

## What is *StreamDevice*?

*StreamDevice* is a generic [EPICS](http://www.aps.anl.gov/epics) [http://www.aps.anl.gov/epics] device support for devices with a "byte stream" based communication interface. That means devices that can be controlled by sending and receiving strings (in the broadest sense, including non-printable characters and even null-bytes). Examples for this type of communication interface are serial line (RS-232, RS-485, ...), IEEE-488 (also known as GPIB or HP-IB), and telnet-like TCP/IP.

*StreamDevice* is not limited to a specific device type or manufacturer nor is it necessary to re-compile anything to support a new device type. Instead, it can be configured for any device type with *protocol files* in plain ASCII text which describes the commands a device understands and the replies it sends. If the device can be controlled with strings like "RF:FREQ 499.655 MHZ" *StreamDevice* can be used. Formatting and parsing of values is done with *format converters* similar to those known from the C functions *printf()* and *scanf()*. To support other formats, it is possible to [write your own converters](#).

Each record with *StreamDevice* support runs one protocol from the protocol file to read or write its value. Protocols can be as simple as just one output string or can consist of many strings sent to and read from the device. However, a protocol is linear. That means it runs from start to end each time the record is [processed](#). It does not provide loops or branches.

*StreamDevice* comes with an interface to [asynDriver](http://www.aps.anl.gov/epics/modules/soft/asyn/) [http://www.aps.anl.gov/epics/modules/soft/asyn/] but can be extended to [support other bus drivers](#). Note that *StreamDevice* is not an alternative or replacement but a supplement for *asynDriver*. *StreamDevice* converts record values to and from strings but leaves it to *asynDriver* (or other bus interfaces) to exchange these strings with the device. Thus any bus type supported by *asynDriver* (to be exact by *asynOctet*) can automatically be used with *StreamDevice*.

*StreamDevice* supports all [standard records](#) of EPICS base which can have device support. It is also possible to [write support for new record types](#).

## What is *StreamDevice* not?

It is not a programming language for a high-level application. It is, for example, not possible to write a complete scanning program in a protocol. Use other tools for that and use *StreamDevice* only for the primitive commands.

It is not a block oriented device support. It is not possible to send or receive huge blocks of data that contain many process variables distributed over many records.

## Recommended Readings

[IOC Application Developer's Guide](http://www.aps.anl.gov/epics/base/R3-14/12-docs/AppDevGuide) [http://www.aps.anl.gov/epics/base/R3-14/12-docs/AppDevGuide]

[EPICS Record Reference Manual](https://wiki-ext.aps.anl.gov/epics/index.php/RRM_3-14) [https://wiki-ext.aps.anl.gov/epics/index.php/RRM\_3-14]

## Color and Style Conventions

In this document, code is written in `green fixed width font`. This marks text you typically type in configuration files etc.

`Longer code segments are often set in a box.`

---

[Next: Setup](#)

## 1. Prerequisites

*StreamDevice* requires either [EPICS base R3.14.6 or higher](http://www.aps.anl.gov/epics/base/R3-14/index.php) [http://www.aps.anl.gov/epics/base/R3-14/index.php] or [EPICS base R3.13.7 or higher](http://www.aps.anl.gov/epics/base/R3-13.php) [http://www.aps.anl.gov/epics/base/R3-13.php]. How to use *StreamDevice* on EPICS R3.13 is described on a [separate page](#).

### Fix required for base R3.14.8.2 and earlier on Windows

Up to release R3.14.8.2, a fix in EPICS base is required to build *StreamDevice* on Windows (not cygwin). Add the following line to `src/iocsh/iocsh.h` and rebuild base.

```
epicsShareFunc int epicsShareAPI iocshCmd(const char *command);
```

## Configuration

*StreamDevice* does not come with its own `<top>` location and `<top>/configure` directory. It expects to be put into an already existing `<top>` directory structure. You can simply create one with `makeBaseApp.pl` (which is part of EPICS base):

```
mkdir top
cd top
makeBaseApp.pl -t support
```

When asked for "Application names" just hit Enter. Then go to the newly created `configure` subdirectory and edit the `RELEASE` file you find there according to the instructions below.

After changing any configuration, you should run `make` in this directory.

For details on `<top>` directories and `RELEASE` files please refer to the [IOC Application Developer's Guide](http://www.aps.anl.gov/epics/base/R3-14/8-docs/AppDevGuide.pdf) [http://www.aps.anl.gov/epics/base/R3-14/8-docs/AppDevGuide.pdf] chapter 4: EPICS Build Facility.

### Support for *asynDriver*

You most probably want *asynDriver* support included, because that is the standard way for *StreamDevice* to talk to hardware. First get and install [asynDriver](http://www.aps.anl.gov/epics/modules/soft/asyn/) [http://www.aps.anl.gov/epics/modules/soft/asyn/] version 4-3 or higher before you build *StreamDevice*. I have tested *StreamDevice* with *asynDriver* versions up to 4-17. Make sure that the *asyn* library can be found by adding the path to the `<top>` directory of your *asyn* installation to the `RELEASE` file:

```
ASYN=/home/epics/asyn4-17
```

### Support for *sCalcout* record

The *sCalcout* [http://www.aps.anl.gov/bcda/synApps/calc/R2-8/sCalcoutRecord.html] record is part of *synApps* [http://www.aps.anl.gov/aod/bcda/synApps/index.php]. If *streamDevice* should be built with support for this record, you have to install the *calc* module from *SynApps* first. Add references to the `RELEASE` file as shown here:

```
CALC=/home/epics/synApps/calc-2-8
```

Up to *calc* release R2-6 (*synApps* release R5\_1), the *sCalcout* record needs a fix. (See separate [scalcout page](#).) And the *calc* module had dependencies on other *SynApps* modules. Release R2-8 or newer is recommended. Support for the *scalcout* is optional. *StreamDevice* works as well without *scalcout* or *SynApps*.

### Support for regular expression matching

If you want to enable regular expression matching, you need the *PCRE* package. For most Linux systems, it is already installed. In that case add the locations you have to make the locations of the *PCRE* header file and library known. However, the pre-installed package can only be used for the host architecture. Thus, add them not to `RELEASE` but to `RELEASE.Common.linux-x86` (if `linux-x86` is your `EPICS_HOST_ARCH`). Note that different Linux distributions may locate the files in different directories.

```
PCRE_INCLUDE=/usr/include/pcre
```

```
PCRE_LIB=/usr/lib
```

A pre-compiled Windows version of *PCRE* is available at [sourceforge](http://sourceforge.net/projects/gnuwin32/files/pcre/7.0/pcre-7.0.exe/download) [<http://sourceforge.net/projects/gnuwin32/files/pcre/7.0/pcre-7.0.exe/download>]

If you want to have *PCRE* support on platforms that don't support it natively, e.g. vxWorks, it is probably the easiest to build *PCRE* as an EPICS application.

### Building the *PCRE* package as an EPICS module

1. Download the *PCRE* package from [www.pcre.org](http://www.pcre.org) [<http://www.pcre.org>].
2. Extract the *PCRE* package in the <top> directory of *StreamDevice* or create a separate <top> location using `makeBaseApp.pl`.
3. Download this `Makefile` [<http://epics.web.psi.ch/software/streamdevice/pcre/Makefile>] and this `fixforvxworks.pl` [<http://epics.web.psi.ch/software/streamdevice/pcre/fixforvxworks.pl>] script and save them to the extracted pcre directory.
4. Change into the pcre directory and run `perl fixforvxworks.pl`
5. Run `make` (or `gmake`)

Define the location of the pcre <top> in the RELEASE file for *StreamDevice*.

```
PCRE=/home/epics/pcre
```

Regular expressions are optional. If you don't want them, you don't need this.

## 2. Build the *StreamDevice* Library

Unpack the [StreamDevice package](#) in the <top> directory of your application build area. (You might probably have done this already.) Go to the newly created *StreamDevice* directory and run `make` (or `gmake`). This will create and install the *stream* library and the `stream.dbd` file.

## 3. Build an Application

To use *StreamDevice*, your application must be built with the *asyn* and *stream* libraries and must load `asyn.dbd` and `stream.dbd`.

Include the following lines in your application Makefile:

```
PROD_LIBS += stream
PROD_LIBS += asyn
```

Include the following lines in your xxxAppInclude.dbd file to use *stream* and *asyn* with serial lines, IP sockets, and vxi11 ("GPIB over ethernet") support.

```
include "base.dbd"
include "stream.dbd"
include "asyn.dbd"
registrar(drvAsynIPPortRegisterCommands)
registrar(drvAsynSerialPortRegisterCommands)
registrar(vxi11RegisterCommands)
```

You can find an example application in the `streamApp` subdirectory.

## 4. The Startup Script

*StreamDevice* is based on [protocol files](#). To tell *StreamDevice* where to search for protocol files, set the environment variable `STREAM_PROTOCOL_PATH` to a list of directories to search. On Unix and vxWorks systems, directories are separated by `:`, on Windows systems by `;`. The default value is `STREAM_PROTOCOL_PATH=.`, i.e. the current directory.

Also configure the buses (in *asynDriver* terms: ports) you want to use with *StreamDevice*. You can give the buses any name you want, like `COM1` or `socket`, but I recommend to use names related to the connected device.

### Example:

A power supply with serial communication (9600 baud, 8N1) is connected to `/dev/ttyS1`. The name of the power supply is `PS1`. Protocol files are either in the current working directory or in the `../protocols` directory.

Then the startup script must contain lines like this:

```
epicsEnvSet ("STREAM_PROTOCOL_PATH", "../protocols")

drvAsynSerialPortConfigure ("PS1", "/dev/ttyS1")
asynSetOption ("PS1", 0, "baud", "9600")
asynSetOption ("PS1", 0, "bits", "8")
asynSetOption ("PS1", 0, "parity", "none")
asynSetOption ("PS1", 0, "stop", "1")
asynSetOption ("PS1", 0, "clocal", "Y")
asynSetOption ("PS1", 0, "crtsets", "N")
```

If the power supply was connected via telnet-style TCP/IP at address 192.168.164.10 on port 23, the startup script would contain:

```
epicsEnvSet ("STREAM_PROTOCOL_PATH", "../protocols")

drvAsynIPPortConfigure ("PS1", "192.168.164.10:23")
```

With a VXI11 (GPIB via TCP/IP) connection, e.g. a HP E2050A on IP address 192.168.164.10, it would look like this:

```
epicsEnvSet ("STREAM_PROTOCOL_PATH", "../protocols")

vxi11Configure ("PS1", "192.168.164.10", 1, 1000, "hpib")
```

## 5. The Protocol File

For each different type of hardware, create a protocol file which defines protocols for all needed functions of the device. The file name is arbitrary, but I recommend that it contains the device type. It must not contain spaces and should be short. During `iocInit`, `streamDevice` loads and parses the required protocol files. If the files contain errors, they are printed on the IOC shell. Put the protocol file in one of the directories listed in `STREAM_PROTOCOL_PATH`.

