MicroGate Serial Communications Software Development Kit Windows Guide

MicroGate Systems, Ltd

http://www.microgate.com

Contents

OVERVIEW

This guide describes developing with SyncLink serial communication devices and the Windows operating system.

Use the guide in the order:

Software Installation
Hardware Installation
Verifying Installation
Serial API Programming

Other sections cover general serial communication topics.

Supported Windows Versions

32-bit and 64-bit Windows is supported.

Windows XP (Server 2003/2003R2)
Windows Vista (Server 2008)
Windows 7 (Server 2008R2)
Windows 8 (Server 2012)
Windows 8.1 (Server 2012R2)
Windows 10 (Server 2016/2019)

Required Developer Knowledge

Developing with SyncLink devices on Windows ***requires*** the following knowledge:

1. A supported language: C, C++, C# or Python
2. Basic Windows administration
3. Serial communication details for target application
4. Reading MicroGate documentation

MicroGate offers paid consulting and development services for projects where this knowledge is absent. Contact MicroGate for details.

Software Installation

Before using or installing SyncLink hardware, install supporting software and device drivers. The MicroGate software package is included on media shipped with your hardware. The latest version can be downloaded from:

http://www.microgate.com/ftp/hdlcapi.sdk/hdlcsdk.exe

`hdlcsdk.exe` is a self extracting executable that expands by default to `c:\mgapi`. Run the package from a command line or from the Windows explorer.

Development Files
These files are required to develop applications and are not required for application use.

`readme.txt`	General information
`serial-api-windows.pdf`	This document
`c`	C/C++ development files
`csharp`	C# development files
`python`	Python development files

Run Time Kit
These files are required for application use and should be distributed with the application and hardware.

`rtk\drivers\winXP-8.1\win32`	drivers for 32-bit Windows XP/Server 2002 to 8.1/Server 2012R2
`rtk\drivers\winXP-8.1\win64`	drivers for 64-bit Windows XP/Server 2002 to 8.1/Server 2012R2
`rtk\drivers\win10\win32`	drivers for 32-bit Windows 10
`rtk\drivers\win10\win64`	drivers for 64-bit Windows 10, Server 2016 and later
`rtk\tools\win32`	32-bit support tools
`rtk\tools\win64`	64-bit support tools

Run `setup.exe` in the `drivers` directory as Administrator. This installs drivers so Windows can automatically detect and support SyncLink hardware. Setup updates previously installed devices to the latest drivers.

```
C:\mgapi\rtk\drivers\>setup
```

After software installation, install hardware as described in the next section. Windows detects the hardware and installs drivers for each device. If Windows does not find the drivers automatically, manually specify the search location as the above directories and follow the displayed instructions to complete device installation.

Software Removal
Run `setup.exe` in the appropriate directory with the `/u` option to remove all device instances and the driver packages.

```
C:\mgapi\rtk\drivers>setup /u
```

Hardware Installation

This section describes the configuration and installation of the serial hardware. Configuration must match application requirements.

Each SyncLink port must configured for one of three electrical specifications using jumpers or DIP switches on PCI cards and software for USB.

RS-232/V.28	single ended, unbalanced signals
V.35	differential data/clock signals and single ended control/status signals
RS-422/RS-485/V.11	differential signals

Refer to the hardware guide (PDF) for details. Hardware guides are available at www.microgate.com

PCI Cards

PCI and PCI Express cards are installed into internal expansion slots on the host system. The card type must match the expansion slot type. SyncLink PCI cards are "universal" and are compatible with 3.3V, 5V, 32-bit, 64-bit and PCI-X expansion slots. Do not confuse PCI-X with PCI Express, they are different slot types. SyncLink PCI Express cards are compatible with 1x, 4x, and 16x PCI Express expansion slots.

- Verify card interface selection (RS232,V.35,RS422) via jumpers or DIP switches.
- Shutdown system.
- Remove system case cover.
- Insert adapter in compatible slot.
- Secure card bracket with screw or clamp.
- Replace system case cover.
- Start system.

USB Adapter

The USB serial adapter plugs into a host USB port using the supplied Type B male to Type A male USB cable.

SyncLink USB should be plugged into a USB 2.0 or later Hi-speed (480Mbps) USB port. Operating on a slower USB port is not recommended. Install directly into a host USB port instead of a USB hub for better performance.

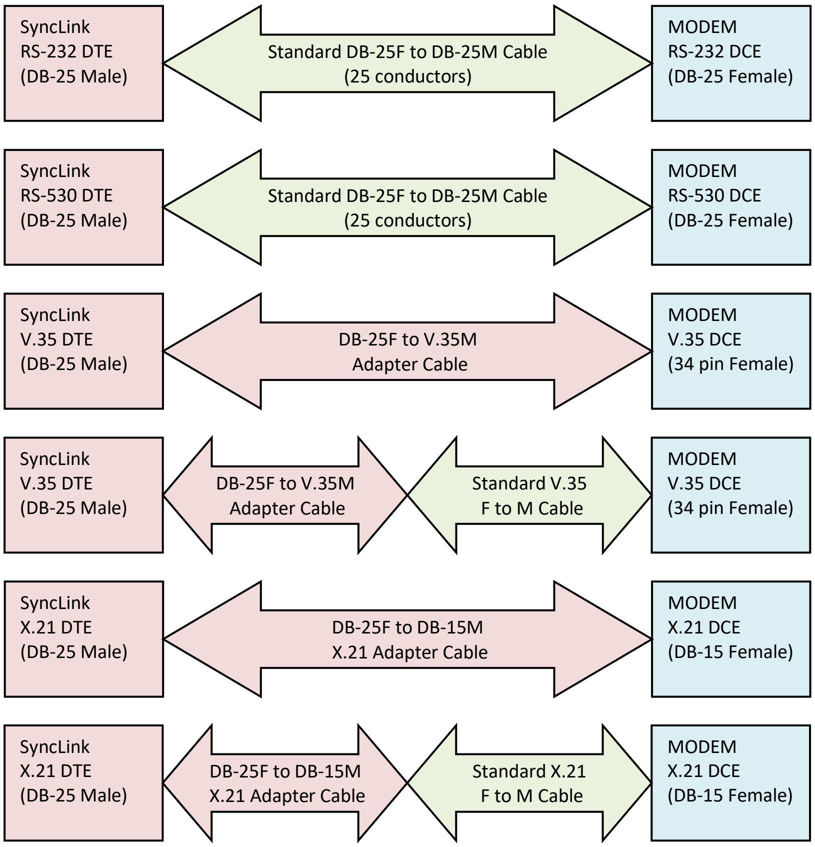
SyncLink USB requires 500mA of power from the USB port, which is standard and supported by most USB ports. Some USB ports may not provide a full 500mA, such as unpowered hubs or ports in small mobile devices.

After insertion and driver installation select the SyncLink USB interface type in the Windows Device Manager.

CABLES

Serial devices are DTE (data terminal equipment) or DCE (data circuit-terminating equipment). A DTE connects directly to a DCE. Two DTEs connect with a cross over cable or null MODEM. A DTE sends and receives data. A DCE converts data to a signal suitable for links like a phone line or radio. SyncLink devices are DTE with a DB25 male connector.

The following diagram shows where standard cables (green) are used and where adapter cables (red) must be purchased from MicroGate to convert the SyncLink DB25 connector to the DCE connector.


SyncLink RS-232 DTE (DB-25 Male)
Standard DB-25F to DB-25M Cable (25 conductors)
MODEM RS-232 DCE (DB-25 Female)
SyncLink RS-530 DTE (DB-25 Male)
Standard DB-25F to DB-25M Cable (25 conductors)
MODEM RS-530 DCE (DB-25 Female)
SyncLink V.35 DTE (DB-25 Male)
DB-25F to V.35M Adapter Cable
MODEM V.35 DCE (34 pin Female)
SyncLink V.35 DTE (DB-25 Male)
DB-25F to V.35M Adapter Cable
Standard V.35 F to M Cable
MODEM V.35 DCE (34 pin Female)
SyncLink X.21 DTE (DB-25 Male)
DB-25F to DB-15M X.21 Adapter Cable
MODEM X.21 DCE (DB-15 Female)
SyncLink X.21 DTE (DB-25 Male)
DB-25F to DB-15M X.21 Adapter Cable
Standard X.21 F to M Cable
MODEM X.21 DCE (DB-15 Female)

If the attached device does not use any of the above connectors, consult documentation for the SyncLink and attached devices to create a custom cable. Pinouts, electrical specification and configuration options are contained in the hardware user’s manual (PDF) for your SyncLink device. Pay close attention to differential signal polarity.

VERIFYING INSTALLATION

Before developing an application, verify the correct installation of serial hardware and device drivers.

WINDOWS DEVICE MANAGER

The primary tool for verification is the Windows Device Manager, an administrative tool included with Windows. This tool displays a tree diagram of hardware devices arranged by type or connection.

The Windows device manager can be started from a command prompt.

To open a command prompt, click the **Start** button, select **All Programs** then **Accessories** and right click on **Command Prompt**. Finally, select **Run as administrator** from the pop-up menu. If prompted for permission to continue, select allow. In the command prompt, run the following command:

```
C:\>devmgmt.msc
```

Once the device manager is running, look for a branch labeled **SyncLink Adapters**.

If you do not see SyncLink Adapters, try selecting the **Computer** branch, click the Action menu and select the “Scan for hardware changes” menu item. This should prompt Windows to detect new hardware and install drivers.

If you still do not see SyncLink Adapters, look for entries with a yellow question mark symbol labeled either “PCI Simple Communications Controller” or “SyncLink USB”. Right click each of these entries and select “Update Driver” from the pop-up menu and follow any displayed instructions. If for some reason Windows can’t find the drivers automatically, manually specify the search location as the driver directory In the SDK package.

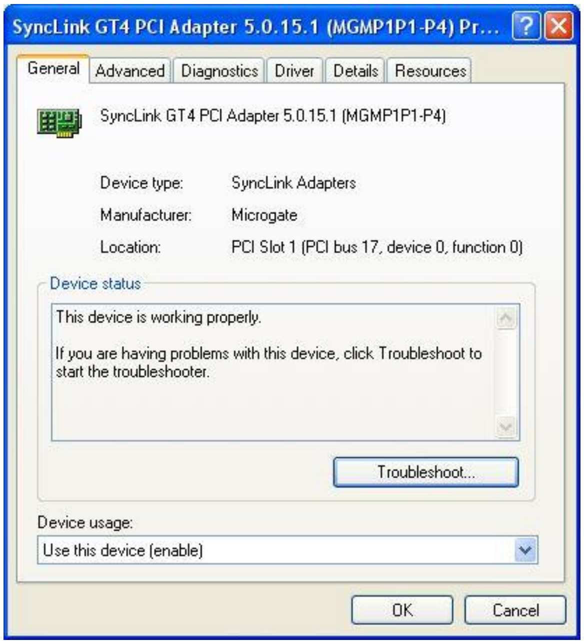
Once the SyncLink Adapter branch is present, expand the branch to display SyncLink devices. Devices with a yellow symbol on the icon have a problem. If no yellow symbol is visible, the device and driver have been correctly installed. Right click on a device entry and select Properties from the pop-up menu.

SyncLink Device Properties

The SyncLink device properties in the Windows device manager displays device and driver information, configures the device and allows testing the device. The device properties window has multiple tabs for different purposes.

General Tab

This tab displays the device status and location (slot or port number). If the device status is working properly then proceed to the next tab. Otherwise note the error message for diagnosing an installation problem. The device driver version and device names are displayed to the right of the card icon near the top.



SyncLink GT4 PCI Adapter 5.0.15.1 (MGMP1P1-P4) Pr...

SyncLink GT4 PCI Adapter 5.0.15.1 (MGMP1P1-P4)

Device type: SyncLink Adapters

Manufacturer: Microgate

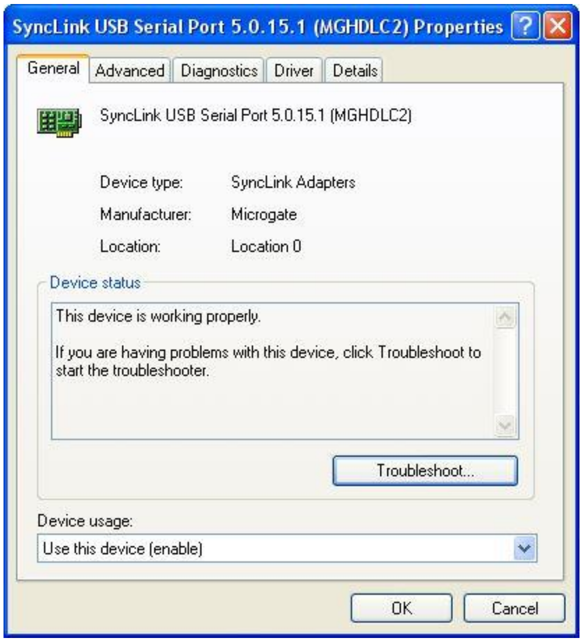
Location: PCI Slot 1 (PCI bus 17, device 0, function 0)

Device status

This device is working properly.

If you are having problems with this device, click Troubleshoot to start the troubleshooter.

Device usage: Use this device (enable)



SyncLink USB Serial Port 5.0.15.1 (MGHDLC2) Properties

SyncLink USB Serial Port 5.0.15.1 (MGHDLC2)

Device type: SyncLink Adapters

Manufacturer: Microgate

Location: Location 0

Device status

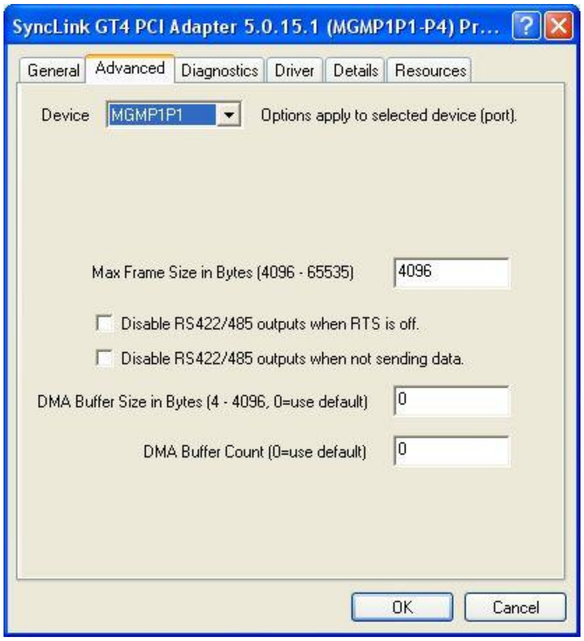
This device is working properly.

If you are having problems with this device, click Troubleshoot to start the troubleshooter.

Device usage: Use this device (enable)

ADVANCED TAB

These settings are applied at system start or device insertion. Some settings apply only to select device types.



SyncLink GT4 PCI Adapter 5.0.15.1 (MGMP1P1-P4) Pr...

Device MGMP1P1 Options apply to selected device (port).

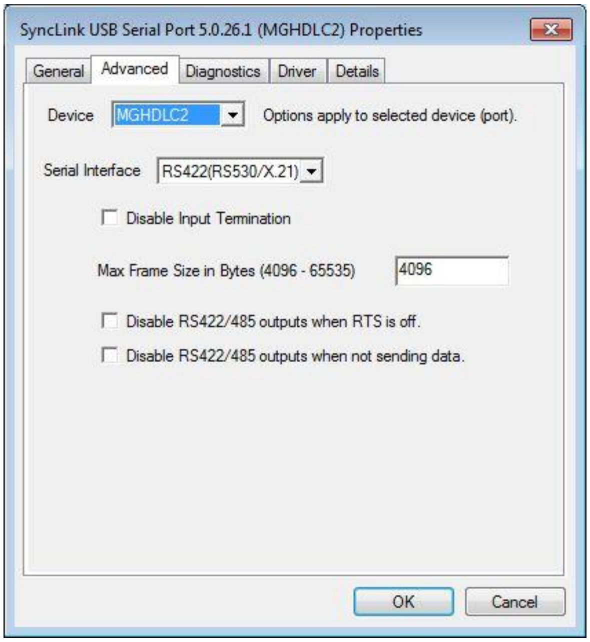
Max Frame Size in Bytes (4096 - 65535) 4096

□ Disable RS422/485 outputs when RTS is off.

□ Disable RS422/485 outputs when not sending data.

DMA Buffer Size in Bytes (4 - 4096, 0=use default) 0

DMA Buffer Count (0=use default) 0



SyncLink USB Serial Port 5.0.26.1 (MGHDLC2) Properties

Device MGHDLC2 Options apply to selected device (port).

Serial Interface RS422(RS530/X.21)

□ Disable Input Termination

Max Frame Size in Bytes (4096 - 65535) 4096

□ Disable RS422/485 outputs when RTS is off.

□ Disable RS422/485 outputs when not sending data.

The **Device** pull down list has an entry for each port on the device. Other settings apply to the selected port.

Max Frame Size
The maximum data size sent or received in a single API call. Default:4096.

Serial Interface (SyncLink USB and SyncLink PCIe Only)
Select the serial interface electrical specification (RS232, V.35, RS422, etc). GT series cards select the interface with jumper settings. Choose the interface type required by your application. Choosing the incorrect interface type prevents correct operation and may damage the device.

Disable Input Termination (SyncLink USB and SyncLink PCIe Only)
When checked, this option disables 120ohm input termination on the USB adapter serial interface when differential modes are selected (RS422/RS530/V.35/X.21). GT series cards enable/disable termination with resistor packs or DIP switch settings.

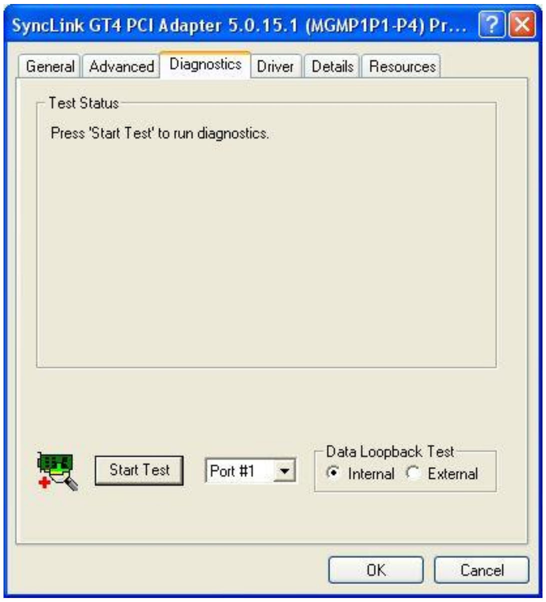
Disable RS422/485 Outputs when RTS is off.
Choose this option when the state of the RTS output signal should be used to control output drivers (enabled or tri-state). An application controls RTS to manually tri-state drivers in a 2-wire half duplex (multidrop) environment.

Disable RS422/485 Outputs when not sending data.
Choose this option if outputs should be disabled (tri-state) when not sending data. Hardware automatically controls driver outputs in a 2-wire half duplex (multidrop) environment.

DMA Buffer Size and DMA Buffer Count
These options control buffer allocation in the driver. Use 0 unless otherwise directed by Microgate support.

DIAGNOSTICS TAB

The **Diagnostics** tab allows you to test the device using an internal or external looback of data.

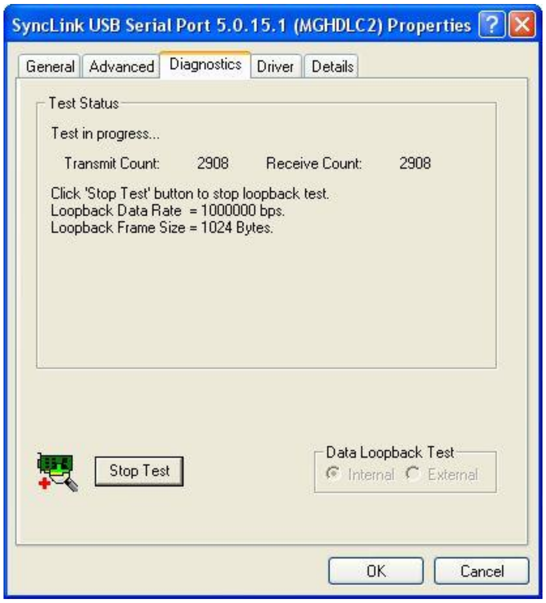


Test Status

Press 'Start Test' to run diagnostics.

Port #1

Data Loopback Test ● Internal ○ External



Test Status

Test in progress...

Transmit Count: 2908 Receive Count: 2908

Click 'Stop Test' button to stop loopback test.
Loopback Data Rate = 1000000 bps.
Loopback Frame Size = 1024 Bytes.

Data Loopback Test ● Internal ○ External

Devices with more than one port will have a pull down list of ports. Select the port to test before proceeding. Devices with only one port will not have this list.

Select the **Data Loopback Test** type: **Internal** or **External**. The internal test does not access the serial connector and only tests the ability of Windows to talk to the device. If external is chosen, install the loopback plug that came with the device on the serial connector.

Then click the **Start Test** button. Send and receive data counts will start incrementing in the **Test Status** area. The test continues until the **Stop Test** button is clicked. Check the **Test Status** area for any error indications.

TRACE UTILITY

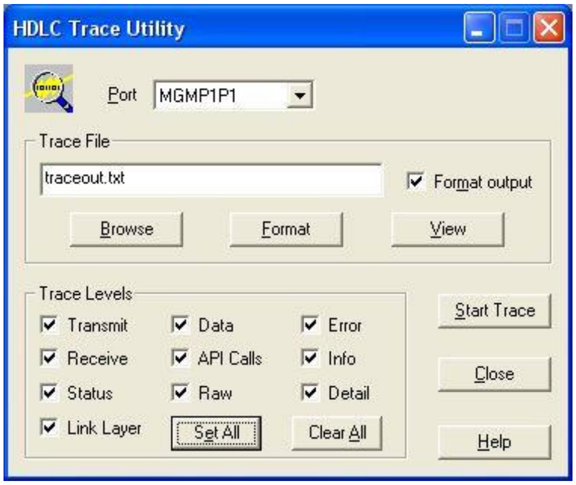
The trace utility records API events for debugging applications and is located at:

```
rtk\tools\win32\mgsltrc.exe     32-bit Windows
rtk\tools\win64\mgsltrc.exe     64-bit Windows
```

Requirements

- Trace type (32-bit/64-bit) must match application. 32-bit applications may run on 64-bit Windows but never at the same time as 64-bit applications. Application type is shown in Windows Task Manager. For C, C++ and C# the application type is determined by the build target. For Python, the application type is determined by the Python installation type.
- Trace must run as same user as application.
- The user must have administrative privilege.

If an **access denied** or **port in use** error is reported by the application or trace utility then carefully review the above requirements. Tracing may start before or after starting application. Run trace from a command line or the Windows explorer. The trace program appears as shown below.



Port: MGMP1P1

Trace File: traceout.txt

☑ Format output

Trace Levels

☑ Transmit ☑ Data ☑ Error

☑ Receive ☑ API Calls ☑ Info

☑ Status ☑ Raw ☑ Detail

☑ Link Layer

1. Enter the trace file name and select **Format output**.
2. Select trace levels. If unsure, enable all levels.
3. Select the port in the **Port** pull down list.
4. Click the **Start Trace** button to start the trace.
5. Perform the tasks to record (run application, connect, etc)
6. Click the **Stop Trace** button to stop the trace.

The output file may be examined for debugging or provided to MicroGate for support.

Device Instances and Names

When hardware is first installed, Windows creates a unique collection of data for the hardware describing the location, configuration and associated software. This data is called a device instance. Each device instance for hardware present in the system is displayed in the Windows device manager. When hardware is removed from the system, the device instance remains but is not displayed in the device manager.

Location Locked and Device Locked Instances

Devices with a unique serial number accessible to Windows (SyncLink USB) use a device instance tied to the specific device called a device locked instance. PCI cards use a device instance tied to the location (PCI slot) called a location locked instance.

Hardware with a device locked instance may be moved to any location (USB port) and the same device instance is used. If the hardware is replaced, a new device instance is created for the new hardware with a different serial number.

Hardware with a location locked instance (PCI cards) may be replaced with the same card type in the same location and the same device instance is used. Moving hardware to a different location (PCI slot) creates a new device instance.

SERIAL API NAMES

When drivers are installed for a SyncLink hardware device, a device name is assigned to the device instance. The name is used to access the device with the serial API. The name is based on an instance number that is assigned sequentially starting with one.

Example:

The first SyncLink GT4 PCI card is adapter number one, with device names MGMP1P1 to MGMP1P4.
The second SyncLink GT4 PCI card is adapter number two, with device names MGMP2P1 to MGMP2P4.

If a PCI card is moved to a different slot, a new device instance is created with a different device name and applications using the original name will fail. The application must use the new name or the old device instance must be removed before creating a new device instance with the old name.

Example:

The first SyncLink GT4 PCI card is adapter number one, with device names MGMP1P1 to MGMP1P4.
The second SyncLink GT4 PCI card is adapter number two, with device names MGMP2P1 to MGMP2P4.
If the first card is moved to a different PCI slot, it becomes MGMP3P1 to MGMP3P2.

To reuse the name MGMP1P1 to MGMP1P4 in the new location, the first device instance must be removed before installing the card in the new location using these steps:

1. Remove hardware from system.
2. Start system and remove device instance. (see next section)
3. Install hardware in new location.

Removing Device Instances of Non-present Hardware

Device instances of non-present hardware are not displayed in the device manager by default. The device manager can be configured using an environment variable to display device instances of non-present hardware:

1. Set the environment variable `devmgr_show_nonpresent_devices = 1`.
2. Start device manager.
3. Select **Show Hidden Devices** item from **View** menu.

