MICROGATE SERIAL COMMUNICATIONS

LINUX GUIDE

MicroGate Systems, Ltd

http://www.microgate.com

CONTENTS

OVERVIEW

This guide describes developing with SyncLink serial communication devices and the Linux operating system. Use the guide in the order:

Software Installation
Hardware Installation
Verifying Installation
Serial API Programming

Other sections cover general serial communication topics.

Supported Linux Versions

- Red Hat Enterprise Linux/CentOS 6.X, 7.X, 8.X
- openSUSE Leap 15.X
- Debian 8.X, 9.X, 10.X
- Ubuntu 14.04LTS, 16.04LTS, 18.04LTS, 20.04LTS
- Raspbian 8, 9 on Raspberry Pi 3 Model B/Pi 4 Model B

Other distributions and kernel versions may be compatible but have not been tested. Before purchasing a SyncLink device, build and install the free downloadable drivers in the target environment to verify compatibility.

MicroGate may be able to offer help porting to the target environment. Depending on the environment, the port may or may not be possible. Depending on the difficulty of the port, a development fee may be required.

SyncLink GT requires kernel version 2.6.5 or later.
SyncLink USB requires kernel version 2.6.28 or later.
Tested with generic (non-vendor specific) kernel versions up to 5.7

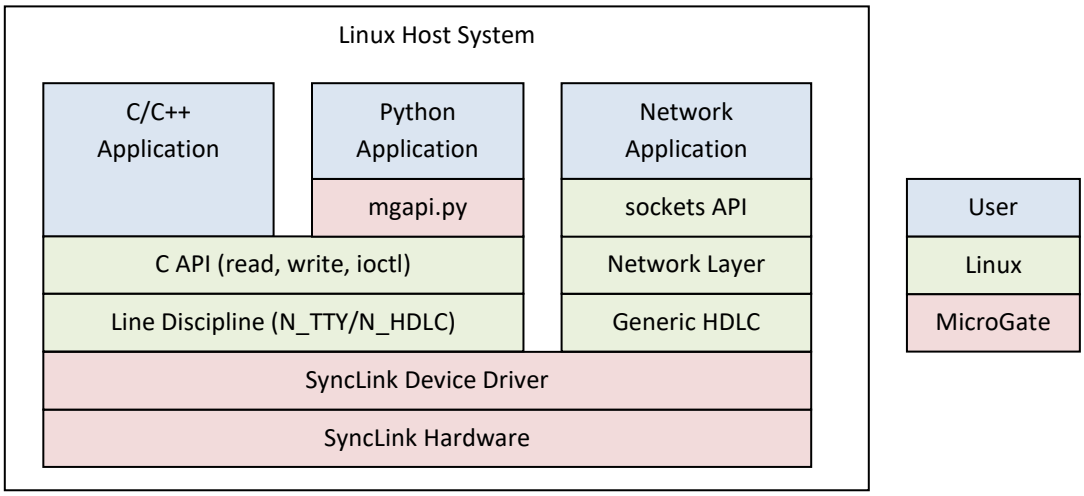
Required Knowledge

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

1. C programming
2. Basic Linux administration
3. Building and installing Linux device drivers
4. Serial communication details for target application
5. Reading supplied MicroGate documentation

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

User applications access a serial device directly with the C API, Python API or as a network device with the sockets API. These methods are describes in following sections. It is important to obtain all application specific details, documentation and specifications so the serial device can be correctly configured.



SOFTWARE ORGANIZATION

HARDWARE INSTALLATION

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

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

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

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

PCI CARDS

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

- Verify card interface selection 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.

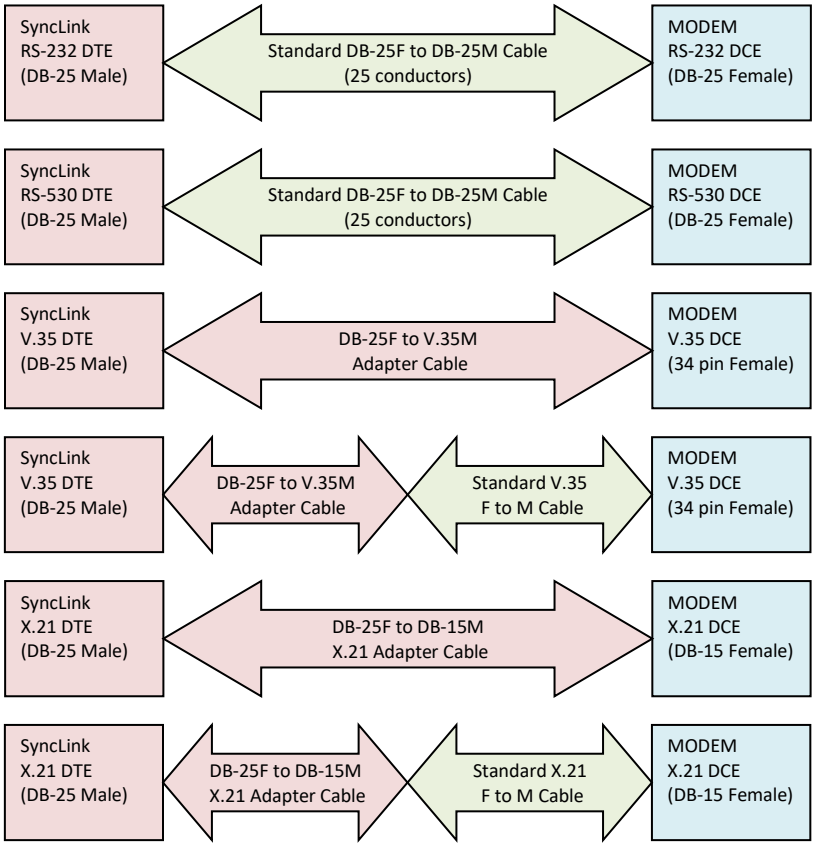
SyncLink USB requires 500mA of power from the USB port, which is standard and supported by most USB ports. Some USB ports may not provide a full 500mA, such as unpowered hubs or ports in small mobile devices.

After insertion and driver installation select the SyncLink USB interface type with the API or `mgslutil` tool.

CABLES

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

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



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

SOFTWARE INSTALLATION

Before using 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/linux/linuxwan.tar.xz

Copy `linuxwan.tar.xz` to the the target system. Extract the contents with the command:

```
#tar xJvf linuxwan.tar.xz
```

This creates a directory `synclink` containing documentation, sample code and drivers.

`c/include/synclink.h`	C API header file
`c/samples`	C API sample applications
`python/mgapi.py`	Python API module
`python/mgapi.tar.gz`	Python API installable package
`python/samples`	Python API sample applications
`drivers`	Device driver installation and source
`tools`	Configuration and testing tools

In the `tools` directory, run the following command to build the tools:

```
#make
```

Run the following command (as root) to install the tools into `/usr/local/sbin`:

```
#make install
```

`mgslutil`	configuration tool
`mgsltest`	test tool
`sethdlc`	tool for configuring serial device as network interface
`testloop`	test bash script (uses `mgslutil` and `mgsltest`)
`netctl`	bash script to configure serial device as network interface (uses `sethdlc`)

The functions performed by these tools can be implemented by applications using the C or Python API.

PREBUILT DRIVERS

Microgate recommends using the latest drivers from the downloadable driver package. If your distribution includes prebuilt drivers for your SyncLink device and you prefer using older prebuilt drivers, skip to the Load/Unload Driver section. Prebuilt drivers may lack features and bug fixes in the latest driver.

Prebuilt drivers are located in:

```
/lib/modules/2.6.32-220.13.1.el6.i686
```

The last part of the path is the target kernel version. If the target kernel is the same as the running kernel, the version is obtained with this command:

```
#uname –r
```

DRIVER BUILD REQUIREMENTS

Drivers must be compiled for specific kernels. This requires parts of the kernel source and supporting files. Distribution supplied packages provide these requirements for distribution supplied kernels. Building custom kernels is beyond the scope of this document and should only be attempted by those with the necessary knowledge. Build tools required to compile the drivers must be installed and are contained in the `build-essential` (or similar) package for your Linux distribution.

Run the `setup` script in the `drivers` directory to build and install drivers. The script searches for the source of the currently running kernel. For unsupported distributions or custom kernels edit `setup` and set the `KERNEL_SOURCE` variable to the source location.

RHEL/CentOS

Install driver build requirements with the command:

```
#sudo yum install kernel-devel
```

Kernel source from package `kernel-devel` is located in:

```
/usr/src/kernels/2.6.32-220.13.1.el6.i686
```

The last portion of this path is the kernel version as determined with the command:

```
#uname –r
```

openSUSE Leap

Install driver build requirements with the command:

```
#sudo zypper install kernel-devel
```

Kernel source from package `kernel-devel` is located in:

```
/usr/src/linux-5.3.18-lp152.44-obj/x86_64/defaults
```

Components of this path are determined with the commands (release and machine):

```
#uname –r
#uname –m
```

Debian/Ubuntu

Install driver build requirements with the command:

```
#sudo apt install linux-headers-$(uname -r)
```

Kernel source from package `linux-headers` is located in:

```
/usr/src/linux-headers-4.4.0-53-generic
```

The last portion of this path is the kernel version as determined with the command:

```
#uname –r
```

Raspbian

Install driver build requirements with the command:

```
#sudo apt-get install raspberrypi-kernel-headers
```

Kernel source from package raspberrypi-kernel-headers is located in the directory:

```
/usr/src/linux-headers-4.4.34
```

The last portion of this path is the kernel version as determined with the command:

```
#uname –r
```

Note: Raspbian was tested with Raspberry Pi 3 Model B and Pi 4 Model B hardware. Other hardware may work but is not supported. The SyncLink USB requires up to 500mA of power depending on loading, cable length and data rate. The amount of power supplied by the Pi 3 USB port depends on the power supply and other connected peripherals. Specific applications may require use of a powered USB hub.

BUILD AND INSTALL DRIVERS

Device drivers are located in `drivers`.

`synclink_gt.c`	PCI cards
`synclink_usb.c`	USB Adapter

Build and install the drivers from the `drivers` directory with the command:

```
#./setup
```

Errors are caused by missing build requirements or incompatibility between the driver source and the kernel version. Correct errors by inspection if possible. If errors persist contact MicroGate with detailed distribution, kernel version and error information.

MODEMMANAGER

ModemManager is a Linux service that controls modem hardware. SyncLink devices **MUST** be excluded from ModemManager control or ModemManager must be disabled.

The Linux service udev notifies ModemManager when serial devices are present. The `setup` script copies a udev rules file (`10-microgate.rules`) to the udev rules directory (`/etc/udev/rules.d`) to exclude SyncLink devices from ModemManager control. Reboot for the rules file to take effect.

LOAD/UNLOAD DRIVER

After installing the hardware and device driver, the driver must be loaded. This is usually done by the operating system without user intervention once the kernel detects the installed device. A reboot after initial driver installation may be necessary for the system to automatically load the driver.

Verify the device driver is loaded using the `lsmod` command:

```
#lsmod | grep synclink
```

Sample output:

```
synclink_usb           42221  0
synclink_gt            39878  0
```

If a required driver is not listed load the driver manually with (example for PCI driver):

```
#modprobe synclink_gt
```

Manually unload drivers with (example for PCI driver):

```
#modprobe -r synclink_gt
```

DEVICE NAMES

Applications access devices using device files. Each device file represents a serial port (device instance). Hardware with more than one serial port (example: GT4e) has multiple device instances. Applications use the `open` system call with the device file name.

Device files are located in the `/dev` directory. The name depends on the hardware type and instance number.

