## CONTENTS

Preface........................................................................................................................................6
Supported Versions .......................................................................................................................6
Required Knowledge....................................................................................................................6
Overview.....................................................................................................................................7
Software Installation ..................................................................................................................9
Software Removal ......................................................................................................................9
Hardware Configuration and Installation ..................................................................................10
Serial Interface Selection ........................................................................................................10
  RS-232......................................................................................................................................10
  V.35..........................................................................................................................................10
  RS-422/RS-485......................................................................................................................10
Hardware Installation ................................................................................................................10
  PCI Cards...............................................................................................................................11
  USB Adapter.........................................................................................................................11
Verifying Hardware and Driver Installation ............................................................................12
  Windows Device Manager....................................................................................................12
  SyncLink Device Properties...............................................................................................13
Trace Utility ..............................................................................................................................17
Device Instances and Names ....................................................................................................18
  Location Locked and Device Locked Instances ..................................................................18
  Serial API Names..................................................................................................................18
Removing Device Instances of Non-present Hardware ..........................................................19
Serial Device Connection .........................................................................................................20
  Standard Cables ..................................................................................................................20
  Custom Cables.....................................................................................................................21
Software Development Kit (SDK) Components ......................................................................22
  Run Time Files.....................................................................................................................22
Serial API Programming ..........................................................................................................23
  C/C++ ...................................................................................................................................23
  C# .......................................................................................................................................23
  Python..................................................................................................................................23
Open/Close Device .....................................................................................................................24
Configure Device ......................................................................................................................... 25
Receiving Data ............................................................................................................................. 27
Sending Data ............................................................................................................................... 29
Waiting for All Data Sent ............................................................................................................... 30
Asynchronous API Notification ................................................................................................... 31
API Function Reference .............................................................................................................. 32
MgslCancelGetTraceEvent ........................................................................................................... 33
MgslCancelGetWaitGpio ................................................................................................................ 34
MgslCancelReceive ...................................................................................................................... 35
MgslCancelTransmit ..................................................................................................................... 36
MgslCancelWaitEvent .................................................................................................................. 37
MgslClose ..................................................................................................................................... 38
MgslEnableReceiver .................................................................................................................... 39
MgslEnableTransmitter ................................................................................................................. 40
MgslEnumeratePorts ..................................................................................................................... 41
MgslGetAssignedResources ......................................................................................................... 42
MgslGetGpio ................................................................................................................................. 43
MgslGetOption .............................................................................................................................. 44
MgslGetParams ............................................................................................................................ 45
MgslGetPortConfigEx ................................................................................................................... 46
MgslGetSerialSignals .................................................................................................................. 47
MgslGetTraceEvent ..................................................................................................................... 48
MgslGetTraceLevel ....................................................................................................................... 49
MgslOpen ..................................................................................................................................... 50
MgslOpenByName .......................................................................................................................... 51
MgslOpenTraceHandle .................................................................................................................. 52
MgslPutTraceEvent ....................................................................................................................... 53
MgslReceive .................................................................................................................................. 54
MgslRead ...................................................................................................................................... 56
MgslReadWithStatus .................................................................................................................... 58
MgslResetTraceBuffers .............................................................................................................. 59
MgslSetIdleMode .......................................................................................................................... 60
MgslSetGpio .................................................................................................................................. 61
MgslSetOption .............................................................................................................................. 62
PREFACE

SUPPORTED VERSIONS

The serial API supports these 32-bit and 64-bit Windows versions:

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)

REQUIRED KNOWLEDGE

Developing with SyncLink devices on Windows requires the following knowledge:

1. C programming
2. Basic Windows administration
3. Serial communication details for target application
4. Reading supplied MicroGate documentation

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

This guide describes the use of MicroGate serial communication devices with the Windows operating system. Serial communication transfers data between systems similar to network devices like Ethernet and WiFi, but using different physical characteristics and protocols. There are many different types of serial communications and a successful connection requires compatible physical characteristics and protocols for all participating systems. The SyncLink family of serial devices may be configured for a variety of physical characteristics and protocols.

Correctly installing and configuring a SyncLink serial device requires a description of the physical characteristics and protocols required for the specific application. Physical characteristics include the electrical signal specification, connector types, signal pin assignments and cable wiring. Protocols include data signal format, clock signal source and a description of control and status signal functions if applicable. These details are application specific and must be obtained by the user of the SyncLink serial device.

Once application specific details are known the serial device can be configured and installed. For PCI serial cards this includes configuration of jumpers and installation into a compatible PCI system slot. USB to Serial converters do not have jumper settings and are connected to system USB ports with USB cables. Next, a device driver must be installed in the system. The device driver is software that provides an interface between the hardware and the operating system. The final step is configuring the application that uses the hardware.

Applications access the serial device through an application programming interface (API). Two levels of API are provided: a low level API to completely control detailed operation and a higher level (link layer) API that performs many protocol operations at the driver level.

![Diagram](image-url)
This document only covers the lower level serial API. The Link Layer API is documented in the Windows help file hdlcapi\docs\llapi.chm.