Setting environment variable in Windows XP

- Click the **Start** button in the lower left of the desktop
- Right click **My Computer**
- Select **Properties** from pop-up menu
- Click the **Advanced** tab
- Click the **Environment Variables** button near bottom of window
- In the **System variables** section, click the **New** button
- In the **Variable name** field, type `devmgr_show_nonpresent_devices`
- In the **Variable value** field, type 1
- Click the **OK** button to dismiss **New System Variable** window
- Click the **OK** button to dismiss **Environment Variables** window
- Click the **OK** button to dismiss **System Properties** window

Setting environment variable in Windows Vista and Windows 7

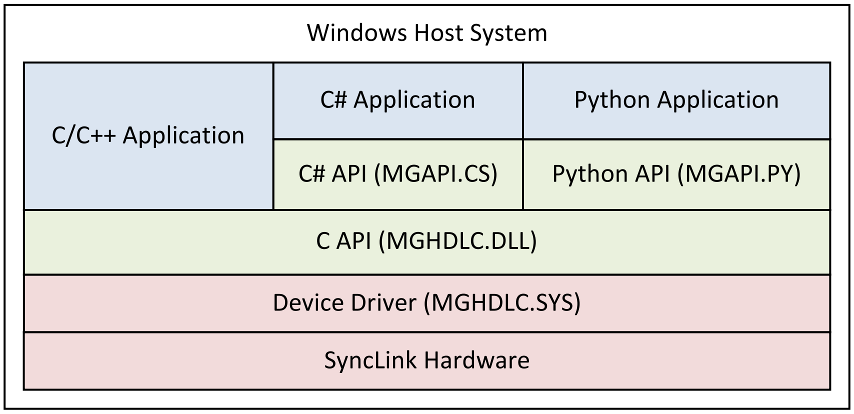
- Click the **Start** button in the lower left of the desktop
- Right click **Computer**
- Select **Properties** from pop-up menu
- Click the **Advanced system settings** in left side of window
- Click the **Environment Variables** button near bottom of window
- In the **System variables** section, click the **New** button
- In the **Variable name** field, type `devmgr_show_nonpresent_devices`
- In the **Variable value** field, type 1
- Click the **OK** button to dismiss **New System Variable** window
- Click the **OK** button to dismiss **Environment Variables** window
- Click the **OK** button to dismiss **System Properties** window

Be careful typing the variable name to ensure it is entered **exactly** the same as above.

Now non-present devices are displayed in the device manager with a grayed out icon. Uninstall the device instance by right clicking on the non-present device and selecting **Uninstall** from the pop-up menu. Repeat this for all grayed out devices in the **SyncLink Adapters** and **SyncLink USB Service Ports** branches of the device tree. Once the non-present device instances have been removed, new hardware can be installed or old hardware moved to a new location.

Serial API Programming

This section describes control of a serial device by an application using the serial API. The primary API is contained in a Windows library (DLL) which provides a C language interface using the Windows standard call convention. C# and Python APIs are provided by modules (mgapi.cs and mgapi.py) that translate the C interface. A working knowledge of the target language in a Windows environment is required.



SOFTWARE ORGANIZATION

Sample code is provided for each supported language. Use the sample code as a starting point for writing your own application. Samples operate on a single device with a loopback plug/cable or two devices connected by a null modem or crossover cable.

Note: The sample code is not intended to be run without modification as an end user application. The sample code WILL NOT match the specific requirements of your environment. The sample code is provided ONLY as a development aid.

C/C++

The following files are used to develop C/C++ serial applications.

`mgapi\c\include\mghdlc.h`	header file defining functions and structures
`mgapi\c\lib\mghdlc.lib`	import library for linking 32-bit applications
`mgapi\c\lib\x64\mghdlc.lib`	import library for linking 64-bit applications

Include the header file in the application source.

```
#include "mghdlc.h"
```

Link the application against the import library. The method for integrating these files into a build environment depends on the tools used. Microsoft Visual Studio requires the import library to be specified in the project settings for **additional libraries**.

Sample Visual Studio 2010 projects are provided for each serial protocol.

`mgapi\c\samples\hdlc`	synchronous HDLC/SDLC
`mgapi\c\samples\raw`	synchronous raw
`mgapi\c\samples\async`	asynchronous
`mgapi\c\samples\commapi`	using Windows asynchronous COM API
`mgapi\c\samples\bisync`	byte synchronous (BISYNC/MONOSYNC)
`mgapi\c\samples\2wire`	HDLC in 2 wire half duplex mode
`mgapi\c\samples\tdm`	Time Division Multiplexing (TDM)
`mgapi\c\samples\fsynth`	programming GT4e/USB frequency synthesizer
`mgapi\c\samples\gpio`	control and monitor GPIO signals

OPEN/CLOSE DEVICE

Call `MgslOpen` or `MgslOpenByName` to get a handle to a serial port for use with other API calls. `MgslOpen` takes a port identifier and `MgslOpenByName` takes a port name. If the port exists and is not in use, `ERROR_SUCCESS` and a device handle are returned. The handle is only valid for the Serial API. Do not use the handle with standard Windows calls.

Call `MgslEnumeratePorts` for a list of identifiers, types and names for each available port. The type and name can be used for display purposes when prompting the user for a port selection.

A port identifier is a 32-bit value identifying an adapter and a port. The upper 16 bits is the port number, and the lower 16 bits is the adapter number. The `MGSL_GET_ADAPTER`() macro returns the adapter number of a port ID. `MGSL_GET_PORT`() returns the port number of a port ID. `MGSL_MAKE_PORT_ID`() creates a port ID from an adapter and port number. Single port adapters always have a port number of 0, so the port ID is the same as the adapter number. Multiport adapters have port numbers starting with 1 up to the number of ports on the adapter.

Port names for single port adapters have the form `MGHDLCx`, where 'x' is the adapter number. Port names for multiport adapters have the form `MGMPxPn`, where x is the adapter number and 'n' is the port number. Adapter numbers are assigned automatically when the adapter is installed.

The following two calls both open the third port of the first multiport adapter.

```
/* open device with integer port identifier */
rc = MgslOpen(MGSL_MAKE_PORT_ID(1, 3), &dev);
if (rc != ERROR_SUCCESS)
      printf("MgslOpen error=%d\n", rc);

/* open device with device name */
rc = MgslOpenByName(“MGMP1P3”, &dev);
if (rc != ERROR_SUCCESS)
      printf("MgslOpen error=%d\n", rc);
```

The following two calls both open the only port of the second single port adapter.

```
/* open device with integer port identifier */
rc = MgslOpen(MGSL_MAKE_PORT_ID(2, 0), &dev);
if (rc != ERROR_SUCCESS)
      printf("MgslOpen error=%d\n", rc);

/* open device with device name */
rc = MgslOpenByName(“MGHDLC2”, &dev);
if (rc != ERROR_SUCCESS)
      printf("MgslOpen error=%d\n", rc);
```

Call `MgslClose` with an open handle after the port is no longer needed so other processes can open the port.

CONFIGURE DEVICE

A device must be configured to match application specific requirements. This is done using the following API calls.

`MgslSetPortConfigEx` Set options that take effect at driver load time, such as serial interface type. These settings are stored in the Windows registry and are read by the driver when loading, usually when Windows is starting. These options are usually set using the device properties in the Windows Device Manager instead of from an application with this call.

`MgslSetParams` Set options that take effect at start of communications session. This is the main configuration call. Calling this function resets the transmitter and receiver.

`MgslSetIdleMode` Set the idle (SDLC/HDLC/raw) or sync (monosync/bisync) pattern.

`MgslSetOption` Set options that can change during a communications session.

The main configuration call is `MgslSetParams`, which uses an `MGSL_PARAMS` structure to specify protocol options. This call is documented in the `MgslSetParams` and `MGSL_PARAMS` sections. The following sample configures a port for HDLC mode. The actual settings used depend on the application requirements.

```
HANDLE dev;
int rc;
MGSL_PARAMS params;

/*
 * SDLC/HDLC mode, loopback disabled
 * receive clock source = RxC input pin
 * transmit clock source = TxC input pin
 * NRZ encoding
 * output 9600bps clock on AUXCLK output pin
 * use ITU/CCITT 16-bit CRC frame check
 */
params.Mode = MGSL_MODE_HDLC;
params.Loopback = 0;
params.Flags = HDLC_FLAG_RXC_RXCPIN + HDLC_FLAG_TXC_TXCPIN;
params.Encoding = HDLC_ENCODING_NRZ;
params.ClockSpeed = 9600;
params.CrcType = HDLC_CRC_16_CCITT;

/* set current device parameters */
rc = MgslSetParams(dev, &params);
if (rc != ERROR_SUCCESS)
      printf("MgslSetParams error=%d",rc);

/* set transmit idle pattern (sent between frames) */
rc = MgslSetIdleMode(dev, HDLC_TXIDLE_ONES);
if (rc != ERROR_SUCCESS)
      printf("MgslSetIdleMode error=%d", rc);
```

Receiving Data

An application gets receive data using the `MgslRead` call.

```
unsigned char buf[4096];
int size = sizeof(buf);
int count;

/* get receive data */
count = MgslRead(dev, buf, size);
if (count) {
      /* count bytes returned in buf */
} else {
      /* no data available (polled mode) or error (blocking mode) */
}
```

`MgslReadWithStatus` lets applications process receive errors in addition to valid receive data or inspect received CRC values. `MgslReceive` adds asynchronous notification in addition to error reporting and should only be used in the very rare cases where the extra features are needed and well understood.

- `MgslRead` = preferred method to receive data (easy)
- `MgslReadWithStatus` = receive data and error reporting (more complex)
- `MgslReceive` = receive data, error reporting and asynchronous notification (most complex)

Behavior of the receive functions depends on the configured serial protocol (HDLC, bisync, async, etc) as described in later sections and in the provided sample code.

HDLC/SDLC

Each buffer returned by the API is one frame of variable size. Set `size` argument to largest expected frame size (typically 4K). Actual frame size is reported by return code.

Monosync/Bisync/Raw/Asynchronous/Isosynchronous

Each buffer returned by the API is a fixed size block of data, set by the `size` argument, and does not imply any message boundaries. In these modes the API does not detect message boundaries, which is the responsibility of the application. Data is not returned until `size` bytes are received.

Blocking and Polled Modes

Receive calls can be used in blocking or polled mode by calling `MgslSetOption` with the `MGSL_OPT_RX_POLL` identifier. When this option is enabled, receive calls return immediately when no data is available (polling mode). When not enabled, receive calls block until data is available (blocking mode = default).

A blocked receive call can be canceled from a different application thread with `MgslCancelReceive`.

MgslReceive (Advanced Features)

`MgslReceive` has extra error reporting and asynchronous notification features for advanced applications and uses the `MGSL_RECEIVE_REQUEST` structure. The receive request structure and overlapped structure must be initialized before each call. If data is available, `ERROR_SUCCESS` is returned. If no data is available and the receiver is enabled then `ERROR_IO_PENDING` is returned and the caller monitors the event member of the overlapped structure for notification of request completion. When the request is complete, the request structure contains data and status information.

This sample code demonstrates request preparation, call and processing. `CreateEvent`, `ResetEvent` and `WaitForSingleObject` are Windows system calls. Refer to the Windows SDK documentation for details. The request and overlapped structures **must** remain valid (heap or stack allocation) until the request completes.

```
OVERLAPPED ol;
MGSL_RECEIVE_REQUEST *req;
int buffer_size = 4096;

/* request preparation */
ol.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
req = malloc(sizeof(MGSL_RECEIVE_REQUEST) + buffer_size);
req->DataLength = buffer_size;
ResetEvent(ol.hEvent);

rc = MgslReceive(dev, req, &ol);
if (rc == ERROR_IO_PENDING) {
      rc = WaitForSingleObject(ol.hEvent, INFINITE);
      if (rc != WAIT_OBJECT_0)
            /* wait error */
} else if (rc != ERROR_SUCCESS) {
      /* MgslReceive error */
}

/* process completed request */
if (req->Status == RxStatus_OK) {
      /* req->DataLength contains count of returned data */
      /* req->DataBuffer contains returned data */
} else
      /* receive error (CRC etc), no data returned */
```

Sending Data

An application sends data using the `MgslWrite` call.

```
unsigned char buf[4096];
int size = sizeof(buf);
int count;

/* initialize buffer with application data */

/* send data */
count = MgslWrite(dev, buf, size);
if (count) {
      /* count bytes stored in API send buffers */
} else {
      /* API send buffers full (polled) or error (blocked) */
}
```

`MgslWrite` is a simplified wrapper function for the original `MgslTransmit` function described below. If you do not need the extra error reporting and asynchronous notification features of `MgslTransmit`, use the simpler `MgslWrite`.

The format of the data depends on the protocol. For frame oriented SDLC/HDLC, each call sends a single frame of data. For other protocols, each call sends data with no formatting, requiring the application to implement message boundaries.

Blocking and Polled Modes

MgslWrite (and MgslTransmit) can be used in blocking or polled mode by calling MgslSetOption with the MGSL_OPT_TX_POLL identifier. When this option is enabled, MgslWrite (or MgslTransmit) returns immediately with a return code of zero when all API send buffers are full (polling mode). When not enabled, MgslWrite blocks until API send buffers are available and data is accepted (blocking mode = default).

A blocked call to MgslWrite can be aborted by calling MgslCancelTransmit from a different application thread.

MgslTransmit (advanced)

MgslTransmit offers additional error reporting and asynchronous notification. If data is accepted, the call returns ERROR_SUCCESS. If no buffers are available then ERROR_IO_PENDING is returned and the caller monitors the event member of the overlapped structure for notification of request completion.

Buffers are sent as soon as possible, but the request completes before the data is actually sent. MgslTransmit can be used to determine when all buffered data has been sent. Refer to the function reference for details.

This sample code demonstrates request preparation, call and processing. CreateEvent, ResetEvent and WaitForSingleObject are Windows system calls. Refer to the Windows SDK documentation for details. The send buffer and overlapped structure **must** remain valid (heap or stack allocation) until the request completes.

```
int rc;
int size = 1024;
unsigned char buf[1024];
OVERLAPPED ol;

ol.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

/* submit data to send */
ResetEvent(ol.hEvent);
rc = MgslTransmit(dev, buf, size, NULL, &ol);
if (rc == ERROR_IO_PENDING) {
       /* wait for free buffer */
       rc = WaitForSingleObject(ol.hEvent, INFINITE);
       if (rc != WAIT_OBJECT_0)
              /* wait error */
} else if (rc != ERROR_SUCCESS) {
       /* MgslTransmit error */
} else {
       /* data accepted */
}
```

WAITING FOR ALL DATA SENT

Data given to the API with MgslWrite or MgslTransmit is saved to API buffers and sent as soon as possible. To determine when the buffered data has been completely sent use the MgslWaitAllSent call.

```
int rc;

/* check for all buffered data sent */
rc = MgslWaitAllSent(dev);
if (!rc) {
      /* all buffered data sent, safe to close port */
} else {
      /* busy sending buffered data (polled) or error (blocking) */
}
```

MgslWaitAllSent is a simplified wrapper function for the original MgslTransmit function. If you do not need the extra error reporting and asynchronous notification features of MgslTransmit, use the simpler MgslWaitAllSent.

Blocking and Polled Modes

MgslWaitAllSent can be used in blocking or polled mode by calling MgslSetOption with the MGSL_OPT_TX_POLL identifier. When this option is enabled, MgslWaitAllSent returns immediately with a return code of zero when all data is sent or non zero if still sending (polled mode). When not enabled, MgslWaitAllSent blocks until all buffered data has been sent (blocking mode = default).

A blocked call to MgslWaitAllSent can be aborted by calling MgslCancelTransmit from a different application thread.

Asynchronous API Notification

API calls that may not complete immediately (MgslTransmit, MgslReceive, MgslWaitEvent, MgslWaitGpio, MgslGetTraceEvent) require an application allocated Windows OVERLAPPED structure.

Initialize the hEvent member of the overlapped structure with the handle of a Windows manual reset event allocated with CreateEvent. Set the other members of the overlapped structure to zero. If an API call returns ERROR_IO_PENDING, monitor the event with the Windows functions WaitForSingleObject or WaitForMultipleObjects**.** When the API signals the event object, the call is complete.

The overlapped structure must remain allocated (on the stack or heap) until request completion.

For more information about the OVERLAPPED structure, Windows events and synchronization, refer to the Windows SDK documentation.

API calls using asynchronous notification can have only one pending instance of each API call. For example, if a MgslTransmit call is pending, the application cannot call MgslTransmit again until the original call completes. Different API calls, such as MgslTransmit and MgslReceive, may be simultaneously pending. Calling an API function that uses asynchronous notification while an instance of that call is already pending results in a return code of ERROR_BUSY.

Functions

This section documents the application programming interface (API) calls used with serial devices. The calls use the Windows standard call (__stdcall) calling convention used by most Windows libraries. The documentation uses the C programming language, but other languages that can access the standard call conventions can also be used.

Call interface details are defined in the C language header file `mghdlc.h`, which must be included in C language source files. The user mode application interface is implemented in the `mghdlc.dll` library which translates the calls to access the device driver `mghdlc.sys`. Applications need to link against the import library `mghdlc.lib`.

MgslCancelGetTraceEvent

```
ULONG MgslCancelGetTraceEvent(HANDLE dev);
```

Arguments

`dev`	handle to open device

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	device handle is invalid

This function cancels a pending call to `MgslGetTraceEvent`. The API completes the pending request with `EventType_None` and signals the event member of the overlapped structure passed to `MgslGetTraceEvent`. Pending requests are automatically cancelled when a device handle is closed.

MgslCancelGetWaitGpio

```
ULONG MgslCancelGetWaitGpio(HANDLE dev);
```

Arguments

`dev`	handle to open device

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	device handle is invalid

This function cancels a pending call to `MgslWaitGpio`. The API signals the `hEvent` member of the overlapped structure passed to `MgslWaitGpio` and the contents of the `GPIO_DESC` structure is undefined. Pending requests are automatically cancelled when a device handle is closed.

MgslCancelReceive

```
ULONG MgslCancelReceive(HANDLE dev);
```

Arguments

`dev`	handle to open device

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	device handle is invalid

This function cancels a pending `MgslReceive` call. The API signals the `hEvent` member of the overlapped structure passed to `MgslReceive` and the receive status is set to `RxStatus_Cancel`. Pending requests are automatically cancelled when a device handle is closed.

MgslCancelTransmit

```
ULONG MgslCancelTransmit(HANDLE dev);
```

Arguments

`dev`	handle to open device

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	device handle is invalid

This function cancels a pending `MgslTransmit` call. The API signals the `hEvent` member of the overlapped structure passed to `MgslTransmit` and the transmit status is set to `TxStatus_Cancel`. Pending requests are automatically cancelled when a device handle is closed.

Calling this function does not disable the transmitter. The transmitter continues sending the idle pattern until more data is sent or the user disables the transmitter with the `MgslEnableTransmitter` call.

MgslCancelWaitEvent

```
ULONG MgslCancelWaitEvent(HANDLE dev);
```

Arguments

`dev`	handle to open device

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	device handle is invalid

This function cancels a pending `MgslWaitEvent` call. The API signals the `hEvent` member of the overlapped structure passed to `MgslWaitEvent` and the returned value of serial events is set to zero. Pending requests are automatically cancelled when a device handle is closed.

MgslClose

```
ULONG MgslClose(HANDLE dev);
```

Arguments

`dev`	handle to open device

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	device handle is invalid

This function closes an open device handle. The handle is not valid after this call. Pending requests are automatically cancelled when a device handle is closed. A handle must be closed before another process can open the device.

MgslEnableReceiver

```
ULONG MgslEnableReceiver(HANDLE dev, BOOL enable);
```

Arguments

`dev`	handle to open device
`enable`	enable command value 0 = disable 1 = enable 2 = enable and force hunt mode

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	device handle is invalid

This function controls the receiver state:

Disabled	receive data signal ignored
Idle	receive data signal scanned for synchronization pattern (flag, start bit, etc)
Active	receive data signal stored for application

The receiver starts disabled. If the receiver is disabled, the enable command makes the receiver idle and discards buffered data, otherwise it does nothing. The disable command disables the receiver. The hunt mode command makes the receiver idle.

A synchronization pattern (start bit, flag, sync) makes an idle receiver active. Raw synchronous mode does not use a synchronization pattern, and the receiver is either disabled or active with both the enable and hunt mode commands making the receiver active.

```
/* enable receiver (store data on receive data input) */
rc = MgslEnableReceiver(dev, 1);
if (rc != ERROR_SUCCESS)
      /* process error */

/* disable receiver (ignore receive data input) */
rc = MgslEnableReceiver(dev, 0);
if (rc != ERROR_SUCCESS)
      /* process error */
/* force receiver to idle state (look for sync pattern) */
rc = MgslEnableReceiver(dev, 2);
if (rc != ERROR_SUCCESS)
      /* process error */
```

MgslEnableTransmitter

```
ULONG MgslEnableTransmitter(HANDLE dev, BOOL enable);
```

Arguments

dev	handle to open device
enable	enable command value 0 = disable 1 = enable

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	device handle is invalid

This function controls the transmitter state:

Disabled	transmit data signal is constant mark (one)
Idle	transmit data signal sends idle or sync pattern
Active	transmit data signal sends data

The transmitter starts disabled. The enable command makes the transmitter idle if disabled, otherwise it does nothing. The disable command disables the transmitter and discards buffered data. Calling MgslTransmit enables the transmitter if disabled. When data is ready to send, the transmitter is active.

```
/* enable transmitter (start sending idle pattern) */
rc = MgslEnableTransmitter(dev, 1);
if (rc != ERROR_SUCCESS)
      /* process error */

/* disable transmitter (discard unsent data) */
rc = MgslEnableTransmitter(dev, 0);
if (rc != ERROR_SUCCESS)
      /* process error */
```

MgslEnumeratePorts

```
ULONG MgslEnumeratePorts(MGSL_PORT *ports, ULONG size, ULONG *count);
```

Arguments

`ports`	pointer to buffer to receive array of `MGSL_PORT` structures
`size`	size of `ports` buffer in bytes
`count`	returned count of port structures

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_PARAMETER`	ports buffer invalid or too small
`ERROR_GEN_FAILURE`	unspecified failure

This function returns information on available serial API ports in an array of `MGSL_PORT` structures. Use this information to identify and open ports with the `MgslOpen` call. On return the `ports` buffer contains the port information and the `count` argument contains the number of returned entries.

Call this function with the `ports` argument set to `NULL` to return only the number of available ports in the `count` argument. Use this to allocate a buffer for the `ports` argument on a subsequent call.

```
char name[] = “MGHDLC1”;
unsigned long i, rc, count;
int port_id = 0;
MGSL_PORT *ports;

/* get count of available ports */
rc = MgslEnumeratePorts(NULL, 0, &count);
if (rc != ERROR_SUCCESS)
      /* process error */

/* allocate memory to hold port information */
ports = malloc(count * sizeof(MGSL_PORT));

/* get port information */
rc = MgslEnumeratePorts(ports, count * sizeof(MGSL_PORT), &count);
if (rc != ERROR_SUCCESS)
      /* process error */

/* convert device name to port_id */
for (i=0; i < count; i++) {
      if (!stricmp(ports[i].DeviceName, name)) {
            port_id = ports[i].PortID;
            break;
      }
}

free(ports);
```

MgslGetAssignedResources

```
ULONG MgslGetAssignedResources(HANDLE dev, MGSL_ASSIGNED_RESOURCES *res);
```

Arguments

dev	handle to open device
res	pointer to MGSL_ASSIGNED_RESOURCES structure

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	invalid device handle
ERROR_INVALID_PARAMETER	invalid res buffer

This function returns the embedded serial number of the device, if available. Only the SerialNumber field of the structure is used. All other fields of the structure are unused and undefined. Only the SyncLink USB device has an embedded serial number. For all other devices, this function has no purpose.

```
MGSL_ASSIGNED_RESOURCES res;

rc = MgslGetAssignedResources(dev, &res);
if (rc != ERROR_SUCCESS)
      printf("MgslGetAssignedResources error=%d\n", rc);
else
      printf(“serial number is %s\n”, res.SerialNumber);
```

MgslGetGpio

```
ULONG MgslGetGpio(HANDLE dev, GPIO_DESC *gpio);
```

Arguments

dev	handle to open device
gpio	pointer to GPIO_DESC structure

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	invalid device handle
ERROR_INVALID_PARAMETER	invalid gpio buffer

This function returns the current direction configuration and state of all general purpose I/O (GPIO) signals.

```
GPIO_DESC gpio;

rc = MgslGetGpio(dev, &gpio);
if (rc != ERROR_SUCCESS)
      printf("MgslGetGpio error=%d\n", rc);
else {
      /* process structure contents */

      if (gpio.dir & (1 << 5))
            printf(“GPIO 5 is an output\n”);
      else
```

```
            printf("GPIO 5 is an input\n");

      if (gpio.state & (1 << 5))
            printf("GPIO 5 is on\n");
      else
            printf("GPIO 5 is off\n");
}
```

MgslGetOption

```
ULONG MgslGetOption(HANDLE dev, UINT option, UINT *value);
```

Arguments

`dev`	handle to open device
`option`	option identifier
`value`	pointer to returned option value

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	invalid device handle
`ERROR_INVALID_PARAMETER`	invalid `option` identifier or `value` buffer

This function returns the specified configuration or status value. Refer to `MgslSetOption` for a description of valid option identifiers.

```
UINT value;

rc = MgslGetOption(dev, MGSL_OPT_RTS_DRIVER_CONTROL, &value);
if (rc != ERROR_SUCCESS)
      printf("MgslSetOption error=%d\n", rc);
else if (value)
      printf("RTS controls output drivers\n");
else
      printf("RTS does not control output drivers\n");
```

MgslGetParams

```
ULONG MgslGetParams(HANDLE dev, MGSL_PARAMS *params);
```

Arguments

`dev`	handle to open device
`params`	pointer to `MGSL_PARAMS` structure

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	invalid device handle
`ERROR_INVALID_PARAMETER`	invalid params buffer

This function returns the current serial device configuration in a `MGSL_PARAMS` structure. This call gets options that are specified once for a communications session. `MgslGetPortConfigEx` is used for driver load time settings and `MgslGetOption` is used for settings that may change during a communications session.

```
MGSL_PARAMS params;

rc = MgslGetParams(dev, &params);
if (rc != ERROR_SUCCESS)
      printf("MgslGetParams error=%d\n", rc);
else
      /* process structure contents */
```

MgslGetPortConfigEx

```
ULONG MgslGetPortConfigEx(ULONG port_id, MGSL_PORT_CONFIG_EX *config);
```

Arguments

`port_id`	integer port identifier (not an open port handle)
`config`	pointer to returned `MGSL_PORT_CONFIG_EX` structure

Return Value

`ERROR_SUCCESS`	call success
`ERROR_ACCESS_DENIED`	insufficient privilege to access registry
`ERROR_INVALID_PARAMETER`	invalid `config` buffer

This function returns the current serial device configuration in a `MGSL_PORT_CONFIG_EX` structure, which is used for options that take effect at driver load time. These options are usually set in the Windows Device Manager, but this call is implemented so the options can be programmatically set.

```
MGSL_PORT_CONFIG_EX config;
unsigned int port_id;

/* operate on port 3 of first adapter */
port_id = MGSL_MAKE_PORT_ID(1,3);

rc = MgslGetPortConfigEx(port_id, &config);
if (rc != ERROR_SUCCESS)
      printf("MgslSetPortConfigEx error=%d\n", rc);
else
      /* process structure contents */
```

MgslGetSerialSignals

```
ULONG MgslGetSerialSignals(HANDLE dev, UCHAR *signals);
```

Arguments

dev	open port handle
config	returned signal states

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	invalid port handle
ERROR_INVALID_PARAMETER	invalid signals buffer

This function returns the state of serial control and status signals. Signal states are identified by macros defined in the `mghdlc.h` header file. A set bit indicates an active signal. Depending on the application, some signals may not be used or may be used for a non-standard purpose.