`/dev/ttySLGx` `synclink_gt` driver, PCI cards
`/dev/ttyUSBx` `synclink_usb` driver, USB devices

In the above names, x is a zero based device instance number. For example, `/dev/ttySLG3` is the fourth device provided by the `synclink_gt` driver. Device instance numbers are assigned in initialization order. For PCI devices this depends on the PCI slot and is static unless cards are added or removed. For USB devices this depends on the USB port and insertion/removal order. The user must determine the mapping of hardware to device instances for each target environment.

USB device file aliases allow use of specific USB devices regardless of insertion order. Aliases based on the device serial number are located in the `/dev/serial/by-id` directory. Aliases based on the USB port are located in the `/dev/serial/by-path` directory. These aliases may be used in place of the device name.

Example serial number based alias:
`/dev/serial/by-id/usb-MicroGate_SyncLink_USB_1U3-12477-if00-port0`

Example USB port based alias:

```
/dev/serial/by-path/pci-0000:08:00.3-usb-0:3:1.0-port0
```

Use the Linux `udevadm` or `ls` commands to find the aliases to a specific device file:

```
#udevadm info --query symlink --name /dev/ttyUSB0

#ls -l /dev/serial/by-*
```

DRIVER DEBUG OUTPUT

The driver can output debugging information to the system log. Debug output is selected when the driver loads using the `debug_level` module parameter. Module parameters are specified on the command line of the `modprobe` utility or in a configuration file in the `/etc/modprobe.d` directory. See the `modprobe` man page for more information. The value for `debug_level` is an integer from 0 to 5, with 0 disabling debug output and 5 providing the most detail. Higher debug levels include all the output from lower levels.

`DEBUG_LEVEL_DATA (1)`	send and receive data
`DEBUG_LEVEL_ERROR (2)`	error events
`DEBUG_LEVEL_INFO (3)`	informational events
`DEBUG_LEVEL_BH (4)`	deferred interrupt or URB processing events
`DEBUG_LEVEL_ISR (5)`	interrupt or URB processing events

High debug levels save very large amounts of data to the system log. At high data rates, this can cause loss of debug output and may interfere with system operation. A debug level of 3 (info) is the best starting value when diagnosing problems with an application under development. High debug levels are used to diagnose problems with the driver.

A sample module configuration file `synclink-modprobe.conf` is located in `drivers`. Edit this file to adjust `debug_level` to the desired value and copy the file to the `/etc/modprobe.d` directory. Reload the driver for the new value to take effect:

```
#modprobe –r synclink_gt
#cp synclink-modprobe.conf /etc/modprobe.d
#modprobe synclink_gt
```

Debug output is sent to a log file which is usually `/var/log/messages` or `/var/log/kern.log`. Consult the documentation for your distribution to locate the kernel log file if it is not in these locations.

Sample output in the system log will look like:

```
Oct 19 10:21:43 localhost kernel: ttySLG0 open, old ref count = 0
Oct 19 10:21:43 localhost kernel: ttySLG0 startup
Oct 19 10:21:43 localhost kernel: ttySLG0 change_params
Oct 19 10:21:43 localhost kernel: ttySLG0 block_til_ready
Oct 19 10:21:43 localhost kernel: ttySLG0 open rc=0
Oct 19 10:21:43 localhost kernel: ttySLG0 ioctl() cmd=80206D01
Oct 19 10:21:43 localhost kernel: ttySLG0 get_params
Oct 19 10:21:43 localhost kernel: ttySLG0 chars_in_buffer()=0
Oct 19 10:21:43 localhost kernel: ttySLG0 wait_until_sent entry
Oct 19 10:21:43 localhost kernel: ttySLG0 wait_until_sent exit
Oct 19 10:21:43 localhost kernel: ttySLG0 flush_buffer
Oct 19 10:21:43 localhost kernel: ttySLG0 tiocmset(6,0)
```

```
Oct 19 10:21:43 localhost kernel: ttySLG0 write count=512
Oct 19 10:21:43 localhost kernel: ttySLG0 tx data:
Oct 19 10:21:43 localhost kernel: FF A5 01 01 01 01 01 01 01 01 01 01 01 01 01 01
Oct 19 10:21:43 localhost kernel: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
```

VERIFYING INSTALLATION

Run the `testloop` script in the `tools` directory to send and receive data on a device using internal loopback. If successful, this demonstrates correct installation.

```
#./testloop /dev/ttySLG0
```

The above command tests the first PCI device. Sample output:

```
internal loopback test on /dev/ttySLG0, mode=hdlc speed=19200, 10 Packets of
512 bytes

mgsltest $Revision: 1.12 $
mgsltest testing 512 byte packets on /dev/ttySLG0
Asserting DTR and RTS
Internal loopback enabled, ignoring DCD
loop#1, sending 512 bytes...send OK...receive OK
…
loop#10, sending 512 bytes...send OK...receive OK

Master test results:
transmit OK=10, transmit timeouts=0
receive OK=10, receive timeouts=0
receive errors=0, lost frames=0
```

Installation errors are the most common cause of failures. If the installation is correct and errors continue, this may indicate a system incompatibility or hardware fault and you should contact MicroGate for assistance.

SERIAL API PROGRAMMING

This section describes direct control of a serial device by an application using either C/C++ or Python. A working knowledge of the target language in a Linux environment is required.

C/C++

Required API definitions are contained in `c/include/synclink.h`. This header must be included in user code. The standard Linux header file `termios.h` must be included to access standard Linux serial device calls.

Sample applications are included for each protocol in `c/samples`:

`hdlc.c`	HDLC/SDLC
`2wire.c`	2-wire/bussed HDLC/SDLC
`raw.c`	raw bitstream
`async.c`	asynchronous/isochronous
`bisync.c`	BISYNC/MONOSYNC
`xsync.c`	extended byte synchronous
`fsynth.c`	programming GT4e/USB frequency synthesizer
`signals.c`	controlling outputs and monitoring inputs

From `c/samples`, build the samples with the command:

```
#make
```

A sample is run as shown below for the HDLC sample on device `/dev/ttyUSB0`:

```
#./hdlc /dev/ttyUSB0
```

Most samples send and receive data on a single port and require a loopback connector to be installed. Refer to the sample code for more details on setup and operation.

LINUX LINE DISCIPLINES

A **line discipline** is a software layer between a serial device driver and a user application that controls the flow of data. An application selects a line discipline with the `ioctl` system call. The default line discipline `N_TTY` is used for byte oriented communications (all protocols except HDLC). The HDLC protocol uses the frame oriented `N_HDLC` line discipline which returns a single frame for each `read` and sends a frame for each `write`.

SYSTEM CALLS

`open`	open file descriptor to device instance
`read`	get received data from device
`write`	send data to device
`ioctl`	monitor, control and configure device
`close`	close file descriptor to device

In addition to system calls, Linux includes tty/termios library calls used for serial communications. Refer to the man pages or third party books on Linux user mode programming for more details about system calls.

OPEN/CLOSE

A user application gets a file descriptor from the `open` call to supply to other functions. The file descriptor is a token representing a device instance for the specified device name.

```
int fd; /* file descriptor */
fd = open("/dev/ttySLG0", O_RDWR | O_NONBLOCK, 0);
/* other code goes here */
close(fd);
```

`O_RDWR` is required to send and receive data.
`O_NONBLOCK` causes `open` to return without waiting for the DCD input to be active.

CONFIGURATION

A device must be configured to match application specific requirements. This is done using the `ioctl` system call with C structures defined in the `synclink.h` header file.

The main configuration call is `ioctl(MGSL_IOCSPARAMS)`, which uses an `MGSL_PARAMS` structure to specify protocol options. This call is documented in detail in a later section detailing all SyncLink ioctl() calls.

```
int fd;
int rc;
MGSL_PARAMS params;

params.mode = MGSL_MODE_HDLC;
params.loopback = 0;
params.flags = HDLC_FLAG_RXC_RXCPIN + HDLC_FLAG_TXC_TXCPIN;
params.encoding = HDLC_ENCODING_NRZ;
params.clock_speed = 9600;
params.crc_type = HDLC_CRC_16_CCITT;

rc = ioctl(fd, MGSL_IOCSPARAMS, &params);
if (rc < 0) {
      printf("ioctl(MGSL_IOCSPARAMS) error=%d %s\n",
             errno, strerror(errno));
}
```

Other important configuration calls include `ioctl(MGSL_IOCGTXIDLE)` for setting the transmit idle pattern and BISYNC/MONOSYNC sync patterns, and the `ioctl(MGSL_IOCSIF)` call for selecting the serial interface type (RS232, V.35, RS422) on the USB adapter.

RECEIVING DATA

A user mode program gets received data from the driver using the `read()` system call.

```
int fd;                         /* open file descriptor */
char buf[HDLC_MAX_FRAME_SIZE]; /* buffer for received frame */
int rc;                         /* read return value */

rc = read(fd, buf, sizeof(buf));
if (rc < 0) {
   /* process error */
} else {
   /* process rc bytes of data */
}
```

For frame oriented SDLC/HDLC, each `read` call returns a single frame of data. For other protocols, each `read` call returns a stream of bytes.

The `read` call can be blocking or non-blocking. In blocking mode, the call returns when data is available. Non-blocking mode always returns immediately, with data or with an error indicating no data is available. Refer to the `read` man page for details. The blocking mode is set with the `fcntl` call.

```
/* set device to blocking mode for reads and writes */
fcntl(fd, F_SETFL, fcntl(fd,F_GETFL) & ~O_NONBLOCK);
```

SENDING DATA

A user mode program sends data to the driver using the write() system call.

```
int fd;                          /* open file descriptor */
char buf[HDLC_MAX_FRAME_SIZE]; /* buffer for send frame */
int size;                        /* size of data in buffer */
int rc;                          /* write() return value */

/* initialize buf with send data */
/* initialize size with count of send data bytes */

rc = write(fd, buf, size);
if (rc < 0) {
   /* process error */
} else {
   /* rc bytes of data successfully sent */
}
```

For frame oriented SDLC/HDLC, each `write` call sends a single frame of data. For other protocols, each `write` call sends a stream of bytes.

The `write` call can be blocking or non-blocking. In blocking mode, the call returns when data is queued by the driver. Non-blocking mode always returns immediately, with a positive return code indicating data was accepted or with an error indicating all buffers are full. Refer to the `write` man page for details. Blocking mode is set with the `fcntl` call.

```
/* set device to blocking mode for reads and writes */
fcntl(fd, F_SETFL, fcntl(fd,F_GETFL) & ~O_NONBLOCK);
```

IOCTL

The `ioctl` system call is used for configuration, control and monitoring a serial device. This section documents the different codes and structures used with this call.

The general form of `ioctl` is:

```
rc = ioctl(fd, code, arg);
```

where `rc` is the return code, `fd` is an open file descriptor for a device, `code` identifies the task to perform and `arg` is a code specific argument that can be a value or a pointer. The code can be a Linux standard code or a SyncLink specific code.

