MICROGATE SERIAL COMMUNICATIONS

LINUX GUIDE

MicroGate Systems, Ltd

http://www.microgate.com

CONTENTS

PREFACE

VERSION REQUIREMENTS

SyncLink drivers and software have been tested with:

- Red Hat Enterprise Linux/CentOS version 6.X, kernel version 2.6.32-220.13.1.el6
- Red Hat Enterprise Linux/CentOS version 7.X, kernel version 3.10.0-229.14.1.el7

Other kernel versions from RHEL/CentOS 6.X/7.X distributions should work but have not been tested. Other distributions based on RHEL, such as Scientific Linux, have an improved chance of working but have not been tested and are not supported.

Different Linux distributions, different versions of RHEL/CentOS and different kernel versions may not be compatible. Before purchasing a SyncLink device, build and install the freely downloadable drivers in the target environment to verify compatibility.

If you encounter an incompatibility, 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.

Known Version Limits:
Kernels before 2.6.5 are not supported.
SyncLink USB is not supported on kernel versions before 2.6.28

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.

OVERVIEW

This guide describes the use of MicroGate serial communication devices with Linux based operating systems. Linux is a kernel that is combined with libraries and tools to form a complete Unix like operating system. A complete system is called a distribution. Red Hat, SUSE and Ubuntu are examples of Linux distributions. This document targets Red Hat Enterprise Linux and CentOS, a free distribution derived from RHEL. Other distributions may be used, but may require distribution specific procedures for installation and configuration that are beyond the scope of this document.

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

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

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

User mode applications access the serial device in one of two different ways: using standard network system calls or directly using MicroGate serial API (application programming interface) calls. The access method depends on application specific requirements. For network access, the application generally knows nothing about serial communications and relies on configuring the serial device and device driver to appear as a standard network device. The serial API allows direct control of the serial device by a custom serial application that controls the detailed operation.

In current Linux kernels, the Generic HDLC kernel component supplies a framework for using serial devices as network devices. This component provides several protocol options, including frame relay, PPP, and Cisco HDLC. An overview of this layer is provided later in this document. Generic HDLC is only used when configuring the SyncLink serial device as a network device. Custom serial applications directly accessing the SyncLink device driver do not use this layer. The Generic HDLC layer is not written or maintained by MicroGate.

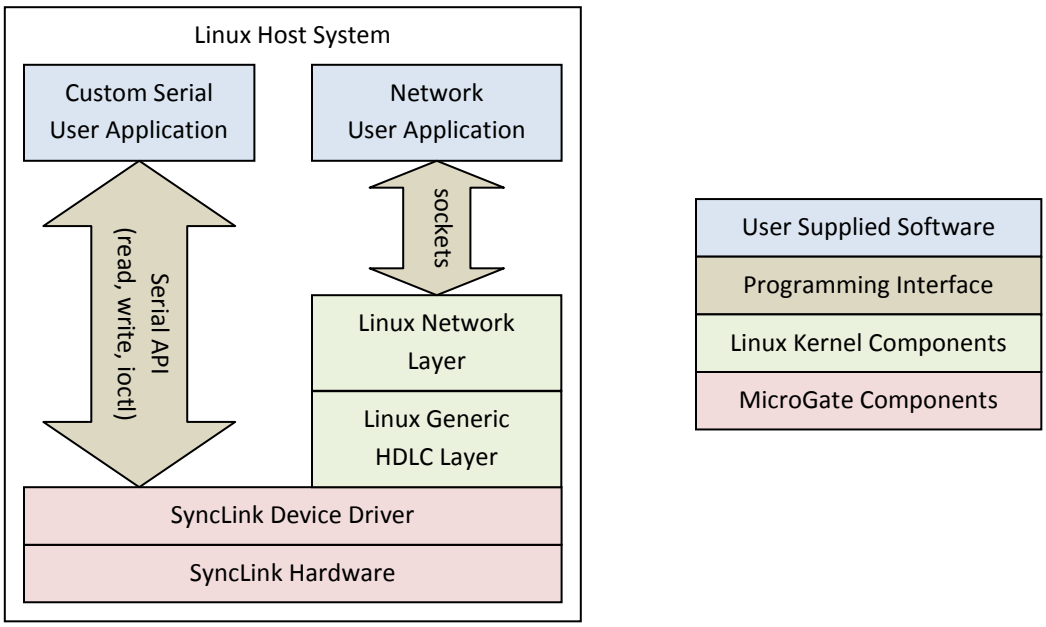


FIGURE 1 SOFTWARE OVERVIEW DIAGRAM

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

PHYSICAL CONFIGURATION AND INSTALLATION

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

SERIAL INTERFACE SELECTION

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

RS-232

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

V.35

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

RS-422/RS-485

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

HARDWARE INSTALLATION

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

VERIFY SYSTEM RECOGNIZES HARDWARE

After installing the hardware and starting the system, verify the system and Linux kernel recognizes the installed hardware. This step does not require the device driver and only verifies that the system sees that the hardware is present in the system. The procedure depends on the hardware type.

PCI/PCI Express/PC104+ Cards

Use the `lspci` command to list recognized PCI cards:

```
#lspci
```

This results in an entry for each recognized device. You should see an entry for a MicroGate Communication controller such as:

```
30:00.0 Communication controller: Microgate Corporation SyncLink GT4 Adapter
```

If you do not see an entry for the MicroGate device, verify the hardware is correctly installed. Try reseating the card in the slot or moving the card to a different slot.

USB to Serial Converter

Use the `lsusb` command to list recognized USB devices:

```
#lsusb
```

This results in an entry for each recognized device. You should see an entry for a MicroGate device, identified with and ID that starts with 2618:

```
Bus 002 Device 004: ID 2618:00b0
```

If you do not see an entry for the MicroGate device, verify the hardware is correctly installed. Try reseating the device cable or moving the device to a different USB connector. Verify the USB port is enabled in the host system BIOS setup. Verify the device driver for the host USB controller is configured and installed. Try a different USB cable.

SERIAL DEVICE CONNECTION

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

STANDARD CABLES

A cable connects the SyncLink serial device to another device. If the attached device is a DCE using a standard connector (RS-232, V.35, RS-530, etc) then a standard cable is used. See Figure 2 Standard cable connections for examples. In some cases, an adapter cable must be purchased from MicroGate to convert the DB-25M connector to the appropriate standard connector. The adapter cable can either plug directly to the attached device or can be used with a standard cable to increase the cable length if the adapter cable is not long enough.