SerialSignal_DCD	Data Carrier Detect input (MODEM or DCE detects signal from remote device)
SerialSignal_DSR	Data Set Ready input (MODEM or DCE is turned on)
SerialSignal_DTR	Data Terminal Ready output (serial device is active)
SerialSignal_RTS	Request to Send output (serial device needs to send data)
SerialSignal_CTS	Clear to Send input (serial device is allowed to send data)
SerialSignal_RI	Ring Indicator input (MODEM or DCE detects incoming call)

```
UCHAR signals;

rc = MgslGetSerialSignals(dev, &signals);
if (rc != ERROR_SUCCESS)
      /* process error */
else if (signals & SerialSignal_DCD)
      /* DCD is active */
```

MgslGetTraceEvent

```
ULONG MgslGetTraceEvent(HANDLE dev, MGSL_TRACE_EVENT *event, OVERLAPPED *ol);
```

Arguments

`dev`	open port handle
`event`	pointer returned trace event structure
`ol`	pointer to Windows overlapped structure for asynchronous notification of call completion

Return Value

`ERROR_SUCCESS`	call success (immediate completion)
`ERROR_IO_PENDING`	waiting for trace event (monitor overlapped structure)
`ERROR_INVALID_HANDLE`	invalid port handle
`ERROR_INVALID_PARAMETER`	invalid event buffer
`ERROR_BUSY`	request already pending

This function returns a trace event. If a trace event is immediately available, ERROR_SUCCESS is returned, otherwise ERROR_IO_PENDING is returned and the application should monitor the hEvent member of the overlapped structure for indication of request completion. `MgslCancelGetTraceEvent` cancels a pending request. Only one `MgslGetTraceEvent` request can be active at a time.

This function uses standard Windows asynchronous notification. This sample code demonstrates request preparation, call and processing. `CreateEvent`, `ResetEvent` and `WaitForSingleObject` are Windows system calls. Refer to Windows SDK documentation for details. The request and overlapped structures **must** remain valid (heap or stack allocation) until the request completes.

```
OVERLAPPED ol;
MGSL_TRACE_EVENT event;

ol.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

/* request preparation */
ResetEvent(ol.hEvent);

rc = MgslGetTraceEvent(dev, &event, &ol);
if (rc == ERROR_IO_PENDING) {
      rc = WaitForSingleObject(ol.hEvent, INFINITE);
      if (rc != WAIT_OBJECT_0)
            /* wait error */
} else if (rc != ERROR_SUCCESS) {
      /* MgslGetTraceEvent error */
}

/* process contents of MGSL_TRACE_EVENT structure */
```

MgslGetTraceLevel

```
ULONG MgslGetTraceLevel(HANDLE dev, ULONG *level);
```

Arguments

`dev`	open port handle
`level`	pointer to returned trace level value

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	invalid port handle
`ERROR_INVALID_PARAMETER`	invalid level buffer

This function returns the current trace level that specifies which events are recorded in the trace buffer. Levels are identified with `TraceLevel_XXX` macros defined in the `mghdlc.h` header file. See `MgslSetTraceLevel` for a description of the different trace levels.

```
ULONG level;

rc = MgslGetTraceLevel(dev, &level);
if (rc != ERROR_SUCCESS)
      printf("MgslGetTraceLevel error=%d\n", rc);
else if (level & TraceLevel_Data)
      printf("Data tracing is enabled\n");
```

MgslOpen

```
ULONG MgslOpen(ULONG port_id, HANDLE *dev);
```

Arguments

`port_id`	port identifier
`dev`	pointer to returned port handle

Return Value

`ERROR_SUCCESS`	call success
`ERROR_DEVICE_IN_USE`	port is in use by another application
`ERROR_BAD_DEVICE`	`port_id` does not exist or is not functioning

This function opens a serial device identified by `port_id` and returns a port handle for use by other API functions. Call `MgslClose` to close the returned handle when it is no longer needed. The port ID can be obtained from `MgslEnumeratePorts`.

The format of the port ID depends on the hardware type:

Single Port Adapter	32-bit integer adapter number Example: 3 (single port adapter 3)
Multiple Port Adapter	32-bit integer with upper 16 bits = port number and lower 16 bits = adapter number Example: 0x00040003 (port 4 of adapter 3)

Macros for manipulating port IDs are defined in the `mghdlc.h` header file:

```
MGSL_MAKE_PORT_ID(AdapterNumber,PortNumber)
MGSL_GET_ADAPTER(PortID)
MGSL_GET_PORT(PortID)

unsigned int port_id;
HANDLE dev;

/* operate on port 3 of first adapter */
port_id = MGSL_MAKE_PORT_ID(1,3);

rc = MgslOpen (port_id, &dev);
if (rc != ERROR_SUCCESS)
      printf("MgslOpen error=%d\n", rc);
else
      /* dev contains valid device handle */
```

MgslOpenByName

```
ULONG MgslOpenByName(char *name, HANDLE *dev);
```

Arguments

name	port name string
dev	pointer to returned port handle

Return Value

ERROR_SUCCESS	call success
ERROR_DEVICE_IN_USE	port is in use by another application
ERROR_BAD_DEVICE	port_id does not exist or is not functioning

This function opens a serial device identified by name and returns a port handle for use by other API functions. Call `MgslClose` to close the returned handle when it is no longer needed.

The format of the name depends on the hardware type:

Single Port Adapter	MGHDLCx, where x = adapter number Example: MGHDLC3
Multiple Port Adapter	MGMPxPy, where x = adapter number and y = port number Example: MGMP2P4

The port name can be obtained from the `MgslEnumeratePorts` call and from the Windows Device Manager.

```
HANDLE dev;

/* open port 3 of first adapter */
rc = MgslOpenByName(“MGMP1P3”, &dev);
if (rc != ERROR_SUCCESS)
      printf("MgslOpenByName error=%d\n", rc);
else
      /* dev contains valid device handle */
```

MgslOpenTraceHandle

```
ULONG MgslOpenTraceHandle(ULONG port_id, HANDLE *dev);
```

Arguments

port_id	port identifier
dev	pointer to returned port handle

Return Value

ERROR_SUCCESS	call success
ERROR_DEVICE_IN_USE	port is in use by another tracing application
ERROR_BAD_DEVICE	port_id does not exist or is not functioning
ERROR_ACCESS_DENIED	insufficient privilege to open a trace handle

This function opens a serial device identified by `port_id` and returns a port handle for use by other API functions. Call `MgslClose` to close the returned handle when it is no longer needed. The port ID can be obtained from MgslEnumeratePorts. This call requires Administrative privilege to succeed.

This call allows two processes to access a single port, one for normal use and the other for tracing. Normally only a single process can open a port. This function is used by the `mgsltrc.exe` tracing utility included with the Serial API. Source code for `mgsltrc.exe` is provided. The tracing API calls allow a custom serial application to integrate tracing into an application.

```
unsigned int port_id;
HANDLE dev;

/* operate on port 3 of first adapter */
port_id = MGSL_MAKE_PORT_ID(1,3);

rc = MgslOpenTraceHandle(port_id, &dev);
if (rc != ERROR_SUCCESS)
      printf("MgslOpenTraceHandle error=%d\n", rc);
else
      /* dev contains valid device handle */
```

MgslPutTraceEvent

```
ULONG MgslPutTraceEvent(HANDLE dev, MGSL_TRACE_EVENT *event);
```

Arguments	
dev	returned port handle
event	MGSL_TRACE_EVENT structure to add to trace buffer

Return Value	
ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	port handle is invalid
ERROR_INVALID_PARAMETER	event buffer is invalid

This function adds a trace event to a serial device's trace buffer. The event is described by an MGSL_TRACE_EVENT structure. The call fills out the EventType, DataLength and EventData fields. The API sets the TimeStamp field. See the documentation for the MGSL_TRACE_EVENT structure for a description of the fields and associated constants.

```
MGSL_TRACE_EVENT event;

/* initialize structure */

rc = MgslPutTraceEvent(dev, &event);
if (rc != ERROR_SUCCESS)
      /* process error */
```

MgslReceive

```
ULONG MgslReceive(HANDLE dev, MGSL_RECEIVE_REQUEST *req, OVERLAPPED *ol);
```

Arguments	
dev	open port handle
req	pointer to receive request structure
ol	pointer to Windows overlapped structure for asynchronous notification of request completion

Return Value	
ERROR_SUCCESS	receive data returned
ERROR_IO_PENDING	waiting for receive data (monitor overlapped structure)
ERROR_INVALID_HANDLE	port handle is invalid
ERROR_INVALID_PARAMETER	request buffer is invalid
ERROR_NOT_READY	receiver is disabled and no data is available

This function returns received data to the application. The receive request structure and overlapped structure must be initialized before each call. If data is available, ERROR_SUCCESS is returned. If data is not available and the receiver is enabled then ERROR_IO_PENDING is returned and the caller monitors the hEvent member of the overlapped structure for notification of request completion.

Call MgslCancelReceive to stop a pending call to MgslReceive. When the cancellation is complete, the hEvent member of the overlapped structure will be signaled.

This sample code demonstrates request preparation, call and processing. CreateEvent, ResetEvent and WaitForSingleObject are Windows system calls. Refer to Windows SDK documentation for details. The request and overlapped structures **must** remain valid (heap or stack allocation) until the request completes.

```
OVERLAPPED ol;
int buffer_size = 4096;
MGSL_RECEIVE_REQUEST *req;

ol.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
req = malloc(sizeof(MGSL_RECEIVE_REQUEST) + buffer_size);

/* preparation and call */
req->DataLength = buffer_size;
ResetEvent(ol.hEvent);
rc = MgslReceive(dev, req, &ol);
if (rc == ERROR_IO_PENDING) {
      rc = WaitForSingleObject(ol.hEvent, INFINITE);
      if (rc != WAIT_OBJECT_0)
            /* wait error */
} else if (rc != ERROR_SUCCESS)
      /* MgslReceive error */

/* process completed request */
if (req->Status == RxStatus_OK)
      /* req->DataLength set to count of data in req->DataBuffer */
else
      /* receive error (CRC etc), no data returned */
```

The amount of data available in the API buffers is obtained using MgslGetOption with MGSL_OPT_RX_COUNT. Refer to the MGSL_RECEIVE_REQUEST structure reference for detailed descriptions of the fields and values.

Raw, Bisync/Monosync Buffer Fill Level:

For raw, bisync and monosync modes, MgslReceive returns data when a driver receive buffer (256 bytes for PCI cards, 128 bytes for USB device) fills. At low data rates this may cause too much delay between receipt of a byte and that byte being returned to the application.

The application can reduce the number of bytes returned per call by setting the DataLength member of the MGSL_RECEIVE_REQUEST structure to the desired count. When the value of DataLength changes from the previous value, the receiver is reset discarding all data. Values over the default value (256 for PCI, 128 for USB) result in the default number of bytes returned per call.

The value also controls the data transfer mode used by hardware (PIO or DMA).

128 to 256	DMA mode, value MUST be a multiple of 4
1 to 127	PIO mode, any value in this range is valid

Use lower values as needed for lower data rates and lower latency. At high data rates PIO mode may cause data loss.

Polled Mode

Use MgslSetOption with MGSL_OPT_RX_POLL to enable polled mode. When enabled, MgslReceive returns ERROR_BUSY and cancels the request instead of returning ERROR_IO_PENDING if no data is available to return. The application must still allocate an overlapped structure and event even when using polled mode.

MGSLREAD

```
int MgslRead(HANDLE dev, unsigned char *buf, int size);
```

Arguments

dev	open port handle
buf	pointer to buffer to hold receive data
size	size of buffer in bytes

Return Value
Number of bytes returned in buffer, or zero if timeout or error.

MgslRead returns received data to the application when available. MgslRead is a simplified wrapper function for the MgslReceive. If you do not need the error reporting and asynchronous notification features of MgslReceive, use the simpler MgslRead.

```
unsigned char buf[4096];
int size = sizeof(buf);
int count;

/* get receive data */
count = MgslRead(dev, buf, size);
if (count) {
	/* count bytes returned in buf */
} else {
	/* no data available (polled) or error (blocking) */
}
```

The amount of data available in the API buffers is obtained using MgslGetOption with MGSL_OPT_RX_COUNT.

MgslRead behavior depends on the serial protocol.

HDLC/SDLC
Each buffer returned by the API is one frame of variable size. Set size argument to largest expected frame size (typically 4K). Actual frame size is reported by return code.

Monosync/Bisync/Raw/Asynchronous/Isosynchronous
Each buffer returned by the API is a fixed size block of data, set by the size argument, and does not imply any message boundaries. In these modes the API does not detect message boundaries, which is the responsibility of the application. Data is not returned until size bytes are received.

For best performance set size to 256 to match the default hardware data transfer size. At low data rates this may cause too much delay (latency) between receipt of a byte and that byte being returned to the application. Use a lower value of size to reduce latency. When the size argument changes from the previous value, the receiver is reset, discarding all data and the new value is used. The value also controls the data transfer mode used by hardware (PIO or DMA/Packet).

128 to 256	DMA/Packet mode (MUST be a multiple of 4)
1 to 127	PIO mode, may cause data loss at high speed

Blocking and Polled Modes

`MgslRead` can be used in blocking or polled mode by calling `MgslSetOption` with the `MGSL_OPT_RX_POLL` identifier. When this option is enabled, `MgslRead` returns immediately with a return code of zero when no data is available (polling mode). When not enabled, `MgslRead` blocks until data is available (blocking mode = default).

A blocked call to `MgslRead` can be aborted by calling `MgslCancelReceive` from a different application thread.

MgslReadWithStatus

```
int MgslReadWithStatus(HANDLE dev, unsigned char *buf,
      int size, int *status);
```

Arguments

dev	open port handle
buf	pointer to buffer to hold receive data
size	size of buffer in bytes
status	buffer to hold returned status

Return Value

Number of bytes returned in buffer, or zero if error.

`MgslReadWithStatus` returns received data and/or receive error indications to the application when available. `MgslReadWithStatus` is similar to `MgslRead` except receive errors can be returned in addition to data. Inspect the returned status value to determine if data is valid or an error was encountered.

```
unsigned char buf[4096];
int size = sizeof(buf);
int count;
int status;

/* get receive data or error indication */
count = MgslReadWithStatus(dev, buf, size, &status);
if (status == RxStatus_OK) {
      // count bytes of valid data returned
} else if (status == RxStatus_CrcError) {
      // CRC error (HDLC only)
} else if (status == RxStatus_Abort) {
      // receive idle or abort pattern (HDLC only)
} else if (status == RxStatus_ShortFrame) {
      // malformed/short frame (HDLC only)
} else if (status == RxStatus_Cancel) {
      // polled mode = no data or indication available
      // blocked mode = application canceled blocked call
}
```

Refer to MgslRead documentation in the previous section for more details. MgslRead is the preferred call unless an application processes receive errors or must inspect received CRC values. Refer to the receive-status sample program that is part of the HDLC sample project for more information on processing receive errors and inspecting received CRC values.

MgslResetTraceBuffers

```
ULONG MgslResetTraceBuffers(HANDLE dev);
```

Arguments

dev	returned port handle

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	port handle is invalid

This function discards all data in the API trace buffers for a serial device.

```
rc = MgslResetTraceBuffers(dev);
if (rc != ERROR_SUCCESS)
      /* call error */
```

MgslSetIdleMode

```
ULONG MgslSetIdleMode(HANDLE dev, ULONG idle);
```

Arguments

dev	returned port handle
idle	idle or sync pattern

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	port handle is invalid

This function sets the idle or sync pattern. In bisync mode, this sets a 16-bit sync pattern. In monosync mode, this sets an 8-bit sync pattern. In other modes, this sets a pattern transmitted when there is no data to send.

HDLC_TXIDLE_FLAG	8-bit flag pattern (0x7e)
HDLC_TXIDLE_ALT_ZEROS_ONES	8-bit alternating zero and one pattern (0xaa)
HDLC_TXIDLE_ZEROS	8-bit all zero pattern (0x00)
HDLC_TXIDLE_ONES	8-bit all one pattern (0xff), also called HDLC abort
HDLC_TXIDLE_CUSTOM8	arbitrary 8-bit pattern in bits [7:0]
HDLC_TXIDLE_CUSTOM16	arbitrary 16-bit pattern in bits [15:0]

The following code demonstrates the call.

```
unsigned char syn1 = 0x32;
unsigned char syn2 = 0x32;

/* set 16-bit sync pattern for bisync mode */
rc = MgslSetIdleMode(dev, HDLC_TXIDLE_CUSTOM_16 | (syn2 << 8) | syn1);
if (rc != ERROR_SUCCESS)
      /* call error */

/* set 8-bit sync pattern for monosync mode */
rc = MgslSetIdleMode(dev, HDLC_TXIDLE_CUSTOM_8 | syn1);
if (rc != ERROR_SUCCESS)
      /* call error */

/* set flag idle for HDLC mode */
rc = MgslSetIdleMode(dev, HDLC_TXIDLE_FLAGS);
if (rc != ERROR_SUCCESS)
      /* call error */
```

MgslSetGpio

```
ULONG MgslSetGpio(HANDLE dev, GPIO_DESC *gpio);
```

Arguments

dev	returned port handle
gpio	pointer to GPIO structure

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	port handle is invalid
ERROR_INVALID_PARAMETER	invalid GPIO structure

This function sets general purpose I/O (GPIO) signal directions and states as described in a `GPIO_DESC` structure. Each bit of the structure fields represent a single signal, with GPIO #0 located in bit 0, GPIO #1 located in bit 1, etc. The number of GPIO signals depends on the specific hardware. Refer to the hardware user's guide for your hardware for a description of available GPIO signals.

```
GPIO_DESC gpio;
ULONG rc;

memset(&gpio, 0, sizeof(gpio));
gpio.dmask = (1 << 4); /* set direction of GPIO[4] */
gpio.dir   = (1 << 4); /* GPIO[4] set to output */
gpio.smask = (1 << 4); /* set state of GPIO[4] */
gpio.dir   = (1 << 4); /* GPIO[4] set high */

rc = MgslSetGpio(dev, &gpio);
if (rc != ERROR_SUCCESS)
      /* call error */
```

MgslSetOption

```
ULONG MgslSetOption(HANDLE dev, UINT option, UINT value);
```

Arguments

`dev`	handle to open device
`option`	option identifier
`value`	new option value

Return Value

`ERROR_SUCCESS`	call success
`ERROR_INVALID_HANDLE`	invalid device handle
`ERROR_INVALID_PARAMETER`	invalid option identifier or value

This function sets the specified configuration value. These settings take effect at application run time. Some of these settings can be set at driver load time (system boot) with `MgslSetPortConfigEx`. Settings that have an equivalent `MgslSetPortConfigEx` value are noted below.

Most `MgslSetOption` settings take effect immediately, some are deferred until the next call to `MgslSetParams`. Deferred options are identified in the descriptions below. Unless otherwise specified, the option takes effect immediately.

The following is a list of option identifiers.

MGSL_OPT_AUXCLK_ENABLE	Select AUXCLK output function. 1 = clock output (default), BRG enabled and used as clock source 0 = static low, BRG enabled for internal clock 2 = static high, BRG disabled (not compatible if internal clock required)
MGSL_OPT_CLOCK_BASE_FREQ	Set base block frequency. Use only with hardware ordered with a custom base clock or hardware with a frequency synthesizer programmed for a custom base clock (GT2e/GT4e/USB). This option takes effect immediately, but should be set BEFORE calling `MgslSetParams` so generated clocks are calculated correctly.
MGSL_OPT_DPLL_RESET	Controls DPLL reset behavior. 0 = no automatic DPLL reset 1 = automatically reset DPLL when enabling receiver 2 = one time manual DPLL reset
MGSL_OPT_ENABLE_LOCALLOOPBACK	Local Loopback output state. 0=off, 1=on, default=off
MGSL_OPT_ENABLE_REMOTELOOPBACK	Remote Loopback output state. 0=off, 1=on, default=off
MGSL_OPT_HALF_DUPLEX	0=disabled, 1=enabled, default=disabled When enabled, RS422/485 outputs (TxD,AUXCLK,RTS,DTR) are active when sending data, otherwise outputs are tri-stated (high impedance). When sending data, the receiver input is ignored.
MGSL_OPT_INTERFACE	Select interface type (RS232, V.35, RS422) with MGSL_INTERFACE_XXX macros. Only valid for USB or SyncLink PCIe hardware. Set this option at driver load time with `MgslSetPortConfigEx`.
MGSL_OPT_MSB_FIRST	Serial bit order. 0=LSB first, 1=MSB first, default = LSB first This option does not take effect until the next call to MgslSetParams.

MGSL_OPT_NO_TERMINATION
Disable serial input termination for differential interface modes (RS-422/RS-485/V.35/RS-530A/X.21) Only valid for USB or SyncLink PCIe hardware. GT series cards use jumpers and DIP switches to enable/disable termination.
Set this option at driver load time with `MgslSetPortConfigEx`.

MGSL_OPT_RTS_DRIVER_CONTROL
0=disabled, 1=enabled, default=disabled
When enabled, RTS signal state controls output drivers.
RTS on = outputs active
RTS off = outputs tri-stated (high impedance)
Set this option at driver load time with `MgslSetPortConfigEx`.

MGSL_OPT_RS422_OE
SyncLink USB/PCIe Only: 8 bit value controls RS422 output enable behavior and state for four serial signal outputs.
[7] TXD OE Select (0=auto, 1=manual)
[6] AUXCLK OE Select (0=auto, 1=manual)
[5] DTR OE Select (0=auto, 1=manual)
[4] RTS OE Select (0=auto, 1=manual)
[3] TXD OE Manual State (0=tristate/disabled, 1=enabled)
[2] AUXCLK OE Manual State (0=tristate/disabled, 1=enabled)
[1] DTR OE Manual State (0=tristate/disabled, 1=enabled)
[0] AUXCLK OE Manual State (0=tristate/disabled, 1=enabled).
Select automatic or manual behavior. Automatic behavior is always enabled unless half duplex or RTS control selected. Manual behavior sets output enable state to that selected by manual state bits.

MGSL_OPT_RX_DISCARD_TOO_LARGE
0=disabled, 1=enabled, default = disabled
Silently discard receive frames larger than MgslReceive buffer.

MGSL_OPT_RX_ERROR_MASK
0=disabled, 1=enabled, default=disabled
Silently discard HDLC receive frames with errors (CRC, etc).

MGSL_OPT_RX_COUNT
Read only value indicates number of bytes in receive buffers.

MGSL_OPT_TX_COUNT
Read only value indicates number of bytes in send buffers.

MGSL_OPT_RX_POLL
Enable polling mode for MgslReceive calls. When enabled, MgslReceive returns ERROR_BUSY and cancels the request instead of ERROR_IO_PENDING if no data is available to return.

MGSL_OPT_TX_POLL
Enable polling mode for MgslTransmit calls. When enabled, MgslTransmit returns ERROR_BUSY and cancels the request instead of ERROR_IO_PENDING if no send buffers are free to hold data.

MGSL_OPT_TX_IDLE_COUNT
Statistic value incremented when transmitter becomes idle after sending data. Clear the count by calling MgslSetOption with value of 0.