The remainder of this document describes the procedures listed above in detail. It is important to obtain all application specific details, documentation and specifications so the serial device can be correctly configured. Without these user provided details, correct operation cannot be achieved.
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, and the latest version can be downloaded from:

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

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

- 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
- rtk\tools\win32 32-bit support tools
- rtk\tools\win64 64-bit support tools

See supported Windows versions at start of this document for more details.

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

C:\hdlcap\rtk\drivers\>setup

After software installation, install hardware as described in the Hardware Installation section and reboot the system. Windows automatically 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:\hdlcap\rtk\drivers>setup /u
HARDWARE CONFIGURATION AND INSTALLATION

This section describes the physical configuration and installation of the serial device. This must match the application specific requirements provided by the user.

SERIAL INTERFACE SELECTION

The SyncLink serial device has a 25 pin connector for each serial port. This connector can be configured for different electrical specifications. The three options are: RS-232 (single ended signals), V.35 (combined single ended and differential signals) and RS-422/RS-485 (differential signals). The selection of interface type is controlled by jumper settings on PCI cards and through software for the USB to serial device.

RS-232
RS-232 (also called EIA-232) is a specification that defines single ended signals (one wire per signal) for use in low data rate applications, usually less than 120Kbps. This is common in legacy applications such as connecting to analog phone line MODEMs. The standard defines pin assignments on a 25 pin connector.

V.35
V.35 is a specification that defines a combination of single ended signals (one wire per signal) and differential signals (two wires per signal). Data and clock signals are differential for high speed. Control and status signals are single ended. The standard defines pin assignments on a 34-pin “block” connector. An adapter cable available from MicroGate is required to convert the device’s 25 pin connector to a standard V.35 34 pin connector.

RS-422/RS-485
RS-422 (also called EIA-422) is an electrical specification for differential signals (two wires per signal). RS-485 is an improved specification that is compatible with RS-422. Neither standard specifies pin assignments or a connector. The SyncLink card assigns RS-422/485 pins using the RS-530 specification for a 25 pin connector. MicroGate offers adapter cables to convert the RS-530 pin assignments to RS-449 (37 pins) or X.21 (15 pins).

HARDWARE INSTALLATION

The exact hardware installation procedure depends on the hardware and the system type. Some hardware plugs into external system ports, other hardware is installed into internal expansion slots on the system. Refer to the hardware user’s guide (PDF) that came with your hardware for detailed specification and configuration information. Hardware user’s guides are also available at www.micogate.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 jumpers (RS232,V.35,RS422) are correctly installed.
- 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.
VERIFYING HARDWARE AND DRIVER 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.

Starting Windows Device Manager

The Windows device manager can be started from a command prompt or through the Windows Control Panel. The device manager must be run with Administrative privilege.

Windows XP Control Panel

For Windows XP, click the Start button and select Control Panel. Double click the System icon. Select the Hardware tab then click the Device Manager button.

Windows Vista, Windows 7 Control Panel

For Windows Vista and Windows 7, click the Start button and select Control Panel. Type “device manager” into the search box in the upper right of the Control Panel window. Then select the Device Manager item in the search results.

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.
ADVANCED TAB

The advanced tab contains device settings that take effect at system startup. Not all settings are available for all SyncLink devices.

![Advanced Tab Screenshot]

The **Device** pull down list includes one entry for each port on the device. The other settings apply to the currently selected port.

**Max Frame Size**

This setting chooses the largest HDLC frame or block of data that can be sent or received in a single API call. 4096 is the best choice for most applications. HDLC frames larger than 4K bytes should only be used with the CRC-32 frame check.

**Serial Interface**

Select the serial interface electrical specification (RS232, V.35, RS422, etc). This option is only available on the SyncLink PCMCIA and USB devices. PCI 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 (USB Only)**

When checked, this option disables the input termination on the USB adapter serial interface when differential modes are selected (RS422/RS530/V.35/X.21). Normally, differential inputs are terminated with 120 Ohms.

**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 loopback of data.

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**

A trace utility is provided in the run time kit for recording events during a communications session for diagnostics purposes. The recorded events are saved to a selectable trace file that can be supplied to support staff. The program is located in the RTK at the location:

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

Run the program from a command line or the Windows explorer. The user must have administrative privilege to run the trace utility. The running trace program will appear as shown below.

1. Enter the desired output file name into the **Trace File** edit field.
2. Click the **Set All** button to enable all trace levels.
3. Select the desired 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.

Supply the output file to support staff.
**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 DEVICE CONNECTION

Serial devices are either DTE (data terminal equipment) or DCE (data circuit-terminating equipment). A DTE device connects directly to a DCE device. Connecting two DTE devices requires a special cross over cable or intermediate device called a null MODEM. A DTE is usually a system consuming and generating data. A DCE is a device that converts data into a format suitable for a communications medium like a phone line or radio link. The SyncLink serial device is a DTE with a DSUB 25 pin male connector.