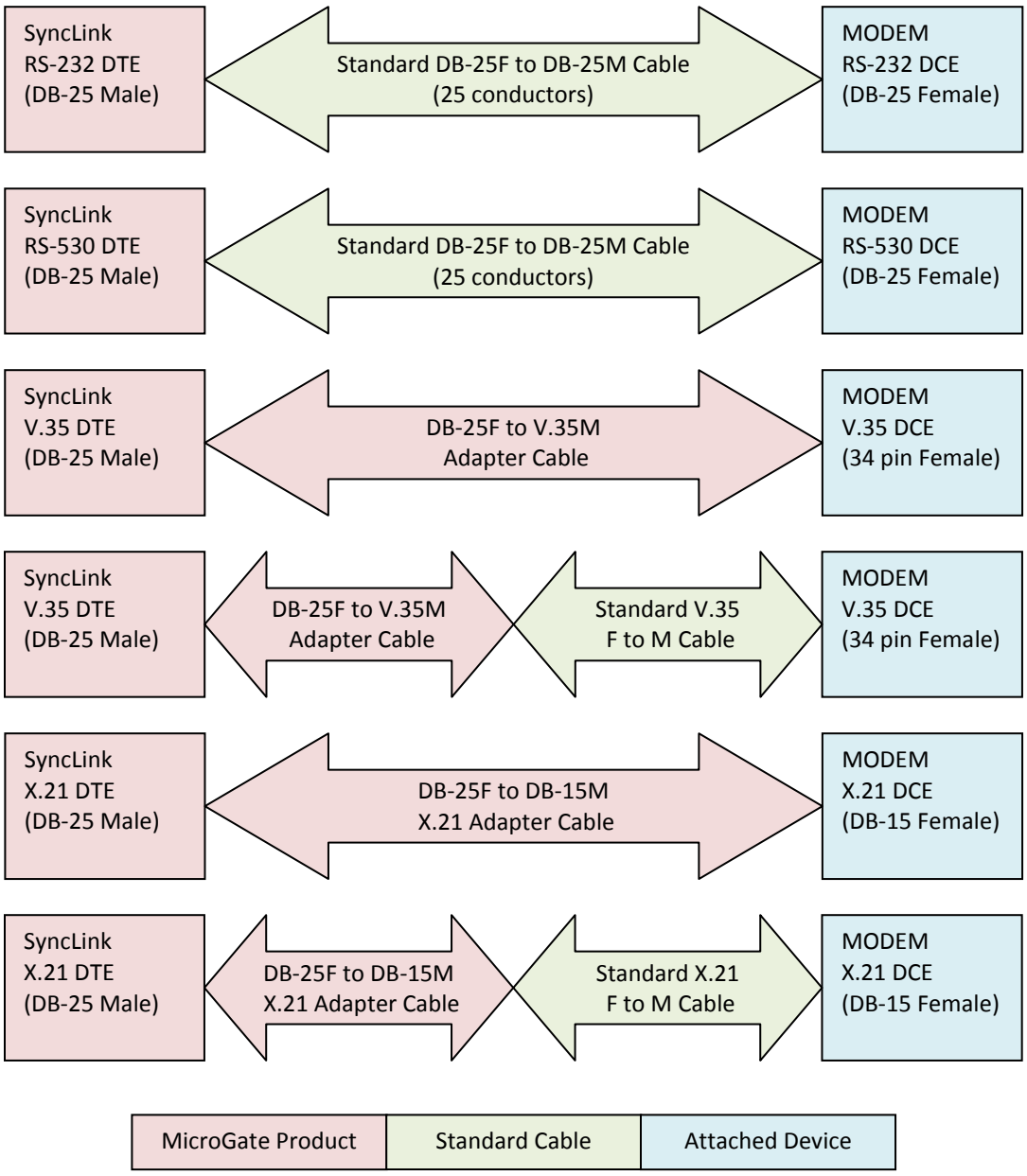


FIGURE 2 STANDARD CABLE CONNECTIONS

CUSTOM CABLES

If the attached device does not use a standard connector, such as custom control and measurement devices, then use the documentation for the SyncLink and custom devices to create an appropriate custom cable. Pinouts, electrical specification and configuration options are contained in the hardware user's manual (PDF) for your SyncLink device.

The first step in specifying a custom cable is determining which signals are required. The SyncLink device implements a full set of data, clock, control and status signals. Many custom devices only require data and clock signals. Some applications use clock recovery and only have data signals. Make a list of signals that are required by

the custom device and find the equivalent signal on the SyncLink device. SyncLink signals that are not required for an application can be left unconnected. A signal ground connection is usually required between endpoints, except for differential signals where the common mode voltage between endpoints can be guaranteed to meet electrical specifications without a signal ground connection. If in doubt, include a signal ground connection. A chassis/earth ground connection is recommended, which should be tied to the cable shield.

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

SOFTWARE 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.gz

Copy `linuxwan.tar.gz` to the `/usr/src` directory on the target system. Extract the package contents with the following command:

```
#tar xzvf linuxwan.tar.gz
```

This creates a new directory `synclink` that contains documentation, sample code and drivers.

In the `/usr/src/synclink` directory, run the following command to build sample code and supporting software:

```
#make
```

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

```
#make install
```

User space programs are included for configuring and testing the hardware from the command line.

`mgslutil.c`	command line configuration program
`mgsltest.c`	command line loopback test program
`sethdlc.c`	command line program for configuring serial device as network interface

These programs are included for testing and demonstration purposes. The functions they perform are usually implemented in custom serial programs written by the customer. In addition, sample shell scripts are included for testing and demonstration purposes that make use of the above programs.

`testloop`	peform loopback test on a device
`testmaster`	perform back to back testing through NULL modem as master
`testslave`	perform back to back testing through NULL modem as slave
`netctl`	configure, enable and disable serial device as network interface
`load-drivers.sh`	manually load drivers (rarely used)
`unload-drivers.sh`	manually unload drivers (rarely used)

CHECKING FOR PREBUILT DRIVERS

If your distribution includes prebuilt drivers for your SyncLink device, skip to the Loading Device Driver section. If prebuilt drivers are not available or you want to use the latest drivers, continue with the next section to build and install drivers.

Prebuilt drivers on RHEL/CentOS are located in the following directories:

RHEL/CentOS 6.X, kernel 2.6.32-220.13.1.el6: `/lib/modules/2.6.32-220.13.1.el6.i686`

RHEL/CentOS 7.X, kernel 3.10.0-229.14.1.el6: `/lib/modules/3.10.0-229.14.1.el7.x86_64`

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

```
#uname –r
```

Search for the SyncLink drivers with this command (use correct directory for distribution/kernel version):

```
#find /lib/modules/2.6.32-220.13.1.el6.i686 –name "synclink*"
```

Compare the results against the driver names:

`synclink_gt.ko`	PCI/PCI Express/PC104+ SyncLink GT cards
`synclink_usb.ko`	SyncLink USB Adapter

Note: The location of installed drivers for other distributions may be different than above. If so, consult the documentation for your distribution for the location of installed drivers.

DRIVER AND KERNEL VERSIONS

Device drivers must be compiled for the exact kernel version used for a system. A driver compiled for one kernel version will not load on a different kernel version. This means driver installation always includes compiling drivers from source code.

LOCATING KERNEL SOURCE

Drivers must be compiled for specific kernel versions. Kernel source code and a kernel configuration file are required to compile and install device drivers. RHEL/CentOS include the minimal kernel source required to compile and install drivers as part of the kernel-devel package. For other distributions the location of partial kernel source, if available, will be documented by the distribution.