MGSL_OPT_UNDERRUN_COUNT
Statistic value incremented for each transmitter underrun. Clear the count by calling MgslSetOption with value of 0.

```
/* set RTS to control serial output state (on or tristate) */
rc = MgslSetOption(dev, MGSL_OPT_RTS_DRIVER_CONTROL, 1);
if (rc != ERROR_SUCCESS)
            printf("MgslSetOption error=%d\n", rc);
```

MgslSetParams

```
ULONG MgslSetParams(HANDLE dev, MGSL_PARAMS *params);
```

Arguments

dev	handle to open device
params	pointer to configuration structure

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	invalid device handle
ERROR_INVALID_PARAMETER	invalid option identifier or value

This function sets the specified configuration values. The options are usually set once per communications session. Calling this function resets the transmitter and receiver, discarding all buffered data. Refer to the MGSL_PARAMS structure section for a description of the fields and values.

```
MGSL_PARAMS params;

/* initialize structure with desired settings */

rc = MgslSetParams(dev, &params);
if (rc != ERROR_SUCCESS)
      printf("MgslSetParams error=%d\n", rc);
```

MgslSetPortConfigEx

```
ULONG MgslSetPortConfigEx(ULONG port_id, MGSL_PORT_CONFIG_EX *config);
```

Arguments

port_id	integer port identifier
config	pointer to configuration structure

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	invalid device handle
ERROR_INVALID_PARAMETER	invalid option identifier or value

This function sets the specified configuration values that are used at driver load time. Refer to the MGSL_PORT_CONFIG_EX structure section for a description of the fields and values. These options are usually set in the Windows Device Manager, but this call is implemented so the options can be set programmatically.

```
MGSL_PORT_CONFIG_EX config;
unsigned int port_id;

/* operate on port 3 of first adapter */
port_id = MGSL_MAKE_PORT_ID(1,3);

/* initialize structure with desired settings */

rc = MgslSetPortConfigEx(port_id, &config);
if (rc != ERROR_SUCCESS)
      printf("MgslSetPortConfigEx error=%d\n", rc);
```

MgslSetSerialSignals

```
ULONG MgslSetSerialSignals(HANDLE dev, UCHAR *signals);
```

Arguments

dev	handle to open device
signals	new serial signals value

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	invalid device handle

This function sets the serial control output states using SerialSignal_xxx macros defined in the mghdlc.h header file. A set bit indicates an active signal. Depending on the application, some signals may not be used or may be used for a non-standard purpose.

SerialSignal_DTR	Data Terminal Ready output (serial device is active)
SerialSignal_RTS	Request to Send output (serial device wants to send data)

```
/* turn on both DTR and RTS output signals */
rc = MgslSetSerialSignals(dev, SerialSignal_RTS + SerialSignal_DTR);
if (rc != ERROR_SUCCESS)
      printf("MgslSetSerialSignals error=%d\n", rc);
```

MgslSetTraceLevel

```
ULONG MgslSetTraceLevel(HANDLE dev, ULONG *level);
```

Arguments

dev	handle to open device
level	new trace level value

Return Value

ERROR_SUCCESS	call success
ERROR_INVALID_HANDLE	invalid device handle

This function sets the current trace level for a serial device using TraceLevel_XXX macros. Most applications do not use this call. The mgsltrc.exe trace utility provided with the Serial API uses this call.

TraceLevel_API	API calls recorded
TraceLevel_Status	serial input (CTS, DCD, DSR, RI) events recorded
TraceLevel_Transmit	transmit events (underrun, completion) recorded
TraceLevel_Receive	receive events (overrun, completion, idle) recorded
TraceLevel_Data	send and receive data contents recorded
TraceLevel_DataLink	high level link layer events (SNRM, RR, etc) recorded
TraceLevel_Error	driver error conditions recorded
TraceLevel_Info	driver general events recorded
TraceLevel_Detail	driver detailed operation recorded

MgslTransmit

```
ULONG MgslTransmit(HANDLE dev, UCHAR *buf, ULONG size, ULONG *status, OVERLAPPED *ol);
```

Arguments

dev	handle to open device
buf	send data buffer
size	number of bytes in send data buffer (may be zero, see below)
status	returned send status (may be NULL, see below)
ol	pointer to Windows overlapped structure used for asynchronous notification of send completion

Return Value

ERROR_SUCCESS	call success
ERROR_IO_PENDING	waiting for call to complete (monitor overlapped structure)
ERROR_BUSY	previous MgslTransmit call is pending
ERROR_INVALID_HANDLE	invalid device handle

This function performs one of three functions depending on the arguments:

If status is NULL, call completes after buffering data to send. This is the typical case.
If status is not NULL, call completes after sending data and status indicates success or error.
If size is zero, call completes when previously buffered data is sent.

ERROR_SUCCESS indicates immediate completion and ERROR_IO_PENDING indicates the application should monitor the hEvent member of the overlapped structure for indication of call completion. Call MgslCancelTransmit to cancel a pending call before normal completion. When the cancellation completes, the event member of the overlapped structure is signaled.

If size is not zero, this call enables the transmitter. The transmitter remains enabled sending the idle pattern after all data is sent.

These values are returned if the status argument is not NULL. Values other than TxStatus_OK indicate data was not sent. Send status is rarely required by an application. Requiring status for each call prevents a continuous flow of data and reduces throughput.

TxStatus_OK	data sent successfully
TxStatus_Underrun	hardware not supplied data fast enough
TxStatus_Cancel	user cancelled request before data sent
TxStatus_CtsFailure	CTS went inactive in middle of sending data

Call MgslGetOption with MGSL_OPT_TX_COUNT to get the number of buffered send bytes. This is useful for maintaining a continuous flow of data while limiting output latency by controlling the amount of buffered data.

Below is sample code demonstrating MgslTransmit.

```
int rc;
int size = 1024;
unsigned char buf[1024];
OVERLAPPED ol;

ol.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

/* submit buffer to send, do not wait for send completion */
ResetEvent(ol.hEvent);
rc = MgslTransmit(dev, buf, size, NULL, &ol);
if (rc == ERROR_IO_PENDING) {
       /* wait for completion (wait for free buffer) */
       rc = WaitForSingleObject(ol.hEvent, INFINITE);
       if (rc != WAIT_OBJECT_0)
              /* wait error */
} else if (rc != ERROR_SUCCESS) {
       /* MgslTransmit error */
} else {
       /* immediate completion */
}

/* wait for previously submitted data to be sent */
ResetEvent(ol.hEvent);
rc = MgslTransmit(dev, NULL, 0, NULL, &ol);
if (rc == ERROR_IO_PENDING) {
       /* wait for completion (wait for all sent) */
       rc = WaitForSingleObject(ol.hEvent, INFINITE);
       if (rc != WAIT_OBJECT_0)
              /* wait error */
} else if (rc != ERROR_SUCCESS) {
       /* MgslTransmit error */
} else {
       /* immediate completion */
}
```

Polled Mode

Use MgslSetOption with MGSL_OPT_TX_POLL to enable polled mode. When enabled, MgslTransmit returns ERROR_BUSY and cancels the request instead of returning ERROR_IO_PENDING if no free buffers are available to hold data. The application must still allocate an overlapped structure and event even when using polled mode.

MgslWaitEvent

```
ULONG MgslWaitEvent(HANDLE dev, ULONG mask, ULONG *events, OVERLAPPED *ol);
```

Arguments

dev	handle to open device
mask	bit flags indicating events of interest
events	returned event bit flags
ol	pointer to Windows overlapped structure used for asynchronous notification of request completion

Return Value

ERROR_SUCCESS	call success
ERROR_IO_PENDING	waiting for call to complete (monitor overlapped structure)
ERROR_BUSY	a previous MgslWaitEvent call is still pending
ERROR_INVALID_HANDLE	invalid device handle
ERROR_INVALID_PARAMETER	invalid events buffer

This function waits for one or more events specified in the mask argument to occur. When the first specified event occurs, the call completes with the events argument set to reflect which events occurred.

The return code ERROR_SUCCESS indicates immediate completion and ERROR_IO_PENDING indicates the application should monitor the event member of the overlapped structure for indication of call completion.

Call MgslCancelWaitEvent to stop a pending call to MgslWaitEvent. When the cancellation is complete, the event member of the overlapped structure will be signaled.

Both mask and events use the following bit flags defined in the mghdlc.h header file:

MgslEvent_DsrActive	wait for DSR (Data Set Ready) input active
MgslEvent_DsrInactive	wait for DSR (Data Set Ready) input inactive
MgslEvent_CtsActive	wait for CTS (Clear to Send) input active
MgslEvent_CtsInactive	wait for CTS (Clear to Send) input inactive
MgslEvent_DcdActive	wait for DCD (Data Carrier Detect) input active
MgslEvent_DcdInactive	wait for DCD (Data Carrier Detect) input inactive
MgslEvent_RiActive	wait for RI (Ring Indicator) input active
MgslEvent_RiInactive	wait for RI (Ring Indicator) input inactive
MgslEvent_ExitHuntMode	wait for receiver to become active (sync/flag detect)
MgslEvent_IdleReceived	wait for receiver to become idle (idle detect)

MgslWaitGpio

```
ULONG MgslWaitGpio(HANDLE dev, GPIO_DESC *gpio, OVERLAPPED *ol);
```

Arguments

dev	handle to open device
gpio	pointer to GPIO structure
ol	pointer to Windows overlapped structure used for asynchronous notification of request completion

Return Value

ERROR_SUCCESS	call success
ERROR_IO_PENDING	waiting for call to complete (monitor overlapped structure)
ERROR_BUSY	a previous MgslWaitGpio call is still pending
ERROR_INVALID_HANDLE	invalid device handle
ERROR_INVALID_PARAMETER	invalid events buffer

This function waits for one or more GPIO signals to reach the specified states. When the first specified state occurs, the call completes with the `gpio` argument set to reflect the current GPIO signal states. The `dmask` and `dir` fields of the `GPIO_DESC` structure are ignored.

Before making this call, set the bits in the `smask` field of the `GPIO_DESC` structure to indicate which GPIO signals to monitor. Set bits in the `state` field to specify the target state.

When the call completes, the `state` member of the `GPIO_DESC` structure is set to indicate the state of all signals at the time the first target state is reached.

The return code `ERROR_SUCCESS` indicates immediate completion and `ERROR_IO_PENDING` indicates the application should monitor the event member of the overlapped structure for indication of call completion.

Call `MgslCancelWaitGpio` to stop a pending call to `MgslWaitGpio`. When the cancellation is complete, the event member of the overlapped structure will be signaled.

MgslWaitAllSent

```
int MgslWaitAllSent(HANDLE dev);
```

Arguments

dev	open port handle

Return Value

Zero if success (all data sent) or error code (timeout or other error).

Check data passed to `MgslWrite` or `MgslTransmit` has been completely sent. Use this call to determine when it is safe to reset the transmitter or close the port.

```
int rc;

/* check for all buffered data sent */
rc = MgslWaitAllSent(dev);
if (!rc) {
      /* all buffered data sent, safe to close port */
} else {
      /* busy sending buffered data (polled) or error (blocking) */
}
```

MgslWaitAllSent is a simplified wrapper function for the original MgslTransmit function. If you do not need the extra error reporting and asynchronous notification features of MgslTransmit, use the simpler MgslWaitAllSent.

Blocking and Polled Modes

MgslWaitAllSent can be used in blocking or polled mode by calling MgslSetOption with the MGSL_OPT_TX_POLL identifier. When this option is enabled, MgslWaitAllSent returns immediately with a return code of zero when all data is sent or non zero if still sending (polled mode). When not enabled, MgslWaitAllSent blocks until all buffered data has been sent (blocking mode = default).

A blocked call to MgslWaitAllSent can be aborted by calling MgslCancelTransmit from a different application thread.

```
int rc;

/* submit send data to API with MgslTransmit or MgslWrite */

/* block until all data is completely sent (INFINITE = no timeout) */
rc = MgslWaitAllSent(dev, INFINITE);
if (!rc) {
      /* all data sent */
} else {
      /* timeout or error */
}

/* safe to close port or wait for response */
```

The amount of queued send data in the API buffers is obtained using MgslGetOption with MGSL_OPT_TX_COUNT.

MgslWrite

```
int MgslWrite(HANDLE dev, unsigned char *buf, int size);
```

Arguments

dev	open port handle
buf	pointer to buffer containing send data
size	size of send data in bytes

Return Value

Number of bytes accepted by API or zero if buffers are full (polled) or error (blocking).

MgslWrite is a simplified wrapper function for MgslTransmit which passes data to the API to send as soon as possible.

```
unsigned char buf[4096];
int size = sizeof(buf);
int count;

/* initialize buffer with application data */

/* send data */
count = MgslWrite(dev, buf, size);
if (count) {
      /* count bytes stored in API send buffers */
} else {
      /* API send buffers full (polled) or error (blocked) */
}
```

The amount of queued send data in the API buffers is obtained using MgslGetOption with MGSL_OPT_TX_COUNT.

A blocked call to MgslWrite can be aborted by calling MgslCancelTransmit from a different application thread.

The format of the data depends on the protocol. For frame oriented SDLC/HDLC, each call sends a single frame of data. For other protocols, each call sends data with no formatting, requiring the application to implement message boundaries.

Blocking and Polled Modes

MgslWrite can be used in blocking or polled mode by calling MgslSetOption with the MGSL_OPT_TX_POLL identifier. When this option is enabled, MgslWrite returns immediately with a return code of zero when all API send buffers are full (polling mode). When not enabled, MgslWrite blocks until API send buffers are available and data is accepted (blocking mode = default).

A blocked call to MgslWrite can be aborted by calling MgslCancelTransmit from a different application thread.

STRUCTURES

The following C language structures are defined in the `mghdlc.h` header file. This header should be included in source files that access the serial API.

GPIO_DESC

This structure is used with the general purpose I/O (GPIO) API calls `MgslGetGpio`, `MgslSetGpio`, `MgslWaitGpio`.

```
typedef struct _GPIO_DESC
{
	UINT state;
	UINT smask;
	UINT dir;
	UINT dmask;

} GPIO_DESC;
```

All fields are 32 bits. Each bit represents an I/O signal. I/O signal 0 is bit 0, signal 1 is bit 1, etc.

`state`	each bit represents the state of an I/O signal
`smask`	specifies which bits in state field are used
`dir`	each bit specifies the direction of an I/O signal (0=input, 1=output)
`dmask`	specifies which bits in dir field are used

MGSL_ASSIGNED_RESOURCES

This structure is used with the API call `MgslGetAssignedResources`. Only the `SerialNumber` field is used. All other fields are unused and undefined. The serial number is only available for SyncLink USB devices.

```
typedef struct _MGSL_ASSIGNED_RESOURCES
{
   ULONG BusType;
   ULONG BusNumber;
   ULONG DeviceNumber;
   ULONG IrqLevel;
   ULONG DmaChannel;
   ULONG IoAddress1;
   ULONG IoAddress2;
   ULONG IoAddress3;
   ULONG MemAddress1;
   ULONG MemAddress2;
   ULONG MemAddress3;
   USHORT DeviceId;
   USHORT SubsystemId;
   char SerialNumber[MGSL_MAX_SERIAL_NUMBER];

} MGSL_ASSIGNED_RESOURCES, *PMGSL_ASSIGNED_RESOURCES;
```

MGSL_PARAMS

This structure is used with the configuration API calls MgslGetParams and MgslSetParams.

```
typedef struct _MGSL_PARAMS
{
   /* common */

   ULONG     Mode;         /* HDLC, asynchronous, raw, bisync, monosync */
   UCHAR     Loopback;     /* internal loopback mode */
   USHORT    Flags;        /* bit field flags */

   /* synchronous modes */

   UCHAR     Encoding;     /* serial encoding NRZ, NRZI, etc. */
   ULONG     ClockSpeed;   /* external clock speed in bits per second */
   UCHAR     Addr;         /* rx HDLC address filter, 0xFF = disable */
   USHORT    CrcType;      /* None, CRC16-CCITT, or CRC32-CCITT */
   UCHAR     PreambleLength;
   UCHAR     PreamblePattern;

   /* asynchronous mode */

   ULONG     DataRate;     /* async data rate in bits per second */
   UCHAR     DataBits;     /* 5 to 8 */
   UCHAR     StopBits;     /* 1 or 2 */
   UCHAR     Parity;       /* none, even, odd */

} MGSL_PARAMS, *PMGSL_PARAMS;
```

Note: Many constants are defined as HDLC_XXX for historical reasons, but apply to all synchronous modes (raw/bisync/monosync/HDLC).

Mode

The mode field of the MGSL_PARAMS structure specifies the communications protocol with an MGSL_MODE_XXX macro. The mode determines the framing, synchronization, transparency and clocking characteristics of the serial protocol.

MGSL_MODE_ASYNC
character oriented
no external clocks
per character hardware framing
per character parity check (none/even/odd)
used for isochronous mode when DataRate set to 0

MGSL_MODE_HDLC
bit synchronous
hardware framing and synchronization (flags)
hardware transparency (0 bit stuff/removal)
hardware CRC check/generation (none/16 bit/32 bit)
Note: SDLC and HDLC are the same in this context

MGSL_MODE_RAW	bit synchronous no hardware framing no hardware synchronization no hardware transparency
MGSL_MODE_BISYNC	byte synchronous 16 bit hardware synchronization no hardware framing no hardware transparency
MGSL_MODE_MONOSYNC	byte synchronous 8 bit hardware synchronization no hardware framing no hardware transparency
MGSL_MODE_TDM	SyncLink PCIe, GT4e, USB Only byte synchronous external control signal synchronization external clock signal framing defined by fixed size data grouping

Loopback

The Loopback field of the MGSL_PARAMS structure enables or disables the internal loopback mode. 0 = normal operation, 1 = loopback mode. When enabled, the transmit data signal is connected internally to the receive data signal and clocks are generated internally by the baud rate generator as specified by the ClockSpeed field. Use internal loopback mode to test the operation of the serial controller without external line drivers or devices.

Flags

The Flags field of the MGSL_PARAMS structure specifies options using HDLC_FLAG_XXX macros. The HDLC_FLAG prefix is used for historical reasons. Unless otherwise specified flags apply to all modes.

Receive Clock Source

These flags select the serial receive clock source.

HDLC_FLAG_RXC_DPLL Recover receive clock from receive data signal. Specify data rate in ClockSpeed member of MGSL_PARAMS structure.

HDLC_FLAG_RXC_BRG Generate receive clock internally. Specify data rate in ClockSpeed member of MGSL_PARAMS structure.

HDLC_FLAG_RXC_RXCPIN Receive clock is supplied by external device on RxC input pin.

HDLC_FLAG_RXC_TXCPIN Receive clock is supplied by external device on TxC input pin.

HDLC_FLAG_RXC_INV Combine with receive clock source flag to invert receive clock polarity.

Transmit Clock Source

These flags select the serial transmit clock source.

`HDLC_FLAG_TXC_DPLL` Recover transmit clock from receive data signal. Specify data rate in `ClockSpeed` member of `MGSL_PARAMS` structure.

`HDLC_FLAG_TXC_BRG` Generate transmit clock internally. Specify data rate in `ClockSpeed` member of `MGSL_PARAMS` structure.

`HDLC_FLAG_TXC_RXCPIN` Transmit clock is supplied by external device on RxC input pin.

`HDLC_FLAG_TXC_TXCPIN` Transmit clock is supplied by external device on TxC input pin.

`HDLC_FLAG_TXC_INV` Combine with transmit clock source flag to invert transmit clock polarity.

Digital Phase Lock Loop Divisor

The DPLL is used to recover a clock from the receive data signal. This is done using a sample/reference clock that is a multiple of the expected data rate. This multiple is either 8 or 16. A higher sample rate (larger divisor) results in a more accurate recovered clock. A lower divisor allows a higher expected data rate. The sample clock is limited by the base clock frequency (default 14.7456MHz).

`HDLC_FLAG_DPLL_DIV8` data rate = reference clock divided by 8

`HDLC_FLAG_DPLL_DIV16` data rate = reference clock divided by 16

Miscellaneous Flags (any combination allowed)

`HDLC_FLAG_AUTO_CTS` Enable transmitter only when CTS input is active.

`HDLC_FLAG_AUTO_DCD` Enable receiver only when DCD input is active.

`HDLC_FLAG_AUTO_RTS` When set, the driver automatically asserts RTS at when sending data and negates RTS when done sending. If RTS is active when a transmit request is made, the driver will not manipulate the state of RTS.

Encoding

Encoding specifies physical data signal representation of logical 1 or 0. Equivalent encoding names are shown in parenthesis. Use BIPHASE encodings with clock recovery to guarantee one data transition per bit. The levels specified below are the data signal from the serial controller, but before the line drivers which invert the signal.

`HDLC_ENCODING_NRZ`	NRZ (NRZ-L) unencoded data signal. high = 1, low = 0
`HDLC_ENCODING_NRZB`	NRZB inverted data signal. high = 0, low = 1
`HDLC_ENCODING_NRZI_MARK`	(NRZ-M) invert at start of bit if 1
`HDLC_ENCODING_NRZI_SPACE`	(NRZI or NRZ-S) invert at start of bit if 0
`HDLC_ENCODING_BIPHASE_MARK`	(FM1) invert at start of bit, invert at bit center if 1
`HDLC_ENCODING_BIPHASE_SPACE`	(FM0) invert at start of bit, invert at bit center if 0
`HDLC_ENCODING_BIPHASE_LEVEL`	(Manchester) start of bit: high=1, low=0, invert at bit center
`HDLC_ENCODING_DIFF_BIPHASE_LEVEL`	invert at start of bit if 1, invert at bit center

ClockSpeed

The `ClockSpeed` field of the `MGSL_PARAMS` structure specifies the data rate of the generated (BRG) or recovered (DPLL) clock. A clock generated by the BRG is output on the AUXCLK output pin for use by an external device. Set to zero to disable clock generation.

The clock is generated by dividing a fixed base clock by a 16-bit integer divisor:

divisor = (base clock/data rate) - 1

Only discrete rates can be generated exactly because the divisor is a 16-bit integer. The default base clock of 14.7456MHz allows exact generation of common rates: 9600, 57600, 115200 etc.

The serial card can be purchased with a different base clock. This option is installed at the factory. When a base clock other than 14.7456MHz is installed, the driver must be configured to use the new value by calling `MgslSetOption(MGSL_OPT_CLOCK_BASE_FREQ)`.

Addr

8-bit address field that filters received SDLC/HDLC frames. 0xFF = return all frames (no filtering), otherwise discard received HDLC frames with addresses other than 0xFF (broadcast) or `Addr` value.

CrcType

The `CrcType` member of the `MGSL_PARAMS` structure specified the frame check type used with SDLC/HDLC frames. The selected Cyclic Redundancy Check (CRC) code is appended to sent frames and verified on receive frames.

HDLC_CRC_NONE	Don't send CRCs on transmit, don't check CRCs on receive.
HDLC_CRC_16_CCITT	16 bit CRC Polynomial: $x^{16} + x^{12} + x^5 + 1$
HDLC_CRC_32_CCITT	32 bit CRC Polynomial: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

HDLC_CRC_RETURN_CRC combine with HDLC_CRC_16_CCITT or HDLC_CRC_32_CCITT to return CRC value (2 or 4 bytes) to application as the final bytes in the receive frame buffer

HDLC_CRC_RETURN_CRCERR_FRAME combine with HDLC_CRC_16_CCITT or HDLC_CRC_32_CCITT to return received frames with CRC errors to application

CRC Errors and Accessing CRC Codes

By default HDLC frames with CRC errors are discarded. The API can be configured to return both good frames and frames with CRC errors to the application. Frames with CRC errors are identified by the status code in the receive request structure.

Add the HDLC_CRC_RETURN_CRCERR_FRAME flag to the CRC type in the `MGSL_PARAMS` structure to receive both good and bad frames as shown below.

```
params.CrcType = HDLC_CRC_16_CCITT | HDLC_CRC_RETURN_CRCERR_FRAME;
```

By default the API strips the CRC value from a received frame and only passes back the data. Add the HDLC_CRC_RETURN_CRC flag to the CRC type in the `MGSL_PARAMS` structure to append the 2 or 4 byte CRC value to the end of the frame passed back to the application.

```
params.CrcType = HDLC_CRC_16_CCITT | HDLC_CRC_RETURN_CRC;
```

These two CRC flags can be combined or used independently to achieve the desired behavior. The following returns all frames, regardless of CRC errors, and appends the CRC value to the end of the frame buffer.

```
params.CrcType = HDLC_CRC_16_CCITT | HDLC_CRC_RETURN_CRCERR_FRAME |
      HDLC_CRC_RETURN_CRC;
```

PreamblePattern

A preamble is a pattern sent before an SDLC/HDLC frame. The usual purpose of a preamble is to synchronize a remote DPLL that is recovering clock information from a data stream. The length of the preamble is specified in the `PreambleLength` member of the `MGSL_PARAMS` structure.

`HDLC_PREAMBLE_PATTERN_NONE`	no preamble (preamble_length ignored)
`HDLC_PREAMBLE_PATTERN_ZEROS`	all zeroes
`HDLC_PREAMBLE_PATTERN_FLAGS`	all flags
`HDLC_PREAMBLE_PATTERN_10`	alternating 1's and 0's
`HDLC_PREAMBLE_PATTERN_01`	alternating 0's and 1's
`HDLC_PREAMBLE_PATTERN_ONES`	all ones

PreambleLength

This member of the `MGSL_PARAMS` structure selects the length in bytes of the preamble pattern using the following macros:

```
HDLC_PREAMBLE_LENGTH_8BITS
HDLC_PREAMBLE_LENGTH_16BITS
HDLC_PREAMBLE_LENGTH_32BITS
HDLC_PREAMBLE_LENGTH_64BITS
```

DataRate

This member of the `MGSL_PARAMS` structure selects the data rate for asynchronous operation. In asynchronous mode the clock is generated internally by the BRG at x8 or x16 the data rate specified in this field. The data rate must match that of the remote device.