### Example:

`PS1` is an *ExamplePS* power supply. It communicates via ASCII strings which are terminated by <carriage return> <line feed> (ASCII codes 13, 10). The output current can be set by sending a string like `"CURRENT 5.13"`. When asked with the string `"CURRENT?"`, the device returns the last set value in a string like `"CURRENT 5.13 A"`.

Normally, an analog output record should write its value to the device. But during startup, the record should be initialized from the device. The protocol file `ExamplePS.proto` defines the protocol `setCurrent`.

```
Terminator = CR LF;

setCurrent {
    out "CURRENT %.2f";
    @init {
        out "CURRENT?";
        in "CURRENT %f A";
    }
}
```

### Reloading the Protocol File

During development, the protocol files might change frequently. To prevent restarting the IOC all the time, it is possible to reload the protocol file of one or all records with the shell function `streamReload("record")`. If `"record"` is not given, all records using `StreamDevice` reload their protocols. Furthermore, the `streamReloadSub` function can be used with a subroutine record to reload all protocols.

Reloading the protocol file aborts currently running protocols. This might set `SEVR=INVALID` and `STAT=UDF`. If a record can't reload its protocol file (e.g. because of a syntax error), it stays `INVALID/UDF` until a valid protocol is

loaded.

See the [next chapter](#) for protocol files in depth.

## 6. Configure the Records

To make a record use *StreamDevice*, set its *DTYP* field to "stream".

The *INP* or *OUT* link has the form "*@file protocol bus [address [parameters]]*".

Here, *file* is the name of the protocol file and *protocol* is the name of a protocol defined in this file. (See the [next chapter](#).)

If the protocol requires [arguments](#), specify them enclosed in parentheses: *protocol (arg1, arg2, ...)*.

The communication channel is specified with *bus* (aka *asynDriver* "port") and *addr*. If the bus does not have addresses, *addr* is dispensable. Optional *parameters* are passed to the bus driver.

### Example:

Create an output record to set the current of PS1. Use protocol *setCurrent* from file *ExamplePS.proto*. The bus is called *PS1* like the device.

```
record (ao, "PS1:I-set")
{
    field (DESC, "Set current of PS1")
    field (DTYP, "stream")
    field (OUT, "@ExamplePS.proto setCurrent PS1")
    field (EGU, "A")
    field (PREC, "2")
    field (DRVL, "0")
    field (DRVH, "60")
    field (LOPR, "0")
    field (HOPR, "60")
}
```

---

[Next: Protocol Files](#)

## 1. Prerequisites

*StreamDevice* version 2.2 and higher can run on EPICS 3.13. However, this requires some preparation, because EPICS 3.13 is missing some libraries and header files. Also *asynDriver* needs to be modified to compile with EPICS 3.13. Due to the limitations of EPICS 3.13, you can build *streamDevice* only for vxWorks systems.

Of course, you need an installation of [EPICS 3.13](http://www.aps.anl.gov/epics/base/R3-13.php) [http://www.aps.anl.gov/epics/base/R3-13.php]. I guess you already have that, otherwise you would want to [install \*StreamDevice\* on EPICS 3.14](#). I have tested *StreamDevice* with EPICS versions 3.13.7 up to 3.13.10 with vxWorks 5.3.1 and 5.5 on a ppc604 processor.

Download my [compatibility package](#), *asynDriver* [http://www.aps.anl.gov/epics/modules/soft/asyn/] version 4-3 or higher, and my [configure patches](#).

## 2. Build the Compatibility Package

Unpack `compat-1-0.tgz` in the `<top>` directory of your application build area. (Please refer to the [EPICS IOC Software Configuration Management](#) [http://www.aps.anl.gov/epics/EpicsDocumentation/AppDevManuals/iocScm-3.13.2/managingATop.html#3] document.)

Change to the `compat` directory and run `make`. This installs many EPICS 3.14-style header files and a small library (`compatLib`).

## 3. Build the *asynDriver* Library

Unpack the *asynDriver* package and change to its top directory.

Unpack `configure.tgz` here. This will modify files in the `configure` directory. Change to the `configure` directory and edit `CONFIG_APP`. Set `COMPAT=...` to the `<top>` directory where you have installed the compatibility package before. (This patch might also allow you to compile other 3.14-style drivers for 3.13. It has absolutely no effect if you use EPICS 3.14.)

Edit `RELEASE` and comment out `IPAC=...` (unless you have the *ipac* package and somehow made it compatible to EPICS 3.13). Set `EPICS_BASE` to your EPICS 3.13 installation.

Run `make` in the `configure` directory.

Change to `../asyn/devGpib` and edit `devGpib.h` and `devSupportGpib.c`. Change all occurrences of `static gDset` to `gDset`.

Go one directory up (to `asyn`) and run `make` twice! (The first run will just create `Makefile.Vx`.) Ignore all compiler warnings.

Do not try to build the test applications. It will not work.

## 4. Build the *StreamDevice* Library

Go to the `<top>` directory of your application build area.

Edit `config/RELEASE` and add the variable `ASYN`. Set it to the location of the *asynDriver* installation. Also set the `COMPAT` variable to the location of the compatibility package. Run `make` in the `config` directory.

Unpack the *StreamDevice* package in your `<top>` directory. Change to the newly created *StreamDevice* directory and run `make`.

## 5. Build an Application

To use *StreamDevice*, your application must be built with the *asyn*, *stream*, and *compat* libraries and must load `asyn.dbd` and `stream.dbd`. Also, as the *stream* library contains C++ code, the application must be munched. Therefore, include `$(TOP)/config/RULES.munch`. (Put your application in the same `<top>` as the *StreamDevice* installation.)

Include the following lines in your `Makefile.Vx`:

```
LDLIBS += $(COMPAT_BIN)/compatLib
LDLIBS += $(ASYN_BIN)/asynLib
```

```
LDLIBS += $(INSTALL_BIN)/streamLib

include $(TOP)/config/RULES.munch
```

Include the following lines in your `xxxAppInclude.dbd` file to use *stream* and *asyn* (you also need a `base.dbd`):

```
include "base.dbd"
include "stream.dbd"
include "asyn.dbd"
```

You can find an example application in the `streamApp` subdirectory.

## 6. The Startup Script

*StreamDevice* is based on *protocol files*. To tell *StreamDevice* where to search for protocol files, set the environment variable `STREAM_PROTOCOL_PATH` to a list of directories to search. Directories are separated by `..`. The default value is `STREAM_PROTOCOL_PATH=.`, i.e. the current directory.

Also configure the buses (in *asynDriver* terms: ports) you want to use with *StreamDevice*. You can give the buses any name you want, like `COM1` or `socket`, but I recommend to use names related to the connected device.

### Example:

A power supply with serial communication (9600 baud, 8N1) is connected to `/dev/ttyS1`. The name of the power supply is `PS1`. Protocol files are either in the current working directory or in the `../protocols` directory.

Then the startup script must contain lines like this:

```
ld < iocCore
ld < streamApp.munch
dbLoadDatabase ("streamApp.dbd")

putenv ("STREAM_PROTOCOL_PATH=../protocols")

drvAsynSerialPortConfigure ("PS1", "/dev/ttyS1")
asynSetOption ("PS1", 0, "baud", "9600")
asynSetOption ("PS1", 0, "bits", "8")
asynSetOption ("PS1", 0, "parity", "none")
asynSetOption ("PS1", 0, "stop", "1")
```

An alternative approach is to skip step 5 (do not build an application) and load all components explicitly in the startup script. The `STREAM_PROTOCOL_PATH` variable can also be a `vxWorks` shell variable.

```
ld < iocCore
ld < compatLib
ld < asynLib
ld < streamLib.munch
dbLoadDatabase ("asyn.dbd")
dbLoadDatabase ("stream.dbd")

STREAM_PROTOCOL_PATH=../protocols"

drvAsynSerialPortConfigure ("PS1", "/dev/ttyS1")
asynSetOption ("PS1", 0, "baud", "9600")
asynSetOption ("PS1", 0, "bits", "8")
asynSetOption ("PS1", 0, "parity", "none")
asynSetOption ("PS1", 0, "stop", "1")
```

## 7. Continue as with EPICS 3.14.

## 1. General Information

A protocol file describes the communication with one device type. It contains *protocols* for each function of the device type and *variables* which affect how the *commands* in a protocol work. It does not contain information about the individual device or the used communication bus.

Each device type should have its own protocol file. I suggest to choose a file name that contains the name of the device type. Don't use spaces in the file name and keep it short. The file will be referenced by its name in the [INP](#) or [OUT](#) link of the records which use it. The protocol file must be stored in one of the directories listed in the environment variable `STREAM_PROTOCOL_PATH` (see chapter [Setup](#)).

The protocol file is a plain text file. Everything not enclosed in quotes (single ' or double ") is not case sensitive. This includes the names of [commands](#), [protocols](#) and [variables](#). There may be any amount of whitespaces (space, tab, newline, ...) or comments between names, [quoted strings](#) and special characters, such as `={};`. A comment is everything starting from an unquoted `#` until the end of the line.

### Example Protocol File:

```
# This is an example protocol file

Terminator = CR LF;

# Frequency is a float
# use ai and ao records

getFrequency {
    out "FREQ?"; in "%f";
}

setFrequency {
    out "FREQ %f";
    @init { getFrequency; }
}

# Switch is an enum, either OFF or ON
# use bi and bo records

getSwitch {
    out "SW?"; in "SW %{OFF|ON}";
}

setSwitch {
    out "SW %{OFF|ON}";
    @init { getSwitch; }
}

# Connect a stringout record to this to get
# a generic command interface.
# After processing finishes, the record contains the reply.

debug {
    ExtraInput = Ignore;
    out "%s"; in "%39c"
}
```

## 2. Protocols

For each function of the device type, define one protocol. A protocol consists of a name followed by a body in braces



`{}`. The name must be unique within the protocol file. It is used to reference the protocol in the `INP` or `OUT` link of the record, thus keep it short. It should describe the function of the protocol. It must not contain spaces or any of the characters `, ; = { } ( ) $ ' " \ #`.