RHEL/CENTOS 6.X

Partial kernel source is part of the `kernel-devel` package, which is installed by default. Partial kernel source is located in the following directory:

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

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

```
#uname –r
```

The MicroGate software package defaults to using `/usr/src/linux` as the location of the kernel source. Create a symbolic link from the actual kernel source location to `/usr/src/linux`:

```
#ln –s /usr/src/kernels/2.6.32-220.13.1.el6.i686 /usr/src/linux
```

RHEL/CentOS 7.X

RHEL/CentOS 7 does **not** automatically install the `kernel-devel` package needed to build drivers. Run the following commands as root to install the required package:

```
#yum install kernel-devel
#cd /usr/src/kernels/3.10.0-123.6.3.el7.x86_64
#make prepare
```

The `make` command reports errors even when successful. Building drivers as outlined in the next section is the only way to verify successful preparation. RHEL/CentOS 7 is only available for 64-bit processors.

Partial kernel source is now located in the following directory:

`/usr/src/kernels/`3.10.0-123.6.3.el7.x86_64

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

```
#uname –r
```

The MicroGate software package defaults to using `/usr/src/linux` as the location of the kernel source. Create a symbolic link from the actual kernel source location to `/usr/src/linux`:

```
#ln –s /usr/src/kernels/3.10.0-123.6.3.el7.x86_64 /usr/src/linux
```

Other Linux Distributions

Check the distribution documentation for procedures to install components needed to build drivers.

If your distribution does not have a partial kernel source package available for building drivers, you must download the complete kernel source package and perform the steps necessary to prepare the source for building drivers. The steps to do so are beyond the scope of this document. Consult your distribution documentation for details on getting kernel source code, creating a kernel configuration file to match the running kernel and preparing the source for building externally supplied drivers.

Compiling And Installing Drivers

With the kernel source setup completed, you can build and install device drivers. The device drivers are located in `/usr/src/synclink/drivers/kernel-2.6` for kernel versions 2.6 and later. The driver name depends on the hardware:

`synclink_gt.c`	PCI/PCI Express/PC104+ SyncLink GT cards
`synclink_usb.c`	SyncLink USB Adapter

Select drivers to build by editing:

`/usr/src/synclink/drivers/kernel-2.6/Makefile`

SyncLink GT and USB drivers are built by default. Add or remove synclink_gt.o or other entries from the obj-m line in the Makefile as needed.

Run the following command to build and install the device driver(s):

```
#make driver_install
```

The compiled driver will be installed in a subdirectory of the following directory:

```
RHEL/CentOS 6.X: /lib/modules/2.6.32-220.13.1.el6.i686

RHEL/CentOS 7.X: /lib/modules/3.10.0-123.6.3.el7.x86_64
```

If you encounter an error while building the driver, you may have missed or incorrectly performed a step in setting up the kernel source. If the kernel source is correctly setup, then there may be an incompatibility between the driver source and the kernel version. You may be able determine the problem from the error output and modify the driver source to match the kernel version. If not, then contact MicroGate with detailed distribution and kernel version information and report the specific errors.

If you encounter an incompatibility, 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.

Known Version Limits:
SyncLink USB is not supported on kernel versions before 2.6.28

LOADING DEVICE 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 is required for the system to automatically load the driver.

Verify the device driver is loaded using the lsmod command:

```
#lsmod | grep synclink
```

This command outputs a line of text for each loaded driver and displays only the lines containing synclink. Sample output is shown below.

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

This shows that the synclink_usb (USB serial converter) and synclink_gt (PCI/PCI Express) drivers are loaded. If you do not see the driver required for your device, try loading the device manually.

MANUALLY LOADING DEVICE DRIVER

Device drivers can be manually loaded using the modprobe command:

```
#modprobe synclink_gt
```

The above command loads the driver for PCI cards.

MANUALLY UNLOADING DEVICE DRIVER

Device drivers can be manually unloaded using the `modprobe` command. If a driver is in use, then the command will fail. Unloading a driver is usually not necessary, but is useful for updating a driver on the system without rebooting.

```
#modprobe -r synclink_gt
```

The above command unloads the driver for PCI cards.

DEVICE SPECIAL FILES

Devices are accessed by user mode applications using a device special file. This is a named entity in the file system that represents a device instance. Applications use the open() system call with the device special file name.

In modern Linux distributions, the device special files are created automatically by the system when a device driver is loaded and device instances are detected by the driver. The device special file entries are created in the /dev directory. The exact name depends on the hardware type.

`/dev/ttySLGx`	`synclink_gt` driver, PCI/PCI Express/PC104+ cards
`/dev/ttyUSBx`	`synclink_usb` driver, USB converters

In the above names, x is a zero based device instance number. For example /dev/ttySLG3 is the forth device provided by the synclink_gt driver. The assignment of device instance numbers depends on the order the system detects and initializes hardware. For PCI cards, this depends on the slot number. For USB devices, this depends on the port number. This order can change after installing or removing hardware or updating system software. The end user must determine the mapping of hardware to device instances for each target environment that uses multiple devices of the same type.

MANUALLY CREATING DEVICE SPECIAL FILES

Some Linux systems, particularly older systems, may require manual creation of device special files after the driver has loaded. Before trying to manual create device special files, be sure your system does not automatically create them. Most modern Linux systems automatically create device special files.

To manually create the device special file, use the mknod command. This command requires the device major number assigned to the driver and the device minor number assigned to the device instance. The device major number can be determined using the following command:

#cat /proc/devices

The output of this command looks like:

```
Character devices:
  1 mem
  4 /dev/vc/0
…
188 ttyUSB
248 ttySLG

Block devices:
  1 ramdisk
259 blkext
```

```
  7 loop
…
```

A device prefix is displayed to the right of the device major number. Serial devices are listed in the `Character devices` section. In the above example, the synclink_gt driver is assigned device major number 248 and the synclink_usb driver is assigned device major number 188.

Device minor numbers are assigned in ascending numerical order starting with a driver specific value:

```
synclink_gt        64
synclink_usb       0
synclink_cs        64
```

Now the mknod command can be executed to create the device special files:

```
#mknod /dev/ttySLG0 248 64
#mknod /dev/ttySLG1 248 65
#mknod /dev/ttySLG2 248 66
#mknod /dev/ttySLG3 248 67
```

The above commands create four device special files for four device instances of the synclink_gt driver for PCI cards using the device major number determined from /proc/devices.

WARNING: Device major numbers can change each time the device driver is loaded. This requires old device special files to be deleted and new ones created after each driver load. This is only required if the system does not automatically create device special files.

A sample shell script `load-drivers.sh` is included with the MicroGate software package. This script demonstrates manually loading device drivers and manually creating device special files.

DRIVER DEBUG OUTPUT

The device driver has the ability to 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 for the SyncLink drivers is included with the MicroGate software package: /usr/src/synclink/synclink-modprobe.conf

```
#
# sample modprobe configuration file for synclink drivers
#
# copy this file to the /etc/modprobe.d directory and edit
# the values below to control debug output to the system log
#

# SyncLink PCI, PCI Express, PC104+ cards
options synclink_gt  debug_level=3

# SyncLink USB
options synclink_usb debug_level=0

# N_HDLC line discipline
# Note: omit underscore in parameter name for this module
options n_hdlc       debuglevel=3
```

Edit this file to adjust debug_level to the desired value and copy the file to the /etc/modprobe.d directory. Then reload the driver for the new value to take effect. Be sure to disable logging when testing is complete. The example debug configuration enables informational events for both the synclink_gt driver and the N_HDLC line discipline. Both are needed to capture all data and information.