STANDARD IOCTL CODES

```
TIOCSETD      set current line discipline
TIOCMGET      get modem control and status signal states
TIOCMBIS      enable specified modem control signals
TIOCMBIC      disable specified modem control signals
TIOCMSET      set state of specified modem control signals
TIOCMIWAIT    wait for serial status signal changes
TIOCGICOUNT   get count of serial status signal changes
TIOCOUTQ      get count of pending send data
```

SYNCLINK SPECIFIC IOCTL CODES

```
MGSL_IOCGPARAMS      get device configuration
MGSL_IOCSPARAMS      set device configuration
MGSL_IOCGTXIDLE      get transmit idle mode
MGSL_IOCSTXIDLE      set transmit idle mode
MGSL_IOCTXENABLE     enable/disable transmitter
MGSL_IOCRXENABLE     enable/disable receiver
MGSL_IOCTXABORT      abort HDLC send frame in progress
MGSL_IOCGSTATS       get or reset device statistics
MGSL_IOCWAITEVENT    wait for specified event
MGSL_IOCSIF          set serial interface mode and options
MGSL_IOCGIF          get serial interface mode and options
MGSL_IOCSGPIO        set general purpose I/O options
MGSL_IOCGGPIO        get general purpose I/O options
MGSL_WAITGPIO        wait for specified GPIO input state
```

TIOCSETD

```
int rc;
int ldisc = N_HDLC;
rc = ioctl(fd, TIOCSETD, &ldisc);
```

Set the line discipline. Choose the appropriate line discipline for the operating mode as part of the configuration process. Use `N_HDLC` for HDLC/SDLC protocol and `N_TTY` for other protocols.

TIOCMGET

```
int rc, sigs;
rc = ioctl(fd, TIOCMGET, &sigs);
```

Return serial control and status signals in `sigs` argument. Active signals are indicated with set bit flags:

```
TIOCM_RTS   Request To Send (output)
TIOCM_DTR   Data Terminal Ready (output)
TIOCM_CAR   Data Carrier Detect (input)
TIOCM_RNG   Ring Indicator (input)
TIOCM_DSR   Data Set Ready (input)
TIOCM_CTS   Clear To Send (input)
```

TIOCMBIS

```
int sigs = TIOCM_RTS | TIOCM_DTR
rc = ioctl(fd, TIOCMBIS, &sigs);
```

Turn on modem outputs specified by `sigs` argument. Altered signals are indicated with bit flags, other signals are unchanged.

```
TIOCM_RTS   Request To Send
TIOCM_DTR   Data Terminal Ready
```

TIOCMBIC

```
int sigs = TIOCM_RTS | TIOCM_DTR
rc = ioctl(fd, TIOCMBIC, &sigs);
```

Turn off modem outputs specified by `sigs` argument. Altered signals are indicated with bit flags, other signals are unchanged.

```
TIOCM_RTS   Request To Send
TIOCM_DTR   Data Terminal Ready
```

TIOCMSET

```
int sigs = TIOCM_RTS | TIOCM_DTR
rc = ioctl(fd, TIOCMSET, &sigs);
```

Set state of modem control signals specified by `sigs` argument. Set a bit flag for outputs to turn on, other outputs are turned off.

```
TIOCM_RTS   Request To Send
TIOCM_DTR   Data Terminal Ready
```

TIOCMIWAIT

```
int sigs = TIOCM_RNG | TIOCM_DSR | TIOCM_CD | TIOCM_CTS;
rc = ioctl(fd, TIOCMGET, &sigs);
```

Wait for one or more serial inputs to change. Specify signals of interest with set bit flags.

```
TIOCM_CAR   Data Carrier Detect
TIOCM_RNG   Ring Indicator
TIOCM_DSR   Data Set Ready
TIOCM_CTS   Clear To Send
```

On return, the application should use `ioctl(TIOCMGET)` determine current signal states. Use `ioctl(TIOCGICOUNT)` before and after this call to see how many transitions have occurred on the serial inputs.

TIOCGICOUNT

```
struct serial_icounter_struct icount;
rc = ioctl(fd, TIOCGICOUNT, &icount);
```

Return a `serial_icounter_struct` structure with counts of serial status signal changes and line errors.

`serial_icounter_struct` is typically found in `/usr/include/linux/serial.h` and is defined as:

```
struct serial_icounter_struct {
      int cts, dsr, rng, dcd;
      int rx, tx;
      int frame, overrun, parity, brk;
      int buf_overrun;
      int reserved[9];
};
```

```
cts             number of transitions in Clear to Send (CTS)
dsr             number of transitions in Data Set Ready (DSR)
rng             number of transitions in Ring Indicator (RI)
dcd             number of transitions in Data Carrier Detect (DCD)
rx              number of received asynchronous data bytes
tx              number of transmitted asynchronous data bytes
frame           number of asynchronous framing errors detected
overrun         number of asynchronous receive overruns detected
parity          number of asynchronous parity errors detected
brk             number of asynchronous break sequences detected
buf_overrun     number of times the driver's receive data buffer overflowed
```

TIOCOUTQ

```
int count;
rc = ioctl(fd, TIOCOUTQ, &count);
```

Returns a byte count of queued send data. This can be used to poll a device for send completion. When rc is zero, all data has been sent.

MGSL_IOCSPARAMS

```
MGSL_PARAMS params;

params.mode = MGSL_MODE_HDLC;
/* ... other application defined settings ... */
rc = ioctl(fd, MGSL_IOCSPARAMS, &params);
```

Set device configuration specified by MGSL_PARAMS structure defined in synclink.h:

```
typedef struct _MGSL_PARAMS
{
   /* Common */

   unsigned long  mode;      /* HDLC, async, raw, bisync, monosync, xsync */
   unsigned char  loopback;  /* internal loopback mode */
   unsigned short flags;

   /* synchronous modes */

   unsigned char encoding;    /* NRZ, NRZI, etc. */
   unsigned long clock_speed; /* external clock speed in bits per second */
   unsigned char addr_filter; /* rx HDLC address filter, 0xFF = disable */
   unsigned short crc_type;   /* None, CRC16-CCITT, or CRC32-CCITT */
   unsigned char  preamble_length;
   unsigned char  preamble;

   /* override asynchronous termios settings if necessary */

   unsigned long  data_rate;
   unsigned char  data_bits;
   unsigned char  stop_bits;
   unsigned char  parity;

} MGSL_PARAMS, *PMGSL_PARAMS;
```

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

MODE

The mode field of the MGSL_PARAMS structure specifies the protocol with an MGSL_MODE_XXX macro.

MGSL_MODE_ASYNC	asynchronous/isochronous
MGSL_MODE_HDLC	HDLC/SDLC
MGSL_MODE_RAW	unformatted bitstream
MGSL_MODE_BISYNC	byte synchronous, 2 byte sync pattern
MGSL_MODE_MONOSYNC	byte synchronous, 1 byte sync pattern
MGSL_MODE_XYNC	extended byte synchronous (1 - 4 byte sync pattern)
MGSL_MODE_BASE_CLOCK	set base clock to clock_speed field (no protocol selected)

LOOPBACK

Enable or disable 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 clock_speed field. Use internal loopback mode to test the operation of the serial controller without external line drivers or devices.

FLAGS

The `flags` field specifies protocol and clocking 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

The `HDLC_FLAG_RXC_xxx` macros select the receive clock source.

`HDLC_FLAG_RXC_DPLL`	Receive clock is recovered from the receive data signal. The `clock_speed` field 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 `clock_speed` field of the `MGSL_PARAMS` structure.
`HDLC_FLAG_RXC_RXCPIN`	Receive clock is supplied on the RxC input.
`HDLC_FLAG_RXC_TXCPIN`	Receive clock is supplied on the TxC input.
`HDLC_FLAG_RXC_INV`	Combine with other flags to invert clock polarity. Example: `HDLC_FLAG_RXC_RXCPIN + HDLC_FLAG_RXC_INV` uses the RxC input as the receiver clock source with inverted polarity. Note: This flag is available only on PCI Express cards (GT2e/GT4e) shipped after 02/2016 and the USB device. It is not available on legacy PCI cards (GT/GT2/GT4)

Transmit Clock Source

The `HDLC_FLAG_TXC_xxx` macros select the transmit clock source.

`HDLC_FLAG_TXC_DPLL`	Transmit clock is recovered from the receive data signal. The `clock_speed` field 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 `clock_speed` field of the `MGSL_PARAMS` structure.
`HDLC_FLAG_TXC_RXCPIN`	Transmit clock is supplied on the RxC input.
`HDLC_FLAG_TXC_TXCPIN`	Transmit clock is supplied on the TxC input.
`HDLC_FLAG_TXC_INV`	Combine with other flags to invert clock polarity. Example: `HDLC_FLAG_TXC_TXCPIN + HDLC_FLAG_TXC_INV` uses the TxC input as the transmitter clock source with inverted polarity. Note: This flag is available only on PCI Express cards (GT2e/GT4e) shipped after 02/2016 and the USB device. It is not available on legacy PCI cards (GT/GT2/GT4)

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 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

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

Specify data signal representation of logical 1 or 0. Equivalent encoding names are shown in parenthesis. BIPHASE encodings are usually used with 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.

HDLC_ENCODING_NRZ	unencoded data
HDLC_ENCODING_NRZB	inverted data
HDLC_ENCODING_NRZI_MARK	invert at start of bit if 1
HDLC_ENCODING_NRZI_SPACE	(NRZI) invert at start of bit if 0
HDLC_ENCODING_BIPHASE_MARK	(FM1) invert at start of bit, invert at middle of bit if 1
HDLC_ENCODING_BIPHASE_SPACE	(FM0) invert at start of bit, invert at middle of bit if 0
HDLC_ENCODING_BIPHASE_LEVEL	(Manchester) bit start: high=1, low=0, invert at middle of bit
HDLC_ENCODING_DIFF_BIPHASE_LEVEL	invert at start of bit if 1, invert at middle of bit

CLOCK_SPEED

Specifies data rate of the generated (BRG) or recovered (DPLL) clock. A clock generated by the BRG is output on the AUXCLK output pin for use by an external device. Set to zero to disable clock generation.

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

divisor = (base clock/data rate) - 1

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

The serial card can be purchased with a different base clock. This option is installed at the factory. When a base clock other than 14.7456MHz is installed, the driver must be configured to use the new value by calling `ioctl(MGSL_IOCSPARAMS)` as shown in the example below:

```
params.mode = MGSL_MODE_BASE_CLOCK;
params.clock_speed = 32000000; /* 32MHz optional base clock */
ioctl(fd, &params);
```

ADDR_FILTER

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 this value.

CRC_TYPE

Selects the frame check type used with SDLC/HDLC frames. The selected Cyclic Redundancy Check (CRC) code is appended to sent frames and verified on receive frames.

HDLC_CRC_NONE	Don't send CRCs on transmit, don't check CRCs on receive.
HDLC_CRC_16_CCITT	16 bit CRC
HDLC_CRC_32_CCITT	32 bit CRC
HDLC_CRC_RETURN_EX	combine with other CRC flags to return CRC value and a status to application

Normally, only HDLC frames without error are returned to the application. HDLC frames with CRC errors are discarded after updating the CRC error count in the adapter statistics. The driver can be configured to return both good frames and frame with CRC errors.

```
params.crc_type = HDLC_CRC_16_CCITT | HDLC_CRC_RETURN_EX;
```

When HDLC_CRC_RETURN_EX is set, `read` returns frame data followed by 2 (CRC-16) or 4 (CRC-32) CRC bytes, followed by a status byte with the value of `RX_OK` or `RX_CRC_ERROR`.

PREAMBLE

A preamble is a pattern sent before an SDLC/HDLC frame or a BISYNC/MONOSYNC/XSYNC sync pattern. The preamble assists remote clock recovery. The length of the preamble is specified in the `preamble_length` field of the `MGSL_PARAMS` structure.

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

PREAMBLE_LENGTH

Select the length of the preamble pattern.

```
HDLC_PREAMBLE_LENGTH_16BITS
HDLC_PREAMBLE_LENGTH_32BITS
HDLC_PREAMBLE_LENGTH_64BITS
```

MGSL_IOCGTXIDLE

```
int idlemode;
rc = ioctl(fd, MGSL_IOCGTXIDLE, &idlemode);
```

Get transmit idle pattern or the sync pattern in MONOSYNC/BISYNC modes. See `MGSL_IOCSTXIDLE` for more details.