STANDARD CABLES

A cable connects the SyncLink device to another device. If the attached device is a DCE using a standard connector (RS-232, V.35, RS-530, etc) then use a standard cable. In some cases, an adapter cable must be purchased from MicroGate to convert the DB-25M connector to the appropriate standard connector. The adapter cable can either plug directly to the attached device or can be used with a standard cable to increase the cable length if needed.
CUSTOM CABLES
If the attached device does not use a standard connector, such as custom control and measurement devices, then use the documentation for the SyncLink and custom devices to create an appropriate custom cable. Pinouts, electrical specification and configuration options are contained in the hardware user’s manual (PDF) for your SyncLink device.

The first step in specifying a custom cable is determining which signals are required. The SyncLink device implements a full set of data, clock, control and status signals. Many custom devices only require data and clock signals. Some applications use clock recovery and only have data signals. Make a list of signals that are required by the custom device and find the equivalent signal on the SyncLink device. SyncLink signals that are not required for an application can be left unconnected. A signal ground connection is usually required between endpoints, except for differential signals where the common mode voltage between endpoints can be guaranteed to meet electrical specifications without a signal ground connection. If in doubt, include a signal ground connection. A chassis/earth ground connection is recommended, which should be tied to the cable shield.

If the attached device uses differential signals (RS-422/RS-485) with two conductors per signal, you must verify the polarity of the signals. Each differential signal on the SyncLink device is designated A/+ and B/-.. Documentation for other devices may use A and B or + and – for each of the two conductors of a signal. Usually, you connect A/+ to A/+, B/- to B/-. Some manufacturers use designations with an opposite sense than MicroGate. For these devices, connect A/+ to B/-. Reversing the conductors of a signal changes the polarity. The SyncLink hardware user’s manual has detailed descriptions of relative voltages and transitions required for signals. If you lack such documentation for the attached device, try normal polarity first and if that does not work try reversing the polarity. Polarity is never a problem when using standard connectors and cables as pin assignments are contained in the specifications for the connector.
SOFTWARE DEVELOPMENT KIT (SDK) COMPONENTS

This section describes contents of the software development kit (SDK). The serial API is a set of C language functions called by an application. Any language supporting the Windows “standard call” calling convention can be used. API wrappers are provided for C# and Python 3 in the hdlcapipsrc directory. The two main development files are:

- hdlcapiinclude\mghdlc.h  C language header file defining calls and structures
- hdlcapi\lib\mghdlc.lib  C language import library for linking 32-bit applications
- hdlcapi\lib\x64\mghdlc.lib  C language import library for linking 64-bit applications

A C language application includes the header file so API calls and structures can be referenced. The compiled application is linked against the import library to resolve API calls. 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”. These files are only used for development and are not necessary for installation on an end user system.

RUN TIME FILES

Files required to support hardware and applications on an end user system (run time files) are contained in the rtk directory as described in the software installation section. These files should be distributed with the application and hardware. The complete list of run time files is subject to change.

The following programs are included in the rtk\tools directory for testing and information collection purposes. These files are not copied as part of the hardware driver installation procedure. The application writer is responsible for installing them as needed on end user systems.

- mgs1trc.exe  trace utility to record device events and API calls to a file
- mgs1trc.chm  Windows help for trace utility
- mgdump.exe  saves device configuration to a text file
SERIAL API PROGRAMMING

This section describes control of a serial device by a user mode application using serial API calls. The API is written in the C language. This document contains a reference for the C API. Sample code is provided for C/C++, C# and Python 3 programming languages. A working knowledge of the target language in a Windows environment is required.

Use the sample code as a starting point for writing your own code. Read the README.TXT files and source code for details on building and using the sample code. Most samples loopback data on a single device or send from one device to another 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 an aid to developing an application.

C/C++

Required API definitions are contained in the mghdlc.h header file that must be included in all applications. Sample Visual Studio 2010 projects are provided for each serial protocol.

hdlicapi\src\hdlc          synchronous HDLC/SDLC
hdlicapi\src\raw           synchronous raw
hdlicapi\src\async         asynchronous
hdlicapi\src\commapi       using Windows asynchronous COM API
hdlicapi\src\bisync        byte synchronous (BISYNC/MONOSYNC)
hdlicapi\src\2wire         HDLC in 2 wire half duplex mode
hdlicapi\src\fsynth        programming GT4e/USB frequency synthesizer
hdlicapi\src\gpio          control and monitor GPIO signals

C#

A wrapper file, MgApi.cs, provides C# versions of the C API and adds C# specific helper functions. Examine MgApi.cs and the application code for details. A sample Visual Studio 2010 project using the wrapper is provided.

hdlicapi\src\csharp         synchronous HDLC/SDLC

Python

A Python wrapper, mgapi.py, provides Python versions of the C API and adds Python specific helper functions. Examine mgapi.py and application examples for details. The wrapper follows the naming conventions of the C API. This violates Python style standards but allows easy use of the C API reference documentation. Sample applications are provided for each serial protocol.