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

Debug output is sent to the system log, which is usually the file /var/log/messages. Consult the documentation for your distribution to locate the system log output file if it is not in the default location.

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: ttySLG 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
```

TESTING INSTALLATION

Once hardware, device drivers and supporting software have been built, installed and the driver loaded you are ready to test your setup.

INTERNAL LOOPBACK TEST

This test opens a device instance, configures the device for internal loopback then sends and receives data. If successful, this demonstrates the hardware, drivers and supporting software are correctly installed.

The shell script testloop is provided in the MicroGate software package to peform this test. The script can be edited to alter the test configuration. The script takes a device name as an argument:

#./testloop /dev/ttySLG0

The above command tests the first PCI device. The results should look like:

```
internal loopback test on /dev/ttySLG0, mode=hdlc speed=19200, 10 Packets of
512 bytes
Removing existing device nodes ...
Creating new device nodes ...

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#2, sending 512 bytes...send OK...receive OK
loop#3, sending 512 bytes...send OK...receive OK
loop#4, sending 512 bytes...send OK...receive OK
loop#5, sending 512 bytes...send OK...receive OK
loop#6, sending 512 bytes...send OK...receive OK
loop#7, sending 512 bytes...send OK...receive OK
loop#8, sending 512 bytes...send OK...receive OK
loop#9, 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
```

If the test fails, review previous sections and verify installation is correct. 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 a user mode application using standard systems calls. All documentation and sample code uses the C programming language. A working knowledge of C programming in a Linux environment is required. C definitions required for configuring a SyncLink device are contained in the `synclink.h` header file included with the MicroGate software package. This header must be included in a

custom serial user mode program. The standard Linux header file `termios.h` should also be included to access standard Linux serial device calls.

Sample source code is included for each serial protocol:

- sample-hdlc synchronous HDLC/SDLC application
- sample-raw synchronous raw application
- sample-async asynchronous application
- sample-bisync byte synchronous (BISYNC/MONOSYNC) application
- sample-xsync extended byte synchronous application
- sample-fsynth programming GT4e/USB frequency synthesizer

The sample code for each protocol is separated into a send program and a receive program for simplicity. Use the sample code as a starting point for writing your own code. The sample code can be run on two SyncLink ports connected by a NULL modem or crossover cable, with the receive code run on one port and the send code on the other.

LINUX LINE DISCIPLINES

Linux uses a kernel component called a **line discipline** to modify the exchange of data between a serial device and a user mode serial application. These modules are part of the Linux kernel and are not provided by MicroGate. An application selects the current line discipline using the ioctl() system call.

The default line discipline N_TTY is used for asynchronous communications and includes optional features for processing special characters and controlling the flow of data. For synchronous (raw, HDLC, bisync, monosync) modes, use the N_HDLC line discipline. This line discipline does not modify or process data but does maintain frame or buffer boundaries such that each call to read() or write() transfers a single frame or buffer of data.

A user mode serial application selects the appropriate line discipline (N_TTY or N_HDLC) for the mode of operation as part of the configuration process. The sample code included with the MicroGate Linux package demonstrates the selection of the line discipline. A line discipline is only used for custom serial applications and is not used for network user applications as described in a later section.

In rare instances, some distributions may not include a prebuilt N_HDLC line discipline module. If necessary, consult your distribution documentation for details on enabling the N_HDLC line discipline in the kernel configuration and rebuilding the kernel.

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 DEVICE

A user application must have a file descriptor (integer identifier) obtained from the operating system through the open() call to supply to other functions. The file descriptor is a token representing a device instance and is returned by open() for the specified special device file name. The code below opens a file descriptor for the first PCI device, performs some function then closes the file descriptor when done.

```
char devname[] = "/dev/ttySLG0"; /* device name */
int fd; /* file descriptor */

/* open device */
fd = open(devname, O_RDWR | O_NONBLOCK, 0);

/* other code goes here */

/* program is done, close device */
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.

CONFIGURE DEVICE

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;

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

/* set current device parameters */
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 size = sizeof(buf);          /* size of buffer */
int rc;                          /* read() return value */

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

The format of the data depends on the protocol. For frame oriented SDLC/HDLC, each read() call returns a single complete frame of data. For other synchronous protocols, each read() call returns a fixed amount of data with no formatting, requiring the application to interpret the data stream and detect message boundaries.

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 can be changed 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 */
}
```

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

The write() call can be blocking or non-blocking. In blocking mode, the call returns when data is accepted 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. The blocking mode can be changed with the fcntl() call.

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

IOCTL CALLS

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 the ioctl() call 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 device specific code.

STANDARD TTY 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 – SET LINE DISCIPLINE

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

Set the current line discipline. Choose the appropriate line discipline for the operating mode as part of the configuration process. Use N_TTY for asynchronous mode and N_HDLC for synchronous modes (SDLC, HDLC, raw, bisync, monosync, xsync).

TIOCMGET - GET MODEM SIGNAL STATES

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

Return serial control and status signals in sigs argument. Active signals are indicated with following bit definitions:

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

TIOCMBIS - ENABLE MODEM OUTPUT SIGNALS

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

Enable modem control signals specified by sigs argument. Enabled signals are indicated with the following bit definitions, with other signals left in the current state:

```
TIOCM_RTS   Request To Send (output signal)
TIOCM_DTR   Data Terminal Ready (output signal)
```

TIOCMBIC DISABLE MODEM CONTROL SIGNALS

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

Disable modem control signals specified by sigs argument. Disabled signals are indicated with the following bit definitions, with other signals left in the current state:

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

TIOCMSET SET MODEM CONTROL SIGNAL STATES

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

Set state of modem control signals specified by sigs argument. Enabled signals are indicated with the following bit definitions, other signals are disabled.

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

TIOCMIWAIT - WAIT FOR SERIAL STATUS SIGNALS TO CHANGE

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

Wait for one or more serial status signals to change. Specify signals of interest with the following bit definitions:

```
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 - GET COUNT OF SERIAL STATUS SIGNAL CHANGES

```
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 - GET COUNT OF PENDING SEND DATA

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

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

MGSL_IOCSPARAMS - SET DEVICE CONFIGURATION