The protocol body contains a sequence of `commands` and optionally `variable assignments` separated by `;`.

## Referencing other protocols

To save some typing, a previously defined protocol can be called inside another protocol like a `command` without parameters. The protocol name is replaced by the commands in the referenced protocol. However, this does not include any `variable assignments` or `exception handlers` from the referenced protocol. See the `@init` handlers in the above example.

## Limitations

The `StreamDevice` protocol is not a programming language. It has neither loops nor conditionals (in this version of `StreamDevice`). However, if an error occurs, e.g. a timeout or a mismatch in input parsing, an `exception handler` can be called to clean up.

## 3. Commands

Seven different commands can be used in a protocol: `out`, `in`, `wait`, `event`, `exec`, `disconnect`, and `connect`. Most protocols will consist only of a single `out` command to write some value, or an `out` command followed by an `in` command to read a value. But there can be any number of commands in a protocol.

`out string;`

Write output to the device. The argument `string` may contain `format converters` which are replaced by the formatted value of the record before sending.

`in string;`

Read and parse input from the device. The argument `string` may contain `format converters` which specify how to interpret data to be put into the record. Input must match the argument string. Any input from the device should be consumed with an `in` command. If a device, for example, acknowledges a setting, use an `in` command to check the acknowledge, even though it contains no user data.

`wait milliseconds;`

Just wait for some milliseconds. Depending on the resolution of the timer system, the actual delay can be slightly longer than specified.

`event(eventcode) milliseconds;`

Wait for event `eventcode` with some timeout. What an event actually means depends on the used `bus`. Some buses do not support events at all, some provide many different events. If the bus supports only one event, `(eventcode)` is dispensable.

`exec string;`

The argument `string` is passed to the IOC shell as a command to execute.

`disconnect;`

Disconnect from the hardware. This is probably not supported by all busses. Any `in` or `out` command will automatically reconnect. Only records reading in "I/O Intr" mode will not cause a reconnect.

`connect milliseconds;`

Explicitly connect to the hardware with `milliseconds` timeout. Since connection is handled automatically, this command is normally not needed. It may be useful after a `disconnect`.

## 4. Strings

In a `StreamDevice` protocol file, strings can be written as quoted literals (single quotes or double quotes), as a sequence of bytes values, or as a combination of both.

Examples for quoted literals are:

```
"That's a string."
```

```
'Say "Hello"'
```

There is no difference between double quoted and single quoted literals, it just makes it easier to use quotes of the other type in a string. To break long strings into multiple lines of the protocol file, close the quotes before the line break and reopen them in the next line. Don't use a line break inside quotes.

As arguments of `out` or `in` commands, string literals can contain `format converters`. A format converter starts with `%` and works similar to formats in the C functions `printf()` and `scanf()`.

*StreamDevice* uses the backslash character `\` to define some escape sequences in quoted string literals:

`\"`, `\'`, `\%`, and `\\` mean literal `"`, `'`, `%`, and `\`.

`\a` means *alarm bell* (ASCII code 7).

`\b` means *backspace* (ASCII code 8).

`\t` means *tab* (ASCII code 9).

`\n` means *new line* (ASCII code 10).

`\r` means *carriage return* (ASCII code 13).

`\e` means *escape* (ASCII code 27).

`\x` followed by up to two hexadecimal digits means a byte with that hex value.

`\0` followed by up to three octal digits means a byte with that octal value.

`\1` to `\9` followed by up to two more decimal digits means a byte with that decimal value.

`\?` in input matches any byte, in output it does not print anything.

`\_` in input matches any amount of white space (including none), in output it prints a single space.

`\$` followed by the name of a [protocol variable](#) is replaced by the contents of that variable.

For non-printable characters, it is often easier to write sequences of byte values instead of escaped quoted string literals. A byte is written as an unquoted decimal, hexadecimal, or octal number in the range of `-128` to `255`, `-0x80` to `0xff` (not case sensitive), or `-0200` to `0377`, respectively.

*StreamDevice* also recognizes the ASCII symbolic names (not case sensitive) for several byte codes:

`NUL` (= `0x00`) *null*

`SOH` (= `0x01`) *start of heading*

`STX` (= `0x02`) *start of text*

`ETX` (= `0x03`) *end of text*

`EOT` (= `0x04`) *end of transmission*

`ENQ` (= `0x05`) *enquiry*

`ACK` (= `0x06`) *acknowledge*

`BEL` (= `0x07`) *bell*

`BS` (= `0x08`) *backspace*

`HT` or `TAB` (= `0x09`) *horizontal tabulator*

`LF` or `NL` (= `0x0A` or `10`) *line feed / new line*

`VT` (= `0x0B` or `11`) *vertical tabulator*

`FF` or `NP` (= `0x0C` or `12`) *form feed / new page*

`CR` (= `0x0D` or `13`) *carriage return*

`SO` (= `0x0E` or `14`) *shift out*

`SI` (= `0x0F` or `15`) *shift in*

`DLE` (= `0x10` or `16`) *data link escape*

`DC1` (= `0x11` or `17`) *device control 1*

`DC2` (= `0x12` or `18`) *device control 2*

`DC3` (= `0x13` or `19`) *device control 3*

`DC4` (= `0x14` or `20`) *device control 4*

`NAK` (= `0x15` or `21`) *negative acknowledge*

`SYN` (= `0x16` or `22`) *synchronous idle*

`ETB` (= `0x17` or `23`) *end of transmission block*

`CAN` (= `0x18` or `24`) *cancel*

`EM` (= `0x19` or `25`) *end of medium*

`SUB` (= `0x1A` or `26`) *substitute*

`ESC` (= `0x1B` or `27`) *escape*

`FS` (= `0x1C` or `28`) *file separator*

`GS` (= `0x1D` or `29`) *group separator*

`RS` (= `0x1E` or `30`) *record separator*

`US` (= `0x1F` or `31`) *unit separator*

`DEL` (= `0x7F` or `127`) *delete*

`SKIP` or `?` matches any input byte

A single string can be built from several quoted literals and byte values by writing them separated by whitespaces or comma.

## Examples:

The following lines represent the same string:

```
"Hello world\r\n"
```

```
'Hello',0x20,"world",CR,LF
72 101 108 108 111 32 119 111 114 108 100 13 10
```

## 5. Protocol Variables

*StreamDevice* uses three types of variables in a protocol file. *System variables* influence the behavior of *in* and *out* commands. *Protocol arguments* work like function arguments and can be specified in the *INP* or *OUT* link of the record. *User variables* can be defined and used in the protocol as abbreviations for often used values.

System and user variables can be set in the global context of the protocol file or locally inside protocols. When set globally, a variable keeps its value until overwritten. When set locally, a variable is valid inside the protocol only. To set a variable use the syntax:

```
variable = value;
```

Set variables can be referenced outside of *quoted strings* by *\$variable* or *\${variable}* and inside quoted strings by *\\$variable* or *\\${variable}*. The reference will be replaced by the value of the variable at this point.

### System variables

This is a list of system variables, their default settings and what they influence.

```
LockTimeout = 5000;
```

Integer. Affects first *out* command in a protocol.

If other records currently use the device, how many milliseconds to wait for exclusive access to the device before giving up?

```
WriteTimeout = 100;
```

Integer. Affects *out* commands.

If we have access to the device but output cannot be written immediately, how many milliseconds to wait before giving up?

```
ReplyTimeout = 1000;
```

Integer. Affects *in* commands.

Different devices need different times to calculate a reply and start sending it. How many milliseconds to wait for the first byte of the input from the device? Since several other records may be waiting to access the device during this time, *LockTimeout* should be larger than *ReplyTimeout*.

```
ReadTimeout = 100;
```

Integer. Affects *in* commands.

The device may send input in pieces (e.g. bytes). When it stops sending, how many milliseconds to wait for more input bytes before giving up? If *InTerminator* = "", a read timeout is not an error but a valid input termination.

```
PollPeriod = $ReplyTimeout;
```

Integer. Affects first *in* command in *I/O Intr* mode (see chapter *Record Processing*).

In that mode, some buses require periodic polling to get asynchronous input if no other record executes an *in* command at the moment. How many milliseconds to wait after last poll or last received input before polling again? If not set the same value as for *ReplyTimeout* is used.

```
Terminator
```

String. Affects *out* and *in* commands.

Most devices send and expect terminators after each message, e.g. *CR LF*. The value of the *Terminator* variable is automatically appended to any output. It is also used to find the end of input. It is removed before the input is passed to the *in* command. If no *Terminator* or *InTerminator* is defined, the underlying driver may use its own terminator settings. For example, *asynDriver* defines its own terminator settings.

```
OutTerminator = $Terminator;
```

String. Affects *out* commands.

If a device has different terminators for input and output, use this for the output terminator.

```
InTerminator = $Terminator;
```

String. Affects *in* commands.

If a device has different terminators for input and output, use this for the input terminator. If no *Terminator* or *InTerminator* is defined, the underlying driver may use its own terminator settings. If *InTerminator* = "", a read timeout is not an error but a valid input termination.

```
MaxInput = 0;
```

Integer. Affects *in* commands.

Some devices don't send terminators but always send a fixed message size. How many bytes to read before

terminating input even without input terminator or read timeout? The value 0 means "infinite".

```
Separator = "";
```

String. Affects `out` and `in` commands.

When formatting or parsing array values in a format converter (see [formats](#) and [waveform record](#)), what string to write or to expect between values? If the first character of the `Separator` is a space, it matches any number of any whitespace characters in an `in` command. To match arbitrary amount of whitespace in input, use `"\ "`.

```
ExtraInput = Error;
```

Error or Ignore. Affects `in` commands.

Normally, when input parsing has completed, any bytes left in the input are treated as parse error. If extra input bytes should be ignored, set `ExtraInput = Ignore`;

## Protocol arguments

Sometimes, protocols differ only very little. In that case it can be convenient to write only one protocol and use *protocol arguments* for the difference. For example a motor controller for the 3 axes X, Y, Z requires three protocols to set a position.

```
moveX { out "X GOTO %d"; }
moveY { out "Y GOTO %d"; }
moveZ { out "Z GOTO %d"; }
```

It also needs three versions of any other protocol. That means basically writing everything three times. To make this easier, *protocol arguments* can be used:

```
move { out "\$1 GOTO %d"; }
```

Now, the protocol can be references in the `OUT` link of three different records as `move(X)`, `move(Y)` and `move(Z)`. Up to 9 parameters, referenced as `$1 ... $9` can be specified in parentheses, separated by comma. The variable `$0` is replaced by the name of the protocol.