DataBits

This member of the `MGSL_PARAMS` structure selects the number of data bits per character for asynchronous operation. This value must match that of the remote device. Valid values are 5 to 8.

StopBits

This member of the `MGSL_PARAMS` structure selects the number of stop bits transmitted per character for asynchronous operation. Valid values are 1 or 2. SyncLink device receivers always recognize 1 or 2 stop bits automatically without configuration.

Parity

This member of the MGSL_PARAMS structure selects the character parity for asynchronous operation. This value must match that of the remote device. When enabled, the selected parity bit is added to transmitted characters and verified on received characters. Parity bits are stripped from received data before data is returned to an application. Characters received with a parity error are discarded. Valid values are specified with the following macros:

ASYNC_PARITY_NONE	no parity used
ASYNC_PARITY_EVEN	0 or 1 bit added to maintain even number of 1 bits in character
ASYNC_PARITY_ODD	0 or 1 bit added to maintain odd number of 1 bits in character

MGSL_PORT_CONFIG_EX

This structure is used with the configuration API calls MgslGetPortConfigEx and MgslSetPortConfigEx. The options specified with this structure take effect at driver load time. These options can also be set in the Windows Device Manager.

```
typedef struct _MGSL_PORT_CONFIG_EX
{
	ULONG BaseAddress;  /* obsolete, ignored */
	ULONG IrqLevel;     /* obsolete, ignored */
	ULONG DmaChannel;   /* obsolete, ignored */
	ULONG BusType;      /* obsolete, ignored */
	ULONG BusNumber;    /* obsolete, ignored */
	ULONG DeviceID;     /* obsolete, ignored */
	ULONG MaxFrameSize; /* max number of bytes per HDLC frame */
	ULONG Flags;        /* various bit flag options */

} MGSL_PORT_CONFIG_EX, *PMGSL_PORT_CONFIG_EX;
```

MaxFrameSize

Set the maximum number of bytes expected per frame of data. Received frames larger than this value are discarded. This also controls the maximum number of bytes accepted per MgslTransmit call.

Flags

This member controls the serial interface physical characteristics. The interface selection flags (MGSL_INTERFACE_XXX) only apply to the PCMCIA and USB devices. All other hardware uses jumpers to select the interface type.

MGSL_INTERFACE_MASK	a bit mask that covers all bits used to select an interface type
MGSL_INTERFACE_DISABLE	serial interface is disabled and tri-stated (high impedance)
MGSL_INTERFACE_RS232	serial interface is set for RS232 (single ended)
MGSL_INTERFACE_V35	serial interface is set for V.35 (mixed differential/single ended)
MGSL_INTERFACE_RS422	serial interface is set for RS422/485 (differential)

`MGSL_NO_TERMINATION` when set, input termination is disabled for differential modes
when clear, differential inputs use 120 Ohm termination
Applies only to USB hardware. PCI cards use jumpers or DIP switches to control input termination.

`MGSL_RTS_DRIVER_CONTROL` when set, RTS controls serial outputs
RTS on = outputs enabled
RTS off = outputs disabled (tri-state, high impedance)

MGSL_RECEIVE_REQUEST

This structure is used with the `MgslReceive` API call.

```
typedef struct _MGSL_RECEIVE_REQUEST
{
      ULONG  Status;            /* returned request status */
      ULONG  DataLength;        /* returned data length */
      Char   DataBuffer[1];     /* variable length data buffer */

} MGSL_RECEIVE_REQUEST, *PMGSL_RECEIVE_REQUEST;
```

The structure defines a header for a receive request. Memory for a receive request should include the size of data buffer required by the application. For example, the following code allocates a receive request with a 4096 byte buffer:

```
MGSL_RECEIVE_REQUEST *req;

req = (MGSL_RECEIVE_REQUEST*)malloc(sizeof(MGSL_RECEIVE_REQUEST) + 4096);
```

The DataLength field of the structure should be initialized before **every** call to `MgslReceive` to indicate the size of the buffer.

```
req->DataLength = 4096;
rc = MgslReceive(hDevice, req, ol);
```

On request completion, the `Status` field indicates success or error. Most error codes apply only to SDLC/HDLC mode. Unless the application is maintaining error statistics, the specific error code can be ignored. When `Status` is `RxStatus_OK`, data is returned in `DataBuffer` and `DataLength` is set to the number of returned bytes.

`RxStatus_OK`	data returned in `DataBuffer`, `DataLength` set to count of bytes
`RxStatus_CrcError`	SDLC/HDLC frame with a CRC error was received and discarded
`RxStatus_FifoOverrun`	hardware FIFO was full when data was received causing data loss
`RxStatus_ShortFrame`	SDLC/HDLC frame with only one data byte was received and discarded
`RxStatus_Abort`	SDLC/HDLC abort or idle sequence was received
`RxStatus_BufferOverrun`	API buffers were full when data was received causing data loss
`RxStatus_Cancel`	receive request was cancelled by user before normal completion
`RxStatus_BufferTooSmall`	SDLC/HDLC frame larger than the receive request buffer was received

C#

The module mgapi.cs provides a C# interface to SyncLink serial devices and is used to build the mgapi.dll C# API assembly. It is built on top of the C API and provides both a C# optimized interface and a C API wrapper. New C# applications should use the C# optimized interface. The C wrapper is used for quickly porting C applications to C# with minimal changes. The wrapper follows the naming conventions of the C API. This reduces the effort needed to port existing code.

Note: The API module uses C# version 4.

A C# Visual Studio 2010 solution is provided containing mgapi.cs and sample programs for each protocol. The API assembly must be added to application references in the Visual Studio project.

```
// C# API (new C# apps)
using Port = Microgate.SerialApi.Port;

// C# wrapper of C API (porting C apps)
// Microgate namespace contains SerialApi class
using Microgate;
```

Contents of mgapi\csharp directory:

csharp.sln	Visual Studio 2010 solution containing sample projects
mgapi\	C# API module
hdlc\	HDLC/SDLC sample
async\	asynchronous sample
bisync\	bisync/monosync sample
raw\	raw bitstream sample
tdm\	TDM sample
2wire\	2-wire (bussed) sample
signals\	control/status signal sample
cwrapper\	C API wrapper sample (for porting C applications)

A C# sample is run as shown below for the HDLC sample on device mghdlc1:

```
C:\mgapi\csharp\hdlc\bin\Release>hdlc mghdlc1
```

Port Object

A Port object is required for most tasks. A port name must be supplied. Port names for single port adapters have the form MGHDLCx, where 'x' is the adapter number. Port names for multiport adapters have the form MGMPxPy, where x is the adapter number and 'y' is the port number. The Port class function enumerate returns a list of available port names.

```
string[] names = Port.enumerate();
foreach (string name in names)
    Console.WriteLine(name);

port = new Port("MGHDLC1");
```

Open/Close

The Port method open claims a port for use and raises an exception if an error occurs. The Port method is_open determines if the port is open. Other attributes, properties and methods must be accessed only when the port is open.

```
try {
      port.open();
}
catch (FileNotFoundException) {
      Console.WriteLine("port not found");
}
catch (UnauthorizedAccessException) {
      Console.WriteLine("access denied or port in use");
}
catch (Win32Exception) {
      Console.WriteLine("open error");
}
```

The Port method close releases the port from use. A port must be closed to allow use by other processes. Ports are automatically closed when a process exits or the Port object is deallocated.

```
port.close();
```

CONFIGURATION

A port must be configured to match application specific requirements. Perform initial configuration with a `Port.Settings` object. The `Port` method `apply_settings()` cancels current operations and shuts down the hardware.

```
var settings = new Port.Settings();
settings.protocol = Port.HDLC;
settings.encoding = Port.MANCHESTER;
settings.crc = Port.CRC16;
settings.transmit_clock = Port.INTERNAL;
settings.receive_clock = Port.RECOVERED;
settings.internal_clock_rate = 9600;
settings.transmit_preamble_pattern = 0x7e;
settings.transmit_preamble_bits = 8;
port.apply_settings(settings);
```

During operation, alter port behavior using various properties.

```
// set pattern sent when transmitter is enabled but
// there is no data to send. HDLC flag = 0x7e
port.transmit_idle_pattern = 0x7e;
```

Set persistent default state that spans system restart and device ejection.

```
// RTS controls RS422 output enable
Port.Defaults defaults = port.get_defaults();
defaults.rts_output_enable = true;
port.set_defaults(defaults);
```

SEND DATA

The Port method write accepts a byte[] containing data to send. It blocks until the data is queued and enables the transmitter if needed. The Port method flush blocks until all queued data has been sent. Both methods return true on success or false if cancelled by another thread. If the Port property blocked_io is false (polling) both methods return false instead of blocking.

```
var send_data = new byte[] {0xff, 0x01};

port.blocked_io = true;
if (port.write(send_data))
    Console.WriteLine("data queued");
else
    Console.WriteLine("blocked write cancelled");
if (port.flush())
    Console.WriteLine("all data sent");
else
    Console.WriteLine("blocked flush cancelled");

port.blocked_io = false;
if (port.write(send_data))
    Console.WriteLine("data queued");
else
    Console.WriteLine("send queue full");
if (port.flush())
    Console.WriteLine("all data sent");
else
    Console.WriteLine("busy sending data");
```

The byte[] length must not exceed the Port.Defaults attribute max_data_size.

HDLC/SDLC protocol
write sends a variable size frame. Hardware adds flags and CRC to mark frame boundary.

TDM protocol
write sends one or more frames of fixed size set by TDM configuration.

Other protocols
write sends a variable length stream of bytes.

RECEIVE DATA

The Port method read accepts an optional size argument (default value of 1) and returns a byte[] containing received data. It blocks until data is available or returns null if cancelled by another thread. The Port method enable_receiver must be called before data can be received. If the Port property blocked_io is false (polling) read returns null instead of blocking.

```
port.blocked_io = true;
receive_data = port.read(size); // block until data available
if (receive_data != null)
    Console.WriteLine("{0} bytes received", receive_data.Length);
else
    Console.WriteLine("blocked read cancelled");

port.blocked_io = false;
receive_data = port.read(size); // poll for data
if (receive_data != null)
    Console.WriteLine("{0} bytes received", receive_data.Length);
else
    Console.WriteLine("no data available");
```

The size argument meaning depends on the protocol:

HDLC/SDLC

Size argument is ignored and should be omitted for clarity. read blocks until a variable size frame is received.

TDM

Size argument is ignored and should be omitted for clarity. read blocks until a fixed size frame or group of frames (as defined by TDM configuration) is received.

Other Protocols

read blocks until size bytes are received. size defaults to 1 and must be in the range 1 to the Port.Defaults attribute max_data_size.

Class Reference

Port

A **Port** object is required for most tasks. A port name must be supplied. Port names for single port adapters have the form MGHDLCx, where 'x' is the adapter number. Port names for multiport adapters have the form MGMPxPy, where x is the adapter number and 'y' is the port number.

```
port = new Port("MGHDLC1");
```

Methods

apply_settings

This method applies configuration contained in a [Port.Settings] object to hardware. Hardware is shutdown, blocked read and write calls are cancelled and the new configuration is applied.

```
var settings = new Port.Settings();  // allocate
settings.protocol = Port.HDLC;   // change
port.apply_settings(settings);   // apply
```

close

This method releases a port from use so it may be used by other processes. A port is automatically closed when a process exits or the port object is deallocated. Methods and properties other than open and is_open must not be accessed when a port is closed. Closing an already closed port has no effect. Port properties may change while port is closed.

disable_receiver

This method disables the receiver hardware and cancels blocked read calls. When disabled, the receiver ignores the receive data input.

disable_transmitter

This method disables the transmitter hardware and cancels blocked write and flush calls. When disabled, the transmit data output is held in a constant one state.

enable_receiver

This method enables the receiver hardware. When enabled, the receiver accepts data from the receive data input.

enable_transmitter

This method enables the transmitter hardware. The transmitter enters the idle state and sends the transmit idle pattern, except for asynchronous/isochronous protocols which always send all ones when idle. When data is passed to the write method, the transmitter becomes active and sends the data. After all data is sent, the transmitter is idle.

enumerate

This class method returns a list of valid port names.

```
string[] names = Port.enumerate();
foreach (string name in names)
    Console.WriteLine(name);
```

get_defaults

This method returns default configuration contained in a Port.Defaults object.

```
Port.Defaults defaults = port.get_defaults();
Console.WriteLine("max_data_size = {0}", defaults.max_data_size);
```

get_settings

This method returns current settings contained in a Port.Settings object.

```
Port Settings settings = port.get_settings();
if (settings.protocol == Port.HDLC)
    Console.WriteLine("HDLC protocol");
```

flush

The Port method flush blocks until all queued send data has been sent. true is returned on success or false if cancelled by another thread. If the Port property blocked_io is false (polling), false is returned instead of blocking.

```
port.blocked_io = true;
if (port.flush())
    Console.WriteLine("all data sent");
else
    Console.WriteLine("blocked flush cancelled");

port.blocked_io = false;
if (port.flush())
    Console.WriteLine("all data sent");
else
    Console.WriteLine("busy sending data");
```

is_open

Return true when port is open or false when port is closed.

open

The `Port` method `open` claims a port for use and raises an exception if an error occurs. The `Port` method `is_open` determines if the port is open. Other properties and methods should be accessed only when the port is open.

```
try {
      port.open();
}
catch (FileNotFoundException) {
      Console.WriteLine("port not found");
}
catch (UnauthorizedAccessException) {
      Console.WriteLine("access denied or port in use");
}
catch (Win32Exception) {
      Console.WriteLine("open error");
}
```

read

This method accepts an optional size argument (default value of 1) and returns a byte[] containing received data. It blocks until data is available or returns null if cancelled by another thread. The Port method enable_receiver must be called before data can be received. If the Port property blocked_io is false (polling) read returns null instead of blocking.

```
port.blocked_io = true;
receive_data = port.read(size); // block until data available
if (receive_data != null)
    Console.WriteLine("{0} bytes received", receive_data.Length);
else
    Console.WriteLine("blocked read cancelled");

port.blocked_io = false;
receive_data = port.read(size); // poll for data
if (receive_data != null)
    Console.WriteLine("{0} bytes received", receive_data.Length);
else
    Console.WriteLine("no data available");
```

The size argument depends on the protocol:

HDLC/SDLC

Size argument is ignored and should be omitted for clarity. read blocks until a variable size frame is received.

TDM

Size argument is ignored and should be omitted for clarity. read blocks until a fixed size frame or group of frames (as defined by TDM configuration) is received.

Other Protocols

read blocks until size bytes are received. size defaults to 1 and must be in the range 1 to the Port.Defaults attribute max_data_size.

set_defaults

This method sets persistent default configuration contained in a Port.Defaults object. Calling this method requires administrative privilege.

```
// RTS controls RS422 output enable
Port.Defaults defaults = port.get_defaults();
defaults.rts_output_enable = true;
port.set_defaults(defaults);
```

set_fsynth_rate

This method programs the frequency synthesizer to the specified rate and returns `true` if successful. The frequency synthesizer replaces the default base clock of 14.7456MHz. Use the frequency synthesizer only when generating an internal clock at a rate that is not a divisor of the default base clock. This method internally sets the property `base_clock_rate` to the same value.

All ports on devices with multiple ports share the same base clock. **Do not call this method when any port on the device is sending or receiving.** When calling this method on a device with multiple ports, set `base_clock_rate` directly on the other ports to the same value so the other ports use the correct value for the common base clock when calculating internal clock settings.

The internal clock generator is used for:

- clock generation for synchronous protocols
- reference clock for asynchronous protocol (at 8 or 16 times data rate)
- reference clock for clock recovery (at 8 or 16 times data rate)

The API tries to use x16 reference clocks for maximum accuracy. If the x16 reference clock is not a divisor of the base clock, the API falls back to x8 reference clocks.

This method uses a table of supported frequencies. If an unsupported frequency is specified then `false` is returned. The tables are located in the API module `mgapi.cs`. New table entries can be created as described in the frequency synthesizer section. The frequency synthesizer is only available on PCI Express cards and the SyncLink USB.

wait

This method waits for a specified event (hardware or signal state). The method blocks until one or more specified events occur. The method accepts an integer containing bit flags specifying events to wait for and returns an integer indicating which events occurred. An optional timeout in milliseconds causes the method to return 0 if the timeout expires.

```
Port.DSR_ON, Port.DSR_OFF
Port.CTS_ON, Port.CTS_OFF
Port.DCD_ON, Port.DCD_OFF
Port.RI_ON, Port.RI_OFF
Port.RECEIVE_ACTIVE, Port.RECEIVE_IDLE
```

```
// wait forever for CTS on or DCD off
uint events = port.wait(Port.CTS_ON | Port.DCD_OFF);
if (events != 0) {
    if ((events & Port.CTS_ON) != 0)
        Console.WriteLine("CTS is on");
    if ((events & Port.DCD_OFF) != 0)
        Console.WriteLine("DCD is off");
} else
    Console.WriteLine("error");


// wait 1 second for CTS on or DCD off
events = port.wait(Port.CTS_ON | Port.DCD_OFF, 1000);
if (events != 0) {
    if ((events & Port.CTS_ON) != 0)
        Console.WriteLine("CTS is on");
    if ((events & Port.DCD_OFF) != 0)
        Console.WriteLine("DCD is off");
} else
    Console.WriteLine("timeout");
```

write

write accepts a byte[] containing data to send. It blocks until the data is queued and enables the transmitter if needed. true is returned if data is queued or false if a blocked write is cancelled by another thread. If the Port property blocked_io is false (polling), false is returned instead of blocking. To determine when data has been completely sent call the flush method.

```
var send_data = new byte[] {0xff, 0x01};

port.blocked_io = true;
if (port.write(send_data))
    Console.WriteLine("data queued");
else
    Console.WriteLine("blocked write cancelled");

port.blocked_io = false;
if (port.write(send_data))
    Console.WriteLine("data queued");
else
    Console.WriteLine("send queue full");
```

The byte[] length must not exceed the Port.Defaults attribute max_data_size.

HDLC/SDLC protocol
write sends a variable size frame. Hardware adds flags and CRC to mark frame boundary.

TDM protocol
write sends one or more frames of fixed size set by TDM configuration.

Other protocols
write sends a variable length stream of bytes.

PROPERTIES AND ATTRIBUTES

base_clock_rate

This integer property is the base clock rate used in calculations for internal clock generation. Setting this property **does not** alter the actual base clock like `set_fsynth_rate`. Use this property to set the base clock when a custom fixed frequency oscillator is installed on the SyncLink device. Also use it after calling `set_fsynth_rate` for one port of a device with multiple ports to set the rate used by the other ports.

blocked_io

This Boolean property controls the behavior of the `read`, `write` and `flush` methods. This property is set to True when the `open` method is called.

`true` (blocking)
`read` blocks until data is available then returns a `byte[]`
`write` blocks until data is queued then returns `true`
`flush` blocks until all data is sent then returns `true`

`false` (polling)
`read` returns `null` instead of blocking
`write` and `flush` return `false` instead of blocking.

cts (Clear To Send)

This read only Boolean property reports the CTS input state. `true` = on, `false` = off.

```
if (port.cts)
    Console.WriteLine("CTS is on");
```

dcd (Data Carrier Detect)

This read only Boolean property reports the DCD input state. `true` = on, `false` = off.

```
if (port.dcd)
    Console.WriteLine("DCD is on");
```

dsr (Data Set Ready)

This read only Boolean property reports the DSR input state. `true` = on, `false` = off.

```
if (port.dsr)
    Console.WriteLine("DSR is on");
```

dtr (Data Terminal Ready)

This Boolean property controls the DTR output state. `true` = on, `false` = off.

```
port.dtr = true; // turn on DTR
```

gpio

This attribute is an array of 32 Port.GPIO objects with each object representing a general purpose I/O signal. Refer to the hardware manual (PDF) for how many GPIO signals are available on a specific device. GPIO signals are controlled and monitored through the output and state properties.

```
// set GPIO #6 direction to output
port.gpio[6].output = true;
// set GPIO #6 state to low
port.gpio[6].state = false;
// set GPIO #7 direction to input
port.gpio[7].output = false;
if (port.gpio[7].state)
    Console.WriteLine("GPIO #7 is on");
else
    Console.WriteLine("GPIO #7 is off");
```

half_duplex

This Boolean property enables a feature where outputs are automatically enabled when sending data and disabled when not sending data. This is used for bussed mode where transmit and receive data share the same cable conductors. This feature only affects RS422/V.11/differential outputs. Applications that require more control over output enable timing should use the rts_output_enable property instead.

ignore_read_errors

This Boolean property controls how receive errors are handled in HDLC/SDLC mode. When True, receive errors are silently discarded. Applications should only set this to false for diagnostic and logging purposes. This property is set to true by the open method.

interface_type

This integer property selects the serial interface standard for SyncLink USB and SyncLink PCIe devices. GT series cards use jumpers to select the serial interface and ignore this property. Interface values are defined by constants.

```
// Port.RS232 = single ended, unbalanced signals (V.28)
// Port.V35 = data/clocks differential (V.11), others single ended (V.28)
// Port.RS422 = differential signals (V.11)
// Port.RS530A = differential signals (V.11), except LL and RL

port.interface_type = Port.RS422;
```

ll (Local Loopback)

This Boolean property controls the LL output state.

```
port.ll = true; // turn on local loopback output
```

name

This read only string property contains the port name.

output_control

This integer property controls individual differential (RS422/V.11) outputs on the SyncLink USB. The integer contains bit flags controlling behavior and state for four output signals. This property does not affect PCI cards or single ended outputs.

[7] TXD select (0=auto, 1=manual)
[6] AUXCLK select (0=auto, 1=manual)
[5] DTR select (0=auto, 1=manual)
[4] RTS select (0=auto, 1=manual)
[3] TXD state (0=tristate/disabled, 1=enabled)
[2] AUXCLK state (0=tristate/disabled, 1=enabled)
[1] DTR state (0=tristate/disabled, 1=enabled)
[0] AUXCLK state (0=tristate/disabled, 1=enabled)

Auto (0) enables the differential output as described by other properties. Manual (1) enables outputs as selected by state bits in this property.

receive_count

This integer read only property indicates how many bytes are available to the application using the `read` method. The granularity of the count depends on the specific hardware and protocol. The count does not include data stored in hardware but not yet transferred to system memory. Use this property to poll for receive data.

ri (Ring Indicator)

This read only Boolean property reports the RI input state. `true` = on, `false` = off.

```
if (port.ri)
    Console.WriteLine("RI is on");
```

rl (Remote Loopback)

This Boolean property controls the RL output state.

```
port.rl = true; // turn on remote loopback output
```

rts (Request To Send)

This Boolean property controls the RTS output state. `true` = on, `false` = off.

```
port.rts = true; // turn on RTS output
```

rts_output_enable

This Boolean property allows RTS to control serial interface outputs. When `true`, RTS on enables outputs and RTS off disables (tristate/hi-Z) outputs. This feature is used in bussed applications where transmit and receive data use the same cable conductors. This feature only affects RS422/V.11/differential outputs. Bussed applications that do not require control over output enable timing should use the `half_duplex` property instead.

```
port.rts_output_enable = true;
port.rts = true; // turn on outputs
// transmit data
port.rts = false; // turn off outputs (tristate/hi-Z)
// receive data
```

signals

This integer property contains control output and status input states with each signal represented by a bit. Individual signal bits are defined as constants. Each access (get or set) of this property results in a hardware access.

```
// Port.DTR = data terminal ready output
// Port.RTS = ready to send output
// Port.CTS = clear to send input
// Port.DSR = data set ready input
// Port.DCD = data carrier detect input
// Port.RI  = ring indicator input

// turn on DTR output
port.signals |= Port.DTR;

// turn off RTS output
port.signals &= ~Port.RTS;

if ((port.signals & Port.DCD) != 0)
    Console.WriteLine("DCD input is on");
else
    Console.WriteLine("DCD input is off");
```

Each signal has a property that is an alias to the `signals` property.

```
// turn on DTR output
port.dtr = true;
// turn off RTS output
port.rts = false;

if (port.dcd)
    Console.WriteLine("DCD input is on");
else
    Console.WriteLine("DCD input is off");
```

transmit_count

This integer read only property indicates the number of bytes in the API send queue. The count does not include data transferred to hardware but not yet sent. The count granularity depends on the specific hardware and protocol. Use this property to maintain queue levels in a desired range for the purpose of controlling transmit latency. Do NOT use this property as an indication of all sent as a zero value may not reflect data still stored in the hardware. Use the `flush` method to determine when all data has been sent.

termination

This Boolean property controls termination of differential (RS422/V.11) inputs on the SyncLink USB/PCIe. When `true`, differential inputs are terminated with 120 ohms. When `false`, differential inputs are not terminated. GT cards select termination with jumpers or switch settings and are not affected by this property.

transmit_idle_pattern

This property specifies the pattern sent on the transmit data output when the transmitter is idle (enabled but no data to send). This can be an arbitrary 8 or 16 bit value. The idle pattern may be used by a remote device to distinguish between an inactive and idle connection. The transmit idle pattern may have additional protocol specific purpose. For monosync and bisync protocols, this property becomes an alias for sync_pattern.

PORT.DEFAULTS

A `Port.Defaults` object contains default configuration that is persistent across system restarts and device ejection. Defaults are accessed with the `Port.get_defaults` and `Port.set_defaults` methods.