```
MGSL_PARAMS params;

params.mode = MGSL_MODE_HDLC;
params.loopback = 0;
/* ... other application defined settings ... */

rc = ioctl(fd, MGSL_IOCSPARAMS, &params);
```

Set device configuration specified by MGSL_PARAMS structure. The MGSL_PARAMS structure is 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 - COMMUNICATIONS MODE/PROTOCOL

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

MGSL_MODE_ASYNC	character oriented no external clocks per character hardware framing per character parity check (none/even/odd)
MGSL_MODE_HDLC	bit synchronous hardware framing and synchronization (flags) hardware transparency (0 bit stuff/removal) hardware CRC check/generation (none/16 bit/32 bit) Note: SDLC and HDLC are the same in this context
MGSL_MODE_RAW	bit synchronous no hardware framing no hardware synchronization no hardware transparency
MGSL_MODE_BISYNC	byte synchronous 16 bit hardware synchronization no hardware framing no hardware transparency
MGSL_MODE_MONOSYNC	byte synchronous 8 bit hardware synchronization no hardware framing no hardware transparency
MGSL_MODE_XYNC	extended byte synchronous (1 - 4 byte sync pattern) no hardware framing no hardware transparency This mode is only available on select GT family cards and only by special order. Contact Microgate for details.
MGSL_MODE_BASE_CLOCK	This option does not select a protocol mode and is a special case used to set the base clock value in cycles per second as specified by clock_speed field. Other fields are ignored when mode is set to this value. See the clock_speed section below for details.

LOOPBACK – ENABLE/DISABLE INTERNAL LOOPBACK MODE

The loopback field of the MGSL_PARAMS structure enables or disables the internal loopback mode. 0 = normal operation, 1 = loopback mode. When enabled, the transmit data signal is connected internally to the receive data

signal and clocks are generated internally by the baud rate generator as specified by the clock_speed field. Use internal loopback mode to test the operation of the serial controller without external line drivers or devices.

FLAGS – PROTOCOL OPTIONS

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

Receive Clock Source Flags

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

HDLC_FLAG_RXC_DPLL	Receive clock is recovered from the receive data signal using the DPLL (digital phase locked loop). The clock_speed member of the MGSL_PARAMS structure specifies the expected data rate.
HDLC_FLAG_RXC_BRG	Receive clock is generated with baud rate generator (BRG) at the speed specified in the clock_speed member of the MGSL_PARAMS structure.
HDLC_FLAG_RXC_RXCPIN	Receive clock is supplied by an external device on the RxC input pin.
HDLC_FLAG_RXC_TXCPIN	Receive clock is supplied by an external device on the TxC input pin.

Transmit Clock Source

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

HDLC_FLAG_TXC_DPLL	Transmit clock is recovered from the receive data signal using the DPLL (digital phase locked loop). The clock_speed member of the MGSL_PARAMS structure specifies the expected data rate.
HDLC_FLAG_TXC_BRG	Transmit clock is generated with baud rate generator (BRG) at the speed specified in the clock_speed member of the MGSL_PARAMS structure.
HDLC_FLAG_TXC_RXCPIN	Transmit clock is supplied by an external device on the RxC input pin.
HDLC_FLAG_TXC_TXCPIN	Transmit clock is supplied by an external device on the TxC input pin.

Digital Phase Lock Loop Divisor

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

HDLC_FLAG_DPLL_DIV8	expected data rate = reference clock divided by 8
HDLC_FLAG_DPLL_DIV16	expected data rate = reference clock divided by 16

Miscellaneous Flags (any combination allowed)

HDLC_FLAG_AUTO_CTS	Enable transmitter only when CTS input is active.
HDLC_FLAG_AUTO_DCD	Enable receiver only when DCD input is active.
HDLC_FLAG_AUTO_RTS	When set, the driver automatically asserts RTS at when sending data and negates RTS when done sending. If RTS is active when a transmit request is made, the driver will not manipulate the state of RTS.

ENCODING – SELECT REPRESENTATION OF LOGICAL 1 OR 0 DATA

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

HDLC_ENCODING_NRZ	NRZ (NRZ-L) is an unencoded data signal. high = 1, low = 0
HDLC_ENCODING_NRZB	NRZB is an inverted data signal. high = 0, low = 1
HDLC_ENCODING_NRZI_MARK	(NRZ-M) invert at start of bit if 1
HDLC_ENCODING_NRZI_SPACE	(NRZI or NRZ-S) invert at start of bit if 0
HDLC_ENCODING_BIPHASE_MARK	(FM1) invert at start of bit, invert at 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) start of bit: 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 – GENERATED OR RECOVERED DATA RATE

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

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

divisor = (base clock/data rate) - 1

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

The serial card can be purchased with a different base clock. This option is installed at the factory. When a base clock other than 14.7456MHz is installed, the driver must be configured to use the new value by calling 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 – SDLC/HDLC ADDRESS FILTER

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

CRC_TYPE – FRAME CHECK OPTION (SDLC/HDLC)

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

HDLC_CRC_NONE	Don't send CRCs on transmit, don't check CRCs on receive.
HDLC_CRC_16_CCITT	16 bit CRC Polynomial: $x^{16} + x^{12} + x^5 + 1$
HDLC_CRC_32_CCITT	32 bit CRC Polynomial: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
HDLC_CRC_RETURN_EX	combine with HDLC_CRC_16_CCITT or HDLC_CRC_32_CCITT to return received CRC value and a status byte to application
HDLC_CRC_RETURN_EX	combine with HDLC_CRC_16_CCITT or HDLC_CRC_32_CCITT to return received CRC value and a status byte to application

HDLC_CRC_RETURN_EX AND ACCESSING CRC CODES

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. For example, an application may be able to apply an error recovery algorithm to a bad frame to rebuild the data.

Add the HDLC_CRC_RETURN_EX flag to the CRC type in the MGSL_PARAMS structure to receive both good and bad frames.

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

configures the driver to use CCITT CRC16, and pass extended frame information back to the application.

When HDLC_CRC_RETURN_EX is set, a read() call returns the received 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 – PREAMBLE PATTERN SELECTION

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

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

PREAMBLE_LENGTH – PREAMBLE LENGTH SELECTION

This member of the MGSL_PARAMS structure selects the length in bytes of the preamble pattern.

HDLC_PREAMBLE_LENGTH_16BITS
HDLC_PREAMBLE_LENGTH_32BITS
HDLC_PREAMBLE_LENGTH_64BITS

MGSL_IOCGTXIDLE - GET TRANSMIT IDLE OR MONOSYNC/BISYNC SYNC PATTERN

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

This ioctl code gets the current transmit idle pattern in SDLC/HDLC mode or the sync pattern in MONOSYNC/BISYNC modes. See MGSL_IOCSTXIDLE (Set Transmit Idle Mode) for more details.

MGSL_IOCSTXIDLE - SET TRANSMIT IDLE MODE OR MONOSYNC/BISYNC SYNC PATTERN

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

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

HDLC_TXIDLE_FLAGS	stream continuous flags.
HDLC_TXIDLE_ONES	send continuous ones.
HDLC_TXIDLE_MARK	send continuous marks.
HDLC_TXIDLE_ALT_ZEROS_ONES	send alternating zeros and ones.
HDLC_TXIDLE_ZEROS	send continuous zeros.

HDLC_TXIDLE_ALT_MARK_SPACE	send alternating marks and space.
HDLC_TXIDLE_SPACE	send continuous space.
HDLC_TXIDLE_CUSTOM_8	send arbitrary 8 bit idle pattern specified in the least significant 8 bits of value. example: (HDLC_TXIDLE_CUSTOM + 0x96) will use an 8 bit idle pattern of 0x96
HDLC_TXIDLE_CUSTOM_16	send arbitrary 16 bit idle pattern specified in the least significant 16 bits of value. example: (HDLC_TXIDLE_CUSTOM + 0x96aa) will use a 16 bit idle pattern of 0x96aa WARNING: specifying 16 bit idle pattern will disable preamble feature. preamble and 16 bit idle patterns can not 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 - ENABLE/DISABLE TRANSMITTER

```
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 mark. When enabled, the transmitter either sends data or the idle pattern.