## User variables

User defined variables are just a means to save some typing. Once set, a user variable can be referenced later in the protocol.

```
f = "FREQ";      # sets f to "FREQ" (including the quotes)
f1 = $f " %f";   # sets f1 to "FREQ %f"

getFrequency {
    out $f "?"; # same as: out "FREQ?";
    in $f1;     # same as: in "FREQ %f";
}

setFrequency {
    out $f1;    # same as: out "FREQ %f";
}
```

## 6. Exception Handlers

When an error happens, an exception handler may be called. Exception handlers are a kind of sub-protocols in a protocol. They consist of the same set of commands and are intended to reset the device or to finish the protocol cleanly in case of communication problems. Like variables, exception handlers can be defined globally or locally. Globally defined handlers are used for all following protocols unless overwritten by a local handler. There is a fixed set of exception handler names starting with `@`.

`@mismatch`

Called when input does not match in an `in` command.

It means that the device has sent something else than what the protocol expected. If the handler starts with an `in` command, then this command reparses the old input from the unsuccessful `in`. Error messages from the unsuccessful `in` are suppressed. Nevertheless, the record will end up in `INVALID/CALC` state (see chapter [Record Processing](#)).

`@writetimeout`

Called when a write timeout occurred in an `out` command.

It means that output cannot be written to the device. Note that `out` commands in the handler are also likely to fail in this case.

#### `@replytimeout`

Called when a reply timeout occurred in an `in` command.

It means that the device does not send any data. Note that `in` commands in the handler are also likely to fail in this case.

#### `@readtimeout`

Called when a read timeout occurred in an `in` command.

It means that the device stopped sending data unexpectedly after sending at least one byte.

#### `@init`

Not really an exception but formally specified in the same syntax. This handler is called from `iocInit` during record initialization. It can be used to initialize an output record with a value read from the device. Also see chapter [Record Processing](#).

### Example:

```
setPosition {
    out "POS %f";
    @init { out "POS?"; in "POS %f"; }
}
```

After executing the exception handler, the protocol terminates. If any exception occurs within an exception handler, no other handler is called but the protocol terminates immediately. An exception handler uses all [system variable](#) settings from the protocol in which the exception occurred.

---

[Next: Format Converters](#)

## 1. Format Syntax

*StreamDevice* format converters work very similar to the format converters of the C functions *printf()* and *scanf()*. But *StreamDevice* provides more different converters and you can also write your own converters. Formats are specified in [quoted strings](#) as arguments of `out` or `in` [commands](#).

A format converter consists of

- The `%` character
- Optionally a field or record name in `( )`
- Optionally flags out of the characters `*# +0-?=!`
- Optionally an integer *width* field
- Optionally a period character `(.)` followed by an integer *precision* field (input only for most formats)
- A conversion character
- Additional information required by some converters

The flags `*# +0-` work like in the C functions *printf()* and *scanf()*. The flags `?, =` and `!` are extensions.

The `*` flag skips data in input formats. Input is consumed and parsed, a mismatch is an error, but the read data is dropped. This is useful if input contains more than one value. Example: `in "%*f%f";` reads the second floating point number.

The `#` flag may alter the format, depending on the converter (see below).

The `' '` (space) and `+` flags print a space or a `+` sign before positive numbers, where negative numbers would have a `-`.

The `0` flag says that numbers should be left padded with `0` if *width* is larger than required.

The `-` flag specifies that output is left justified if *width* is larger than required.

The `?` flag makes failing input conversions succeed with a default zero value (`0`, `0.0`, or `""`, depending on the format type).

The `=` flag allows to compare input with current values. It is only allowed in input formats. Instead of reading a new value from input, the current value is formatted (like for output) and then compared to the input.

The `!` flag demands that input is exactly *width* bytes long (normally *width* defines the maximum number of bytes read in many formats). For example `in "%!5d";` expects exactly 5 digits. Fewer digits are considered loss of data and make the format fail. This feature has been added by Klemen Vodopivec, SNS.

### Examples:

```
in "%f";           float
out "%(HOPR)7.4f"; the HOPR field as 7 char float with precision 4
out "%#010x";      0-padded 10 char alternate hex (with leading 0x)
in "%[_a-zA-Z0-9]"; string of chars out of a charset
in "%*i";          skipped integer number
in "%?d";          decimal number or nothing (read as 0)
in "%=.3f";        compare input to the current value formatted as a float with precision 3
```

## 2. Data Types and Record Fields

### Default fields

Every conversion character corresponds to one of the data types `DOUBLE`, `LONG`, `ENUM`, or `STRING`. In opposite to to *printf()* and *scanf()*, it is not required to specify a variable for the conversion. The variable is typically the `VAL` or `RVAL` field of the record, selected automatically depending on the data type. Not all data types make sense for all record types. Refer to the description of [supported record types](#) for details.

*StreamDevice* makes no difference between `float` and `double` nor between `short`, `int` and `long` values. Thus, data type modifiers like `l` or `h` do not exist in *StreamDevice* formats.

## Redirection

To use other fields of the record or even fields of other records on the same IOC for the conversion, write the field name in parentheses directly after the %. For example `out "%(EGU) s"`; outputs the `EGU` field formatted as a string. Use `in "%(otherrecord.RVAL) f"`; to write the floating point input value into the `RVAL` field of `otherrecord`. If no field is given for an other record `.VAL` is assumed. When a record name conflicts with a field name use `.VAL` explicitly.

This feature is very useful when one line of input contains many values that should be distributed to many records. If `otherrecord` is passive and the field has the PP attribute (see [Record Reference Manual](http://www.aps.anl.gov/asd/controls/epics/EpicsDocumentation/AppDevManuals/RecordRef/Recordref-1.html) [<http://www.aps.anl.gov/asd/controls/epics/EpicsDocumentation/AppDevManuals/RecordRef/Recordref-1.html>]), the record will be processed. It is your responsibility that the data type of the record field is compatible to the the data type of the converter. STRING formats are compatible with arrays of CHAR or UCHAR.

Note that using this syntax is by far not as efficient as using the default field. At the moment it is not possible to set `otherrecord` to an alarm state when anything fails.

## Pseudo-converters

Some formats are not actually converters. They format data which is not stored in a record field, such as a [checksum](#). No data type corresponds to those *pseudo-converters* and the `%(FIELD)` syntax cannot be used.

## 3. Standard DOUBLE Converters (%f, %e, %E, %g, %G)

**Output:** `%f` prints fixed point, `%e` prints exponential notation and `%g` prints either fixed point or exponential depending on the magnitude of the value. `%E` and `%G` use `E` instead of `e` to separate the exponent.

With the `#` flag, output always contains a period character.

**Input:** All these formats are equivalent. Leading whitespaces are skipped.

With the `#` flag additional whitespace between sign and number is accepted.

When a maximum field width is given, leading whitespace only counts to the field width when the space flag is used.

## 4. Standard LONG Converters (%d, %i, %u, %o, %x, %X)

**Output:** `%d` and `%i` print signed decimal, `%u` unsigned decimal, `%o` unsigned octal, and `%x` or `%X` unsigned hexadecimal. `%X` uses upper case letters.

With the `#` flag, octal values are prefixed with `0` and hexadecimal values with `0x` or `0X`.

Unlike `printf`, `%x` and `%X` truncate the output to the the given width (number of least significant half bytes).

**Input:** `%d` matches signed decimal, `%u` matches unsigned decimal, `%o` unsigned octal. `%x` and `%X` both match upper or lower case unsigned hexadecimal. Octal and hexadecimal values can optionally be prefixed. `%i` matches any integer in decimal, or prefixed octal or hexadecimal notation. Leading whitespaces are skipped.

With the `-` negative octal and hexadecimal values are accepted.

With the `#` flag additional whitespace between sign and number is accepted.

When a maximum field width is given, leading whitespace only counts to the field width when the space flag is used.

## 5. Standard STRING Converters (%s, %c)

**Output:** `%s` prints a string. If *precision* is specified, this is the maximum string length. `%c` is a LONG format in output, printing one character!

**Input:** `%s` matches a sequence of non-whitespace characters and `%c` matches a sequence of not-null characters. The maximum string length is given by *width*. The default *width* is infinite for `%s` and 1 for `%c`. Leading whitespaces are skipped with `%s` except when the space flag is used but not with `%c`. The empty string matches.

With the `#` flag `%s` matches a sequence of not-null characters instead of non-whitespace characters.

With the `0` flag `%s` pads with 0 bytes instead of spaces.

## 6. Standard Charset STRING Converter (%[charset])

This is an input-only format. It matches a sequence of characters from *charset*. If *charset* starts with `^`, the format matches all characters not in *charset*. Leading whitespaces are not skipped.



Example: `%[_a-z]` matches a string consisting entirely of `_` (underscore) or letters from `a` to `z`.

## 7. ENUM Converter (`%{string0|string1|...}`)

This format maps an unsigned integer value on a set of strings. The value 0 corresponds to `string0` and so on. The strings are separated by `|`.

Example: `%{OFF|STANDBY|ON}` maps the string `OFF` to the value 0, `STANDBY` to 1 and `ON` to 2.

When using the `#` flag it is allowed to assign integer values to the strings using `=`. Unassigned strings increment their values by 1 as usual.

If one string is the initial substring of another, the substring must come later to ensure correct matching. In particular if one string is the empty string, it must be the last one. Use `#` and `=` to renumber if necessary.

Use the assignment `=?` for the last string to make it the default value for output formats.

Example: `%#{neg=-1|stop|pos|fast=10|rewind=-10}`.

If one of the strings contains `|` or `}` (or `=` if the `#` flag is used) a `\` must be used to escape the character.

**Output:** Depending on the value, one of the strings is printed, or the default if no value matches.

**Input:** If any of the strings matches, the value is set accordingly.

## 8. Binary LONG Converter (`%b, %Bzo`)

This format prints or scans an unsigned integer represented as a binary string (one character per bit). The `%b` format uses the characters `0` and `1`. With the `%B` format, you can choose two other characters to represent zero and one.

With the `#` flag, the bit order is changed to *little endian*, i.e. least significant bit first.

Examples: `%B.!` or `%B\x00\xff`. `%B01` is equivalent to `%b`.

In output, if *width* is larger than the number of significant bits, then the flag `0` means that the value should be padded with the chosen zero character instead of spaces. If *precision* is set, it means the number of significant bits.

Otherwise, the highest 1 bit defines the number of significant bits.

In input, leading spaces are skipped. A maximum of *width* characters is read. Conversion stops with the first character that is not the zero or the one character.