PROPERTIES AND ATTRIBUTES

interface_type

This Boolean attribute sets the default state of the `Port.interface_type` property. This value is labeled "Serial interface" in the Windows device manager properties for SyncLink devices.

max_data_size

This integer attribute specifies the maximum number of bytes that are returned by `read` or supplied to `write`. The default value is 4096. This value is labeled "Max Frame Size" in the Windows device manager properties for SyncLink devices.

rts_output_enable

This Boolean attribute sets the default state of the `Port.rts_output_enable` property. This value is labeled "Disable RS422/485 outputs when RTS is off" in the Windows device manager properties for SyncLink devices.

termination

This Boolean attribute sets the default state of the `Port.termination` property. This value is labeled "Serial interface" in the Windows device manager properties for SyncLink devices.

Port.GPIO

A `Port.GPIO` object represents a single general purpose I/O signal. PCI Express and USB serial devices have GPIO signals accessible to headers on the device PCB. Refer to the hardware manual (PDF) for information on how many GPIO signals are available and header pin assignments.

Properties and Attributes

state

This Boolean property contains the GPIO signal state: `true` = high, `false` = low.

output

This Boolean property contains the GPIO signal direction: `true` = output, `false` = input. Some GPIO signals have a fixed direction, see hardware manual for details.

Port.Settings

A `Port.Settings` object contains configuration used with the Port methods `get_settings` and `apply_settings`.

Properties and Attributes

async_data_bits

This integer attribute specifies the number of data bits in an asynchronous character. Valid values: 5 to 8

async_data_rate

This integer attribute specifies the data rate for the asynchronous protocol in bits per second. The internal clock generator runs at 8 or 16 times the data rate to supply the asynchronous receiver sampling clock.

async_stop_bits

This integer attribute specifies the number of stop bits in an asynchronous character. Valid values: 1 or 2

async_parity

This attribute specifies the parity used in an asynchronous character. Valid values: `Port.OFF`, `Port.EVEN`, `Port.ODD`.

auto_cts

This Boolean attribute controls automatic processing of the CTS (clear to send) input. When `true`, the transmitter waits for CTS on before sending data.

auto_dcd

This Boolean attribute controls automatic processing of the DCD (data carrier detect) input. When `true`, the receiver is disabled when DCD is off.

auto_rts

This Boolean attribute controls automatic processing of the RTS (request to send) output. When `true`, RTS is on when data is available to send and off when all data has been sent. This feature only applies when RTS starts in the off state. If RTS is manually set to on then RTS stays on regardless of this attribute.

crc

This attribute selects the frame check sequence used with the HDLC (SDLC) protocol. Other protocols must set this attribute to `Port.OFF`. Valid values are listed below. The same CRC must be used by all endpoints.

`Port.OFF`	CRC disabled
`Port.CRC16`	16-bit CRC
`Port.CRC32`	32-bit CRC

discard_data_with_error

This Boolean attribute determines how the API handles HDLC (SDLC) frames with a frame check error. When set to `true` (default), frames with errors are silently discarded. When `false`, frames with errors are returned to the application. Only applications that with to inspect invalid data for diagnostics, logging and recovery purposes should set this to `false`.

discard_received_crc

This Boolean attribute controls handling of received CRC values for the HDLC (SDLC) protocol. When `true` (default), the CRC is discarded and the application only gets data. When `false`, the CRC is returned as the final bytes of the frame to the application. Only applications that wish to inspect the CRC value for diagnostics, logging or recovery purposes should set this to `false`.

encoding

This attribute selects the encoding of serial data. Encoding defines how individual logical bits (0 or 1) are translated to physical data signal levels (high or low). Serial link endpoints must use the same encoding. Valid encoding values are listed below. Refer to the Encoding section for details of each value.

```
Port.NRZ
Port.NRZB
Port.NRZI
Port.NRZI_MARK
Port.FM0
Port.FM1
Port.MANCHESTER
Port.DIFF_BIPHASE_LEVEL
```

hdlc_address_filter

This integer attribute selects an 8-bit value used to filter received HDLC (SDLC) frames. Other protocols ignore this attribute. A value of `0xFF` (default) disables filtering and returns all frames to the application. Other values indicate the expected HDLC station address so that only frames with the broadcast address (`0xFF`) and the specified station address are returned to the application. Other frames are silently discarded.

internal_clock_rate

This integer attribute is the **data rate** used to setup the internal clock generator. When using the internal clock directly as a data clock it runs at this rate. When using clock recovery the internal clock runs at 8 or 16 times this rate to create a **reference clock** for sampling the receive data input and recovering a data clock.

The internal clock is generated by dividing the hardware base clock (default 14.7456MHz) by a 16-bit integer. If the specified rate is not a divisor of the base clock it is approximated by selecting the next slower rate that is a divisor. For details, refer to the clock generator section.

The API tries to use a 16x reference clock for maximum accuracy. If a 16x reference clock is not a divisor of the base clock, the API falls back to an 8x reference clock. If an x8 reference clock is not a divisor of the base clock then the specified data rate can't be precisely recovered.

The base clock can be adjusted on PCI Express and USB devices to allow precise internal clock rates when the specified rate (or reference clock) is not a divisor of the default 14.7456MHz base clock. Refer to the `set_fsynth_rate` method.

internal_loopback

This Boolean attribute enables internal loopback of data signals for diagnostic or development purposes. When set to `true`, the transmit data output is connected internally to the data input and data clocks are internally generated.

msb_first

This Boolean attribute selects the serial bit order. Most protocols must use least significant bit (LSB) first. Some protocols (RAW and TDM) may optionally use most significant bit (MSB) first by setting this attribute to `true`.

protocol

This attribute selects the serial protocol. Valid values are listed below. Refer to the Protocol section for details.

`Port.HDLC`	HDLC, also called SDLC
`Port.ASYNC`	Asynchronous and Isochronous
`Port.BISYNC`	Bi-synchronous
`Port.MONOSYNC`	Mono-synchronous
`Port.RAW`	Raw bit stream
`Port.TDM`	Time Division Multiplexing

receive_clock

This attribute selects the source of the receive clock.

`Port.TXC_INPUT`	TxC clock input
`Port.RXC_INPUT`	RxC clock input
`Port.INTERNAL`	internal clock generator
`Port.RECOVERED`	clock recovered from receive data input

receive_clock_invert

This attribute controls receive clock polarity. When set to default value of `false`, the receive clock uses normal polarity. When set to `true`, the receive clock uses inverted clock polarity. Endpoints must use the same clock polarity.

sync_pattern

This integer attribute specifies the synchronization pattern used for byte synchronous protocols (monosync or bisync) and should not be used for other protocols. The transmitter sends the sync pattern when idle (enabled and no data to send). The receiver searches for the sync pattern to determine byte alignment and returns data following a sync pattern to the application until the receiver is disabled or forced to the idle state so it searches for the next sync pattern.

tdm_sync_delay

This integer attribute specifies the number of clock cycles data is delayed from the start of a TDM sync pulse. Valid values are 0 (no delay), 1 and 2.

tdm_sync_short

This Boolean attribute specifies the length of a TDM sync pulse. `true` = one clock cycle, `false` = one slot length

tdm_sync_invert

This Boolean attribute specifies the polarity of the TDM sync signal. `false` = normal, `true` = inverted

tdm_frame_count

This integer attribute specifies the number of TDM frames returned to the application by the `read` method. Valid values are 1 to 256. The frame count time the fixed frame size must not exceed `Port.Defaults.max_data_size`. This allows better performance by reducing the number of `read` calls when the frame size is small and the data rate is high.

tdm_slot_count

This integer attribute specifies the number of TDM slots in a frame. Valid values are in the range 2 to 32 **or** 384.

tdm_slot_bits

This integer attribute specifies the number of bits in a TDM slot. Valid values are 8, 12, 16, 20, 24, 28 and 32.

transmit_clock

This attribute selects the source of the transmit clock.

`Port.TXC_INPUT`	TxC clock input
`Port.RXC_INPUT`	RxC clock input
`Port.INTERNAL`	internal clock generator
`Port.RECOVERED`	clock recovered from receive data input

transmit_clock_invert

This attribute controls transmit clock polarity. When set to default value of `false`, the transmit clock uses normal polarity. When set to `true`, the transmit clock uses inverted clock polarity. Endpoints must use the same clock polarity.

transmit_preamble_bits

This integer attribute selects the number of bits of preamble to prepend to sent data. A preamble is used to assist clock recovery from a data signal. Only synchronous protocols using clock recovery should set this attribute to a non-zero value. The only valid values are 8, 16, 32 and 64.

transmit_preamble_pattern

This 8-bit integer attribute selects the pattern that is prepended to sent data. A preamble is used to assist clock recovery from a data signal. Only synchronous protocols using clock recovery should set this attribute. The length of the preamble is selected by the `transmit_preamble_bits` attribute.

PYTHON

The module mgapi.py provides a Python interface to SyncLink serial devices. It is built on top of the C API and provides both a Python optimized interface and a C API wrapper. New Python applications should use the Python optimized interface. The C wrapper is used for quickly porting C applications to Python with minimal changes. The wrapper follows the naming conventions of the C API. This violates Python style standards but reduces the effort needed to port existing code.

WARNING: The API module uses Python 3 and is incompatible with Python 2.

A Python installation is required to run the Python sample code. The API module must be imported into a Python application with one of the following lines:

```
# Python API (new Python apps)
from mgapi import Port

# Python wrapper of C API (porting C apps)
import mgapi
```

The module must be found in one of three ways:

- mgapi.py is located in same directory as application.
- mgapi.py is located in the Python system path.
- mgapi.tar.gz package is installed.

The package is installed using the Python installation program (pip) in a command prompt with Administrator privilege:

```
C:\mgapi\python>pip install mgapi.tar.gz
```

Contents of mgapi\python directory:

mgapi.py	Python API module
mgapi.tar.gz	Installable Python API package
samples\hdlc.py	HDLC/SDLC sample
samples\async.py	asynchronous sample
samples\bisync.py	bisync/monosync sample
samples\raw.py	raw bitstream sample
samples\tdm.py	TDM sample
samples\2wire.py	2-wire (bussed) sample
samples\commapi.py	asynchronous using pyserial interface
samples\signals.py	control/status signal sample
samples\cwrapper.py	C API wrapper sample (for porting C applications)

A Python application is run as shown below for the HDLC sample on device mghdlc1:

```
C:\mgapi\python\samples>python hdlc.py mghdlc1
```

PORT OBJECT

A Port object is required for most tasks. A port name must be supplied. Port names for single port adapters have the form MGHDLCx, where 'x' is the adapter number. Port names for multiport adapters have the form MGMPxPy, where x is the adapter number and 'y' is the port number. The Port class function enumerate returns a list of available port names.

```
names = Port.enumerate()
for name in names:
    print(name)

port = Port('MGHDLC1')
```

OPEN/CLOSE

The Port method open claims a port for use and raises an exception if an error occurs. The Port method is_open determines if the port is open. Other attributes, properties and methods must be accessed only when the port is open.

```
try:
    port.open()
except FileNotFoundError:
    print('port not found')
except PermissionError:
    print('access denied or port in use')
except OSError:
    print('open error')
if port.is_open():
    print('port is open')
```

The Port method close releases the port from use. A port must be closed to allow use by other processes. Ports are automatically closed when a process exits or the Port object is deallocated.

```
port.close()
```

CONFIGURATION

A port must be configured to match application specific requirements. Perform initial configuration with a Port.Settings() object. The `Port` method `apply_settings()` cancels current operations and shuts down the hardware.

```
settings = Port.Settings()
settings.protocol = Port.HDLC
settings.encoding = Port.MANCHESTER
settings.crc = Port.CRC16
settings.transmit_clock = Port.INTERNAL
settings.receive_clock = Port.RECOVERED
settings.internal_clock_rate = 9600
settings.transmit_preamble_pattern = 0x7e
settings.transmit_preamble_bits = 8
port.apply_settings(settings)
```

During operation, alter port behavior using various properties.

```
# set pattern sent when transmitter is enabled but
# there is no data to send. HDLC flag = 0x7e
port.transmit_idle_pattern = 0x7e
```

Set persistent default state that spans system restart and device ejection.

```
# RTS controls RS422 output enable
defaults = port.get_defaults()
defaults.rts_output_enable = True
port.set_defaults(defaults)
```

SEND DATA

The Port method write accepts a bytearray containing data to send. It blocks until the data is queued and enables the transmitter if needed. The Port method flush blocks until all queued data has been sent. Both methods return True on success or False if cancelled by another thread. If the Port property blocked_io is False (polling) both methods return False instead of blocking.

```
send_data = bytearray.fromhex('FF01')

port.blocked_io = True
if port.write(send_data):
    print('data queued')
else:
    print('blocked write cancelled')
if port.flush():
    print('all data sent')
else:
    print('blocked flush cancelled')

port.blocked_io = False
if port.write(send_data):
    print('data queued')
else:
    print('send queue full')
if port.flush():
    print('all data sent')
else:
    print('busy sending data')
```

The bytearray length must not exceed the Port.Defaults attribute max_data_size.

HDLC/SDLC protocol
write sends a variable size frame. Hardware adds flags and CRC to mark frame boundary.

TDM protocol
write sends one or more frames of fixed size set by TDM configuration.

Other protocols
write sends a variable length stream of bytes.

RECEIVE DATA

The Port method read accepts an optional size argument (default value of 1) and returns a bytearray containing received data. It blocks until data is available or returns None if cancelled by another thread. The Port method enable_receiver must be called before data can be received. If the Port property blocked_io is False (polling) read returns None instead of blocking.

```
port.blocked_io = True
receive_data = port.read(size) # block until data available
if receive_data:
    print(len(receive_data), 'bytes received')
else:
    print('blocked read cancelled')

port.blocked_io = False
receive_data = port.read(size) # poll for data
if receive_data:
    print(len(receive_data), 'bytes received')
else:
    print('no data available')
```

The size argument meaning depends on the protocol:

HDLC/SDLC

Size argument is ignored and should be omitted for clarity. read blocks until a variable size frame is received.

TDM

Size argument is ignored and should be omitted for clarity. read blocks until a fixed size frame or group of frames (as defined by TDM configuration) is received.

Other Protocols

read blocks until size bytes are received. size defaults to 1 and must be in the range 1 to the Port.Defaults attribute max_data_size.

Class Reference

Port

A **Port** object is required for most tasks. A port name must be supplied. Port names for single port adapters have the form MGHDLCx, where 'x' is the adapter number. Port names for multiport adapters have the form MGMPxPy, where x is the adapter number and 'y' is the port number.

```
port = Port('MGHDLC1')
```

Methods

apply_settings

This method applies configuration contained in a Port.Settings object to hardware. Hardware is shutdown, blocked read and write calls are cancelled and the new configuration is applied.

```
settings = Port.Settings()      # allocate
settings.protocol = Port.HDLC   # change
port.apply_settings(settings)   # apply
```

close

This method releases a port from use so it may be used by other processes. A port is automatically closed when a process exits or the port object is deallocated. Methods and properties other than open and is_open must not be accessed when a port is closed. Closing an already closed port has no effect. Port properties may change while port is closed.

disable_receiver

This method disables the receiver hardware and cancels blocked read calls. When disabled, the receiver ignores the receive data input.

disable_transmitter

This method disables the transmitter hardware and cancels blocked write and flush calls. When disabled, the transmit data output is held in a constant one state.

enable_receiver

This method enables the receiver hardware. When enabled, the receiver accepts data from the receive data input.

enable_transmitter

This method enables the transmitter hardware. The transmitter enters the idle state and sends the transmit idle pattern, except for asynchronous/isochronous protocols which always send all ones when idle. When data is passed to the write method, the transmitter becomes active and sends the data. After all data is sent, the transmitter is idle.

enumerate

This class method returns a list of valid port names.

```
names = Port.enumerate()
for name in names:
    print(name)
```

get_defaults

This method returns default configuration contained in a `Port.Defaults` object.

```
defaults = port.get_defaults()
print('max_data_size =', defaults.max_data_size)
```

get_settings

This method returns current settings contained in a `Port.Settings` object.

```
settings = port.get_settings()
if settings.protocol == Port.HDLC:
    print('HDLC protocol')
```

flush

The `Port` method `flush` blocks until all queued send data has been sent. `True` is returned on success or `False` if cancelled by another thread. If the `Port` property `blocked_io` is `False` (polling), `False` is returned instead of blocking.

```
port.blocked_io = True
if port.flush():
    print('all data sent')
else:
    print('blocked flush cancelled')

port.blocked_io = False
if port.flush():
    print('all data sent')
else:
    print('busy sending data')
```

is_open

Return `True` when port is open or `False` when port is closed.

open

The Port method open claims a port for use and raises an exception if an error occurs. The Port method is_open determines if the port is open. Other properties and methods should be accessed only when the port is open.

```
try:
    port.open()
except FileNotFoundError:
    print('port not found')
except PermissionError:
    print('access denied or port in use')
except OSError:
    print('open error =', port.error)
if port.is_open():
    print('port is open')
```

read

This method accepts an optional size argument (default value of 1) and returns a bytearray containing received data. It blocks until data is available or returns None if cancelled by another thread. The Port method enable_receiver must be called before data can be received. If the Port property blocked_io is False (polling) read returns None instead of blocking.

```
port.blocked_io = True
receive_data = port.read(size) # block until data available
if receive_data:
    print(len(receive_data), 'bytes received')
else:
    print('blocked read cancelled')

port.blocked_io = False
receive_data = port.read(size) # poll for data
if receive_data:
    print(len(receive_data), 'bytes received')
else:
    print('no data available')
```

The size argument depends on the protocol:

HDLC/SDLC

Size argument is ignored and should be omitted for clarity. read blocks until a variable size frame is received.

TDM

Size argument is ignored and should be omitted for clarity. read blocks until a fixed size frame or group of frames (as defined by TDM configuration) is received.

Other Protocols

read blocks until size bytes are received. size defaults to 1 and must be in the range 1 to the Port.Defaults attribute max_data_size.

set_defaults

This method sets persistent default configuration contained in a Port.Defaults object. Calling this method requires administrative privilege.

```
defaults = port.get_defaults()
defaults.max_data_size = 8192
port.set_defaults(defaults)
```

set_fsynth_rate

This method programs the frequency synthesizer to the specified rate and returns `True` if successful. The frequency synthesizer replaces the default base clock of 14.7456MHz. Use the frequency synthesizer only when generating an internal clock at a rate that is not a divisor of the default base clock. This method internally sets the property `base_clock_rate` to the same value.

All ports on devices with multiple ports share the same base clock. **Do not call this method when any port on the device is sending or receiving.** When calling this method on a device with multiple ports, set `base_clock_rate` directly on the other ports to the same value so the other ports use the correct value for the common base clock when calculating internal clock settings.

The internal clock generator is used for:

- clock generation for synchronous protocols
- reference clock for asynchronous protocol (at 8 or 16 times data rate)
- reference clock for clock recovery (at 8 or 16 times data rate)

The API tries to use x16 reference clocks for maximum accuracy. If the x16 reference clock is not a divisor of the base clock, the API falls back to x8 reference clocks.

This method uses a table of supported frequencies. If an unsupported frequency is specified then `False` is returned. The tables are located in the API module `mgapi.py`. New table entries can be created as described in the frequency synthesizer section. The frequency synthesizer is only available on PCI Express cards and the SyncLink USB.

wait

This method waits for a specified event (hardware or signal state). The method blocks until one or more specified events occur. The method accepts an integer containing bit flags specifying events to wait for and returns an integer indicating which events occurred. An optional timeout in milliseconds causes the method to return 0 if the timeout expires.

```
Port.DSR_ON, Port.DSR_OFF
Port.CTS_ON, Port.CTS_OFF
Port.DCD_ON, Port.DCD_OFF
Port.RI_ON, Port.RI_OFF
Port.RECEIVE_ACTIVE, Port.RECEIVE_IDLE
```

```
# wait forever for CTS on or DCD off
events = port.wait(Port.CTS_ON | Port.DCD_OFF)
if events:
    if events & Port.CTS_ON:
        print('CTS is on')
    if events & Port.DCD_OFF:
        print('DCD is off')
else:
    print('error')


# wait 1 second for CTS on or DCD off
events = port.wait(Port.CTS_ON | Port.DCD_OFF, 1000)
if events:
    if events & Port.CTS_ON:
        print('CTS is on')
    if events & Port.DCD_OFF:
        print('DCD is off')
else:
    print('timeout')
```

write

`write` accepts a `bytearray` containing data to send. It blocks until the data is queued and enables the transmitter if needed. `True` is returned if data is queued or `False` if a blocked `write` is cancelled by another thread. If the `Port` property `blocked_io` is `False` (polling), `False` is returned instead of blocking. To determine when data has been completely sent call the `flush` method.

```
send_data = bytearray.fromhex('FF01')

port.blocked_io = True
if port.write(send_data):
    print('data queued')
else:
    print('blocked write cancelled')

port.blocked_io = False
if port.write(send_data):
    print('data queued')
else:
    print('send queue full')
```

The `bytearray` length must not exceed the `Port.Defaults` attribute `max_data_size`.

HDLC/SDLC protocol
`write` sends a variable size frame. Hardware adds flags and CRC to mark frame boundary.

TDM protocol
`write` sends one or more frames of fixed size set by TDM configuration.

Other protocols
`write` sends a variable length stream of bytes.

PROPERTIES AND ATTRIBUTES

base_clock_rate

This integer property is the base clock rate used in calculations for internal clock generation. Setting this property **does not** alter the actual base clock like set_fsynth_rate. Use this property to set the base clock when a custom fixed frequency oscillator is installed on the SyncLink device. Also use it after calling set_fsynth_rate for one port of a device with multiple ports to set the rate used by the other ports.

blocked_io

This Boolean property controls the behavior of the read, write and flush methods. This property is set to True when the open method is called.

True (blocking)
read blocks until data is available then returns a bytearray
write blocks until data is queued then returns True
flush blocks until all data is sent then returns True

False (polling)
read returns None instead of blocking
write and flush return False instead of blocking.

cts (Clear To Send)

This read only Boolean property reports the CTS input state. True = on, False = off.

```
if port.cts:
    print('CTS is on')
```

dcd (Data Carrier Detect)

This read only Boolean property reports the DCD input state. True = on, False = off.

```
if port.dcd:
    print('DCD is on')
```

dsr (Data Set Ready)

This read only Boolean property reports the DSR input state. True = on, False = off.

```
if port.dsr:
    print('DSR is on')
```

dtr (Data Terminal Ready)

This Boolean property controls the DTR output state. True = on, False = off.

```
port.dtr = True # turn on DTR
```

gpio

This attribute is a list of 32 Port.GPIO objects with each object representing a general purpose I/O signal. Refer to the hardware manual (PDF) for how many GPIO signals are available on a specific device. GPIO signals are controlled and monitored through the `output` and `state` properties.

```
# set GPIO #6 direction to output
port.gpio[6].output = True
# set GPIO #6 state to low
port.gpio[6].state = False
# set GPIO #7 direction to input
port.gpio[7].output = False
if port.gpio[7].state:
    print('GPIO #7 is on')
else:
    print('GPIO #7 is off')
```

half_duplex

This Boolean property enables a feature where outputs are automatically enabled when sending data and disabled when not sending data. This is used for bussed mode where transmit and receive data share the same cable conductors. This feature only affects RS422/V.11/differential outputs. Applications that require more control over output enable timing should use the `rts_output_enable` property instead.

ignore_read_errors

This Boolean property controls how receive errors are handled in HDLC/SDLC mode. When `True`, receive errors are silently discarded. Applications should only set this to `False` for diagnostic and logging purposes. This property is set to `True` by the `open` method.

interface

This integer property selects the serial interface standard for SyncLink USB and SyncLink PCIe devices. GT series cards use jumpers to select the serial interface and ignore this property. Interface values are defined by constants.

```
# Port.RS232 = single ended, unbalanced signals (V.28)
# Port.V35 = data/clocks differential (V.11), others single ended (V.28)
# Port.RS422 = differential signals (V.11)
# Port.RS530A = differential signals (V.11), except

port.interface = Port.RS422
```

ll (Local Loopback)

This Boolean property controls the LL output state.

```
port.ll = True # turn on local loopback output
```

name

This read only string property contains the port name.

output_control

This integer property controls individual differential (RS422/V.11) outputs on the SyncLink USB. The integer contains bit flags controlling behavior and state for four output signals. This property does not affect PCI cards or single ended outputs.