MGSL_IOCRXENABLE - ENABLE/DISABLE RECEIVER

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

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

The serial receiver has three possible 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 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 data rates and 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 - SET EXTENDED SYNC PATTERN

```
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 - GET EXTENDED SYNC PATTERN

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

Get the current 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 - SET EXTENDED SYNC CONTROL

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

This 32-bit value controls 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 - GET EXTENDED SYNC CONTROL

```
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 - ABORT HDLC SEND FRAME IN PROGRESS

```
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 - GET STATISTICS

```
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`	asynchronous mode: number of times the driver's receive buffer overflowed. HDLC mode: number of times the driver's receive buffers were exhausted and had to shutdown the receiver until buffers became available. (data lost).
`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 - WAIT FOR SPECIFIED EVENTS

```
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 - GET SERIAL INTERFACE MODE AND OPTIONS

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

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

MGSL_IOCSIF - SET SERIAL INTERFACE MODE AND OPTIONS

```
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 (USB adapter). The serial interface options may not be available on all adapters.

MODES

The portion of the mode value identified by MGSL_INTERFACE_MASK selects the serial interface mode (rs232, v35, rs422). The value of the mode can be one of the following:

MGSL_INTERFACE_DISABLED	high impedance state which does not drive outputs or monitor inputs
MGSL_INTERFACE_RS232	RS-232 : single ended, low speed interface commonly used with modems
MGSL_INTERFACE_V35	V.35 is a combination single ended (status/control) and differential (data/clocks) legacy standard used for CSU/DSU connections at medium speeds.
MGSL_INTERFACE_RS422	Differential serial standard for high speeds. Depending on the attached cable, this setting is used for the RS-530, RS-449, and X.21 standards.

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 for monosync, bisync, xsync, and raw modes. Default bit order is LSB first. HDLC and asynchronous always use LSB first. This feature is only available on select GT family cards and only by special order. Contact Microgate for more details.
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 option only applies to the USB adapter. PCI card 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 - SET GENERAL PURPOSE I/O OPTIONS

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 - GET GENERAL PURPOSE I/O OPTIONS

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 - WAIT FOR SPECIFIED GPIO INPUT STATE