## 9. Raw LONG Converter (`%r`)

The raw converter does not really "convert". A signed or unsigned integer value is written or read in the internal (usually two's complement) representation of the computer. The normal byte order is *big endian*, i.e. most significant byte first. With the `#` flag, the byte order is changed to *little endian*, i.e. least significant byte first. With the `0` flag, the value is unsigned, otherwise signed.

In output, the *precision* (or `sizeof(long)` whatever is less) least significant bytes of the value are sign extended or zero extended (depending on the `0` flag) to *width* bytes. The default for *precision* is 1. Thus if you do not specify the *precision*, only the least significant byte is written! It is common error to write `out "%2r";` instead of `out "%.2r";`.

In input, *width* bytes are read and put into the value. If *width* is larger than the size of a `long`, only the least significant bytes are used. If *width* is smaller than the size of a `long`, the value is sign extended or zero extended, depending on the `0` flag.

Examples: `out "%.2r"; in "%02r";`

## 10. Raw DOUBLE Converter (`%R`)

The raw converter does not really "convert". A float or double value is written or read in the internal (maybe IEEE) representation of the computer. The normal byte order is *big endian*, i.e. most significant byte first. With the `#` flag, the byte order is changed to *little endian*, i.e. least significant byte first. The *width* must be 4 (float) or 8 (double). The default is 4.

## 11. Packed BCD (Binary Coded Decimal) LONG Converter (`%D`)

Packed BCD is a format where each byte contains two binary coded decimal digits (0 ... 9). Thus a BCD byte is in the range from `0x00` to `0x99`. The normal byte order is *big endian*, i.e. most significant byte first. With the `#` flag, the byte order is changed to *little endian*, i.e. least significant byte first. The `+` flag defines that the value is signed, using the upper half of the most significant byte for the sign. Otherwise the value is unsigned.

In output, *precision* decimal digits are printed in at least *width* output bytes. Signed negative values have `0xF` in their



most significant half byte followed by the absolute value.

In input, *width* bytes are read. If the value is signed, a one in the most significant bit is interpreted as a negative sign. Input stops with the first byte (after the sign) that does not represent a BCD value, i.e. where either the upper or the lower half byte is larger than 9.

## 12. Checksum Pseudo-Converter (`%<checksum>`)

This is not a normal "converter", because no user data is converted. Instead, a checksum is calculated from the input or output. The *width* field is the byte number from which to start calculating the checksum. Default is 0, i.e. the first byte of the input or output of the current command. The last byte is *precision* bytes before the checksum (default 0). For example in "`abcdefg%<xor>`" the checksum is calculated from `abcdefg`, but in "`abcdefg%2.1<xor>`" only from `cdef`.

Normally, multi-byte checksums are in *big endian* byteorder, i.e. most significant byte first. With the `#` flag, the byte order is changed to *little endian*, i.e. least significant byte first.

The `0` flag changes the checksum representation to hexadecimal ASCII (2 chars per checksum byte).

The `-` flag changes the checksum representation to "poor man's hex": `0x30 ... 0x3f` (2 chars per checksum byte).

The `+` flag changes the checksum representation to decimal ASCII (formatted with `%d`).

In output, the checksum is appended.

In input, the next byte or bytes must match the checksum.

### Implemented checksum functions

`%<sum>` or `%<sum8>`

One byte. The sum of all characters modulo  $2^8$ .

`%<sum16>`

Two bytes. The sum of all characters modulo  $2^{16}$ .

`%<sum32>`

Four bytes. The sum of all characters modulo  $2^{32}$ .

`%<negsum>`, `%<nsum>`, `%<-sum>`, `%<negsum8>`, `%<nsum8>`, or `%<-sum8>`

One byte. The negative of the sum of all characters modulo  $2^8$ .

`%<negsum16>`, `%<nsum16>`, or `%<-sum16>`

Two bytes. The negative of the sum of all characters modulo  $2^{16}$ .

`%<negsum32>`, `%<nsum32>`, or `%<-sum32>`

Four bytes. The negative of the sum of all characters modulo  $2^{32}$ .

`%<notsum>` or `%<~sum>`

One byte. The bitwise inverse of the sum of all characters modulo  $2^8$ .

`%<xor>`

One byte. All characters xor'ed.

`%<xor7>`

One byte. All characters xor'ed & `0x7F`.

`%<crc8>`

One byte. An often used 8 bit crc checksum (poly=`0x07`, init=`0x00`, xorout=`0x00`).

`%<ccitt8>`

One byte. The CCITT standard 8 bit crc checksum (poly=`0x31`, init=`0x00`, xorout=`0x00`, reflected).

`%<crc16>`

Two bytes. An often used 16 bit crc checksum (poly=`0x8005`, init=`0x0000`, xorout=`0x0000`).

`%<crc16r>`

Two bytes. An often used reflected 16 bit crc checksum (poly=`0x8005`, init=`0x0000`, xorout=`0x0000`, reflected).

`%<modbus>`

Two bytes. The modbus 16 bit crc checksum (poly=`0x8005`, init=`0xffff`, xorout=`0x0000`, reflected)

`%<ccitt16>`

Two bytes. The usual (but [wrong?](http://srecord.sourceforge.net/crc16-ccitt.html) [\[http://srecord.sourceforge.net/crc16-ccitt.html\]](http://srecord.sourceforge.net/crc16-ccitt.html)) implementation of the CCITT standard 16 bit crc checksum (poly=`0x1021`, init=`0xFFFF`, xorout=`0x0000`).

`%<ccitt16a>`

Two bytes. The unusual (but [correct?](http://srecord.sourceforge.net/crc16-ccitt.html) [\[http://srecord.sourceforge.net/crc16-ccitt.html\]](http://srecord.sourceforge.net/crc16-ccitt.html)) implementation of the

CCITT standard 16 bit crc checksum with augment. (poly=0x1021, init=0x1D0F, xorout=0x0000).

`%<ccitt16x>` or `%<crc16c>` or `%<xmodem>`

Two bytes. The XMODEM checksum. (poly=0x1021, init=0x0000, xorout=0x0000).

`%<crc32>`

Four bytes. The standard 32 bit crc checksum. (poly=0x04C11DB7, init=0xFFFFFFFF, xorout=0xFFFFFFFF).

`%<crc32r>`

Four bytes. The standard reflected 32 bit crc checksum. (poly=0x04C11DB7, init=0xFFFFFFFF, xorout=0xFFFFFFFF, reflected).

`%<jamcrc>`

Four bytes. Another reflected 32 bit crc checksum. (poly=0x04C11DB7, init=0xFFFFFFFF, xorout=0x00000000, reflected).

`%<adler32>`

Four bytes. The Adler32 checksum according to [RFC 1950](http://www.ietf.org/rfc/rfc1950.txt) [<http://www.ietf.org/rfc/rfc1950.txt>].

`%<hexsum8>`

One byte. The sum of all hex digits. (Other characters are ignored.)

### 13. Regular Expression STRING Converter (`%/regex/`)

This input-only format matches [Perl compatible regular expressions \(PCRE\)](http://www.pcre.org/) [<http://www.pcre.org/>]. It is only available if a PCRE library is installed.

If PCRE is not available for your host or cross architecture, download the sourcecode from [www.pcre.org](http://www.pcre.org) [<http://www.pcre.org/>] and try my EPICS compatible [Makefile](http://epics.web.psi.ch/software/streamdevice/pcre/Makefile) [<http://epics.web.psi.ch/software/streamdevice/pcre/Makefile>] to compile it like a normal EPICS support module. The Makefile is known to work with EPICS 3.14.8 and PCRE 7.2. In your RELEASE file define the variable `PCRE` so that it points to the install location of PCRE.

If PCRE is already installed on (some of) your systems, you may add architectures where PCRE can be found in standard include and library locations to the variable `WITH_SYSTEM_PCRC`. If either the header file or the library are in a non-standard place, set in your RELEASE file the variables `PCRE_INCLUDE_arch` and/or `PCRE_LIB_arch` for the respective architectures to the correct directories or set `PCRE_INCLUDE` and/or `PCRE_LIB` in architecture specific RELEASE.Common.arch files.

If the regular expression is not anchored, i.e. does not start with `^`, leading non-matching input is skipped. To match in multiline mode (across newlines) add `(?m)` at the beginning of the pattern. To match case insensitive, add `(?i)`.

A maximum of *width* bytes is matched, if specified. If *precision* is given, it specifies the sub-expression whose match is returned. Otherwise the complete match is returned. In any case, the complete match is consumed from the input buffer. If the expression contains a `/` it must be escaped like `\/`.

Example: `%.1/<title>(.*?)</title>/` returns the title of an HTML page, skips anything before the `<title>` tag and leaves anything after the `</title>` tag in the input buffer.

### 14. Regular Expression Substitution Pseudo-Converter

(`%#/regex/subst/`)

This is a variant of the previous converter (note the `#`) but instead of returning the matching string, it can be used as a pre-processor for input or as a post-processor for output.

Matches of the *regex* are replaced by the string *subst* with all `&` or `\0` in *subst* replaced with the match itself and all `\1` through `\9` replaced with the match of the corresponding sub-expression. To get a literal `&` or `\` or `/` in the substitution write `\&` or `\\` or `\/`. There is no way to specify literal bytes with values less or equal to 9 in the substitution!

If *width* is specified, it limits the number of characters processed. If the `-` flag is used (i.e. *width* looks like a negative number) only the last *width* characters are processed, else the first. Without *width* (or 0) all available characters are processed.

If *precision* is specified, it indicates which matches to replace. With the `+` flag given, *precision* is the maximum number of matches to replace. Otherwise *precision* is the index (counting from 1) of the match to replace. Without *precision* (or 0), all matches are replaced.

In input this converter pre-processes data received from the device before following converters read it. Converters preceding this one will read unmodified input. Thus place this converter before those whose input should be pre-processed.

In output it post-processes data already formatted by preceding converters before sending it to the device. Converters following this one will send their output unmodified. Thus place this converter after those whose output should be post-processed.

Examples:

`%#+-10.2/ab/X/` replaces the string `ab` with `X` maximal 2 times in the last 10 characters. (`abcabcabcabc` becomes `abcXcXcabc`)

`%#/\\\/` replaces all `\` with `/` (`\dir\file` becomes `/dir/file`)

`%#/. .\B/&:/` inserts `:` after every second character which is not at the end of a word. (`0b19353134` becomes `0b:19:35:31:34`)

`%#/://` removes all `:` characters. (`0b:19:35:31:34` becomes `0b19353134`)

`%#/([^+-]) * ([+-]) / \2\1/` moves a postfix sign to the front. (`1.23-` becomes `-1.23`)

## 15. MantissaExponent DOUBLE converter (%m)

This exotic and experimental format matches numbers in the format `[sign] mantissa sign exponent`, e.g. `+123-4` meaning `123e-4` or `0.0123`. Mantissa and exponent are decimal integers. The sign of the mantissa is optional. Compared to the standard `%e` format, this format does not contain the characters `.` and `e`.