[7] TXD select (0=auto, 1=manual)
[6] AUXCLK select (0=auto, 1=manual)
[5] DTR select (0=auto, 1=manual)
[4] RTS select (0=auto, 1=manual)
[3] TXD state (0=tristate/disabled, 1=enabled)
[2] AUXCLK state (0=tristate/disabled, 1=enabled)
[1] DTR state (0=tristate/disabled, 1=enabled)
[0] AUXCLK state (0=tristate/disabled, 1=enabled)

Auto (0) enables the differential output as described by other properties. Manual (1) enables outputs as selected by state bits in this property.

receive_count

This integer read only property indicates how many bytes are available to the application using the `read` method. The granularity of the count depends on the specific hardware and protocol. The count does not include data stored in hardware but not yet transferred to system memory. Use this property to poll for receive data.

ri (Ring Indicator)

This read only Boolean property reports the RI input state. `True` = on, `False` = off.

```
if port.ri:
    print('RI is on')
```

rl (Remote Loopback)

This Boolean property controls the RL output state.

```
port.rl = True # turn on remote loopback output
```

rts (Request To Send)

This Boolean property controls the RTS output state. `True` = on, `False` = off.

```
port.rts = True # turn on RTS output
```

rts_output_enable

This Boolean property allows RTS to control serial interface outputs. When `True`, RTS on enables outputs and RTS off disables (tristate/hi-Z) outputs. This feature is used in bussed applications where transmit and receive data use the same cable conductors. This feature only affects RS422/V.11/differential outputs. Bussed applications that do not require control over output enable timing should use the `half_duplex` property instead.

```
port.rts_output_enable = True
port.rts = True # turn on outputs
# transmit data
port.rts = False # turn off outputs (tristate/hi-Z)
# receive data
```

signals

This integer property contains control output and status input states with each signal represented by a bit. Individual signal bits are defined as constants. Each access (get or set) of this property results in a hardware access.

```
# Port.DTR = data terminal ready output
# Port.RTS = ready to send output
# Port.CTS = clear to send input
# Port.DSR = data set ready input
# Port.DCD = data carrier detect input
# Port.RI  = ring indicator input

# turn on DTR output
port.signals |= Port.DTR

# turn off RTS output
port.signals &= ~Port.RTS

if port.signals & Port.DCD:
    print('DCD input is on')
else:
    print('DCD input is off')
```

Each signal has a property that is an alias to the `signals` property.

```
# turn on DTR output
port.dtr = True
# turn off RTS output
port.rts = False

if port.dcd:
    print('DCD input is on')
else:
    print('DCD input is off')
```

transmit_count

This integer read only property indicates the number of bytes in the API send queue. The count does not include data transferred to hardware but not yet sent. The count granularity depends on the specific hardware and protocol. Use this property to maintain queue levels in a desired range for the purpose of controlling transmit latency. Do NOT use this property as an indication of all sent as a zero value may not reflect data still stored in the hardware. Use the `flush` method to determine when all data has been sent.

termination

This Boolean property controls termination of differential (RS422/V.11) inputs on the SyncLink USB and SyncLink PCIe hardware. When `True`, differential inputs are terminated with 120 ohms. When `False`, differential inputs are not terminated. GT series cards select termination with jumpers or switch settings and are not affected by this property.

transmit_idle_pattern

This property specifies the pattern sent on the transmit data output when the transmitter is idle (enabled but no data to send). This can be an arbitrary 8 or 16 bit value. The idle pattern may be used by a remote device to distinguish between an inactive and idle connection. The transmit idle pattern may have additional protocol specific purpose. For monosync and bisync protocols, this property becomes an alias for sync_pattern.

Port.Defaults

A `Port.Defaults` object contains default configuration that is persistent across system restarts and device ejection. Defaults are accessed with the `Port.get_defaults` and `Port.set_defaults` methods.

Properties and Attributes

interface

This Boolean attribute sets the default state of the `Port.interface` property. This value is labeled "Serial interface" in the Windows device manager properties for SyncLink devices.

max_data_size

This integer attribute specifies the maximum number of bytes that are returned by `read` or supplied to `write`. The default value is 4096. This value is labeled "Max Frame Size" in the Windows device manager properties for SyncLink devices.

rts_output_enable

This Boolean attribute sets the default state of the `Port.rts_output_enable` property. This value is labeled "Disable RS422/485 outputs when RTS is off" in the Windows device manager properties for SyncLink devices.

termination

This Boolean attribute sets the default state of the `Port.termination` property. This value is labeled "Serial interface" in the Windows device manager properties for SyncLink devices.

Port.GPIO

A Port.GPIO object represents a single general purpose I/O signal. PCI Express and USB serial devices have GPIO signals accessible to headers on the device PCB. Refer to the hardware manual (PDF) for information on how many GPIO signals are available and header pin assignments.

Properties and Attributes

state

This Boolean property contains the GPIO signal state: True = high, False = low.

output

This Boolean property contains the GPIO signal direction: True = output, False = input. Some GPIO signals have a fixed direction, see hardware manual for details.

Port.Settings

A Port.Settings object contains configuration used with the Port methods get_settings and apply_settings.

Properties and Attributes

async_data_bits

This integer attribute specifies the number of data bits in an asynchronous character. Valid values: 5 to 8

async_data_rate

This integer attribute specifies the data rate for the asynchronous protocol in bits per second. The internal clock generator runs at 8 or 16 times the data rate to supply the asynchronous receiver sampling clock.

async_stop_bits

This integer attribute specifies the number of stop bits in an asynchronous character. Valid values: 1 or 2

async_parity

This attribute specifies the parity used in an asynchronous character. Valid values: Port.OFF, Port.EVEN, Port.ODD.

auto_cts

This Boolean attribute controls automatic processing of the CTS (clear to send) input. When True, the transmitter waits for CTS on before sending data.

auto_dcd

This Boolean attribute controls automatic processing of the DCD (data carrier detect) input. When True, the receiver is disabled when DCD is off.

auto_rts

This Boolean attribute controls automatic processing of the RTS (request to send) output. When True, RTS is on when data is available to send and off when all data has been sent. This feature only applies when RTS starts in the off state. If RTS is manually set to on then RTS stays on regardless of this attribute.

crc

This attribute selects the frame check sequence used with the HDLC (SDLC) protocol. Other protocols must set this attribute to `Port.OFF`. Valid values are listed below. The same CRC must be used by all endpoints.

`Port.OFF`	CRC disabled
`Port.CRC16`	16-bit CRC
`Port.CRC32`	32-bit CRC

discard_data_with_error

This Boolean attribute determines how the API handles HDLC (SDLC) frames with a frame check error. When set to `True` (default), frames with errors are silently discarded. When `False`, frames with errors are returned to the application. Only applications that with to inspect invalid data for diagnostics, logging and recovery purposes should set this to `False`.

discard_received_crc

This Boolean attribute controls handling of received CRC values for the HDLC (SDLC) protocol. When `True` (default), the CRC is discarded and the application only gets data. When `False`, the CRC is returned as the final bytes of the frame to the application. Only applications that wish to inspect the CRC value for diagnostics, logging or recovery purposes should set this to `False`.

encoding

This attribute selects the encoding of serial data. Encoding defines how individual logical bits (0 or 1) are translated to physical data signal levels (high or low). Serial link endpoints must use the same encoding. Valid encoding values are listed below. Refer to the Encoding section for details of each value.

```
Port.NRZ
Port.NRZB
Port.NRZI
Port.NRZI_MARK
Port.FM0
Port.FM1
Port.MANCHESTER
Port.DIFF_BIPHASE_LEVEL
```

hdlc_address_filter

This integer attribute selects an 8-bit value used to filter received HDLC (SDLC) frames. Other protocols ignore this attribute. A value of `0xFF` (default) disables filtering and returns all frames to the application. Other values indicate the expected HDLC station address so that only frames with the broadcast address (`0xFF`) and the specified station address are returned to the application. Other frames are silently discarded.

internal_clock_rate

This integer attribute is the **data rate** used to setup the internal clock generator. When using the internal clock directly as a data clock it runs at this rate. When using clock recovery the internal clock runs at 8 or 16 times this rate to create a **reference clock** for sampling the receive data input and recovering a data clock.

The internal clock is generated by dividing the hardware base clock (default 14.7456MHz) by a 16-bit integer. If the specified rate is not a divisor of the base clock it is approximated by selecting the next slower rate that is a divisor. For details, refer to the clock generator section.

The API tries to use a 16x reference clock for maximum accuracy. If a 16x reference clock is not a divisor of the base clock, the API falls back to an 8x reference clock. If an x8 reference clock is not a divisor of the base clock then the specified data rate can't be precisely recovered.

The base clock can be adjusted on PCI Express and USB devices to allow precise internal clock rates when the specified rate (or reference clock) is not a divisor of the default 14.7456MHz base clock. Refer to the `set_fsynth_rate` method.

internal_loopback

This Boolean attribute enables internal loopback of data signals for diagnostic or development purposes. When set to True, the transmit data output is connected internally to the data input and data clocks are internally generated.

msb_first

This Boolean attribute selects the serial bit order. Most protocols must use least significant bit (LSB) first. Some protocols (RAW and TDM) may optionally use most significant bit (MSB) first by setting this attribute to `True`.

protocol

This attribute selects the serial protocol. Valid values are listed below. Refer to the Protocol section for details.

`Port.HDLC`	HDLC, also called SDLC
`Port.ASYNC`	Asynchronous and Isochronous
`Port.BISYNC`	Bi-synchronous
`Port.MONOSYNC`	Mono-synchronous
`Port.RAW`	Raw bit stream
`Port.TDM`	Time Division Multiplexing

receive_clock

This attribute selects the source of the receive clock.

`Port.TXC_INPUT`	TxC clock input
`Port.RXC_INPUT`	RxC clock input
`Port.INTERNAL`	internal clock generator
`Port.RECOVERED`	clock recovered from receive data input

receive_clock_invert

This attribute controls receive clock polarity. When set to default value of `False`, the receive clock uses normal polarity. When set to `True`, the receive clock uses inverted clock polarity. Endpoints must use the same clock polarity.

sync_pattern

This integer attribute specifies the synchronization pattern used for byte synchronous protocols (monosync or bisync) and should not be used for other protocols. The transmitter sends the sync pattern when idle (enabled and no data to send). The receiver searches for the sync pattern to determine byte alignment and returns data following a sync pattern to the application until the receiver is disabled or forced to the idle state so it searches for the next sync pattern.

tdm_sync_delay

This integer attribute specifies the number of clock cycles data is delayed from the start of a TDM sync pulse. Valid values are 0 (no delay), 1 and 2.

tdm_sync_short

This Boolean attribute specifies the length of a TDM sync pulse. `True` = one clock cycle, `False` = one slot length

tdm_sync_invert

This Boolean attribute specifies the polarity of the TDM sync signal. `False` = normal, `True` = inverted

tdm_frame_count

This integer attribute specifies the number of TDM frames returned to the application by the `read` method. Valid values are 1 to 256. The frame count time the fixed frame size must not exceed `Port.Defaults.max_data_size`. This allows better performance by reducing the number of `read` calls when the frame size is small and the data rate is high.

tdm_slot_count

This integer attribute specifies the number of TDM slots in a frame. Valid values are in the range 2 to 32 **or** 384.

tdm_slot_bits

This integer attribute specifies the number of bits in a TDM slot. Valid values are 8, 12, 16, 20, 24, 28 and 32.

transmit_clock

This attribute selects the source of the transmit clock.

`Port.TXC_INPUT`	TxC clock input
`Port.RXC_INPUT`	RxC clock input
`Port.INTERNAL`	internal clock generator
`Port.RECOVERED`	clock recovered from receive data input

transmit_clock_invert

This attribute controls transmit clock polarity. When set to default value of `False`, the transmit clock uses normal polarity. When set to `True`, the transmit clock uses inverted clock polarity. Endpoints must use the same clock polarity.

transmit_preamble_bits

This integer attribute selects the number of bits of preamble to prepend to sent data. A preamble is used to assist clock recovery from a data signal. Only synchronous protocols using clock recovery should set this attribute to a non-zero value. The only valid values are 8, 16, 32 and 64.

transmit_preamble_pattern

This 8-bit integer attribute selects the pattern that is prepended to sent data. A preamble is used to assist clock recovery from a data signal. Only synchronous protocols using clock recovery should set this attribute. The length of the preamble is selected by the `transmit_preamble_bits` attribute.

PROTOCOLS

This section provides an overview of supported serial protocols. Protocols define how framing, transparency, and timing are implemented.

Framing

Framing is a hardware mechanism to identify data boundaries and optionally check data integrity. HDLC uses flag patterns to mark the start and end of frames and CRC for integrity. TDM uses a sync signal to mark the start of fixed length frames. Asynchronous data uses start/stop bits on each character with parity for integrity. Bisync and monosync use sync patterns to indicate start of data and the application must detect end of data. Raw protocol has no hardware framing and the application is responsible for interpreting the stream of bits.

Transparency

Transparency is a mechanism to distinguish between data and non data patterns. HDLC uses zero insertion/deletion to distinguish between data and flags. HDLC is the only supported protocol that implements transparency at the hardware level.

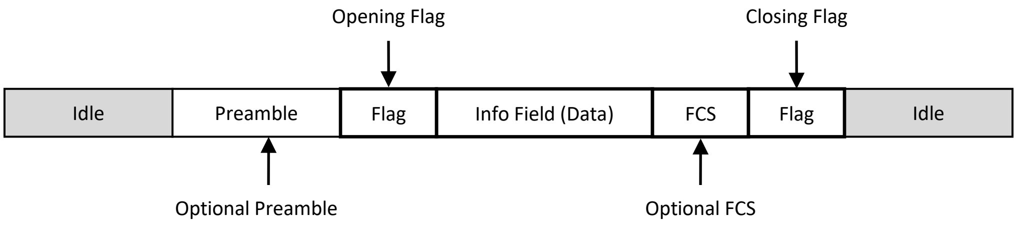
Timing

Serial communication requires a timing mechanism to coordinate data transfer. This can be an external clock signal, an internally generated clock, or a clock recovered from a data signal. The clock frequency determines the data transfer rate. The receiver and transmitter have separate clock configuration.

HDLC/SDLC

High Level Data Link Control (HDLC) is an international standard (ISO3309) based on SDLC (Synchronous Data Link Control), a protocol developed by IBM. This document uses HDLC to refer to both HDLC and SDLC.

Data is grouped into an information field of two or more bytes. The information field may be followed by an optional frame check sequence (FCS) such as CRC16 or CRC32. The FCS is calculated on the bits in the information field. The information field and FCS are framed with a non data pattern 01111110 (0x7e) called a flag. The collection of an opening flag, information field, FCS, and a closing flag is called a frame. A frame in progress can be aborted before the closing flag by sending a non data pattern called an abort, which is 7 or more consecutive ones. Aborted frames or frames with a FCS error should be ignored by the receiver.



HDLC Frame Format

An optional preamble may be sent before each frame. The preamble is useful for synchronizing a DPLL for clock recovery. The preamble pattern should be chosen to provide the maximum transitions for a given serial encoding standard. Refer to the DPLL section for details. Preambles are usually not used for applications with a separate data clock signal.

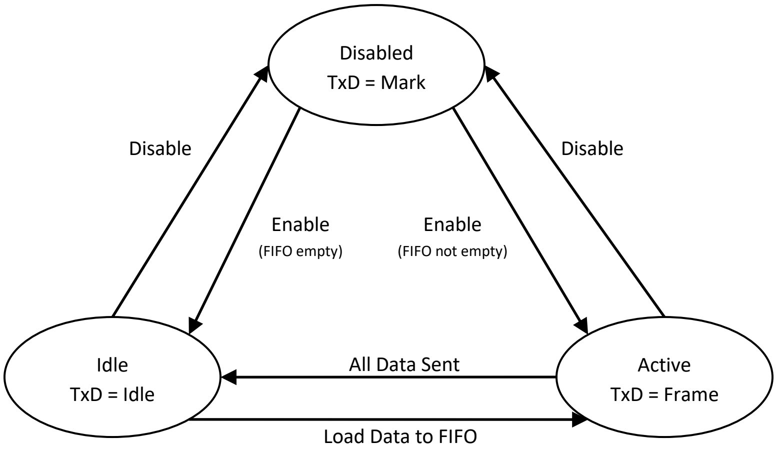
Leading bytes of the information field contain variable length address and control fields. The serial controller does not process the address or control fields, and treats the entire information field as data. Interpretation of the address and control fields is the responsibility of the device driver or application.

Data transparency is provided to distinguish between data and flag or abort patterns. This is accomplished with zero insertion and deletion. The controller automatically inserts a zero after any sequence of five consecutive ones in send data and automatically deletes a zero after any sequence of five consecutive ones in receive data. Zero insertion and deletion is only applied to the information field and FCS.

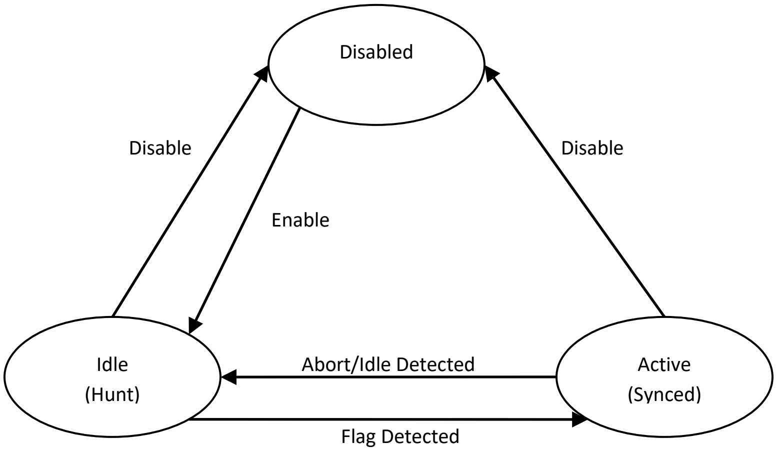
HDLC may use separate data clock signals or can recover data clocks from a data signal using DPLL (digital phase locked loop) clock recovery. There is one clock cycle per bit.

The HDLC transmitter has three states: disabled, idle, and active. The transmitter starts in the disabled state with the transmit data signal set to a constant mark. When software enables the transmitter with a bit in a control register the transmitter becomes idle. An idle transmitter sends a user configurable idle pattern, usually all ones or repeated flags. When software provides data to send, the transmitter becomes active and sends a frame containing the data. When the frame completes, the transmitter becomes idle. Software can disable the transmitter at any time using control bits in a register.

The HDLC receiver has three states: disabled, idle (hunt), and active (synced). The receiver start in the disabled state. When software enables the receiver with a bit in a control register, the receiver becomes idle and starts hunting for an opening flag. When a flag is detected, the receiver is active. An active receiver stores data between flags. When an abort sequence is detected, the receiver becomes idle. Software can disable the receiver at any time using control bits in a register.



HDLC Transmitter State Diagram

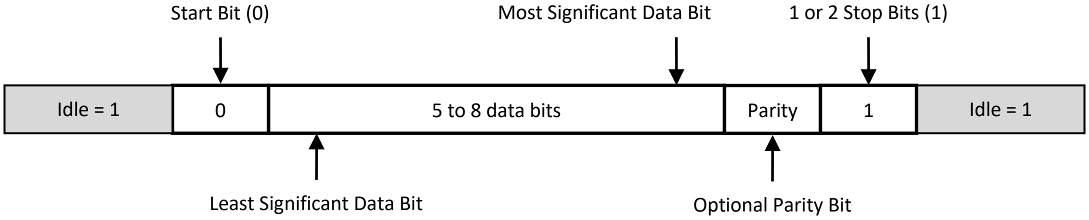


HDLC Receiver State Diagram

ASYNCHRONOUS

Asynchronous communication frames each character with a single start bit and one or two stop bits. Data length is configurable for 5 to 8 data bits per character. An optional parity bit (odd or even) is appended to the data. The idle line state is a logical 1. The start bit is a logical 0. Stop bits are a logical 1. Data is transmitted least significant bit first followed by the optional parity bit. The total character size is the combination of the start bit, data bits, optional parity bit, and stop bits. The total character size range is 7 to 12 bits. The number of data bits, stop bits, and use of parity must be configured in advance to match the settings of a remote station.

Data clocks are generated internally. The data rate must be chosen in advance to match that of a remote station. The receive clock runs at 8 or 16 times the selected data rate. This clock is used to sample the receive data line. The start bit is detected as the falling edge from the idle condition or the stop bits of the preceding character (1 to 0).



Asynchronous Character Format

ISOCHRONOUS

Isochronous is identical to asynchronous as described in the previous section, except a separate physical clock signal is used with the same frequency as the data rate (1x clock). Some models of SyncLink hardware (GT4e and USB) support the isochronous protocol by configuring the device for asynchronous communications with the `DataRate` member of the `MGSL_PARAMS` structure set to zero.

With `MGSL_MODE_ASYNC` and a `DataRate` of zero, the `Flags` and `ClockSpeed` fields of the `MGSL_PARAMS` structure control the clocking configuration. Usually a single clock supplied by a remote device drives the SyncLink transmitter and receiver. It is possible to use different clocks for transmit and receive or to generate the clock on the `AUXCLK` output by setting `ClockSpeed` to a non-zero value.

The benefit of isochronous is the data rate does not have to be identically configured in advance on both ends of the connection since timing is derived from a separate clock signal. The trade off is the added expense of the extra signal.

The following code fragment demonstrates isochronous using a single clock signal connected to the `RxC` input pin.

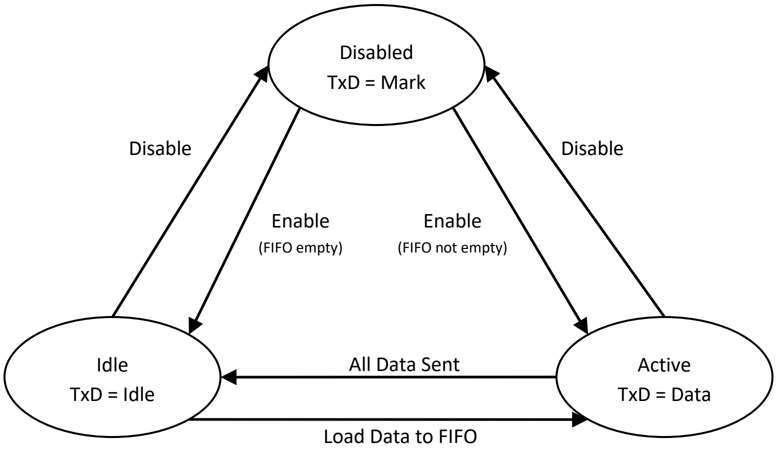
```
/*
 * Isochronous mode, format N-8-1
 * receive clock  = RxC input pin
 * transmit clock = RxC input pin
 * single clock from remote device connected to RxC input pin
 */
params.Mode = MGSL_MODE_ASYNC;
params.Loopback = 0;
params.Flags = HDLC_FLAG_RXC_RXCPIN + HDLC_FLAG_TXC_RXCPIN;
params.Encoding = HDLC_ENCODING_NRZ; /* ignored for isochronous */
params.ClockSpeed = 0;
params.CrcType = HDLC_CRC_NONE; /* ignored for isochronous */
params.DataBits = 8;
params.StopBits = 1;
params.Parity = ASYNC_PARITY_NONE;
params.DataRate = 0; /* selects isochronous */

/* set current device parameters */
rc = MgslSetParams(dev, &params);
```

RAW SYNCHRONOUS

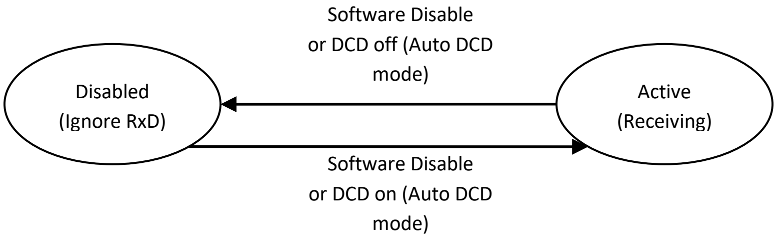
Raw synchronous operation performs no framing or synchronization. Data is sent bit for bit as supplied to the controller. Data is received bit for bit as seen on the receive data signal.

The raw transmitter has three states: disabled, idle, and active. The transmitter starts in the disabled state with the transmit data signal set to a constant mark. When software enables the transmitter with a bit in a control register the transmitter becomes idle. An idle transmitter sends a user configurable idle pattern. When software provides data to send, the transmitter becomes active and sends the data in an exact bit for bit representation. When no more data is available to send, the transmitter becomes idle. Software can disable the transmitter at any time using control bits in a register.



Raw Mode Transmitter State Diagram

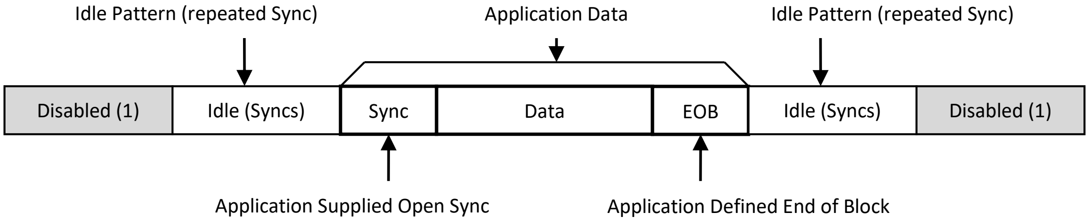
The raw receiver has two states: disabled, and active. The receiver start in the disabled state. When software enables the receiver with a bit in a control register, the receiver becomes active and starts storing receive data exactly bit for bit as seen on the receive data signal. Software can disable the receiver at any time using control bits in a register.



Raw Mode Receiver State Diagram

MONOSYNC AND BISYNC

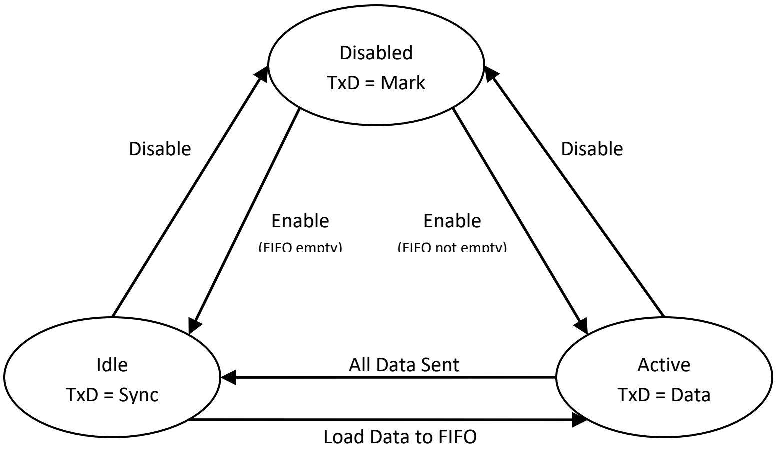
Monosync and Bisync operation is similar to raw synchronous operation. The difference is the receiver looks for an 8-bit (monosync) or 16-bit (bisync) pattern to signal synchronization and the following data is byte aligned to the synchronization pattern.