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 */
}
```

SERIAL PROTOCOL OVERVIEW

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

FRAMING AND TRANSPARENCY

Framing is a mechanism to identify boundaries of a data grouping, such as a separate control signal or non data patterns in a data signal. Transparency is a mechanism to distinguish between data and non data patterns.

SYNCHRONIZATION AND ALIGNMENT

Synchronization is a mechanism to identify the presence of a data signal and to define the alignment and boundaries of data units. The synchronization mechanism is often closely related to the framing mechanism.

TIMING AND CLOCK SOURCE

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

HDLC/SDLC

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

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

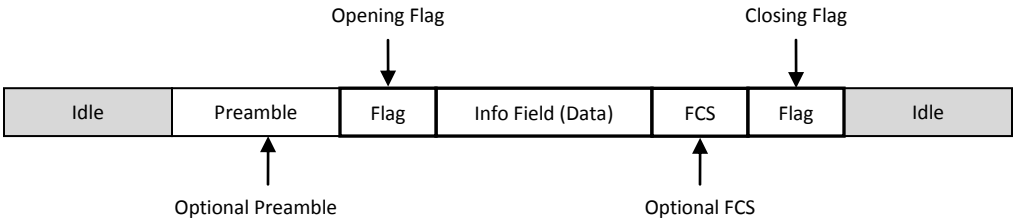


Illustration 3: HDLC Frame

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.

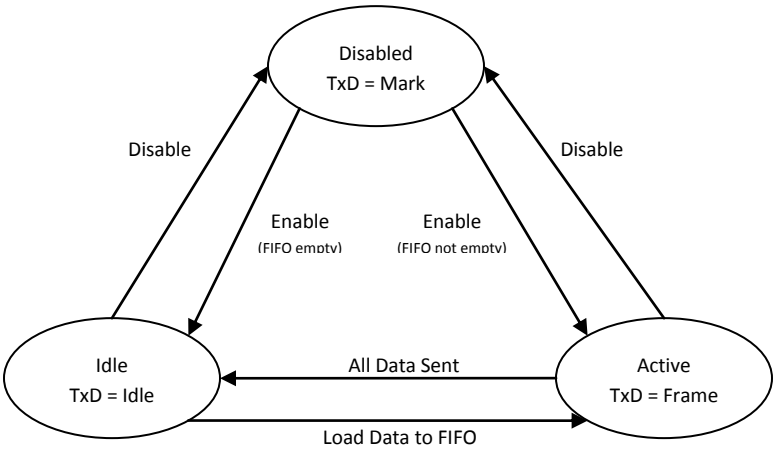


Illustration 4: HDLC Transmitter State Diagram

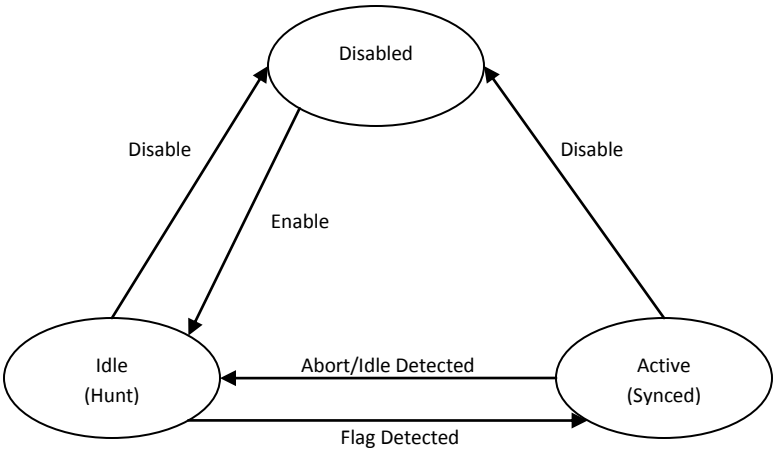


Illustration 5: HDLC Receiver State Diagram

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).

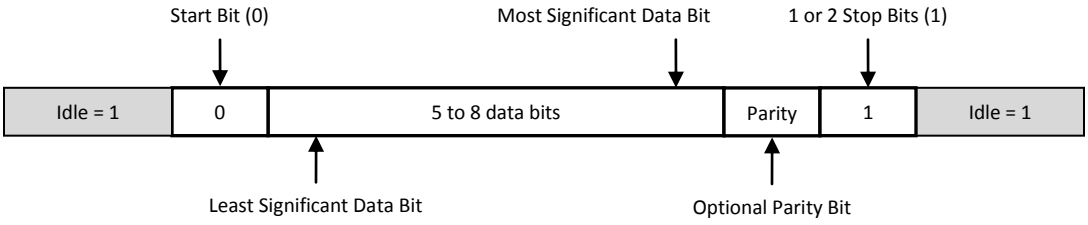


Illustration 6: Asynchronous Character

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.

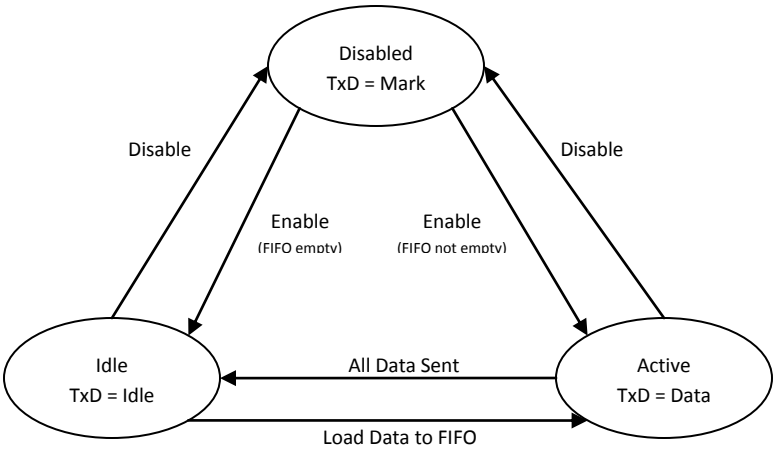


Illustration 7: Raw Transmitter State Diagram

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.

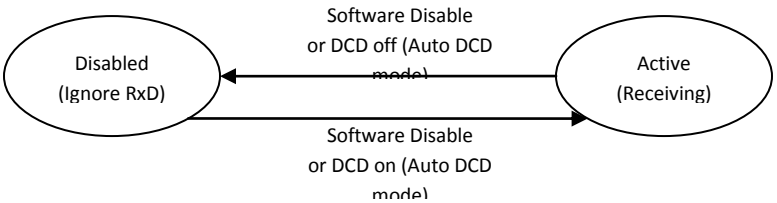


Illustration 8: Raw Receiver State Diagram

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.

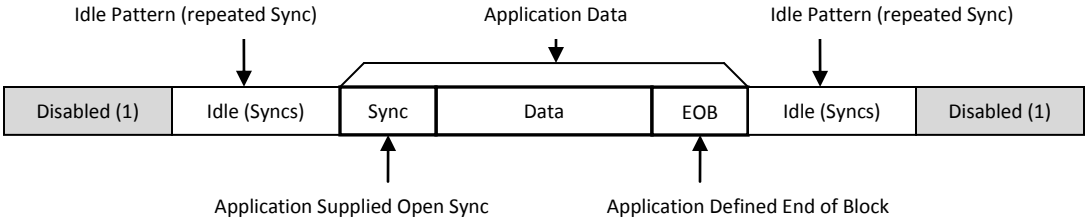


Illustration 9: Monosync/Bisync Block

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.

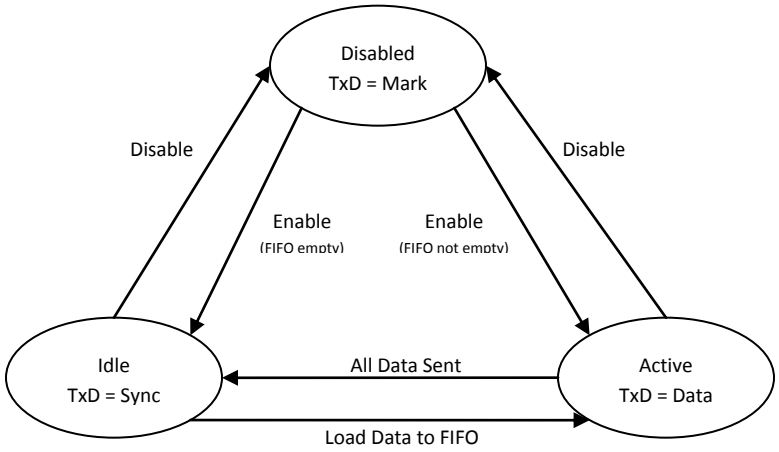


Illustration 10: Monosync/Bisync Transmitter State Diagram

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 (RCR[1]=1), the receiver becomes idle and starts hunting for the sync pattern (8-bit for monosync and 16-bit for bisync). When a sync pattern is detected, the receiver is active. An active receiver stores data bit for bit exactly as seen on the receive data signal. All data is byte aligned to the sync pattern. The receiver remains active until software disables the receiver (RCR[1]=0) or forces the receiver to idle/hunt (RCR[3]=1).

Hardware does not detect the end of a data block. The end of block indication varies widely for monosync and bisync implementations and is the responsibility of software to detect. Typically an application enables the receiver

and processes received data until the end of block condition is detected. The application then forces the receiver to idle/hunt by setting RCR[3] to 1.

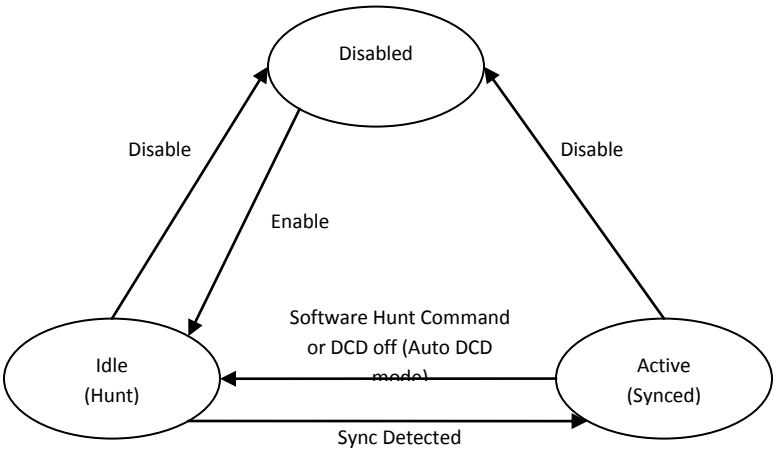


Illustration 11: Monosync/Bisync Receiver State Diagram

EXTENDED BYTE SYNCHRONOUS

Extended byte synchronous operation is similar to monosync and bisync operation. The difference is the receiver looks for a selectable one to four byte pattern to signal synchronization and the following data is byte aligned to the synchronization pattern. The transmit idle pattern is distinct from the sync pattern. The sync pattern and size is selected in the extended sync register (XSR) and extended control register (XCR).

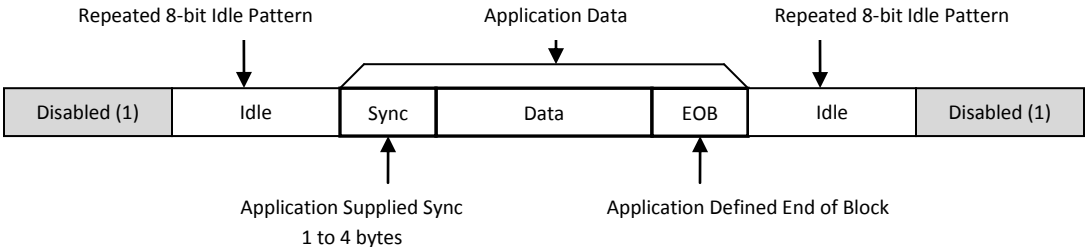


Illustration 12: Extended Sync Block

