u32 S26_ComSetBreakChar( u32 hbd, u8 dev, u8 BreakChar, u32 msec, u32 retries );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`dev`	`u8`	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
`BreakChar`	`u8`	Specifies the character that is to be inserted into the receive buffer upon detection of an incoming break condition. The default break character is 0x00 upon power-up or reset of the MM.
`msec`	`u32`	Maximum time, in milliseconds, to wait for the MM to respond.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: Following a MM reset, `S26_ComSetBreakChar()` may be called for each comport to specify the port's break character. The break character will be automatically inserted into the comport's receive buffer in the event a break

condition is detected on the comport's receive line. The break character may be set to a "printable" character, such as a carriage return character, to provide a "visual" indication that a break was detected.

Break conditions are sometimes employed as message delimiters. For example, a hand-held barcode scanner may assert a break when its trigger is squeezed, and again when the trigger is released. The resulting break characters will then serve as delimiters for the barcode data.

The target comport may be either open or closed when this function is called.

Example:

```
// Configure COM1 on MM number 0 to use a carriage return as its break character.
u32 errstat = S26_ComSetBreakChar( 0, LOGDEV_COM1, 13, 1000, 1 );
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else
    printf( "Successfully set COM1 break char.\n" );
```

8.2.3 S26_ComOpen()

Function: Enable transmit and receive operations on a comport.

Prototype: `u32 S26_ComOpen( HMM hbd, u8 LogDev, u32 msec, u32 retries );`

Parameter	Type	Description
hbd	u32	MM handle.
LogDev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: By default, all comports are closed after a MM reset. `S26_ComOpen()` must be called to enable each comport that will be used. A comport must be enabled before attempting to send data to or receive data from the its remote serial device. Before calling `S26_ComOpen()`, the application should call `S26_ComSetMode()` to configure the comport.

The target comport must be closed when this function is called. If the comport is already open when this command is issued, the command will be rejected and the status byte's `COM_REJECTED` flag will be set.

The `COM_ISOPEN` flag will be asserted in the returned status byte if this function executes successfully.

Example:

```
// Open COM1 on MM number 0.
u32 errstat = S26_ComOpen( 0, 1, 1000, 1 );
if ( errstat & GWERRMASK )
    printf( "COM1 communication problem detected.\n" );
else if ( errstat & COM_REJECTED )
    printf( "COM1 already open.\n" );
else
    printf( "COM1 is %s.\n", ( status & COM_ISOPEN ) ? "open" : "closed" );
```

8.2.4 S26_ComClose()

Function: Disable transmit and receive operations on a comport.

Prototype: `u32 S26_ComClose( HMM hbd, u8 LogDev, u32 msec, u32 retries );`

Parameter	Type	Description
hbd	u32	MM handle.
LogDev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: This function flushes the target comport's serial transmitter and receiver queues. Any character transmission that is in progress is completed.

The target comport must be open when this function is called. If the comport is closed when this command is issued, the command will be rejected and the status byte's `COM_REJECTED` flag will be set.

The `COM_ISOPEN` flag will be negated in the returned status byte if this function executes successfully.

Example:

```
// Close COM1 on MM number 0 and flush all transmit and receive buffers.
u32 errstat = S26_ComClose( 0, 1, 1000, 1 );
if ( errstat & ( GWERRMASK | COM_REJECTED ) )
    printf( "COM1 communication problem detected.\n" );
else if ( errstat & COM_REJECTED )
    printf( "COM1 already closed.\n" );
else
    printf( "COM1 is %s.\n", ( status & COM_ISOPEN ) ? "open" : "closed" );
```

8.3 Communication

8.3.1 S26_ComSend()

Function: Sends data bytes to a comport.

Prototype: `u32 S26_ComSend( u32 hbd, u8 LogDev, char *MsgBuf, u16 MsgLen, u32 msec, u32 retries );`

Parameter	Type	Description
hbd	u32	MM handle.
LogDev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
MsgBuf	char*	Address of a buffer that contains the data bytes to be sent to the target comport.
MsgLen	u16	Number of bytes in `MsgBuf[]` that are to be sent to the target comport.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: This function transfers data bytes to the target comport's transmitter queue. The comport transmitter queue is a FIFO queue, so any bytes that are already pending in the queue will be transmitted before the new bytes are transmitted.

If the transmitter queue would overflow as a result of adding the new data bytes to it, all of the data bytes in the comport command packet will be discarded and the status byte's `COM_REJECTED` flag will be set.

The target comport must be open when this function is called. If the comport is closed, the command will be rejected and the status byte's `COM_REJECTED` flag will be set.