MGSL_IOCSTXIDLE

```
int idlemode = HDLC_TXIDLE_FLAGS;
rc = ioctl(fd, MGSL_IOCGTXIDLE, idlemode);
```

Select the transmit idle pattern or the sync pattern in MONOSYNC/BISYNC modes. When the transmitter is enabled and not sending data, this pattern is repeated. The transmitter always sends continuous marks when idle in asynchronous/isochronous mode.

`HDLC_TXIDLE_FLAGS`	flags (0x7e)
`HDLC_TXIDLE_ONES`	all ones (0xFF)
`HDLC_TXIDLE_ALT_ZEROS_ONES`	alternating zeros and ones (0x55)
`HDLC_TXIDLE_ZEROS`	all zeros (0x00)
`HDLC_TXIDLE_CUSTOM_8`	arbitrary 8 bit pattern specified in least significant 8 bits of value example: `HDLC_TXIDLE_CUSTOM + 0x96`
`HDLC_TXIDLE_CUSTOM_16`	arbitrary 16 bit pattern specified in least significant 16 bits of value example: `HDLC_TXIDLE_CUSTOM + 0x96aa` WARNING: preamble and 16 bit idle can't be used at the same time

The transmit idle pattern is also used as the synchronization pattern in bisync and monosync modes. For monosync, use HDLC_TXIDLE_CUSTOM_8 to specify the 8 bit sync pattern. For bisync, use HDLC_TXIDLE_CUSTOM_16 to specify the 16 bit sync pattern.

MGSL_IOCTXENABLE

```
int enable = 1;
rc = ioctl(fd, MGSL_IOCTXENABLE, enable);
```

Enable or disable the transmitter. 0=disable, 1=enable

The driver automatically enables the transmitter on a `write` call. When disabled, transmit data signal is a constant one value. When enabled, the transmitter either sends data or the idle pattern.

The following macros are defined for use as the `enable` argument for select SyncLink hardware that supports the low send latency feature:

```
MGSL_ENABLE_PIO         programmed I/O (PIO) data transfer
MGSL_ENABLE_DMA         direct memory access (DMA) data transfer (default)
```

Send Latency

Latency is the time from `write` until data appears on the serial data output. Queued data increases latency. Limit latency by limiting the amount of queued data.

Continuous Transmission

Usually, the serial transmitter cycles between active (sending data) and idle (sending idle pattern) as data becomes available and is sent. Some applications require continuous transmission (always active, no idle). Continuous transmission requires supplying data fast enough to keep the transmitter active. Buffering more data in the driver decreases the chance of an idle transmitter at the expense of increased send latency.

Direct Memory Access (DMA)

DMA is the most efficient transfer method and is preferred for high data rates. Continuous transmission in DMA mode requires sending at least 128 bytes per `write` call (latency of at least 128 bytes).

Programmed I/O (PIO)

PIO transfers data to hardware byte by byte. This allows greater control over latency when using continuous transmission but limits the maximum data rate. Use `ioctl(TIOCOUTQ)` to monitor count of queued send data and limit `write` calls. The `raw` sample demonstrates continuous low latency transmission.

MGSL_IOCRXENABLE

```
int enable = 1;
rc = ioctl(fd, MGSL_IOCRXENABLE, enable);
```

0=disable, 1=enable, 2=force hunt mode

The serial receiver has three states: disabled, idle, active.

A disabled receiver does not store data and ignores the receive data signal. Calling `ioctl(MGSL_IOCRXENABLE, 0)` immediately disables the receiver. Calling `ioctl(MGSL_IOCRXENABLE, 1)` puts the receiver in the idle state.

An idle receiver monitors the receive data signal for incoming data. Idle is also called "hunt mode". When a synchronization pattern (start bit, sync pattern, flag) is detected the receiver becomes active.

An active receiver saves data for use by the application. Depending on the mode, the receiver also monitors the receive data signal for an idle pattern (stop bits, flag). If an idle pattern is detected the receiver becomes idle. Calling `ioctl(MGSL_IOCRXENABLE, 2)` immediately returns the receiver to the idle state.

Note: The receiver is automatically enabled (idle state) after an `open` or `ioctl(MGSL_IOCSPARAMS)` call if the `O_RDWR` is passed to `open` system call.

When the receiver goes from disabled to enabled, all receive buffers internal to the driver are reset.

Raw, Bisync/Monosync Buffer Fill Level:

For raw, bisync and monosync modes, `read` returns data when a driver receive buffer (256 bytes) fills. At low data rates this may cause too much latency, the delay between receipt of a byte and that byte being returned to the application.

Bits 31..16 of the `MGSL_IOCRXENABLE ioctl` argument specify the receive buffer fill level. When the specified number of bytes are received, `read` returns the data. 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
0	No operation - do not alter the fill level

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

The line below sets the fill level to 8 bytes and selects PIO mode:

```
ioctl(fd, MGSL_IOCRXENABLE, ((8 << 16) | 1);
```

MGSL_IOCSXSYNC

```
int sync = 0x3232;
rc = ioctl(fd, MGSL_IOCSXSYNC, sync);
```

Set the extended sync mode synchronization pattern. This pattern is 1 to 4 bytes with the pattern located in the least significant bytes of the value. The size of the sync pattern is specified with `ioctl(MGSL_IOCSXTRL)`.

MGSL_IOCGXSYNC

```
int sync;
rc = ioctl(fd, MGSL_IOCGXSYNC, &sync);
```

Get extended sync mode synchronization pattern. This pattern is 1 to 4 bytes with the pattern located in the least significant bytes of the value. The size of the sync pattern is specified with `ioctl(MGSL_IOCSXTRL)`.

MGSL_IOCSXCTRL

```
int xctrl;
rc = ioctl(fd, MGSL_IOCSXCTRL, xctrl);
```

Set the 32-bit value controlling the operation in extended byte synchronous mode:

```
xctrl[31:19] reserved, must be zero
xctrl[18:17] extended sync pattern length in bytes
             00 = 1 byte  in xsr[7:0]
             01 = 2 bytes in xsr[15:0]
             10 = 3 bytes in xsr[23:0]
             11 = 4 bytes in xsr[31:0]
xctrl[16]    1 = enable terminal count, 0=disabled
xctrl[15:0]  receive terminal count for fixed length packets
             value is count minus one (0 = 1 byte packet)
             when terminal count is reached, receiver
             automatically returns to hunt mode and receive
             FIFO contents are flushed to DMA buffers with
             end of frame (EOF) status
```

MGSL_IOCGXCTRL

```
int xctrl;
rc = ioctl(fd, MGSL_IOCGXCTRL, &xctrl);
```

Return the current 32-bit value controlling extended byte synchronous mode operation. See `MGSL_IOCSXCTRL` for details of this value.

MGSL_IOCTXABORT

```
rc = ioctl(fd, MGSL_IOCTXABORT, 0);
```

Abort an HDLC send frame in progress with an HDLC abort pattern (7 or more contiguous ones). Only valid in HDLC mode.

MGSL_IOCGSTATS

```
struct mgsl_icount icount;
rc = ioctl(fd, MGSL_IOCGSTATS, &icount);
```

Return or reset the current `mgsl_icount` structure values maintained by the driver. Pass an argument of 0 instead of a pointer to a structure to reset the statistics. Statistics are automatically reset on the first open on a device instance.

The mgsl_icount structure is synclink.h:

```
struct mgsl_icount {
    __u32   cts, dsr, rng, dcd, tx, rx;
    __u32   frame, parity, overrun, brk;
    __u32   buf_overrun;
    __u32   txok;
    __u32   txunder;
    __u32   txabort;
    __u32   txtimeout;
    __u32   rxshort;
    __u32   rxlong;
    __u32   rxabort;
    __u32   rxover;
    __u32   rxcrc;
    __u32   rxok;
    __u32   exithunt;
    __u32   rxidle;
};
```

cts	number of transitions in Clear to Send (CTS).
dsr	number of transitions in Data Set Ready (DSR).
rng	number of transitions in Ring Indicator (RI).
dcd	number of transitions in Data Carrier Detect (DCD).
tx	number of transmitted asynchronous data bytes.
rx	number of received asynchronous data bytes.
frame	number of asynchronous framing errors detected.
parity	number of asynchronous parity errors detected.
overrun	number of asynchronous receive overruns detected.
brk	number of asynchronous break sequences detected.
buf_overrun	number of times the driver's receive buffer overflowed
txok	This value increments each time the transmitter finishes sending all queued data without error and becomes idle. If multiple HDLC frames are queued, txok is incremented only once.
txunder	number of times an HDLC frame transmit underrun occurred.
txabort	number of times an HDLC frame was aborted with an abort sequence.
txtimeout	number of times a transmit operation timed out.
rxshort	number of received HDLC short frames (less than two bytes if no CRC generation is configured or less than four bytes if CRC generation is configured. (frame discarded).
rxlong	number of received HDLC frames larger than 4096 bytes. (frame discarded).
rxabort	number of received HDLC abort sequences detected. (frame discarded).
rxover	number of times a received HDLC frame terminated due to a receiver overrun error. (frame discarded).
rxcrc	number of received HDLC frames received in error. (frame discarded).
rxok	number of successfully received HDLC frames.
exithunt	number of times the receiver exitted hunt mode while in HDLC mode (enabled via a MGSL_IOCWAITEVENT request).
rxidle	number of times the receiver detected an idle sequence while in HDLC mode (enabled via a MGSL_IOCWAITEVENT request).

MGSL_IOCWAITEVENT

```
int events = MgslEvent_DcdActive + MgslEvent_CtsInactive;
rc = ioctl(fd, MGSL_IOCWAITEVENT, &events);
```

Wait for specified event. Specify event of interest in events variable. On return, inspect events variable to determine which event occurred.

`MgslEvent_DsrActive`	wait for Data Set Ready (DSR) active
`MgslEvent_DsrInactive`	wait for Data Set Ready (DSR) inactive
`MgslEvent_Dsr`	same as specifying both DsrActive and DsrInactive use to poll current DSR state
`MgslEvent_CtsActive`	wait for Clear to Send (CTS) active
`MgslEvent_CtsInactive`	wait for Clear to Send (CTS) inactive.
`MgslEvent_Cts`	same as specifying both CtsActive and CtsInactive use to poll current CTS state
`MgslEvent_DcdActive`	wait for Data Carrier Detect (DCD) active
`MgslEvent_DcdInactive`	wait for Data Carrier Detect (DCD) inactive
`MgslEvent_Dcd`	same as specifying both DcdActive and DcdInactive use to poll current DCD state
`MgslEvent_RiActive`	wait for Ring Indicator (RI) active
`MgslEvent_RiInactive`	wait for Ring Indicator (RI) inactive
`MgslEvent_Ri`	same as specifying both RiActive and RiInactive use to poll current RI state
`MgslEvent_ExitHuntMode`	wait for receiver to detect opening flag or sync pattern
`MgslEvent_IdleReceived`	wait for receiver to detect idle pattern

MGSL_IOCGIF

```
int mode;
rc = ioctl(fd, MGSL_IOCGIF, &mode);
```

Get current serial interface mode and options. See `MGSL_IOCSIF` for option details.

MGSL_IOCSIF

```
int mode = MGSL_INTERFACE_RS232;
rc = ioctl(fd, MGSL_IOCSIF, mode);
```

Set serial interface mode (rs232, v35, rs422) and options. The mode applies only to adapters that have a software selectable serial interface (SyncLink PCIe and USB adapter). The serial interface options may not be available on all adapters.