Output formatting is ambiguous (e.g. `123-4` versus `1230-5`). I chose the following convention: Format `precision` defines number of digits in mantissa. No leading '0' in mantissa (except for 0.0 of course). Number of digits in exponent is at least 2. Format flags `+`, `-`, and space are supported in the usual way (always sign, left justified, space instead of `+` sign). Flags `#` and `0` are unsupported.

## 16. Timestamp DOUBLE converter (%T(*timeformat*))

This format reads or writes timestamps and converts them to a double number. The value represents the number of seconds since 1970 (the UNIX epoch). The precision of a double is large enough for microseconds (but not for nanoseconds). This format is probably used best in combination with a redirection to the `TIME` field. In this case, the value is converted to EPICS timestamps (seconds since 1990 and nanoseconds). The timestamp format understands the usual converters that the C function `strftime()` understands. In addition, fractions of a second can be specified and the time zone can be set in the format string.

Example: `%(TIME)T(%d %b %Y %H:%M:%.3S %z)` may print something like `3 Sep 2010 15:45:59 +0200`.

Fractions of a second can be specified as `%.nS` (seconds with `n` fractional digits), as `%0nF` or `%nF` (`n` fractional digits) or as `%N` (nanoseconds). In input, `n` is the maximum number of digits parsed, there may be actually less digits in the input. If `n` is not specified (`%.S` or `%F`) it uses a default value of 6.

In input, the time zone can be specified in the format like `+%hmm` or `-%hmm` for cases where the parsed time stamp does not specify the time zone, where `hmm` is a 4 digit number specifying the offset in hours and minutes.

In output, the system function `strftime()` is used to format the time. There may be differences in the implementation between operating systems.

In input, `StreamDevice` uses its own implementation because many systems are missing the `strptime()` function and additional formats are supported.

Day of the week can be parsed but is ignored because the information is redundant when used together with day, month and year and more or less useless otherwise. No check is done for consistency.

Because of the complexity of the problem, locales are not supported. Thus, only the English month names can be used (week day names are ignored anyway).

---

[Next: Record Processing](#)

## 1. Normal Processing

*StreamDevice* is an asynchronous device support (see [IOC Application Developer's Guide](http://www.aps.anl.gov/epics/base/R3-14/8-docs/AppDevGuide.pdf) [http://www.aps.anl.gov/epics/base/R3-14/8-docs/AppDevGuide.pdf] chapter 12: Device Support). Whenever the record is processed, the `protocol` is scheduled to start and the record is left active (`PACT=1`). The protocol itself runs in another thread. That means that any waiting in the protocol does not delay any other part of the IOC.

After the protocol has finished, the record is processed again, leaving `PACT=0` this time, triggering monitors and processing the forward link `FLNK`. Note that input links with PP flag pointing to a *StreamDevice* record will read the old value first and start the protocol afterward. This is a problem all asynchronous EPICS device supports have.

The first `out` command in the protocol locks the device for exclusive access. That means that no other record can communicate with that device. This ensures that replies given by the device reach the record which has sent the request. On a bus with many devices on different addresses, this normally locks only one device. The device is unlocked when the protocol terminates. Another record trying to lock the same device has to wait and might get a `LockTimeout`.

If any error happens, the protocol is aborted. The record will have its `SEVR` field set to `INVALID` and its `STAT` field to something describing the error:

### TIMEOUT

The device could not be locked (`LockTimeout`) or the device did not reply (`ReplyTimeout`).

### WRITE

Output could not be written to the device (`WriteTimeout`).

### READ

Input from the device started but stopped unexpectedly (`ReadTimeout`).

### COMM

The device driver reported some other communication error (e.g. unplugged cable).

### CALC

Input did not match the argument string of the `in` command or it contained values the record did not accept.

### UDF

Some fatal error happened or the record has not been initialized correctly (e.g. because the protocol is erroneous).

If the protocol is aborted, an `exception handler` might be executed if defined. Even if the exception handler can complete with no further error, the protocol will not resume and `SEVR` and `STAT` will be set according to the original error.

## 2. Initialization

Often, it is required to initialize records from the hardware after booting the IOC, especially output records. For this purpose, initialization is formally handled as an `exception`. The `@init` handler is called as part of the `initRecord()` function during `iocInit` before any scan task starts.

In contrast to `normal processing`, the protocol is handled synchronously. That means that `initRecord()` does not return before the `@init` handler has finished. Thus, the records initialize one after the other. The scan tasks are not started and `iocInit` does not return before all `@init` handlers have finished. If the handler fails, the record remains uninitialized: `UDF=1`, `SEVR=INVALID`, `STAT=UDF`.

The `@init` handler has nothing to do with the `PINI` field. The handler does not process the record nor does it trigger forward links or other PP links. It runs before `PINI` is handled. If the record has `PINI=YES`, the `PINI` processing is a `normal processing` after the `@init` handlers of all records have completed.

Depending on the record type, format converters might work slightly different from normal processing. Refer to the description of `supported record types` for details.

If the `@inithandler` has read a value and has completed without error, the record starts in a defined state. That means `UDF=0`, `SEVR=NO_ALARM`, `STAT=NO_ALARM` and the `VAL` field contains the value read from the device.

If no `@init` handler is installed, `VAL` and `RVAL` fields remain untouched. That means they contain the value defined in the record definition, read from a constant `INP` or `DOL` field, or restored from a bump-less reboot system (e.g.

*autosave* from the *synApps* package).

### 3. I/O Intr

*StreamDevice* supports I/O event scanning. This is a mode where record processing is triggered by the device whenever the device sends input.

In terms of protocol execution this means: When the `SCAN` field is set to `I/O Intr` (during `iocInit` or later), the protocol starts without processing the record. With the first `in` command, the protocol is suspended. If the device has been locked (i.e there was an `out` command earlier in the protocol), it is unlocked now. That means that other records can communicate to the device while this record is waiting for input. This `in` command ignores `replyTimeout`, it waits forever.

The protocol now receives any input from the device. It also gets a copy of all input directed to other records. Non-matching input does not generate a mismatch `exception`. It just restarts the `in` command until matching input is received.

After receiving matching input, the protocol continues normally. All other `in` commands are handled normally. When the protocol has completed, the record is processed. It then triggers monitors, forward links, etc. After the record has been processed, the protocol restarts.

This mode is useful in two cases: First for devices that send data automatically without being asked. Second to distribute multiple values in one message to different records. In this case, one record would send a request to the device and pick only one value out of the reply. The other values are read by records in `I/O Intr` mode.

#### Example:

Device `dev1` has a "region of interest" (ROI) defined by a start value and an end value. When asked "`ROI?`", it replies something like "`ROI 17.3 58.7`", i.e. a string containing both values.

We need two ai records to store the two values. Whenever record `ROI:start` is processed, it requests ROI from the device. Record `ROI:end` updates automatically.

```
record (ai "ROI:start") {
  field (DTYP, "stream")
  field (INP, "@myDev.proto getROIstart dev1")
}
record (ai "ROI:end") {
  field (DTYP, "stream")
  field (INP, "@myDev.proto getROIend dev1")
  field (SCAN, "I/O Intr")
}
```

Only one of the two protocols sends a request, but both read their part of the same reply message.

```
getROIstart {
  out "ROI?";
  in "ROI %f %*f";
}
getROIend {
  in "ROI %*f %f";
}
```

Note that the other value is also parsed by each protocol, but skipped because of the `%*` format. Even though the `getROIend` protocol may receive input from other requests, it silently ignores every message that does not start with "`ROI`", followed by two floating point numbers.

---

[Next: Supported Record Types](#)

## Supported Record Types

*StreamDevice* comes with support for all standard record types in EPICS base which can have device support.

There is a separate page for each supported record type:

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

Each page describes which record fields are used in input and output for different [format data types](#) during [normal record processing](#) and [initialization](#).

It is also possible to [write support for other recordtypes](#).

---

**Note:** aai record support is disabled per default. Enable it in `src/CONFIG_STREAM`.

## Normal Operation

With aai records, the format converter is applied to each element. Between the elements, a separator is printed or expected as specified by the `Separator variable` in the protocol. When parsing input, a space as the first character of the `Separator` matches any number of any whitespace characters.

During input, a maximum of `NELM` elements is read and `NORD` is updated accordingly. Parsing of elements stops when the separator does not match, conversion fails, or the end of the input is reached. A minimum of one element must be available.

During output, the first `NORD` elements are written.

The format data type must be convertible to or from the type specified in the `FTVL` field. The variable `x[i]` stands for one element of the written or read value.

DOUBLE format (e.g. `%f`):

Output: `x[i]=double (VAL[i])`

`FTVL` can be "DOUBLE", "FLOAT", "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

Input: `VAL[i]=FTVL(x[i])`

`FTVL` must be "FLOAT" or "DOUBLE"

LONG or ENUM format (e.g. `%i` or `%{}`):

Output: `x[i]=long (VAL[i])`

`FTVL` can be "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

Signed values are sign-extended to long, unsigned values are zero-extended to long before converting them.

Input: `VAL[i]=FTVL(x[i])`

`FTVL` can be "DOUBLE", "FLOAT", "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

The value is truncated to the least significant bytes if `FTVL` has a smaller data size than `long`.

STRING format (e.g. `%s`):

If `FTVL=="STRING"`:

Output: `x[i]=VAL[i]`

Input: `VAL[i]=x[i]`

Note that this is an array of strings, not an array of characters.

If `FTVL=="CHAR"` or `FTVL=="UCHAR"`:

In this case, the complete aai is treated as a large single string of size `NORD`. No separators are printed or expected.

Output: `x=range (VAL, 0, NORD)`

The first `NORD` characters are printed, which might be less than `NELM`.

Input: `VAL=x, NORD=length(x)`

A maximum of `NELM-1` characters can be read. `NORD` is updated to the index of the first of the trailing zeros. Usually, this is the same as the string length.

Other values of `FTVL` are not allowed for this format.

## Initialization

During `initialization`, the `@init` handler is executed, if present. All format converters work like in normal operation.

[aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [calcout](#) [scalcout](#)

**Note:** aao record support is disabled per default. Enable it in `src/CONFIG_STREAM`.

## Normal Operation

With aao records, the format converter is applied to each element. Between the elements, a separator is printed or expected as specified by the `Separator` variable in the protocol. When parsing input, a space as the first character of the `Separator` matches any number of any whitespace characters.

During output, the first `NORD` elements are written.

During input, a maximum of `NELM` elements is read and `NORD` is updated accordingly. Parsing of elements stops when the separator does not match, conversion fails, or the end of the input is reached. A minimum of one element must be available.

The format data type must be convertible to or from the type specified in the `FTVL` field. The variable `x[i]` stands for one element of the written or read value.

DOUBLE format (e.g. `%f`):

Output: `x[i]=double (VAL[i])`

`FTVL` can be "DOUBLE", "FLOAT", "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

Input: `VAL[i]=FTVL(x[i])`

`FTVL` must be "FLOAT" or "DOUBLE"

LONG or ENUM format (e.g. `%i` or `%{}`):

Output: `x[i]=long (VAL[i])`

`FTVL` can be "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

Signed values are sign-extended to long, unsigned values are zero-extended to long before converting them.

Input: `VAL[i]=FTVL(x[i])`

`FTVL` can be "DOUBLE", "FLOAT", "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

The value is truncated to the least significant bytes if `FTVL` has a smaller data size than `long`.

STRING format (e.g. `%s`):

If `FTVL=="STRING"`:

Output: `x[i]=VAL[i]`

Input: `VAL[i]=x[i]`

Note that this is an array of strings, not an array of characters.

If `FTVL=="CHAR"` or `FTVL=="UCHAR"`:

In this case, the complete aao is treated as a large single string of size `NORD`. No separators are printed or expected.

Output: `x=range (VAL, 0, NORD)`

The first `NORD` characters are printed, which might be less than `NELM`.

Input: `VAL=x, NORD=length(x)`

A maximum of `NELM-1` characters can be read. `NORD` is updated to the index of the first of the trailing zeros. Usually, this is the same as the string length.

Other values of `FTVL` are not allowed for this format.

## Initialization

During `initialization`, the `@init` handler is executed, if present. All format converters work like in normal operation.

[aai](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [calcout](#) [scalcout](#)



## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):

Output:  $x = (VAL - AOFF) / ASLO$

Input:  $VAL = (x * ASLO + AOFF) * (1.0 - SMOO) + VAL * SMOO$

In both cases, if `ASLO==0.0`, it is treated as `1.0`. Default values are `ASLO=1.0`, `AOFF=0.0`, `SMOO=0.0`.

If input is successful, `UDF` is cleared.

LONG format (e.g. `%i`):

Output:  $x = RVAL$

Input:  $RVAL = x$

Note that the record calculates  $VAL = ((RVAL + ROFF) * ASLO + AOFF) * ESLO + EOFF) * (1.0 - SMOO) + VAL * SMOO$  if `LINR=="LINEAR"`. `ESLO` and `EOFF` might be set in the record definition. *StreamDevice* does not set it. For example, `EOFF=-10` and `ESLO=0.000305180437934` ( $=20.0/0xFFFF$ ) maps `0x0000` to `-10.0`, `0x7FFF` to `0.0` and `0xFFFF` to `10.0`. Using unsigned formats with values  $\geq 0x800000$  gives different results on 64 bit machines.

If `LINR=="NO CONVERSION"` (the default), `VAL` is directly converted from and to `long` without going through `RVAL`. This allows for more bits on 64 bit machines. To get the old behavior, use `LINR=="LINEAR"`.

ENUM format (e.g. `%i`):

Not allowed.

STRING format (e.g. `%s`):

Not allowed.

## Initialization

During [initialization](#), the `@init` handler is executed, if present. In contrast to normal operation, in DOUBLE input `SMOO` is ignored (treated as `0.0`).

[aai](#) [aao](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable  $x$  stands for the written or read value.

DOUBLE format (e.g. %f):

Output:  $x = (\text{OVAL} - \text{AOFF}) / \text{ASLO}$

Input:  $\text{VAL} = x * \text{ASLO} + \text{AOFF}$

In both cases, if  $\text{ASLO} = 0.0$ , it is treated as  $1.0$ . Default values are  $\text{ASLO} = 1.0$ ,  $\text{AOFF} = 0.0$ .

Note that  $\text{OVAL}$  is not necessarily equal to  $\text{VAL}$  if  $\text{OROC} \neq 0.0$ .

LONG format (e.g. %i):

Output:  $x = \text{RVAL}$

Input:  $\text{RBV} = x$

Note that the record calculates  $\text{RVAL} = ((\text{OVAL} - \text{EOFF}) / \text{ESLO}) - \text{AOFF} / \text{ASLO}$  if  $\text{LINR} = \text{"LINEAR"}$ .  $\text{ESLO}$  and  $\text{EOFF}$  might be set in the record definition. *StreamDevice* does not set it. For example,  $\text{EOFF} = -10$  and  $\text{ESLO} = 0.000305180437934$  ( $= 20.0 / 0x\text{FFFF}$ ) maps  $-10.0$  to  $0x0000$ ,  $0.0$  to  $0x7\text{FFF}$  and  $10.0$  to  $0x\text{FFFF}$ . Using unsigned formats with values  $\geq 0x800000$  gives different results on 64 bit machines.

If  $\text{LINR} = \text{"NO CONVERSION"}$  (the default),  $\text{OVAL}$  is directly converted to `long` without going through  $\text{RVAL}$ . This allows for more bits on 64 bit machines. To get the old behavior, use  $\text{LINR} = \text{"LINEAR"}$ .

ENUM format (e.g. %i):

Not allowed.

STRING format (e.g. %s):

Not allowed.

## Initialization

During [initialization](#), the `@init` handler is executed, if present. In contrast to normal operation, output in DOUBLE format uses  $\text{VAL}$  instead of  $\text{OVAL}$ . Note that the record initializes  $\text{VAL}$  from  $\text{DOL}$  if that is a constant. LONG input is put to  $\text{RVAL}$  as well as to  $\text{RBV}$  and converted by the record.

[aai](#) [aao](#) [ai](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable  $x$  stands for the written or read value.

DOUBLE format (e.g. `%f`):  
Not allowed.

LONG format (e.g. `%i`):

Output:  $x=RVAL$

Input:  $RVAL=x \& MASK$

$MASK$  can be set in the record definition. Stream Device does not set it. If  $MASK==0$ , it is ignored (i.e.  $RVAL=x$ ). The record sets  $VAL=(RVAL!=0)$ , i.e. 1 if  $RVAL!=0$  and 0 if  $RVAL==0$ .

ENUM format (e.g. `%{}`):

Output:  $x=VAL$

Input:  $VAL=(x!=0)$

STRING format (e.g. `%s`):

Output: Depending on  $VAL$ ,  $ZNAM$  or  $ONAM$  is written, i.e.  $x=VAL?ONAM:ZNAM$ .

Input: If input is equal to  $ZNAM$  or  $ONAM$ ,  $VAL$  is set accordingly. Other input strings are not accepted.

## Initialization

During [initialization](#), the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):  
Not allowed.

LONG format (e.g. `%i`):

Output: `x=RVAL`

Input: `RBV=x&MASK`

`MASK` can be set in the record definition. Stream Device does not set it. If `MASK==0`, it is ignored (i.e. `RBV=x`).

ENUM format (e.g. `%{}`):

Output: `x=VAL`

Input: `VAL= (x!=0)`

STRING format (e.g. `%s`):

Output: Depending on `VAL`, `ZNAM` or `ONAM` is written, i.e. `x=VAL?ONAM:ZNAM`.

Input: If input is equal to `ZNAM` or `ONAM`, `VAL` is set accordingly. Other input strings are not accepted.

## Initialization

During [initialization](#), the `@init` handler is executed, if present. In contrast to normal operation, LONG input is put to `RVAL` as well as to `RBV` and converted by the record.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):

Not allowed.

LONG format (e.g. `%i`):

If any of `ZRVL ... FFVL` is set (is not 0):

Output: `x=RVAL&MASK`

Input: `RVAL=x&MASK`

Note that the record shifts `RVAL` right by `SHFT` bits, compares the result with all of `ZRVL ... FFVL`, and sets `VAL` to the index of the first match. `MASK` is initialized to `NOBT` 1-bits shifted left by `SHFT`. If `MASK==0` (because `NOBT` was not set) it is ignored, i.e. `x=RVAL` and `RVAL=x`.

If none of `ZRVL ... FFVL` is set (all are 0):

Output: `x=VAL`

Input: `VAL=x`

ENUM format (e.g. `%{}`):

Output: `x=VAL`

Input: `VAL=x`

STRING format (e.g. `%s`):

Output: Depending on `VAL`, one of `ZRST` or `FFST` is written. `VAL` must be in the range 0 ... 15.

Input: If input is equal one of `ZRST ... FFST`, `VAL` is set accordingly. Other input strings are not accepted.

## Initialization

During `initialization`, the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable *x* stands for the written or read value.

DOUBLE format (e.g. %f):

Not allowed.

LONG format (e.g. %i):

If any of ZRVL ... FFVL is set (is not 0):

Output:  $x=RVAL\&MASK$

Note that the record calculates RVAL by choosing one of ZRVL ... FFVL depending on VAL and by shifting it left by SHFT bits.

Input:  $RBV=x\&MASK$

MASK is initialized to NOBT 1-bits shifted left by SHFT. If MASK==0 (because NOBT was not set) it is ignored, i.e.  $x=RVAL$  and  $RBV=x$ .

If none of ZRVL ... FFVL is set (all are 0):

Output:  $x=VAL$

Input:  $VAL=x$

ENUM format (e.g. %{}):

Output:  $x=VAL$

Input:  $VAL=x$

STRING format (e.g. %s):

Output: Depending on VAL, one of ZRST ... FFST is written. VAL must be in the range 0 ... 15.

Input: If input is equal one of ZRST ... FFST, VAL is set accordingly. Other input strings are not accepted.

## Initialization

During *initialization*, the @init handler is executed, if present. In contrast to normal operation, LONG input is put to RVAL as well as to RBV and converted by the record.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbiDirect](#) [mbbo](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):  
Not allowed.

LONG format (e.g. `%i`):

If `MASK==0` (because `NOBT` is not set):

Output: `x=VAL`

Input: `VAL=x`

If `MASK!=0`:

Output: `x=RVAL&MASK`

Input: `RVAL=x&MASK`

`MASK` is initialized to `NOBT` 1-bits shifted left by `SHFT`.