Example:

```
// Send an ASCII string to COM1 on MM number 0.  Note that the message size
// is reduced by 1 because we don't want to transmit the null stored at the
// end of the string.
char Msg[] = "This is a test.";
u32 errstat = S26_ComSend( 0, 1, Msg, sizeof(Msg) - 1, 1000, 1 );
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else if ( errstat & COM_REJECTED )
    printf( "insufficient COM1 buffer space.\n" );
else
    printf( "Sent string to COM1.\n" );
```

Example:

```
// Send a binary string to COM1 on MM number 0.
char Msg[] = { 1, 2, 3, 4, 5 };
u32 errstat = S26_ComSend( 0, 1, Msg, sizeof(Msg), 1000, 1 );
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else if ( errstat & COM_REJECTED )
    printf( "insufficient COM1 buffer space.\n" );
else
    printf( "Sent string to COM1.\n" );
```

8.3.2 S26_ComReceive()

Function: Returns data bytes from a comport's serial receiver queue.

Prototype: `u32 S26_ComReceive(u32 hbd, u8 LogDev, char *MsgBuf, u16 *MsgLen, u32 msec, u32 retries);`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`LogDev`	`u8`	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
`MsgBuf`	`char*`	Address of a buffer that is to receive the data bytes from the target comport.
`MsgLen`	`u16*`	Address of a 16-bit application buffer that contains a byte count. Before calling this function, set the byte count to the maximum number of bytes that are to be transferred from the comport into `MsgBuf[]`. The function will transfer this number of bytes, or all of the unread bytes in the serial receiver queue, whichever is less. When the function returns, the byte count will be set to the number of bytes that were transferred into `MsgBuf[]`.
`msec`	`u32`	Maximum time, in milliseconds, to wait for the MM to respond.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: This function transfers bytes from the comport's serial receiver queue into `MsgBuf[]`. If no bytes are present in the queue, the byte count value at `*MsgLen` will be set to zero, otherwise the number of bytes that were transferred into `MsgBuf[]` will be indicated by `*MsgLen`.

The target comport must be open when this function is called. If the comport is closed, the command will be rejected and the status byte's `COM_REJECTED` flag will be set.

Example:

```
// Fetch and display an ASCII string from COM1 on MM number 0.
char RcvBuf[256];                 // Buffer that will receive the string.
u16 BufLen = sizeof(RcvBuf);      // Max number of characters to receive.
u32 errstat = S26_ComReceive( 0, 1, RcvBuf, &BufLen, 1000, 1 );
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else
{
```

```
    RcvBuf[BufLen] = 0;           // Append null to end of string.
    printf( "%s\n", RcvBuf );     // Display the string.
}
```

8.3.3 S26_ComGetRxCount()

Function: Returns a comport's receive buffer character count.

Prototype: `u32 S26_ComGetRxCount( u32 hbd, u8 LogDev, u16 *CharCount, u32 msec, u32 retries );`

Parameter	Type	Description
hbd	u32	MM handle.
LogDev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
CharCount	u16	Address of a 16-bit application buffer that will receive the character count.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: `S26_ComGetRxCount()` returns the number of pending received characters (i.e., receive characters that have not yet been retrieved by an Ethernet client) remaining in a comport's receive ring buffer.

Example:

```
// Determine the number of characters pending in MM number 0, COM1 receive buffer.
u16 RxCount;
u32 errstat = S26_ComGetRxCount( 0, 1, &RxCount, 1000, 1 );
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else if ( errstat & COM_REJECTED )
    printf( "COM1 is not open.\n" );
else
    printf( "There are %d characters in the Rx buffer.\n", RxCount );
```

8.3.4 S26_ComGetTxCount()

Function: Returns a comport's transmit buffer character count.

Prototype: `u32 S26_ComGetTxCount( u32 hbd, u8 LogDev, u16 *CharCount, u32 msec, u32 retries );`

Parameter	Type	Description
hbd	u32	MM handle.
LogDev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
CharCount	u16	Address of a 16-bit application buffer that will receive the character count.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: `S26_ComGetTxCount()` returns the number of characters remaining in a comport's transmit ring buffer that have not yet been transmitted onto the serial interface. This can be useful if you must determine whether all characters have been sent to a remote serial device, or if you need to find out if there is enough space in the transmit buffer for new characters.

Example:

```
// Determine the number of characters remaining in MM number 0, COM1 transmit buffer.
u16 TxCount;
u32 errstat = S26_ComGetTxCount( 0, 1, &TxCount, 1000, 1 );
```

```
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else if ( errstat & COM_REJECTED )
    printf( "COM1 is not open.\n" );
else
    printf( "There are %d characters in the Tx buffer.\n", TxCount );
```

8.4 Control

8.4.1 S26_ComStartBreak()

Function: Initiates a break transmission on a comport.