The wrapper uses Python 3 and is incompatible with Python 2. A Python 3 installation is required to run the Python sample code. The wrapper must be imported into a Python application file with the following line:
import mgapi

The wrapper 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 wrapper package can be installed using the Python pip installation program in a command prompt with Administrator privilege:

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

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdlcapi\src\python\mgapi.py</td>
<td>Python wrapper</td>
</tr>
<tr>
<td>hdlcapi\src\python\mgapi.tar.gz</td>
<td>Installable Python wrapper package</td>
</tr>
<tr>
<td>hdlcapi\src\python\examples</td>
<td>Python example applications</td>
</tr>
</tbody>
</table>

A Python application is run as shown below for sending HDLC data on device mghdlc1:

```
C:\hdlcapi\src\python\examples\hdlc>python send.py mghdlc1
```

**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.

```c
/* 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.

```c
/* 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.

```c
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.

```c
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 Poll 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.
API FUNCTION REFERENCE

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**

```c
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.
## MgsEnableReceiver

ULONG MgsEnableReceiver(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.

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

/* disable receiver (ignore receive data input) */
rc = MgsEnableReceiver(dev, 0);
if (rc != ERROR_SUCCESS)
    /* process error */

/* force receiver to idle state (look for sync pattern) */
rc = MgsEnableReceiver(dev, 2);
if (rc != ERROR_SUCCESS)
    /* process error */
```
**MgslEnableTransmitter**

```c
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.

```c
/* 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**

```c
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.

```c
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**

```c
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.

```c
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.

```c
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.

```c
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.

```c
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.

The **mgs1trc.exe** trace utility source code is located in **hdlcapi\src\trace**.

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.

```c
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)`

```c
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**

```c
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.

```c
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**

```c
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.

```c
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.

```c
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 */
/* 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**

```c
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*.

```c
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

```c
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.

```c
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.

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

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>returned port handle</td>
</tr>
<tr>
<td>idle</td>
<td>idle or sync pattern</td>
</tr>
</tbody>
</table>

**Return Value**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_SUCCESS</td>
<td>call success</td>
</tr>
<tr>
<td>ERROR_INVALID_HANDLE</td>
<td>port handle is invalid</td>
</tr>
</tbody>
</table>

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.

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

/* set 16-bit sync pattern for bisync mode */
rc = MgslSetIdleMode(dev, HDLC_TXIDLE_CUSTOM16 | (syn2 << 8) | syn1);
if (rc != ERROR_SUCCESS)
    /* call error */

/* set 8-bit sync pattern for monosync mode */
rc = MgslSetIdleMode(dev, HDLC_TXIDLE_CUSTOM8 | 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.

typedef struct _GPIO_DESC
{
    UINT state; /* 0 (low) or 1 (high) */
    UINT smask; /* 0=ignore bit in state, 1=set bit in state*/
    UINT dir; /* 0=input, 1=output */
    UINT dmask; /* 0=ignore bit in dir, 1=set bit in dir */
} GPIO_DESC;

Sample code:

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 */
**MgsSetOption**

```c
ULONG MgsSetOption(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 `MgsSetPortConfigEx`. Settings that have an equivalent `MgsSetPortConfigEx` value are noted below.

Most `MgsSetOption` settings take effect immediately, some are deferred until the next call to `MgsSetParams`. 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 `MgsSetParams` 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**
- **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 PCMCIA hardware.
  - Set this option at driver load time with `MgsSetPortConfigEx`.

- **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 `MgsSetParams`.

- **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 hardware. PCI cards use jumpers and DIP switches to enable/disable termination.
  - Set this option at driver load time with `MgsSetPortConfigEx`.
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 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. Source code for the trace utility is provided in the hdlcapip/src\trace directory.

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
MGLTRANSMIT

ULONG MsglTransmit(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 MsglTransmit 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 MsglCancelTransmit 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 MsglGetOption 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 MsglTransmit.
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.

```c
int rc;

/* check for all buffered data sent */
r = 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 Poll 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**

```c
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.

```c
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.
API STRUCTURE REFERENCE

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 STRUCTURE**

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

```c
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 STRUCTURE**

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.