The extended sync 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 idle patterns. 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.

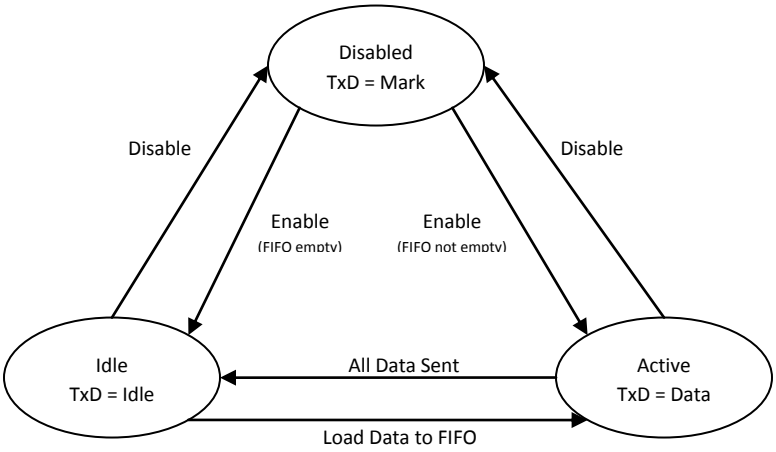


Illustration 13: Extended Sync Transmitter State Diagram

The extended sync receiver has three states: disabled, idle (hunt), and active (synced). The receiver starts in the disabled state. When software enables the receiver (RCR[1]=1), the receiver becomes idle and starts hunting for the sync pattern (8-bit for monosync and 16-bit for bisync). When a sync pattern is detected, the receiver is active. An active receiver stores data bit for bit exactly as seen on the receive data signal. All data is byte aligned to the sync pattern. The receiver remains active until software disables the receiver (RCR[1]=0) or forces the receiver to idle/hunt (RCR[3]=1).

Hardware does not detect the end of a data block. The end of block indication is the responsibility of software to detect. Typically an application enables the receiver and processes received data until the end of block condition is detected. The application then forces the receiver to idle/hunt by setting RCR[3] to 1.

A terminal count feature allows the hardware to automatically force hunt mode after a specified number of bytes are received. This is useful for applications that use fixed length packets. The terminal count is a 16 bit value and is specified in the extended control register (XCR).

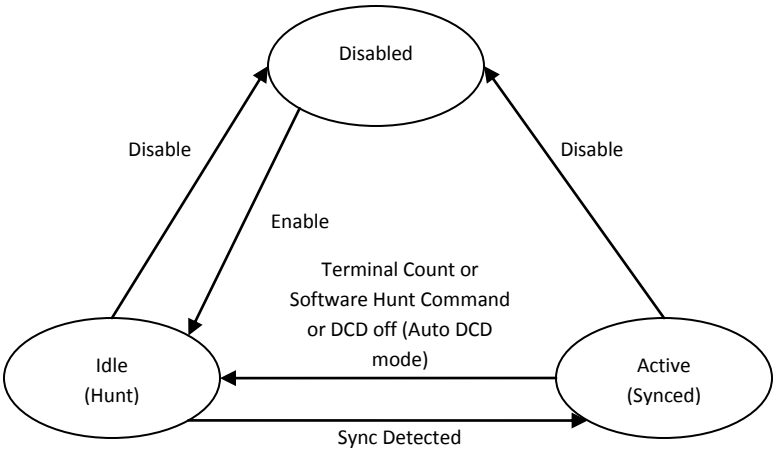


Illustration 14: Extended Sync Receiver

SERIAL ENCODING

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

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

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

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

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.

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

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.

DPLL CLOCK RECOVERY

Synchronous modes usually get transmit and receive timing from the transmit and receive clock inputs. Alternatively, timing can be recovered from a received data signal using a digital phased locked loop (DPLL). This requires the exact data rate to be known in advance and specified in `clock_speed` 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
DPLL Reference Clock = Data Clock * 16 = 153,600Hz
Base Clock = 14.7456MHz
Divisor = (14,745,600/153,600) – 1 = 95
Since 95 is an integer in the range 0 to 65535, the 153,600Hz reference clock can be generated exactly for recovering the 9600bps data clock.

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

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

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

SERIAL ENCODING WITH DPLL

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

PREAMBLE WITH DPLL

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

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

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:

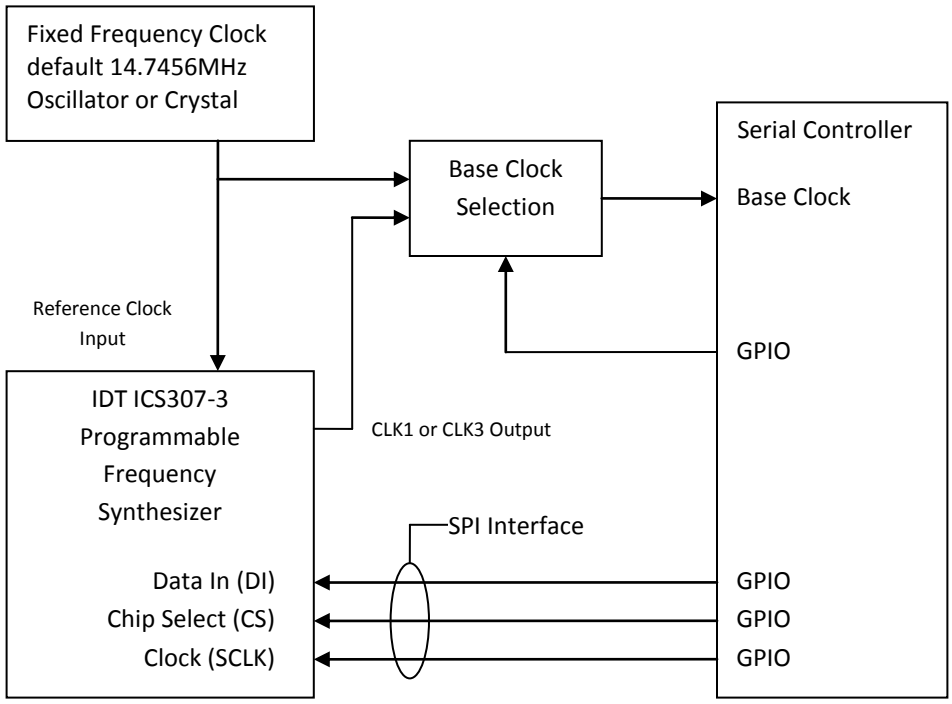
```
#ifconfig hdlc0 192.168.2.1 pointopoint 192.168.2.2 up
```

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.

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 `sample-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.

```
params.mode = MGSL_MODE_BASE_CLOCK;
params.clock_speed = 32768000;
rc = ioctl(fd, MGSL_IOCSPARAMS, &params);
```

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