MODES

Bits identified by `MGSL_INTERFACE_MASK` select the serial interface (rs232, v35, rs422). Interface bits are one of the following:

`MGSL_INTERFACE_DISABLED`	high impedance state which does not drive outputs or monitor inputs
`MGSL_INTERFACE_RS232`	RS-232
`MGSL_INTERFACE_V35`	V.35
`MGSL_INTERFACE_RS422`	RS-422, RS-485, RS-530, RS-449, and X.21

SyncLink PCIe card DIP switches are set to allow software selection or choose a fixed interface. The interface bits contain the current software selection or a constant switch selected value. The SyncLink USB adapter interface is set only by software with the interface bits. Other MicroGate hardware, including the GT series, uses jumpers to select the serial interface and the interface bits are ignored.

OPTIONS

The options portion of the serial interface IOCTL is defined as all other bits than those defined by `MGSL_INTERFACE_MASK`. These options may not be available on all adapters.

`MGSL_INTERFACE_RTS_EN`	Used in RS485 mode to disable serial interface outputs (DTR/RTS/TXD/AUXCLK) when RTS is off. This is used for RS485 bus applications. Only the SyncLink GT and SyncLink AC family of adapters support this option.
`MGSL_INTERFACE_HALF_DUPLEX`	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_INTERFACE_LL`	enable local loopback output signal This signal is used by some DCEs to enable a local loopback mode. This can also be used as a general purpose output.
`MGSL_INTERFACE_RL`	enable remove loopback output signal This signal is used by some DCEs to enable a remote loopback mode. This can also be used as a general purpose output.
`MGSL_INTERFACE_MSB_FIRST`	enable MSB first bit order. HDLC/async always use LSB first
`MGSL_INTERFACE_TERM_OFF`	disable termination for differential interfaces (USB Only) (RS422/485/530/V.35/X.21) Normally differential inputs have 120 Ohm termination. When this option is set, inputs are not terminated. This only applies to SyncLink USB and SyncLink PCIe. Other hardware, including GT series, control termination with jumpers and switch settings.

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 the `ioctl` call as described in the following sections.

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.

If an adapter does not support GPIO, the following `ioctl` codes will return the error `EINVAL`.

All general purpose I/O operations use a `gpio_desc` structure:

```
struct gpio_desc {
      __u32 state;
      __u32 smask;
      __u32 dir;
      __u32 dmask;
};
```

Each bit of each field represents a signal. Bit 0 controls signal 1, bit 1 controls signal 2, etc.
`state` - signal state (0 or 1)
`smask` - state mask 0=ignore associated bit in state
`dir` - signal direction (0=input, 1=output)
`dmask` - direction mask 0=ignore associated bit in dir

MGSL_IOCSGPIO

This ioctl code is used to set the input/output mode for configurable signals, and the signal state for dedicated outputs and I/O configured as outputs.

To set a signal direction (only valid for configurable I/O), set the associated bit in the dmask (direction mask) field to one and set the associated bit in the dir (direction) field to the desired mode (0=input, 1=output).

To set an output state (only valid for dedicated outputs and I/O configured as outputs), set the associated bit in smask (state mask) to one and set the associated bit in the state field to the desired state.

```
struct gpio_desc gpio;

gpio.state = 0  /* set state of signal 1 to 0 */
gpio.smask = 1; /* set state of signal 1 as specified in state field */
gpio.dmask = 3; /* set direction of signal 1 and signal 2 */
gpio.dir   = 1; /* signal 1 = output, signal 2 = input */

rc = ioctl(fd, MGSL_IOCSGPIO, &gpio);
```

MGSL_IOCGGPIO

This ioctl code is used to get the input/output mode for configurable signals, and the signal states. On successful return, smask and dmask are set to all ones (0xffffffff), dir contains the current mode for each signal, and state contains the current state for each signal.

```
struct gpio_desc gpio;
rc = ioctl(fd, MGSL_IOCGGPIO, &gpio);
```

MGSL_WAITGPIO

This ioctl call blocks until at least one of the specified input states is reached. One or more inputs may be monitored. Dedicated outputs and I/O signals configured as outputs cannot be monitored. When the call returns, at least one of the monitored signals is in the desired state. The state field contains the state of all signals at the point the wait is satisfied.

```
gpio.smask = 0xC0; /* monitor signals 7 and 8 */
gpio.state = 0x80; /* wait for signal 8 = 1 or signal 7 = 0 */
gpio.dir   = 0;    /* field ignored */
gpio.dmask = 0;    /* field ignored */
rc = ioctl(fd, MGSL_IOCWAITGPIO, &gpio);
if (rc)
        goto error;

/* test gpio.state */
if (gpio.state & 0x80) {
        /* signal 8 is one, do something */
}
if (!(gpio.state & 0x40)) {
        /* signal 7 is zero, do something else */
}
```

PYTHON

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

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

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

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

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

The module must be found in one of three ways:

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

The package is installed using the Python installation program (`pip`):

```
#pip install mgapi.tar.gz
```

Contents of `python` directory:

`mgapi.py`	Python API module
`mgapi.tar.gz`	Installable Python API package
`samples/2wire.py`	2-wire/bussed HDLC/SDLC sample
`samples/hdlc.py`	HDLC/SDLC sample
`samples/async.py`	asynchronous sample
`samples/bisync.py`	bisync/monosync sample
`samples/raw.py`	raw bitstream sample
`samples/xsync.py`	XSYNC sample
`samples/2wire.py`	2-wire (bussed) sample
`samples/signals.py`	control/status signal sample
`samples/cwrapper.py`	C API wrapper sample (for porting C applications)

A sample is run as shown below for the HDLC sample on device `/dev/ttyUSB0`:

```
#python hdlc.py /dev/ttyUSB0
```

Most samples send and receive data on a single port and require a loopback connector to be installed. Refer to the sample code for more details on setup and operation.

PORT OBJECT

A `Port` object is required for most tasks. A port name must be supplied. Port names for USB adapters have the form `dev/ttyUSBx`, where 'x' is the device number. Port names for PCI cards have the form `/dev/ttySLGx`, where x is the device number. The `Port` class function `enumerate` returns a list of available port names.

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

port = Port('/dev/ttyUSB0')
```

OPEN/CLOSE

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

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

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

```
port.close()
```

CONFIGURATION

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

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

During operation, alter port behavior using various properties.

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

SEND DATA

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

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

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

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

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

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

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

RECEIVE DATA

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

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

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

The `size` argument meaning depends on the protocol:

HDLC/SDLC

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

Other Protocols

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

Class Reference

Port

A **`Port`** object is required for most tasks. A port name must be supplied. PCI cards have a name of the form `/dev/ttySLGx` where x is a port number. USB devices have a name of the form `/dev/ttyUSBx` where x is a port number.

```
port = Port('/dev/ttyUSB0')
```

Methods

apply_settings

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

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

close

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

disable_receiver

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

disable_transmitter

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

enable_receiver

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

enable_transmitter

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

enumerate

This class method returns a list of valid port names.

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

get_defaults

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

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

get_settings

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

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

flush

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

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

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

is_open

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

open

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

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

read

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

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

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

The `size` argument depends on the protocol:

HDLC/SDLC

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

Other Protocols

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

set_fsynth_rate

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

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

The internal clock generator is used for:

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

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

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

wait

This method waits for a specified event (hardware or signal state). The method blocks until one or more specified events occur. The method accepts an integer containing bit flags specifying events to wait for and returns an integer indicating which events occurred.

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

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

write

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

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

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

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

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

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

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

Properties and Attributes

base_clock_rate

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

blocked_io

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

`True` (blocking)
`read` blocks until data is available then returns a `bytearray`

`write` blocks until data is queued then returns `True`
`flush` blocks until all data is sent then returns `True`

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

cts (Clear To Send)

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

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

dcd (Data Carrier Detect)

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

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

dsr (Data Set Ready)

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

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

dtr (Data Terminal Ready)

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

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

gpio

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

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

half_duplex

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

ignore_read_errors

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

interface

This integer property selects the serial interface standard. Valid interface values are defined by constants.

SyncLink PCIe card DIP switches are set to allow software selection or choose a fixed interface. This property contains the current software selection or is a constant switch selected value. The SyncLink USB adapter interface is set only by software with this property. Other MicroGate hardware, including the GT series, uses jumpers to select the serial interface and this property is ignored.

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

port.interface = Port.RS422
```

ll (Local Loopback)

This Boolean property controls the LL output state.

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

name

This read only string property contains the port name.

receive_count

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

receive_transfer_size

This integer property sets how many bytes of receive data are transferred from the device to the system at a time. Lower values reduce receive latency (time from receiving data until it becomes available to system) but increase overhead. (Linux only)

range=1-256, default=256
< 128 : programmed I/O (PIO), low data rate
>= 128 : direct memory access (DMA), MUST be multiple of 4

ri (Ring Indicator)

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

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

rl (Remote Loopback)

This Boolean property controls the RL output state.

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

rts (Request To Send)

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

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

rts_output_enable

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

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

signals

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

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

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

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

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

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

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

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

transmit_count

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

transmit_transfer_mode

This property sets the transmit transfer mode. Valid values are Port.DMA (direct memory access) and Port.PIO (programmed I/O). (Linux only)

termination

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

transmit_idle_pattern

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

PORT.DEFAULTS

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

PROPERTIES AND ATTRIBUTES

max_data_size

This integer attribute specifies the maximum number of bytes that are returned by `read` or supplied to `write`. The default value is 4096.

PORT.GPIO

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

PROPERTIES AND ATTRIBUTES

state

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

output

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

PORT.SETTINGS

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

PROPERTIES AND ATTRIBUTES

async_data_bits

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

async_data_rate

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

async_stop_bits

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

async_parity

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

auto_cts

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

auto_dcd

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

auto_rts

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

crc

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

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

discard_data_with_error

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

discard_received_crc

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

encoding

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

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

hdlc_address_filter

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

internal_clock_rate

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

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

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

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

internal_loopback

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

msb_first

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

protocol

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

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

receive_clock

This attribute selects the source of the receive clock.

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

receive_clock_invert

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

sync_pattern

This integer attribute specifies the synchronization pattern used for monosync, bisync and xsync protocols. The sync pattern size is protocol dependent: monosync = 1 byte, bisync = 2 bytes, xsync = 1 to 4 bytes as set by `xsync_sync_size` attribute. The transmission byte order is most significant to least.

An enabled receiver waits for a sync pattern then stores data byte aligned to the sync pattern until the receiver is disabled or forced to search for the next sync pattern. For monosync and bisync protocols the sync pattern is used as the transmit idle pattern. The xsync protocol has separate sync and idle patterns. The application adds the sync pattern to the start of write data. Specific applications may require more than one sync pattern at the start of data.

transmit_clock

This attribute selects the source of the transmit clock.

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

transmit_clock_invert

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

transmit_preamble_bits

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

transmit_preamble_pattern

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

xsync_block_size

This integer attribute sets the fixed block size for the xsync protocol. If set to 0, data is returned as a stream of bytes and the application determines block boundaries. Non zero values set the fixed block size with one block returned per `read` call. Valid values are 1 to 65535.

xsync_sync_size

This integer attribute sets the sync pattern size in bytes for the xsync protocol. Valid values are 1 to 4.

PROTOCOLS

This section provides an overview of supported serial protocols. Protocols define how framing, transparency, and timing are implemented.

Framing

Framing is a hardware mechanism to identify data boundaries and optionally check data integrity. HDLC uses flag patterns to mark the start and end of frames and CRC for integrity. TDM uses a sync signal to mark the start of fixed length frames. Asynchronous data uses start/stop bits on each character with parity for integrity. Bisync and monosync use sync patterns to indicate start of data and the application must detect end of data. Raw protocol has no hardware framing and the application is responsible for interpreting the stream of bits.