Idle Pattern (repeated Sync)
Application Data
Idle Pattern (repeated Sync)
Disabled (1)
Idle (Syncs)
Sync
Data
EOB
Idle (Syncs)
Disabled (1)
Application Supplied Open Sync
Application Defined End of Block

Monosync/Bisync Block Format

The monosync/bisync transmitter has three states: disabled, idle, and active. The transmitter starts in the disabled state with the transmit data signal set to a constant mark. When software enables the transmitter with a bit in a control register the transmitter becomes idle. An idle transmitter sends repeated sync patterns (8-bit for monosync and 16-bit for bisync). When software provides data to send, the transmitter becomes active and sends the data in an exact bit for bit representation. When no more data is available to send, the transmitter becomes idle. Software can disable the transmitter at any time using control bits in a register.

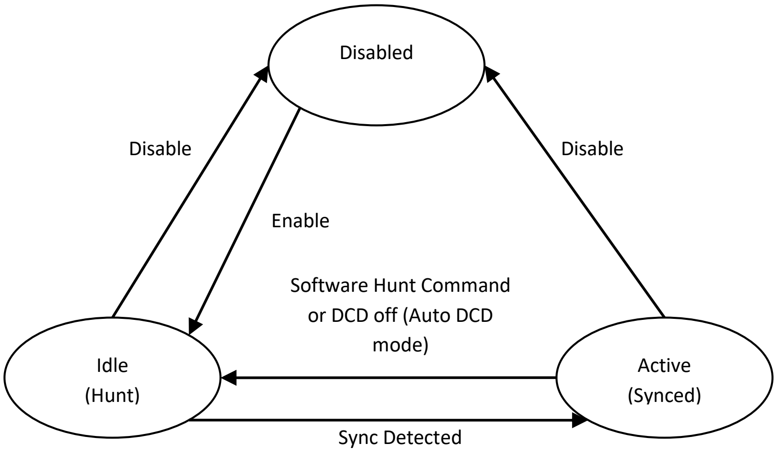
The transmitter does not automatically add a leading sync sequence to send data. Software must add the sync sequence manually to any data supplied to the transmitter.



Monosync/Bisync Transmitter State Diagram

The monosync/bisync receiver has three states: disabled, idle (hunt), and active (synced). The receiver starts in the disabled state. When software enables the receiver, the receiver becomes idle and starts hunting for the sync pattern (8-bit for monosync and 16-bit for bisync). When a sync pattern is detected, the receiver is active. An active receiver stores data bit for bit exactly as seen on the receive data signal. All data is byte aligned to the sync pattern. The receiver remains active until software disables the receiver or forces the receiver to idle/hunt.

Hardware does not detect the end of a data block. The end of block indication varies widely for monosync and bisync implementations and is the responsibility of software to detect. Typically an application enables the receiver and processes received data until the end of block condition is detected. The application then forces the receiver to idle/hunt.



Monosync/Bisync Receiver State Diagram

Time Division Multiplexing (TDM)

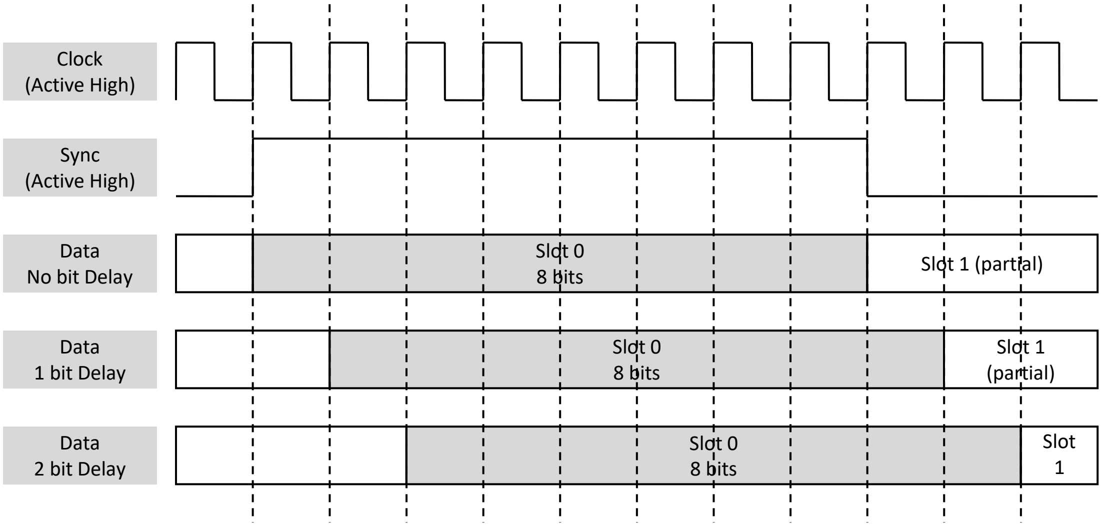
Note: TDM is only supported on SyncLink GT2e, GT4e and USB devices shipped February 2016 and later. Contact Microgate for information on updating these listed devices to support TDM if purchased before that date.

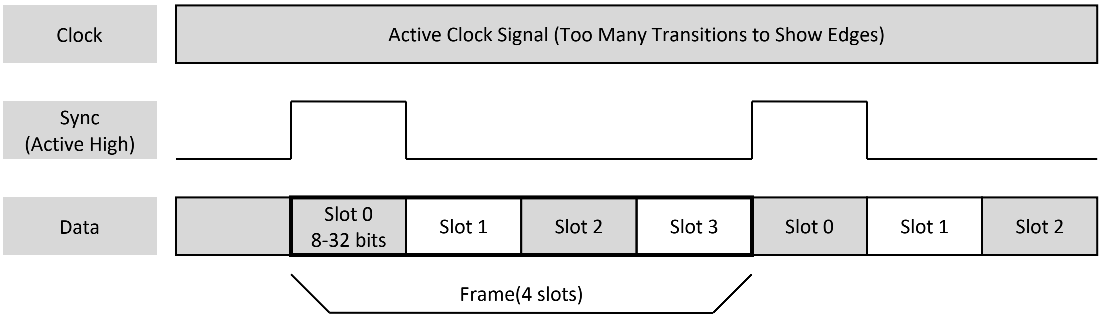
Time Division Multiplexing divides a serial signal by time into two or more **slots**. A **frame** is an ordered set of slots. Each slot represents a communication channel or data sample. The TDM implementation described here is compatible with the TDM mode of the multichannel audio serial port (McASP) of Texas Instruments controllers.

A data signal carries slots and frames. A slot contains 8 to 32 bits of data, in increments of 4 bits. A frame contains 2 to 32, or 384 slots. Slots within a frame are always contiguous. Multiple frames may or may not be contiguous. Data signal polarity (meaning of high or low signal) is selectable. The serial bit order is selectable, least significant bit (LSB) first or most significant bit (MSB) first. For example, a 12 bit slot with hexadecimal value 123 is sent as (first to last) 1100_0100_1000 for LSB first or 0001_0010_0011 for MSB first.

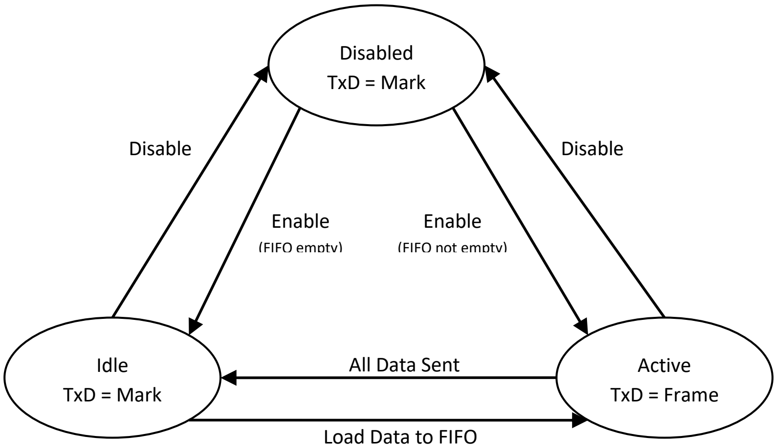
A clock signal provides timing information for the sync and data signals. Each clock cycle equals a single data bit. Selectable clock polarity determines which clock edge (rising or falling) changes output signals and which samples input signals.

A sync signal identifies the start of a frame. The sync signal is one bit or one slot in length. The first bit of a frame occurs with the start of the sync signal or may be delayed up to two clock cycles using the sync delay option. Sync signal polarity, active high or low, is selectable. Transmit sync length is selectable (bit or slot). The receiver automatically accepts either bit or slot length sync pulses.



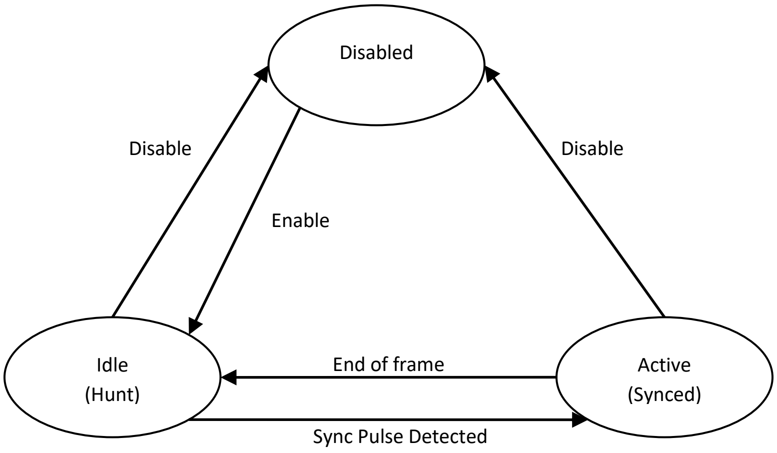


The TDM transmitter has three states: disabled, idle, and active and starts disabled. When enabled, the transmitter becomes idle. When supplied data, the transmitter becomes active. When all data is sent, the transmitter becomes idle. The transmitter may be disabled at any time, discarding unsent data.



TDM Transmitter State Diagram

The TDM receiver has three states: disabled, idle (hunt), and active (synced), and starts disabled. When enabled, the receiver becomes idle. When a sync is detected, the receiver is active. An active receiver stores data until a frame completes. When a frame completes, the receiver becomes idle. The receiver may be disabled at any time.



TDM Receiver State Diagram

TDM Signal Mapping

TDM Signal	Physical Serial Signal
Transmit Data (output)	TxD
Transmit Sync (output)	RTS
Transmit Clock (output)	AUXCLK
Receive Data (input)	RxD
Receive Sync (input)	DCD
Receive Clock (input)	RxC

ENCODING

Serial encoding converts a logical one or zero into a coded signal used on the physical connection between end points. Send data is encoded and receive data is decoded. The table below describes each encoding standard.

The NRZ family (NRZ, NRZB, NRZI-space, NRZI-mark) has zero or one signal transitions per bit located at the start of the bit cell. The receiver samples the data signal at the center of the bit cell. NRZ type encoding is usually used with synchronous protocols that have a separate data clock signal. NRZ has fewer transitions per bit than biphase encoding which allows higher data rates for a bandwidth limited physical connection.

Note: NRZI without a space or mark modifier is often used as short hand for NRZI-space.

The biphase family has one to two transitions per bit located at the beginning and center of the bit cell. The receiver samples the data signal at ¼ and ¾ of the bit cell length. Biphase encoding is usually used with DPLL clock recovery as it guarantees at least one transition per bit cell to keep the recovered clock synchronized.

Serial Encoding	
Name	**Description**
NRZ	TxD is logical value (no encoding)
NRZB	TxD is logical value inverted
NRZI-space	If logical value 0, invert TxD at start of bit
NRZI-mark	If logical value 1, invert TxD at start of bit
Biphase-mark (FM1)	Invert TxD at start of bit. If logical value 1, invert TxD at center of bit.
Biphase-space (FM0)	Invert TxD at start of bit. If logical value 0, invert TxD at center of bit.
Biphase-level (Manchester)	Set TxD to logical value at start of bit. Invert TxD at center of bit.
Differential Biphase-level	If logical value 0, Invert TxD at start of bit. Invert TxD at center of bit.

BAUD RATE GENERATOR

The serial controller has a functional unit called the baud rate generator (BRG). The BRG divides a clock input (the base block) by a 16 bit integer (divisor) to generate a clock output. The clock output can be used internally for transmit and receive timing, output on the AUXCLK serial output pin and as a reference clock for the DPLL described in the next section.

The default base clock on the GT and USB devices is 14.7456MHz. Devices can be special ordered with an alternate base clock frequency. The GT2e/GT4e cards and the USB device include a programmable frequency synthesizer (described in a later section) than can be used as the base clock.

The BRG divides the base clock by a 16-bit integer (0 to 65535) to generate the data clock.

$$f_{data} = \frac{f_{base}}{divisor+1} \qquad divisor = \left(\frac{f_{base}}{f_{data}}\right) - 1$$

Example 1:

Data Clock = 9600bps, Base Clock = 14.7456MHz, Divisor = (14745600/9600) – 1 = 1535
Since 1535 is an integer in the range 0 to 65535 the 9600bps clock can be generated exactly.

Example 2:
Data Clock = 1Mbps, Base Clock = 14.7456MHz, Divisor = (14745600/1000000) – 1 = 13.7456
Since13.7456 is NOT an integer, the 1Mbps clock cannot be generated exactly.

The default 14.7456MHz base clock supports exact data rates of 9600, 38400, 115200, etc.

CLOCK RECOVERY

Synchronous modes usually get transmit and receive timing from the transmit and receive clock inputs. Alternatively, timing can be recovered from a received data signal using a digital phased locked loop (DPLL). This requires the exact data rate to be known in advance and specified in `ClockSpeed` field of the `MGSL_PARAMS` structure.

Use these options in the `flags` field of the `MGSL_PARAMS` structure to use DPLL clock recovery:

For receiver:

`HDLC_FLAG_RXC_DPLL`	Receive clock comes from DPLL (recovered)

For transmitter (use only one):

`HDLC_FLAG_TXC_DPLL`	Transmit clock comes from DPLL (recovered)
`HDLC_FLAG_TXC_BRG`	Transmit clock comes from BRG (generated)

Usually the receiver uses the recovered clock and the transmitter the generated clock.

The BRG supplies the DPLL a reference clock that is 8 or 16 times greater than the data rate. Specify this setting in the `flags` field of the `MGSL_PARAMS` structure:

`HDLC_FLAG_DPLL_DIV8`	reference clock = 8 x data rate (highest max rate)
`HDLC_FLAG_DPLL_DIV16`	reference clock = 16 x data rate (better precision)

The BRG divides the base clock by a 16-bit integer (0 to 65535) to generate the DPLL reference clock.

$$f_{ref} = \frac{f_{base}}{divisor+1} \qquad divisor = \left(\frac{f_{base}}{f_{ref}}\right) - 1$$

Example 1:
Data Clock = 9600bps
Reference Clock = Data Clock * 16 = 153,600Hz
Base Clock = 14.7456MHz
Divisor = (14,745,600/153,600) – 1 = 95
Since 95 is an integer in the range 0 to 65535, the 153,600Hz reference clock can be generated exactly for recovering the 9600bps data clock.

Example 2:
Data Clock = 10,000bps
Reference Clock = Data Clock * 16 = 160,000Hz
Base Clock = 14.7456MHz
Divisor = (14,745,600/160,000) – 1 = 91.16
Since 91.16 is NOT an integer the 160,000Hz reference clock cannot be generated exactly.

If the reference clock can't be generated exactly, clock recovery can still work if the difference between the exact rate and the actual rate is small enough and sufficient data signal transitions are maintained. A 10% difference is acceptable if using a biphase encoding (FM or Manchester) that guarantees a data transition every clock cycle.

Custom base clocks can be ordered and installed at the factory to allow exact recovery of data rates not supported by the standard base clock of 14.7456MHz. Contact Microgate to determine which custom base clock is required for your needs.

Serial Encoding with DPLL

DPLL clock recover is usually used with a biphase encoding (FM or Manchester) which guarantees a data transition every bit. DPLL can be used with NRZI encoding when using SDLC/HDLC mode because that mode guarantees a transition every 6 bits.

Preamble with DPLL

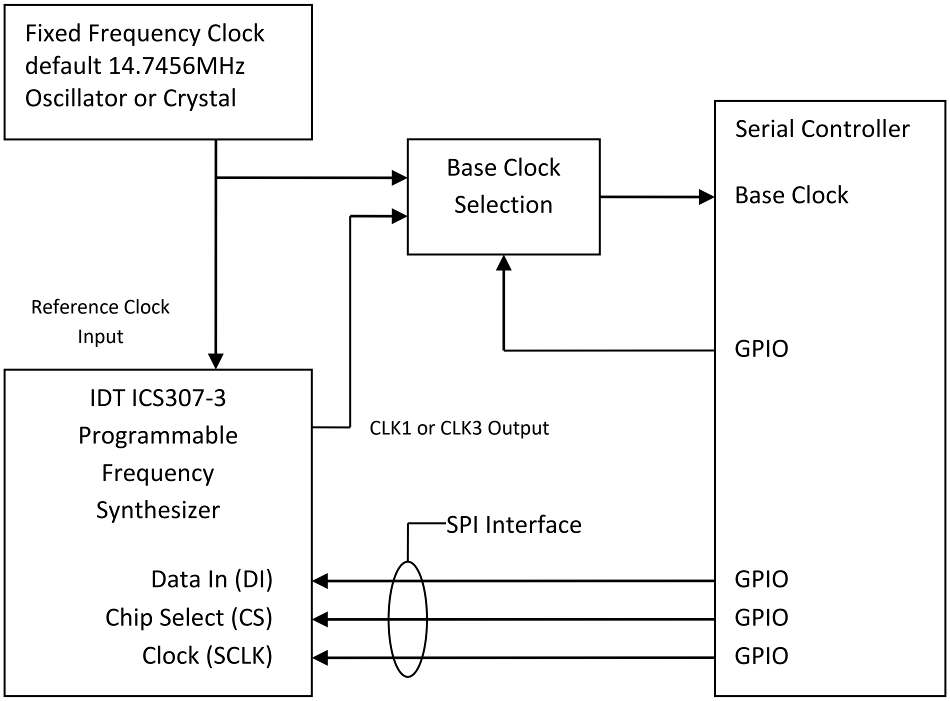
When a data signal is not continuously driven a preamble before each SDLC/HDLC frame is recommended to allow the DPLL to synchronize. Below is a list of suggested preamble patterns for different serial encodings:

Serial Encoding	Preamble Pattern
`HDLC_ENCODING_NRZI_SPACE (NRZI)`	`HDLC_PREAMBLE_PATTERN_ZEROS`
`HDLC_ENCODING_BIPHASE_MARK (FM1)`	`HDLC_PREAMBLE_PATTERN_ZEROS`
`HDLC_ENCODING_BIPHASE_SPACE (FM0)`	`HDLC_PREAMBLE_PATTERN_ONES`
`HDLC_ENCODING_BIPHASE_LEVEL (Manchester)`	`HDLC_PREAMBLE_PATTERN_01`

Frequency Synthesizer

Some models of SyncLink hardware have a programmable frequency synthesizer. The output of the synthesizer may be used as the base clock for the adapter. The synthesizer device is part number ICS307-3 manufactured by Integrated Device Technology (idt.com).

Below is an overview of the connection to the serial controller. The synthesizer reference clock is a fixed frequency clock source (oscillator or crystal). The synthesizer is programmed through an SPI interface connected to GPIO pins on the serial controller. Another GPIO pin selects between the fixed frequency clock source and the synthesizer. Refer to the Hardware User's Guide for your SyncLink device for the exact connections, GPIO assignments and type of fixed frequency clock source (oscillator or crystal).



The GPIO pins are controlled using the GPIO calls of the serial API as described in the GPIO section of this document. Sample code for programming the frequency synthesizer is included in the `mgapi\c\samples\fsynth` directory of the SDK.

Frequency synthesizer programming consists of a 132 bit word. For a description of the fields of the word, refer to the device datasheet for the ICS307-3 from idt.com. The 132 bit word is calculated by the Versaclock 2 Windows software provided by idt.com based on desired output values and error tolerances.

Note: Versions of Versaclock later than 2 do not support the ICS307-3 device. Contact idt.com for the older Versaclock 2 software required to program this device.

Values calculated by Versaclock can be copied to the Windows clipboard and then pasted into the sample `fsynth.c` program. The clipboard value requires manual formatting for use by the sample code. Below are instructions for calculating a value and using it with the sample code.

1. Run Versaclock 2 software and click **Select Part Number**

2. Select **ICS307-03-Clock** for SyncLink GT2e/GT4e cards or **ICS307-03-xtal** for SyncLink USB

3. Click the **Continue** button

4. Select **Manual Pin Assignment** from the **Options** menu

You should see three lines in the "Outputs" section labeled 8, 12, 14
SyncLink GT2e/GT4e uses pin 8 (CLK1) line
SyncLink USB uses pin 14 (CLK3) line

5. In **Ref freq (MHz)** edit box type: 14.7456

6. Leave **Vdd** pull down list set to 3.3V

7. On the appropriate line (pin 8 or pin 14) in **Outputs** section, enter two values:

Desired MHz : desired clock rate in MHz
Error ppm : use 100ppm (default for standard oscillators)

8. Click the **Calculate** button near bottom of window

9. Results appear in **Actual MHz** and **Error ppm** fields in Outputs section.

Verify the calculated error is within the specified range. If not, try generating a frequency that is a multiple of the desired rate and use the serial controller BRG to divide that for use.

10. Click the **Prog. word to Clipboard** button near the bottom of window.

11. Paste the result into the `fsynth.c` file near an existing table entry for the table associated with your SyncLink device type (`gt4e_table` for GT2e/GT4e or `usb_table` for USB).

Example: for 32.768MHz output on the SyncLink GT2e/GT4e card the clipboard value is:
`08001400D8A0000000000000001F9FE2`

12. Divide clip board value into 4 32-bit hex values of 8 digits each, with the 5th value a single digit:

`08001400D8A0000000000000001F9FE2` becomes
`08001400, D8A00000, 00000000, 0001F9FE, 2`

13. Format the values into a table of 5 32-bit values for use as a C language array initializer. The final digit is the most significant digit of a 32-bit value.

```
{0x08001400, 0xD8A00000, 0x00000000, 0x0001F9FE, 0x20000000}
```

14. Use the array initializer from the previous step to create a table entry for the desired frequency and place it in the table.

```
struct freq_table_entry gt4e_table[] =
{
        {12288000, {0x29BFDC00, 0x61200000, 0x00000000, 0x0000A5FF, 0xA0000000}},
        {14745600, {0x38003C05, 0x24200000, 0x00000000, 0x000057FF, 0xA0000000}},
        {16000000, {0x280CFC02, 0x64A00000, 0x00000000, 0x000307FD, 0x20000000}},
        {20000000, {0x00001403, 0xE0C00000, 0x00000000, 0x00045E02, 0xF0000000}},
        {30000000, {0x20267C05, 0x64C00000, 0x00000000, 0x00050603, 0x30000000}},
        {32000000, {0x21BFDC00, 0x5A400000, 0x00000000, 0x0004D206, 0x30000000}},
        {32768000, {0x08001400, 0xD8A00000, 0x00000000, 0x0001F9FE, 0x20000000}},
        {64000000, {0x21BFDC00, 0x12000000, 0x00000000, 0x000F5E14, 0xF0000000}},
        {0, {0, 0, 0, 0, 0}} /* final entry must have zero freq */
};
```

Once the frequency synthesizer has been programmed, it retains that value until reprogrammed or power is lost.

After programming the frequency synthesizer and selecting the synthesizer output as the base clock, use the serial API to inform the driver of the new value. The driver uses this value to calculate BRG and DPLL divisors.

```
rc = MgslSetOption(dev, MGSL_OPT_CLOCK_BASE_FREQ, 32768000);
```

This call needs to be made for every port on the adapter.

WINDOWS COMMUNICATION API (COM PORT MODE)

SyncLink serial devices support the Windows Communication API used with standard, asynchronous serial ports (COM ports). This allows SyncLink devices to work with programs that operate with standard COM ports using the Windows Communication API. Access the Windows Communication API using an alternate, API specific name for a SyncLink device as described in the following section. The Windows Communication API is described in the Windows Software Development Kit (SDK) available from Microsoft. Only asynchronous serial communications are available when using the Windows Communication API. HDLC or other synchronous protocols require the MicroGate Serial API.

Device Names

SyncLink devices have a name used with the MicroGate Serial API. Single port devices use MGHDLC**x**, where **x** is the device instance number (MGHDLC1, MGHDLC2, etc). Devices with multiple ports use MGMP**x**P**y**, where **x** is the device instance number and **y** is the port number. The MicroGate Serial API name is displayed in the Windows Device Manager.

SyncLink devices have an alternate name for use with the Windows Communication API. Single port devices use MGC**x**, where **x** is the device instance number (MGC1, MGC2, etc). Devices with multiple ports use MC**xy**, where **x** is the device instance number and **y** is the port number. MC11 = device 1, port 1. MC12 = device 1, port 2. The Windows Communication API device name does not appear in the Windows device manager, as it is an API specific alias of the actual device.

Both device names are created when a SyncLink device and drivers are installed. Normally, only one of the two device names should be in use, with the open name controlling the hardware. Both names may be used at the same time to auto-dial an SDLC link using asynchronous AT modem commands. When both names are open, the Windows Communication API name controls the hardware until DTR is negated in the Windows Communication API. This behavior allows an SDLC program to operate using the MicroGate Serial API, with an independent Windows Communication API program sending asynchronous AT commands to setup the link. When setup completes, the Windows Communication API program drops DTR or closes the port, returning control of the hardware to the SDLC program.

Sample code demonstrating the use of a SyncLink device with the Windows Communication API is located in the `c\samples\commapi` directory of the MicroGate Serial API development package.