```c
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 STRUCTURE

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

```c
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 - COMMUNICATIONS MODE/PROTOCOL

The mode field of the MGSL_PARAMS structure specifies the communications mode 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
<table>
<thead>
<tr>
<th>MGSL_MODE_RAW</th>
<th>bit synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no hardware framing</td>
</tr>
<tr>
<td></td>
<td>no hardware synchronization</td>
</tr>
<tr>
<td></td>
<td>no hardware transparency</td>
</tr>
<tr>
<td>MGSL_MODE_BISYNC</td>
<td>byte synchronous</td>
</tr>
<tr>
<td></td>
<td>16 bit hardware synchronization</td>
</tr>
<tr>
<td></td>
<td>no hardware framing</td>
</tr>
<tr>
<td></td>
<td>no hardware transparency</td>
</tr>
<tr>
<td>MGSL_MODE_MONOSYNC</td>
<td>byte synchronous</td>
</tr>
<tr>
<td></td>
<td>8 bit hardware synchronization</td>
</tr>
<tr>
<td></td>
<td>no hardware framing</td>
</tr>
<tr>
<td></td>
<td>no hardware transparency</td>
</tr>
<tr>
<td>MGSL_MODE_TDM</td>
<td>SyncLink GT2e, GT4e, USB Only (Feb 2016 or later)</td>
</tr>
<tr>
<td></td>
<td>byte synchronous</td>
</tr>
<tr>
<td></td>
<td>external control signal synchronization</td>
</tr>
<tr>
<td></td>
<td>external clock signal</td>
</tr>
<tr>
<td></td>
<td>framing defined by fixed size data grouping</td>
</tr>
</tbody>
</table>

**LOOPBACK – ENABLE/DISABLE INTERNAL LOOPBACK MODE**

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 – PROTOCOL OPTIONS**

The *Flags* field of the `MGSL_PARAMS` structure specified miscellaneous protocol options using `HDLC_FLAG_XXX` macros. The `HDLC_FLAG` prefix is used for historical reasons, but unless otherwise specified these flags apply to all modes.

**Receive Clock Source Flags**

The serial receiver requires a clock for operation. `HDLC_FLAG_RXC_xxx` macros select the source of the receive clock. The clock can be generated internally, recovered from a data signal or supplied by an external device on one of the clock input pins. These options are mutually exclusive, the receiver can have only one clock source.

- **HDLC_FLAG_RXC_DPLL**  
  Receive clock is recovered from the receive data signal using the DPLL (digital phase locked loop). The *ClockSpeed* member of the `MGSL_PARAMS` structure specifies the expected data rate.

- **HDLC_FLAG_RXC_BRG**  
  Receive clock is generated with baud rate generator (BRG) at the speed specified in the *ClockSpeed* member of the `MGSL_PARAMS` structure.
HDLC_FLAG_RXC_RXCPIN  Receive clock is supplied by an external device on the RxC input pin.

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

**Transmit Clock Source**

The serial transmitter requires a clock for operation. HDLC_FLAG_TXC_xxx macros select the source of the transmit clock. The clock can be generated internally, recovered from a data signal or supplied by an external device on one of the clock input pins. These options are mutually exclusive, the transmitter can have only one clock source.

HDLC_FLAG_TXC_DPLL  Transmit clock is recovered from the receive data signal using the DPLL (digital phase locked loop). The ClockSpeed member of the MGSL_PARAMS structure specifies the expected data rate.

HDLC_FLAG_TXC_BRG  Transmit clock is generated with baud rate generator (BRG) at the speed specified in the ClockSpeed member of the MGSL_PARAMS structure.

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

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

**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 – SELECT REPRESENTATION OF LOGICAL 1 OR 0 DATA
Specify physical data signal representation of logical 1 or 0. Equivalent encoding names are shown in parenthesis. BIPHASE encodings are usually used with DPLL clock recovery because they guarantee one transition per bit. The encoding must match the application specific requirements. The levels specified below are the data signal from the serial controller, but before the line drivers which invert the signal.

<table>
<thead>
<tr>
<th>Encoding Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDLC_ENCODING_NRZ</td>
<td>NRZ (NRZ-L) unencoded data signal. high = 1, low = 0</td>
</tr>
<tr>
<td>HDLC_ENCODING_NRZB</td>
<td>NRZB inverted data signal. high = 0, low = 1</td>
</tr>
<tr>
<td>HDLC_ENCODING_NRZI_MARK</td>
<td>(NRZ-M) invert at start of bit if 1</td>
</tr>
<tr>
<td>HDLC_ENCODING_NRZI_SPACE</td>
<td>(NRZI or NRZ-S) invert at start of bit if 0</td>
</tr>
<tr>
<td>HDLC_ENCODING_BIPHASE_MARK</td>
<td>(FM1) invert at start of bit, invert at bit center if 1</td>
</tr>
<tr>
<td>HDLC_ENCODING_BIPHASE_SPACE</td>
<td>(FM0) invert at start of bit, invert at bit center if 0</td>
</tr>
<tr>
<td>HDLC_ENCODING_BIPHASE_LEVEL</td>
<td>(Manchester) start of bit: high=1, low=0, invert at bit center</td>
</tr>
<tr>
<td>HDLC_ENCODING_DIFF_BIPHASE_LEVEL</td>
<td>invert at start of bit if 1, invert at bit center</td>
</tr>
</tbody>
</table>

CLOCK SPEED – GENERATED OR RECOVERED DATA RATE
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:

\[
\text{divisor} = (\text{base clock/data rate}) - 1
\]

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 Mgs1SetOption(MGSL_OPT_CLOCK_BASE_FREQ).

ADDR – SDLC/HDLC ADDRESS FILTER
The Addr member of the MGSL_PARAMS structure controls filtering of received SDLC/HDLC frames based on an eight bit address field. 0xFF = return all frames (no filtering), otherwise discard received HDLC frames with addresses other than 0xFF (broadcast) or Addr value.
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.

<table>
<thead>
<tr>
<th>CRC Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDLC_CRC_NONE</td>
<td>Don't send CRCs on transmit, don't check CRCs on receive.</td>
</tr>
<tr>
<td>HDLC_CRC_16_CCITT</td>
<td>16 bit CRC Polynomial: $x^{16} + x^{12} + x^5 + 1$</td>
</tr>
<tr>
<td>HDLC_CRC_32_CCITT</td>
<td>32 bit CRC Polynomial: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$</td>
</tr>
<tr>
<td>HDLC_CRC_RETURN_CRC</td>
<td>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</td>
</tr>
<tr>
<td>HDLC_CRC_RETURN_CRCERR_FRAME</td>
<td>Combine with HDLC_CRC_16_CCITT or HDLC_CRC_32_CCITT to return received frames with CRC errors to application</td>
</tr>
</tbody>
</table>

**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.