Transparency

Transparency is a mechanism to distinguish between data and non data patterns. HDLC uses zero insertion/deletion to distinguish between data and flags. HDLC is the only supported protocol that implements transparency at the hardware level.

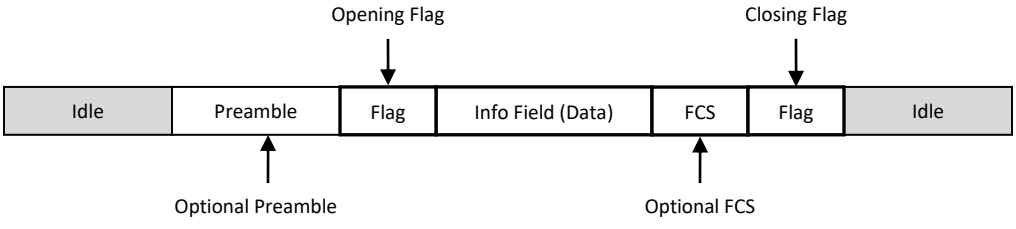
Timing

Serial communication requires a timing mechanism to coordinate data transfer. This can be an external clock signal, an internally generated clock, or a clock recovered from a data signal. The clock frequency determines the data transfer rate. The receiver and transmitter have separate clock configuration.

HDLC/SDLC

High Level Data Link Control (HDLC) is an international standard (ISO3309) based on SDLC (Synchronous Data Link Control), a protocol developed by IBM. This document uses HDLC to refer to both HDLC and SDLC.

Data is grouped into an information field of two or more bytes. The information field may be followed by an optional frame check sequence (FCS) such as CRC16 or CRC32. The FCS is calculated on the bits in the information field. The information field and FCS are framed with a non data pattern 01111110 (0x7e) called a flag. The collection of an opening flag, information field, FCS, and a closing flag is called a frame. A frame in progress can be aborted before the closing flag by sending a non data pattern called an abort, which is 7 or more consecutive ones. Aborted frames or frames with a FCS error should be ignored by the receiver.



HDLC Frame Format

An optional preamble may be sent before each frame. The preamble is useful for synchronizing a DPLL for clock recovery. The preamble pattern should be chosen to provide the maximum transitions for a given serial encoding standard. Refer to the DPLL section for details. Preambles are usually not used for applications with a separate data clock signal.

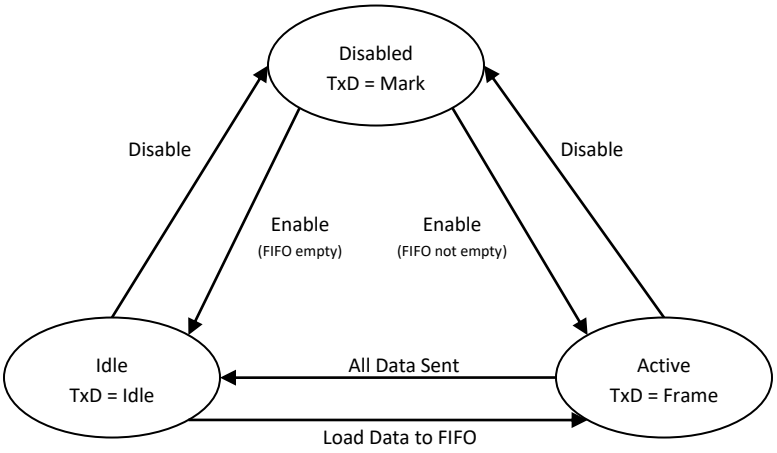
Leading bytes of the information field contain variable length address and control fields. The serial controller does not process the address or control fields, and treats the entire information field as data. Interpretation of the address and control fields is the responsibility of the device driver or application.

Data transparency is provided to distinguish between data and flag or abort patterns. This is accomplished with zero insertion and deletion. The controller automatically inserts a zero after any sequence of five consecutive ones in send data and automatically deletes a zero after any sequence of five consecutive ones in receive data. Zero insertion and deletion is only applied to the information field and FCS.

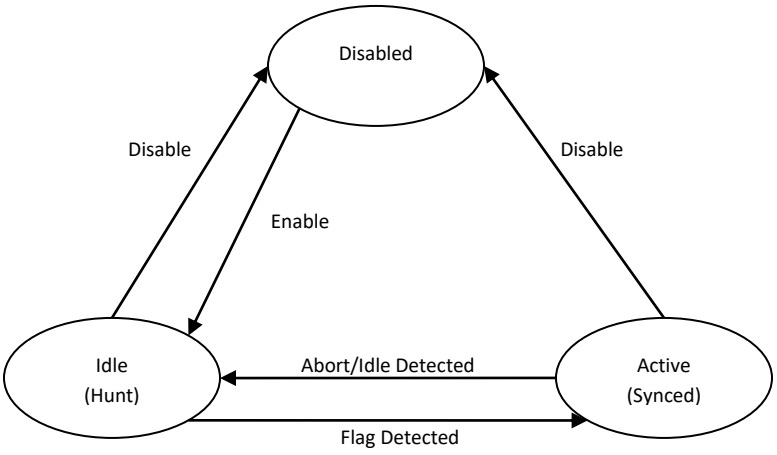
HDLC may use separate data clock signals or can recover data clocks from a data signal using DPLL (digital phase locked loop) clock recovery. There is one clock cycle per bit.

The HDLC transmitter has three states: disabled, idle, and active. The transmitter starts in the disabled state with the transmit data signal set to a constant mark. When software enables the transmitter with a bit in a control register the transmitter becomes idle. An idle transmitter sends a user configurable idle pattern, usually all ones or repeated flags. When software provides data to send, the transmitter becomes active and sends a frame containing the data. When the frame completes, the transmitter becomes idle. Software can disable the transmitter at any time using control bits in a register.

The HDLC receiver has three states: disabled, idle (hunt), and active (synced). The receiver start in the disabled state. When software enables the receiver with a bit in a control register, the receiver becomes idle and starts hunting for an opening flag. When a flag is detected, the receiver is active. An active receiver stores data between flags. When an abort sequence is detected, the receiver becomes idle. Software can disable the receiver at any time using control bits in a register.



HDLC Transmitter State Diagram

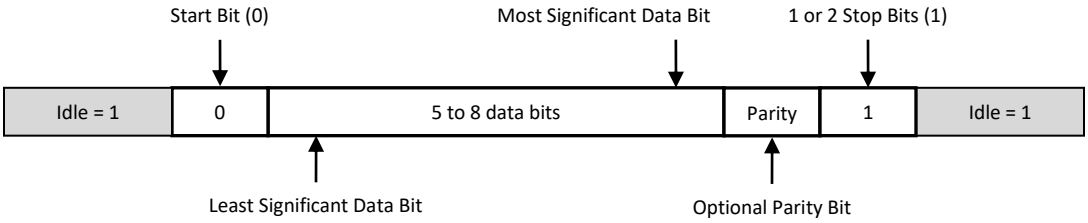


HDLC Receiver State Diagram

ASYNCHRONOUS

Asynchronous communication frames each character with a single start bit and one or two stop bits. Data length is configurable for 5 to 8 data bits per character. An optional parity bit (odd or even) is appended to the data. The idle line state is a logical 1. The start bit is a logical 0. Stop bits are a logical 1. Data is transmitted least significant bit first followed by the optional parity bit. The total character size is the combination of the start bit, data bits, optional parity bit, and stop bits. The total character size range is 7 to 12 bits. The number of data bits, stop bits, and use of parity must be configured in advance to match the settings of a remote station.

Data clocks are generated internally. The data rate must be chosen in advance to match that of a remote station. The receive clock runs at 8 or 16 times the selected data rate. This clock is used to sample the receive data line. The start bit is detected as the falling edge from the idle condition or the stop bits of the preceding character (1 to 0).



Asynchronous Character Format

ISOCHRONOUS

Isochronous is identical to asynchronous as described in the previous section, except a separate physical clock signal is used with the same frequency as the data rate (1x clock). Some models of SyncLink hardware (GT4e and USB) support the isochronous protocol by configuring the device for asynchronous communications with the `DataRate` member of the `MGSL_PARAMS` structure set to zero.

With `MGSL_MODE_ASYNC` and a `DataRate` of zero, the `Flags` and `ClockSpeed` fields of the `MGSL_PARAMS` structure control the clocking configuration. Usually a single clock supplied by a remote device drives the SyncLink transmitter and receiver. It is possible to use different clocks for transmit and receive or to generate the clock on the `AUXCLK` output by setting `ClockSpeed` to a non-zero value.

The benefit of isochronous is the data rate does not have to be identically configured in advance on both ends of the connection since timing is derived from a separate clock signal. The trade off is the added expense of the extra signal.

The following code fragment demonstrates isochronous using a single clock signal connected to the `RxC` input pin.

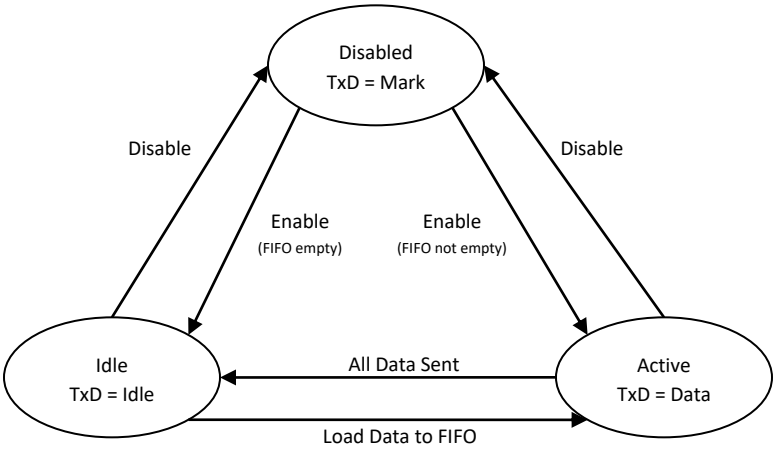
```
/*
 * Isochronous mode, format N-8-1
 * receive clock  = RxC input pin
 * transmit clock = RxC input pin
 * single clock from remote device connected to RxC input pin
 */
params.Mode = MGSL_MODE_ASYNC;
params.Loopback = 0;
params.Flags = HDLC_FLAG_RXC_RXCPIN + HDLC_FLAG_TXC_RXCPIN;
params.Encoding = HDLC_ENCODING_NRZ; /* ignored for isochronous */
params.ClockSpeed = 0;
params.CrcType = HDLC_CRC_NONE; /* ignored for isochronous */
params.DataBits = 8;
params.StopBits = 1;
params.Parity = ASYNC_PARITY_NONE;
params.DataRate = 0; /* selects isochronous */

/* set current device parameters */
rc = MgslSetParams(dev, &params);
```

RAW SYNCHRONOUS

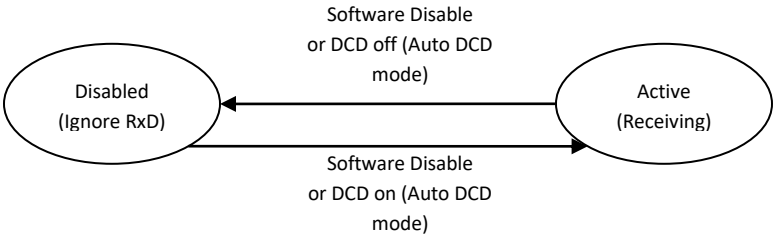
Raw synchronous operation performs no framing or synchronization. Data is sent bit for bit as supplied to the controller. Data is received bit for bit as seen on the receive data signal.