Prototype: `u32 S26_ComStartBreak( u32 hbd, u8 LogDev, u32 msec, u32 retries );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`LogDev`	`u8`	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
`msec`	`u32`	Maximum time, in milliseconds, to wait for the MM to respond.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: `S26_ComStartBreak()` is used to initiate a break transmission on a comport. The break condition will continue until `S26_ComEndBreak()` is called or the MM is reset.

The target comport must be open when this function is called. If the comport is already closed when this command is issued, the command will be rejected and the status byte's `COM_REJECTED` flag will be set.

Example: See the example in section 8.4.2.

8.4.2 S26_ComEndBreak()

Function: Terminates a break transmission on a comport.

Prototype: `u32 S26_ComEndBreak( u32 hbd, u8 LogDev, u32 msec, u32 retries );`

Parameter	Type	Description
`hbd`	`u32`	MM handle.
`LogDev`	`u8`	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
`msec`	`u32`	Maximum time, in milliseconds, to wait for the MM to respond.
`retries`	`u32`	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: `S26_ComEndBreak()` is used to terminate a break transmission that was started by calling the `S26_ComStartBreak()` function.

The target comport must be open when this function is called. If the comport is already closed when this command is issued, the command will be rejected and the status byte's `COM_REJECTED` flag will be set.

Example:

```
// For a duration of 250 milliseconds, transmit a break on MM number 0, COM1.
// Error detection is omitted here for clarity.
S26_ComStartBreak( 0, 1, 1000, 1 );
Sleep( 250 );
S26_ComEndBreak( 0, 1, 1000, 1 );
```

8.4.3 S26_ComClearFlags()

Function: Resets all error flags belonging to a comport.

Prototype: `u32 S26_ComClearFlags( u32 hbd, u8 LogDev, u32 msec, u32 retries );`

Parameter	Type	Description
hbd	u32	MM handle.
LogDev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: If this function executes successfully, the COM_FRAMINGERROR, COM_PARITYERROR and COM_OVERFLOWERROR flags will be reset to zero on the target comport. This will be reflected in the returned status byte as well.

Example:

```
// Reset all COM1 error flags on MM number 0.
// For clarity, error detection is not shown here.
S26_ComClearFlags( 0, 1, 1000, 1 );
```

8.4.4 S26_ComFlush()

Function: Flushes a comport's receiver buffer and resets its error flags.

Prototype: `u32 S26_ComFlush( u32 hbd, u8 LogDev, u32 msec, u32 retries );`

Parameter	Type	Description
hbd	u32	MM handle.
LogDev	u8	Logical device identifier for the target comport. Specify a value from 1 to 4 to address comport 1 to 4, respectively.
msec	u32	Maximum time, in milliseconds, to wait for the MM to respond.
retries	u32	Maximum number of transaction retry attempts.

Returns: Error/status value, as described in section 8.1.1.

Notes: S26_ComFlush() may be used to "reset" the receiver buffer to compensate for a detected error on a received character from the remote serial device; this has the effect of resynchronizing the Ethernet client to the remote serial device.

This function should be called, for example, if a parity, framing or overrun error is detected on a received character. When a receive error occurs, the entire contents of the receiver buffer must be considered corrupt and the receiver buffer should accordingly be dumped in preparation for a communication retry to the remote serial device.

The target comport must be open when this function is called. If the comport is already closed when this command is issued, the command will be rejected and the status byte's COM_REJECTED flag will be set.

Example:

```
// Fetch and display an ASCII string from COM1 on MM number 0.

char RcvBuf[256];                 // Buffer that will receive the string.
u16 BufLen = sizeof(RcvBuf);      // Max number of characters to receive.

u32 errstat = S26_ComReceive( 0, 1, RcvBuf, &BufLen, 1000, 1 );
if ( errstat & GWERRMASK )
    printf( "COM1 error detected.\n" );
else if ( errstat & ( COM_PARITYERROR | COM_OVERFLOWERROR | COM_FRAMINGERROR ) )
{
```

```
    // Received a bad character, so we must flush the receive buffer.
    printf( "Character receive error.\n" );
    S26_ComFlush( 0, 1, 1000, 1 );
}
else
{
    // All is OK, so process the received string.
    RcvBuf[BufLen] = 0;           // Append null to end of character string.
    printf( "%s\n", RcvBuf );     // Display the string.
}
```