```c
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.

```c
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.

```c
params.CrcType = HDLC_CRC_16_CCITT | HDLC_CRC_RETURN_CRCERR_FRAME | HDLC_CRC_RETURN_CRC;
```
**PREAMBLEPATTERN – PREAMBLE PATTERN SELECTION**

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.

<table>
<thead>
<tr>
<th>Preamble Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDLC_PREAMBLE_PATTERN_NONE</td>
<td>no preamble (preamble_length ignored)</td>
</tr>
<tr>
<td>HDLC_PREAMBLE_PATTERN_ZEROS</td>
<td>all zeroes</td>
</tr>
<tr>
<td>HDLC_PREAMBLE_PATTERN_FLAGS</td>
<td>all flags</td>
</tr>
<tr>
<td>HDLC_PREAMBLE_PATTERN_10</td>
<td>alternating 1’s and 0’s</td>
</tr>
<tr>
<td>HDLC_PREAMBLE_PATTERN_01</td>
<td>alternating 0’s and 1’s</td>
</tr>
<tr>
<td>HDLC_PREAMBLE_PATTERN_ONES</td>
<td>all ones</td>
</tr>
</tbody>
</table>

**PREAMBLELENGTH – PREAMBLE LENGTH SELECTION**

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 – ASYNCHRONOUS DATA RATE**

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 – ASYNCHRONOUS DATA BITS PER CHARACTER**

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 – ASYNCHRONOUS STOP BITS PER CHARACTER**

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 – ASYNCHRONOUS CHARACTER 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 STRUCTURE**

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.

```c
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 STRUCTURE**

This structure is used with the MgslReceive API call.

```c
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:

```c
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.

```c
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.

<table>
<thead>
<tr>
<th><code>Status</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RxStatus_OK</code></td>
<td>data returned in <code>DataBuffer</code>, <code>DataLength</code> set to count of bytes</td>
</tr>
<tr>
<td><code>RxStatus_CrcError</code></td>
<td>SDLC/HDLC frame with a CRC error was received and discarded</td>
</tr>
<tr>
<td><code>RxStatus_FifoOverrun</code></td>
<td>hardware FIFO was full when data was received causing data loss</td>
</tr>
<tr>
<td><code>RxStatus_ShortFrame</code></td>
<td>SDLC/HDLC frame with only one data byte was received and discarded</td>
</tr>
<tr>
<td><code>RxStatus_Abort</code></td>
<td>SDLC/HDLC abort or idle sequence was received</td>
</tr>
<tr>
<td><code>RxStatus_BufferOverrun</code></td>
<td>API buffers were full when data was received causing data loss</td>
</tr>
<tr>
<td><code>RxStatus_Cancel</code></td>
<td>receive request was cancelled by user before normal completion</td>
</tr>
<tr>
<td><code>RxStatus_BufferTooSmall</code></td>
<td>SDLC/HDLC frame larger than the receive request buffer was received</td>
</tr>
</tbody>
</table>
GENERAL PURPOSE I/O

Some models of SyncLink adapter have general purpose input/output (GPIO) signals that can be controlled and monitored. This is done with API calls as described below.

Each adapter may have up to 32 signals, each of which may be a dedicated input, a dedicated output, or a configurable input/output. The exact number and configuration of these signals varies with the specific adapter and are documented in the associated Hardware User’s Guide.

If an adapter does not support GPIO, the following functions return ERROR_INVALID_PARAMETER.

All general purpose I/O operations use the following structure:

```c
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

**Mgs1SetGpio** sets the current state and direction configuration of one or more GPIO signals.

**Mgs1GetGpio** gets the current state and direction configuration of all GPIO signals.

**Mgs1WaitGpio** waits for specific GPIO signal states.
SERIAL PROTOCOL OVERVIEW

This section provides an overview of supported serial protocols to assist in selecting the correct configuration.

FRAMING AND TRANSPARENCY

Framing is a mechanism to identify boundaries of a data grouping such as a separate control signal or non data patterns in a data signal. Transparency is a mechanism to distinguish between data and non data patterns. Examples of framing are the flag non data pattern for HDLC or start/stop bits in asynchronous mode. Examples of transparency are zero insertion and deletion for HDLC or escape characters for bisync mode.