The raw transmitter has three states: disabled, idle, and active. The transmitter starts in the disabled state with the transmit data signal set to a constant mark. When software enables the transmitter with a bit in a control register the transmitter becomes idle. An idle transmitter sends a user configurable idle pattern. When software provides data to send, the transmitter becomes active and sends the data in an exact bit for bit representation. When no more data is available to send, the transmitter becomes idle. Software can disable the transmitter at any time using control bits in a register.



Raw Mode Transmitter State Diagram

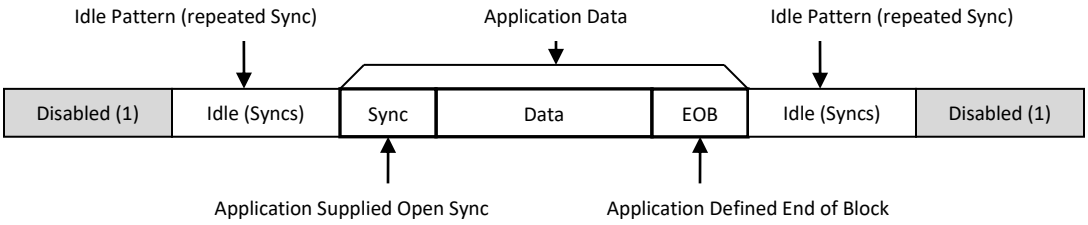
The raw receiver has two states: disabled, and active. The receiver start in the disabled state. When software enables the receiver with a bit in a control register, the receiver becomes active and starts storing receive data exactly bit for bit as seen on the receive data signal. Software can disable the receiver at any time using control bits in a register.



Raw Mode Receiver State Diagram

MONOSYNC AND BISYNC

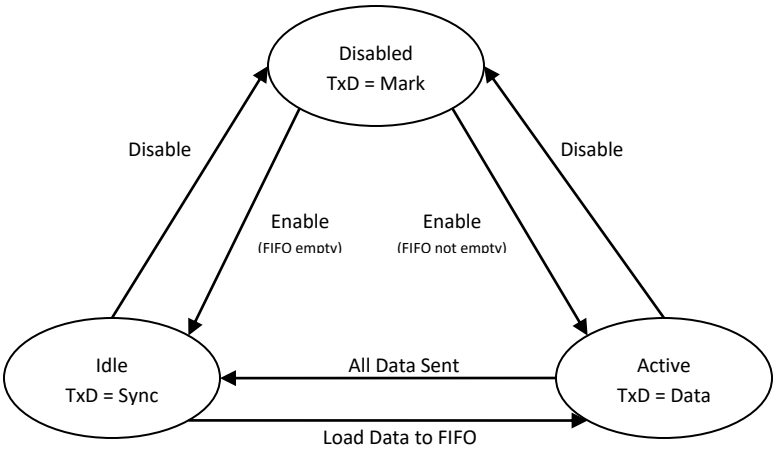
Monosync and Bisync operation is similar to raw synchronous operation. The difference is the receiver looks for an 8-bit (monosync) or 16-bit (bisync) pattern to signal synchronization and the following data is byte aligned to the synchronization pattern.



Monosync/Bisync Block Format

The monosync/bisync transmitter has three states: disabled, idle, and active. The transmitter starts in the disabled state with the transmit data signal set to a constant mark. When software enables the transmitter with a bit in a control register the transmitter becomes idle. An idle transmitter sends repeated sync patterns (8-bit for monosync and 16-bit for bisync). When software provides data to send, the transmitter becomes active and sends the data in an exact bit for bit representation. When no more data is available to send, the transmitter becomes idle. Software can disable the transmitter at any time using control bits in a register.

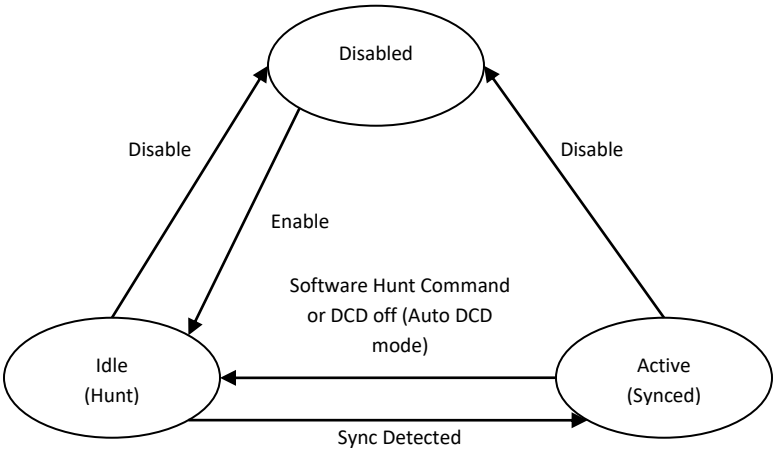
The transmitter does not automatically add a leading sync sequence to send data. Software must add the sync sequence manually to any data supplied to the transmitter.



Monosync/Bisync Transmitter State Diagram

The monosync/bisync receiver has three states: disabled, idle (hunt), and active (synced). The receiver starts in the disabled state. When software enables the receiver, the receiver becomes idle and starts hunting for the sync pattern (8-bit for monosync and 16-bit for bisync). When a sync pattern is detected, the receiver is active. An active receiver stores data bit for bit exactly as seen on the receive data signal. All data is byte aligned to the sync pattern. The receiver remains active until software disables the receiver or forces the receiver to idle/hunt.

Hardware does not detect the end of a data block. The end of block indication varies widely for monosync and bisync implementations and is the responsibility of software to detect. Typically an application enables the receiver and processes received data until the end of block condition is detected. The application then forces the receiver to idle/hunt.



Monosync/Bisync Receiver State Diagram

ENCODING

Serial encoding converts a logical one or zero into a coded signal used on the physical connection between end points. Send data is encoded and receive data is decoded. The table below describes each encoding standard.

The NRZ family (NRZ, NRZB, NRZI-space, NRZI-mark) has zero or one signal transitions per bit located at the start of the bit cell. The receiver samples the data signal at the center of the bit cell. NRZ type encoding is usually used with synchronous protocols that have a separate data clock signal. NRZ has fewer transitions per bit than biphase encoding which allows higher data rates for a bandwidth limited physical connection.

Note: NRZI without a space or mark modifier is often used as short hand for NRZI-space.

The biphase family has one to two transitions per bit located at the beginning and center of the bit cell. The receiver samples the data signal at ¼ and ¾ of the bit cell length. Biphase encoding is usually used with DPLL clock recovery as it guarantees at least one transition per bit cell to keep the recovered clock synchronized.

Serial Encoding	
Name	**Description**
NRZ	TxD is logical value (no encoding)
NRZB	TxD is logical value inverted
NRZI-space	If logical value 0, invert TxD at start of bit
NRZI-mark	If logical value 1, invert TxD at start of bit
Biphase-mark (FM1)	Invert TxD at start of bit. If logical value 1, invert TxD at center of bit.
Biphase-space (FM0)	Invert TxD at start of bit. If logical value 0, invert TxD at center of bit.
Biphase-level (Manchester)	Set TxD to logical value at start of bit. Invert TxD at center of bit.
Differential Biphase-level	If logical value 0, Invert TxD at start of bit. Invert TxD at center of bit.

BAUD RATE GENERATOR

The serial controller has a functional unit called the baud rate generator (BRG). The BRG divides a clock input (the base block) by a 16 bit integer (divisor) to generate a clock output. The clock output can be used internally for transmit and receive timing, output on the AUXCLK serial output pin and as a reference clock for the DPLL described in the next section.

The default base clock on the GT and USB devices is 14.7456MHz. Devices can be special ordered with an alternate base clock frequency. The GT2e/GT4e cards and the USB device include a programmable frequency synthesizer (described in a later section) than can be used as the base clock.

The BRG divides the base clock by a 16-bit integer (0 to 65535) to generate the data clock.

$$f_{data} = \frac{f_{base}}{divisor+1} \qquad divisor = \left(\frac{f_{base}}{f_{data}}\right) - 1$$

Example 1:

Data Clock = 9600bps, Base Clock = 14.7456MHz, Divisor = (14745600/9600) – 1 = 1535
Since 1535 is an integer in the range 0 to 65535 the 9600bps clock can be generated exactly.

Example 2:
Data Clock = 1Mbps, Base Clock = 14.7456MHz, Divisor = (14745600/1000000) – 1 = 13.7456
Since13.7456 is NOT an integer, the 1Mbps clock cannot be generated exactly.

The default 14.7456MHz base clock supports exact data rates of 9600, 38400, 115200, etc.

CLOCK RECOVERY

Synchronous modes usually get transmit and receive timing from the transmit and receive clock inputs. Alternatively, timing can be recovered from a received data signal using a digital phased locked loop (DPLL). This requires the exact data rate to be known in advance and specified in `ClockSpeed` field of the `MGSL_PARAMS` structure.

Use these options in the `flags` field of the `MGSL_PARAMS` structure to use DPLL clock recovery:

For receiver:

`HDLC_FLAG_RXC_DPLL`	Receive clock comes from DPLL (recovered)

For transmitter (use only one):

`HDLC_FLAG_TXC_DPLL`	Transmit clock comes from DPLL (recovered)
`HDLC_FLAG_TXC_BRG`	Transmit clock comes from BRG (generated)

Usually the receiver uses the recovered clock and the transmitter the generated clock.

The BRG supplies the DPLL a reference clock that is 8 or 16 times greater than the data rate. Specify this setting in the `flags` field of the `MGSL_PARAMS` structure:

`HDLC_FLAG_DPLL_DIV8`	reference clock = 8 x data rate (highest max rate)
`HDLC_FLAG_DPLL_DIV16`	reference clock = 16 x data rate (better precision)

The BRG divides the base clock by a 16-bit integer (0 to 65535) to generate the DPLL reference clock.

$$f_{ref} = \frac{f_{base}}{divisor+1} \qquad divisor = \left(\frac{f_{base}}{f_{ref}}\right) - 1$$

Example 1:
Data Clock = 9600bps
Reference Clock = Data Clock * 16 = 153,600Hz
Base Clock = 14.7456MHz
Divisor = (14,745,600/153,600) – 1 = 95
Since 95 is an integer in the range 0 to 65535, the 153,600Hz reference clock can be generated exactly for recovering the 9600bps data clock.

Example 2:
Data Clock = 10,000bps
Reference Clock = Data Clock * 16 = 160,000Hz
Base Clock = 14.7456MHz
Divisor = (14,745,600/160,000) – 1 = 91.16
Since 91.16 is NOT an integer the 160,000Hz reference clock cannot be generated exactly.

If the reference clock can't be generated exactly, clock recovery can still work if the difference between the exact rate and the actual rate is small enough and sufficient data signal transitions are maintained. A 10% difference is acceptable if using a biphase encoding (FM or Manchester) that guarantees a data transition every clock cycle.

Custom base clocks can be ordered and installed at the factory to allow exact recovery of data rates not supported by the standard base clock of 14.7456MHz. Contact Microgate to determine which custom base clock is required for your needs.

Serial Encoding with DPLL

DPLL clock recover is usually used with a biphase encoding (FM or Manchester) which guarantees a data transition every bit. DPLL can be used with NRZI encoding when using SDLC/HDLC mode because that mode guarantees a transition every 6 bits.