SYNCHRONIZATION

Synchronization is a mechanism to indicate the presence of a data signal. This is closely related to framing. HDLC uses flags for framing (start and end of frame markers) and synchronization (continuous flags indicate remote device is active but not sending data). Bisync uses a 16 bit non data pattern to indicate synchronization but may use different non data patterns to indicate the start and end of a block of data.

TIMING AND CLOCK SOURCE

Serial communications requires a timing mechanism to coordinate data transfer. This can be a separate clock signal, an internally generated clock, or a clock recovered from a data signal. The clock frequency determines the data transfer rate.

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.

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 Receiver State Diagram

- Disabled
  - Disable
  - Enable
- Idle (Hunt)
  - Abort/Idle Detected
  - Flag Detected
- Active (Synced)
  - Disable
  - Flag Detected
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).

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**

[Diagram of Raw Mode Transmitter State Diagram]
The raw receiver has two states: disabled, and active. The receiver starts 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.

**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.

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.
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 (RCR[1]=1), 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 (RCR[1]=0) or forces the receiver to idle/hunt (RCR[3]=1).

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 by setting RCR[3] to 1.
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.
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](image)

### TDM SIGNAL MAPPING

<table>
<thead>
<tr>
<th>TDM Signal</th>
<th>Physical Serial Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmit Data (output)</td>
<td>TxD</td>
</tr>
<tr>
<td>Transmit Sync (output)</td>
<td>RTS</td>
</tr>
<tr>
<td>Transmit Clock (output)</td>
<td>AUXCLK</td>
</tr>
<tr>
<td>Receive Data (input)</td>
<td>RxD</td>
</tr>
<tr>
<td>Receive Sync (input)</td>
<td>DCD</td>
</tr>
<tr>
<td>Receive Clock (input)</td>
<td>RxC</td>
</tr>
</tbody>
</table>

### USING TDM PROTOCOL WITH SERIAL API

The directory `src\sample-tdm` contains sample TDM code. A port must be configured with application dependent requirements matching the attached device characteristics. Configuration must be performed in the following order.

**Configuration, Step 1: Call MgsSetOption with MGSL_OPT_TDM**

This call specifies multiple TDM specific options using API header `(mghdlc.h)` macros. Options apply to both transmitter and receiver unless noted otherwise. Macro and setting descriptions follow the sample code.

```c
unsigned int tdm_options =
    TDM_SYNC_DELAY_NONE | TDM_TX_SYNC_WIDTH_SLOT |
    TDM_SYNC_POLARITY_NORMAL | TDM_RX_FRAME_COUNT(10) | |
    TDM_SLOT_COUNT(4) | TDM_SLOT_SIZE_32BITS; 
MgsSetOption(handle, MGSL_OPT_TDM, tdm_options);
```

- **Sync to Data Delay (choose one)**
  - `TDM_SYNC_DELAY_NONE`, `TDM_SYNC_DELAY_1BIT`, `TDM_SYNC_DELAY_2BITS`

- **Transmitter Sync Pulse Length (choose one)**
  - `TDM_TX_SYNC_WIDTH_SLOT`, `TDM_TX_SYNC_WIDTH_BIT`

*Note: receiver always works with either slot or bit length*
Sync Pulse Polarity (choose one)
TDM_SYNC_POLARITY_NORMAL (low line signal)
TDM_SYNC_POLARITY_INVERT (high line signal)

Number of frames returned each MgslReceive call (1-256)
TDM_RX_FRAME_COUNT(frame_count)

Number of slots per frame (384 or 2-32)
TDM_SLOT_COUNT(slot_count)

Number of bits per slot (choose one, 8 to 32, increments of 4)
TDM_SLOT_SIZE_8BITS, TDM_SLOT_SIZE_12BITS, TDM_SLOT_SIZE_16BITS,
TDM_SLOT_SIZE_20BITS, TDM SLOT_SIZE_24BITS, TDM_SLOT_SIZE_28BITS,
TDM SLOT_SIZE_32BITS

Configuration, Step 2: Call MgslSetOption with MGSL_OPT_MSB_FIRST

This call specifies the serial bit order, LSB first (value of 0) or MSB first (value of 1).

/* select bit order (0=LSB first, 1=MSB first) */
MgslSetOption(fd, MGSL_OPT_MSB_FIRST, bit_order);

Configuration, Step 3: Call MgslSetParams

This call specifies the TDM protocol, clock settings, data polarity and transmit clock speed.