Preamble with DPLL

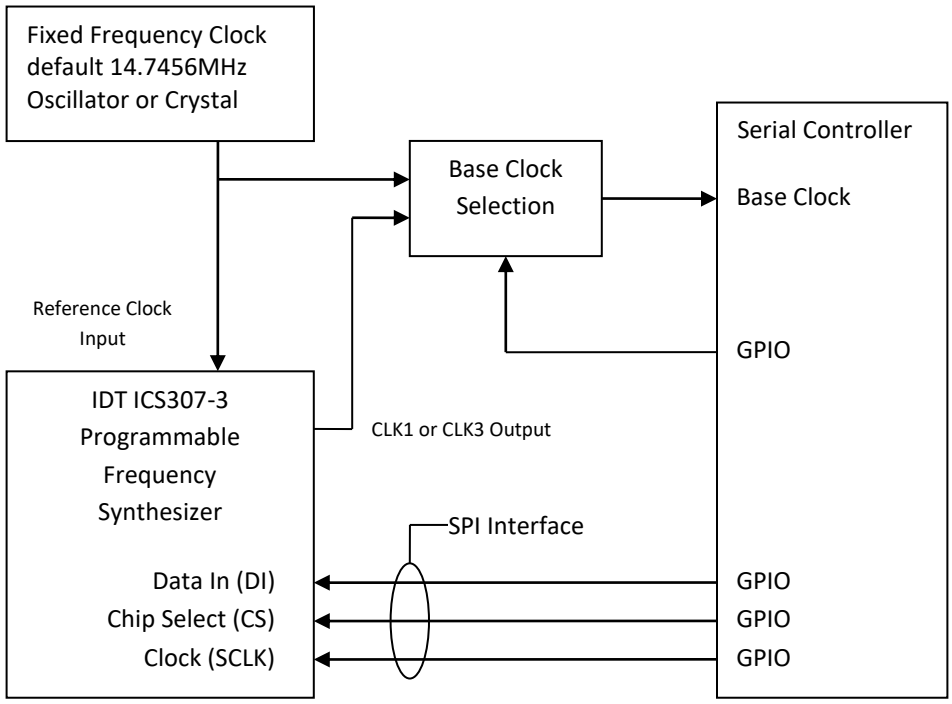
When a data signal is not continuously driven a preamble before each SDLC/HDLC frame is recommended to allow the DPLL to synchronize. Below is a list of suggested preamble patterns for different serial encodings:

Serial Encoding	Preamble Pattern
HDLC_ENCODING_NRZI_SPACE (NRZI)	HDLC_PREAMBLE_PATTERN_ZEROS
HDLC_ENCODING_BIPHASE_MARK (FM1)	HDLC_PREAMBLE_PATTERN_ZEROS
HDLC_ENCODING_BIPHASE_SPACE (FM0)	HDLC_PREAMBLE_PATTERN_ONES
HDLC_ENCODING_BIPHASE_LEVEL (Manchester)	HDLC_PREAMBLE_PATTERN_01

FREQUENCY SYNTHESIZER

Some models of SyncLink hardware have a programmable frequency synthesizer. The output of the synthesizer may be used as the base clock for the adapter. The synthesizer device is part number ICS307-3 manufactured by Integrated Device Technology (idt.com).

Below is an overview of the connection to the serial controller. The synthesizer reference clock is a fixed frequency clock source (oscillator or crystal). The synthesizer is programmed through an SPI interface connected to GPIO pins on the serial controller. Another GPIO pin selects between the fixed frequency clock source and the synthesizer. Refer to the Hardware User's Guide for your SyncLink device for the exact connections, GPIO assignments and type of fixed frequency clock source (oscillator or crystal).



The GPIO pins are controlled using the GPIO calls of the serial API as described in the GPIO section of this document. Sample code for programming the frequency synthesizer is included in the `mgapi/c/samples/fsynth` directory of the SDK.

Frequency synthesizer programming consists of a 132 bit word. For a description of the fields of the word, refer to the device datasheet for the ICS307-3 from idt.com. The 132 bit word is calculated by the Versaclock 2 Windows software provided by idt.com based on desired output values and error tolerances.

Note: Versions of Versaclock later than 2 do not support the ICS307-3 device. Contact idt.com for the older Versaclock 2 software required to program this device.

Values calculated by Versaclock can be copied to the Windows clipboard and then pasted into the sample `fsynth.c` program. The clipboard value requires manual formatting for use by the sample code. Below are instructions for calculating a value and using it with the sample code.

1. Run Versaclock 2 software and click **Select Part Number**

2. Select **ICS307-03-Clock** for SyncLink GT2e/GT4e cards or **ICS307-03-xtal** for SyncLink USB

3. Click the **Continue** button

4. Select **Manual Pin Assignment** from the **Options** menu

You should see three lines in the "Outputs" section labeled 8, 12, 14
SyncLink GT2e/GT4e uses pin 8 (CLK1) line
SyncLink USB uses pin 14 (CLK3) line

5. In **Ref freq (MHz)** edit box type: 14.7456

6. Leave **Vdd** pull down list set to 3.3V

7. On the appropriate line (pin 8 or pin 14) in **Outputs** section, enter two values:

Desired MHz : desired clock rate in MHz
Error ppm : use 100ppm (default for standard oscillators)

8. Click the **Calculate** button near bottom of window

9. Results appear in **Actual MHz** and **Error ppm** fields in Outputs section.

Verify the calculated error is within the specified range. If not, try generating a frequency that is a multiple of the desired rate and use the serial controller BRG to divide that for use.

10. Click the **Prog. word to Clipboard** button near the bottom of window.

11. Paste the result into the `fsynth.c` file near an existing table entry for the table associated with your SyncLink device type (`gt4e_table` for GT2e/GT4e or `usb_table` for USB).

Example: for 32.768MHz output on the SyncLink GT2e/GT4e card the clipboard value is:
`08001400D8A00000000000000001F9FE2`

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

`08001400D8A00000000000000001F9FE2` becomes
`08001400, D8A00000, 00000000, 0001F9FE, 2`

13. Format the values into a table of 5 32-bit values for use as a C language array initializer. The final digit is the most significant digit of a 32-bit value.

```
{0x08001400, 0xD8A00000, 0x00000000, 0x0001F9FE, 0x20000000}
```

14. Use the array initializer from the previous step to create a table entry for the desired frequency and place it in the table.

```
struct freq_table_entry gt4e_table[] =
{
        {12288000, {0x29BFDC00, 0x61200000, 0x00000000, 0x0000A5FF, 0xA0000000}},
        {14745600, {0x38003C05, 0x24200000, 0x00000000, 0x000057FF, 0xA0000000}},
        {16000000, {0x280CFC02, 0x64A00000, 0x00000000, 0x000307FD, 0x20000000}},
        {20000000, {0x00001403, 0xE0C00000, 0x00000000, 0x00045E02, 0xF0000000}},
        {30000000, {0x20267C05, 0x64C00000, 0x00000000, 0x00050603, 0x30000000}},
        {32000000, {0x21BFDC00, 0x5A400000, 0x00000000, 0x0004D206, 0x30000000}},
        {32768000, {0x08001400, 0xD8A00000, 0x00000000, 0x0001F9FE, 0x20000000}},
        {64000000, {0x21BFDC00, 0x12000000, 0x00000000, 0x000F5E14, 0xF0000000}},
        {0, {0, 0, 0, 0, 0}} /* final entry must have zero freq */
};
```

Once the frequency synthesizer has been programmed, it retains that value until reprogrammed or power is lost.

After programming the frequency synthesizer and selecting the synthesizer output as the base clock, use the serial API to inform the driver of the new value. The driver uses this value to calculate BRG and DPLL divisors.

```
rc = MgslSetOption(dev, MGSL_OPT_CLOCK_BASE_FREQ, 32768000);
```

This call needs to be made for every port on the adapter.

GENERIC HDLC NETWORKING

Generic HDLC is a Linux kernel facility supporting synchronous serial hardware and WAN protocols for network communications and applications. SyncLink drivers implement the Generic HDLC interface for use with this facility. When used in this way, the SyncLink device appears to the user mode network application as a standard network interface. Refer to previous sections of this document for details on installing and loading the SyncLink software and drivers.

The configuration of a serial network interface requires knowledge of basic Linux administration and the required protocols. This document is not intended to be a tutorial on Linux administration or network protocols. If you need more information on Linux administration or network protocols, research third party tutorials and books.

KERNEL CONFIGURATION

RHEL/CentOS 6.X includes precompiled Generic HDLC components for the following protocols: raw, Cisco HDLC, PPP and frame relay. If you are using one of these protocols, skip to the next section on using Generic HDLC.

If your distribution does not have the necessary Generic HDLC modules, consult your distribution documentation for details on obtaining the kernel source, configuring the kernel and building the kernel with these components.

USING GENERIC HDLC

Using Generic HDLC networking requires the following steps:

- load SyncLink device driver as described earlier in this document
- configure SyncLink device for the specific low level requirements
- load Generic HDLC module for network protocol (PPP, Cisco HDLC, etc)
- configure protocol module
- configure and enable network interface

When the SyncLink and Generic HDLC protocol modules are loaded, two devices are created for each serial port: a serial device and a network device. Both devices are associated with a single hardware port.

The serial device is named:

`/dev/ttySLGx`	PCI, PCI Express, PC104+ cards
`/dev/ttyUSBx`	SyncLink USB Adapter

where x is zero based instance number.

The network device is named `hdlcx`, where x is a zero based instance number.

Examples:

A single SyncLink GT adapter creates the serial device `/dev/ttySLG0` and the network device `hdlc0`.

A SyncLink GT4 four port adapter creates the serial devices:

`/dev/ttySLG0, /dev/ttySLG1, /dev/ttySLG2, /dev/ttySLG3`
and the network devices `hdlc0, hdlc1, hdlc2, hdlc3`.

Configuring and controlling WAN connections requires the following utility programs:

`mgslutil`	configure hardware options, supplied with synclink drivers, operates on serial device
`sethdlc`	control and configure WAN protocols, supplied with synclink drivers, operates on network device
`ifconfig`	control and configure networking options, part of Linux distribution, operates on network device
`modprobe`	manage kernel modules, part of Linux distribution

A sample shell script is provided to demonstrate control and configuration of connections. This script is a sample and **must** be modified for specific setups and applications. The script combines the individual steps necessary to setup a network connection into a single command for a single device.

`netctl` start/stop a network connection connection

Example use of script to start and stop a network interface:

```
#netctl start
#netctl stop
```

The main functions of the script are:

1. Load protocol module with `modprobe` utility.
2. Select and configure protocol module with `sethdlc` utility.
3. Configure serial port with `mgslutil` utility.
4. Configure network interface with `ifconfig` utility.

The protocol module names are:

`hdlc_cisco`	Cisco HDLC
`hdlc_ppp`	PPP
`hdlc_fr`	Frame Relay
`hdlc_raw`	raw (no encapsulation)

Modules are loaded with the `modprobe` command. This example loads the PPP protocol module:

```
#modprobe hdlc_ppp
```

The protocol is selected with the `sethdlc` command. This example selects the PPP protocol:

```
#sethdlc ppp
```

The serial port is configured with the `mgslutil` command. This example selects HDLC with 16 bit ITU CRC, NRZI encoding and external clocks on the first PCI card serial port:

```
#mgslutil /dev/ttySLG0 hdlc crc16 nrzi rxc txc
```

The network interface is configured with the ifconfig command. This example enables the interface for a PPP link with the local IP address of 192.168.2.1 and remote IP address of 192.168.2.2 on the first serial device:

```
#ifconfig hdlc0 192.168.2.1 pointopoint 192.168.2.2 up
```

These are only examples. The actual commands will differ depending on the requirements of a specific connection. Refer to Linux man pages for the `modprobe` and `ifconfig` commands. Run `setsdlc` without arguments for a list of valid options. Run `mgslutil` without arguments for a list of valid options. Source code is included for `sethdlc` and `mgslutil`. The `mgslutil` program uses the serial API described earlier in this document.