MGSL_PARAMS params;
memset(&params, 0, sizeof(params));
params.Mode = MGSL_MODE_TDM;

/* clock polarity (choose one) */
/* normal, sample on falling edge of line signal */
params.Flags = HDLC_FLAG_RXC_RXCPIN | HDLC_FLAG_TXC_BRG;
/* invert, sample on rising edge of line signal */
params.Flags = HDLC_FLAG_RXC_RXCPIN | HDLC_FLAG_RXC_INV |
HDLC_FLAG_TXC_BRG | HDLC_FLAG_TXC_INV;

/* data polarity (choose one) */
params.Encoding = HDLC_ENCODING_NRZ; /* normal, 1 = low line signal */
params.Encoding = HDLC_ENCODING_NRZB; /* invert, 1 = high line signal */
params.ClockSpeed = 3686400; /* transmit clock rate in bps */
MgslSetParams(handle, &params);

Sending and Receiving TDM Data

Data is transferred with MgslReceive and MgslTransmit API calls.
MgslReceive returns 1 to 256 frames of receive data each call as specified by TDM_RX_FRAME_COUNT above. MgslReceive does not complete until the specified number of frames are received. Use larger values for high data rates and small frame size to reduce system load.

One or more frames are sent with each MgslTransmit call. The number of frames sent per call can change as needed, as long as the frames fit in the maximum buffer size (4096 bytes default). Partial frames must not be submitted to MgslTransmit.

Each slot consumes an integer number of API buffer bytes stored in little endian order as shown below. This applies to both MgslTransmit and MgslReceive. The application uses knowledge of the slot size to determine slot positions within an API buffer.

2 Frames of 2 Slots, Slot=12 bits, Stored in API Buffer

<table>
<thead>
<tr>
<th>Buffer Byte Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Content</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Frame 1 (2 slots, slot=12 bits)  Frame 2 (2 slots, slot=12 bits)
SERIAL 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.

<table>
<thead>
<tr>
<th>Serial Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>NRZ</td>
</tr>
<tr>
<td>NRZB</td>
</tr>
<tr>
<td>NRZI-space</td>
</tr>
<tr>
<td>NRZI-mark</td>
</tr>
<tr>
<td>Biphase-mark (FM1)</td>
</tr>
<tr>
<td>Biphase-space (FM0)</td>
</tr>
<tr>
<td>Biphase-level (Manchester)</td>
</tr>
<tr>
<td>Differential Biphase-level</td>
</tr>
</tbody>
</table>
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.

\[ f_{data} = \frac{f_{base}}{divisor + 1} \quad divisor = \left( \frac{f_{base}}{f_{data}} \right) - 1 \]

**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 = \((14745600/1000000) – 1 = 13.7456\)

Since 13.7456 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.
DPLL 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.

\[ f_{\text{ref}} = \frac{f_{\text{base}}}{\text{divisor}+1} \quad \text{divisor} = \left( \frac{f_{\text{base}}}{f_{\text{ref}}} \right) - 1 \]

Example 1:

Data Clock = 9600bps
DPLL 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:

<table>
<thead>
<tr>
<th>Serial Encoding</th>
<th>Preamble Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDLC_ENCODING_NRZI_SPACE (NRZI)</td>
<td>HDLC_PREAMBLE_PATTERN_ZEROS</td>
</tr>
<tr>
<td>HDLC_ENCODING_BIPHASE_MARK (FM1)</td>
<td>HDLC_PREAMBLE_PATTERN_ZEROS</td>
</tr>
<tr>
<td>HDLC_ENCODING_BIPHASE_SPACE (FM0)</td>
<td>HDLC_PREAMBLE_PATTERN_ONES</td>
</tr>
<tr>
<td>HDLC_ENCODING_BIPHASE_LEVEL (Manchester)</td>
<td>HDLC_PREAMBLE_PATTERN_01</td>
</tr>
</tbody>
</table>
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 hdlcapi\src\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:
   08001400D8A000000000000001F9FE2

12. Divide clipboard value into 4 32-bit hex values of 8 digits each, with the 5th value a single digit:

   08001400D8A000000000000001F9FE2 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.

```c
struct freq_table_entry gt4e_table[] =
{
    {12288000, {0x29BFDC00, 0x61200000, 0x00000000, 0x0000A5FF, 0xA0000000}},
    {14745600, {0x38003C05, 0x24200000, 0x00000000, 0x000057FF, 0xA0000000}},
    {16000000, {0x280CFC02, 0x64A00000, 0x00000000, 0x0000307FD, 0x20000000}},
    {20000000, {0x000001403, 0xE0C00000, 0x00000000, 0x00045E02, 0xF0000000}},
    {30000000, {0x20267C05, 0x64C00000, 0x00000000, 0x00050603, 0x30000000}},
    {32768000, {0x21BFDC00, 0x5A400000, 0x00000000, 0x0004D206, 0x30000000}},
    {32000000, {0x21BFDC00, 0x5A400000, 0x00000000, 0x0004D206, 0x30000000}},
    {32768000, {0x000014003, 0x78A00000, 0x00000000, 0x0001F9FE, 0x20000000}},
    {64000000, {0x21BFDC00, 0x12000000, 0x00000000, 0x000F5E14, 0xF0000000}},
    {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.

```c
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 MGHDLCx, where x is the device instance number (MGHDLC1, MGHDLC2, etc). Devices with multiple ports use MGMPxy, 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 MGCx, where x is the device instance number (MGC1, MGC2, etc). Devices with multiple ports use MCxy, 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 src\sample-commapi directory of the MicroGate Serial API development package.