ENUM format (e.g. `%{`):  
Not allowed.

STRING format (e.g. `%s`):  
Not allowed.

## Initialization

During [initialization](#), the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [calcout](#)

## Normal Operation

Depending on the format type, different record fields are used for output and input. The variable *x* stands for the written or read value.

DOUBLE format (e.g. %f):

Not allowed.

LONG format (e.g. %i):

If *MASK*==0 (because *NOBT* is not set):

Output: *x*=VAL

Input: VAL=*x*

If *MASK*!=0:

Output: *x*=RVAL&*MASK*

Input: RBV=*x*&*MASK*

*MASK* is initialized to *NOBT* 1-bits shifted left by *SHFT*.

ENUM format (e.g. %{}):

Not allowed.

STRING format (e.g. %s):

Not allowed.

## Initialization

During *initialization*, the *@init* handler is executed, if present. In contrast to normal operation, input is put to *RVAL* as well as to *RBV* and converted by the record if *MASK*!=0.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)



## Normal Operation

The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):

Not allowed.

LONG format (e.g. `%i`):

Not allowed.

ENUM format (e.g. `%{}`):

Not allowed.

STRING format (e.g. `%s`):

Output: `x=VAL`

Input: `VAL=x`

## Initialization

During [initialization](#), the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

Dirk Zimoch, 2005

## Normal Operation

The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):

Not allowed.

LONG format (e.g. `%i`):

Not allowed.

ENUM format (e.g. `%{}`):

Not allowed.

STRING format (e.g. `%s`):

Output: `x=VAL`

Input: `VAL=x`

## Initialization

During [initialization](#), the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [waveform](#) [calcout](#) [scalcout](#)

Dirk Zimoch, 2005

## Normal Operation

The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):

Not allowed.

LONG format (e.g. `%i`):

Output: `x=VAL`

Input: `VAL=x` Using unsigned formats with values  $\geq 0x800000$  gives different results on 64 bit machines.

ENUM format (e.g. `%{}`):

Output: `x=VAL`

Input: `VAL=x`

STRING format (e.g. `%s`):

Not allowed.

## Initialization

During [initialization](#), the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

Dirk Zimoch, 2005

## Normal Operation

The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):

Not allowed.

LONG format (e.g. `%i`):

Output: `x=VAL`

Input: `VAL=x` Using unsigned formats with values  $\geq 0x800000$  gives different results on 64 bit machines.

ENUM format (e.g. `%{}`):

Output: `x=VAL`

Input: `VAL=x`

STRING format (e.g. `%s`):

Not allowed.

## Initialization

During [initialization](#), the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#) [scalcout](#)

Dirk Zimoch, 2005

## Normal Operation

With waveform records, the format converter is applied to each element. Between the elements, a separator is printed or expected as specified by the *Separator variable* in the protocol. When parsing input, a space as the first character of the *Separator* matches any number of any whitespace characters.

During input, a maximum of *NELM* elements is read and *NORD* is updated accordingly. Parsing of elements stops when the separator does not match, conversion fails, or the end of the input is reached. A minimum of one element must be available.

During output, the first *NORD* elements are written.

The format data type must be convertible to or from the type specified in the *FTVL* field. The variable  $x[i]$  stands for one element of the written or read value.

DOUBLE format (e.g. %f):

Output:  $x[i]=\text{double}(\text{VAL}[i])$

*FTVL* can be "DOUBLE", "FLOAT", "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

Input:  $\text{VAL}[i]=\text{FTVL}(x[i])$

*FTVL* must be "FLOAT" or "DOUBLE"

LONG or ENUM format (e.g. %i or %{}):

Output:  $x[i]=\text{long}(\text{VAL}[i])$

*FTVL* can be "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

Signed values are sign-extended to long, unsigned values are zero-extended to long before converting them.

Input:  $\text{VAL}[i]=\text{FTVL}(x[i])$

*FTVL* can be "DOUBLE", "FLOAT", "LONG", "ULONG", "SHORT", "USHORT", "CHAR", "UCHAR", or "ENUM" (which is treated as "USHORT").

The value is truncated to the least significant bytes if *FTVL* has a smaller data size than *long*.

STRING format (e.g. %s):

If *FTVL*=="STRING":

Output:  $x[i]=\text{VAL}[i]$

Input:  $\text{VAL}[i]=x[i]$

Note that this is an array of strings, not an array of characters.

If *FTVL*=="CHAR" or *FTVL*=="UCHAR":

In this case, the complete waveform is treated as a large single string of size *NORD*. No separators are printed or expected.

Output:  $x=\text{range}(\text{VAL}, 0, \text{NORD})$

The first *NORD* characters are printed, which might be less than *NELM*.

Input:  $\text{VAL}=x, \text{NORD}=\text{length}(x)$

A maximum of *NELM-1* characters can be read. *NORD* is updated to the index of the first of the trailing zeros. Usually, this is the same as the string length.

Other values of *FTVL* are not allowed for this format.

## Initialization

During *initialization*, the *@init* handler is executed, if present. All format converters work like in normal operation.

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [calcout](#) [scalcout](#)

**Note:** Device support for calcout records is only available for EPICS base R3.14.5 or higher.

## Normal Operation

Different record fields are used for output and input. The variable `x` stands for the written or read value.

DOUBLE format (e.g. `%f`):

Output: `x=OVAL`

Input: `VAL=x`

Note that the record calculates `OVAL` from `CALC` or `OCAL` depending on `DOPT`.

LONG format (e.g. `%i`):

Output: `x=int(OVAL)`

Input: `VAL=x`

ENUM format (e.g. `%{}`):

Output: `x=int(OVAL)`

Input: `VAL=x`

STRING format (e.g. `%s`):

Not allowed.

For calcout records, it is probably more useful to access fields `A` to `L` directly (e.g. `"%(A) f"`). However, even if `OVAL` is not used, it is calculated by the record. Thus, `CALC` must always contain a valid expression (e.g. `"0"`).

## Initialization

During [initialization](#), the `@init` handler is executed, if present. All format converters work like in normal operation.

---

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [scalcout](#)

**Note:** The scalcout record is part of the *calc* module of the *synApps*

[<http://www.aps.anl.gov/aod/bcda/synApps/index.php>] package. Device support for scalcout records is only available for *calc* module release 2-4 or higher. You also need the *synApps* modules *genSub* and *sscan* to build *calc*.

Up to release 2-6 (*synApps* release 5.1), the scalcout record needs a fix. In *sCalcout.c* at the end of `init_record` add before the final `return(0)`:

```
if(pscalcoutDSET->init_record ) {
return (*pscalcoutDSET->init_record)(pcalc);
}
```

## Normal Operation

Different record fields are used for output and input. The variable *x* stands for the written or read value.

DOUBLE format (e.g. %f):

Output: *x*=OVAL

Input: VAL=*x*

Note that the record calculates OVAL from CALC or OCAL depending on DOPT.

LONG format (e.g. %i):

Output: *x*=int(OVAL)

Input: VAL=*x*

ENUM format (e.g. %i):

Output: *x*=int(OVAL)

Input: VAL=*x*

STRING format (e.g. %s):

Output: *x*=OSV

Input: SVAL=*x*

For scalcout records, it is probably more useful to access fields *A* to *L* and *AA* to *LL* directly (e.g. "%(A) f" or "%(BB) s"). However, even if OVAL is not used, it is calculated by the record. Thus, CALC must always contain a valid expression (e.g. "0").

## Initialization

During *initialization*, the `@init` handler is executed, if present. All format converters work like in normal operation.

[aai](#) [aao](#) [ai](#) [ao](#) [bi](#) [bo](#) [mbbi](#) [mbbo](#) [mbbiDirect](#) [mbboDirect](#) [longin](#) [longout](#) [stringin](#) [stringout](#) [waveform](#) [calcout](#)

## I have many almost identical protocols

You can give [arguments](#) to a protocol. In the `INP` or `OUT` link, write:

```
field (OUT, "@protocolfile protocol(arg1,arg2,arg3) bus")
```

In the protocol, reference arguments as `$1` `$2` `$3` or inside strings as `"\ $1 \ $2 \ $3"`.

```
moveaxis {out "move\ $1 \ %.6f";}
field (OUT, "@motor.proto moveaxis(X) motor1")
readpressure {out 0x02 0x00 $1; in 0x82 0x00 $1 "%2r";}
field (INP, "@vacuumgauge.proto readpressure(0x84) gauge3")
```

## I have a device that sends unsolicited data

Use `I/O Intr` [processing](#). The record receives any input and processes only when the input matches.

```
read {in "new value = %f";}
record (ai, "$(RECORD)") {
  field (DTYP, "stream")
  field (INP, "@$(DEVICETYPE).proto read $(BUS)")
  field (SCAN, "I/O Intr")
}
```

## I have a device that sends multi-line messages

```
Here is the value:
3.1415
```

Use as many `in` commands as you get input lines.

```
read_value {in "Here is the value: "; in "%f";}
```

## I need to write more than one value in one message

There is more than one solution to this problem. Different approaches have different requirements.

### A) All values have the same type and are separated by the same string

Use array records (e.g. [waveform](#), [aao](#)).

```
array_out {separator=", "; out "an array: (%.2f)";}
```

The format `%.2f` is repeated for each element of the array. All elements are separated by `", "`.

Output will look like this:

```
an array: (3.14, 17.30, -12.34)
```

### B) We have up to 12 numeric values

Use a [calcout](#) record and [redirection to fields](#).

```
write_ABC {out "A=%(A) .2f B=%(B) .6f C=%(C) .0f";}
```

You must specify a valid expression in the `CALC` field even if you don't use it.

```
record (calcout, "$(RECORD)") {
  field (INPA, "$(A_RECORD)")
  field (INPB, "$(B_RECORD)")
  field (INPC, "$(C_RECORD)")
  field (CALC, "0")
  field (DTYP, "stream")
  field (OUT, "@$(DEVICETYPE).proto write_ABC $(BUS)")
}
```



## C) Values are in other records on the same IOC

Use [redirection to records](#).

```
acquire {out 'ACQUIRE "%(\$1:directory)s/%s",%(\$1:time).3f;';}
```

You can specify a record name or record.FIELD in parentheses directly after the %. To avoid plain record names in protocol files use [protocol arguments](#) like \\$1. In the link, specify the record name or just the basename of the other records (device name) in parentheses.

```
record (stringout, "$(DEVICE):getimage") {
  field (DTYP, "stream")
  field (OUT, "@$(DEVICETYPE).proto acquire($(DEVICE)) $(BUS)")
}
```

## I need to read more than one value from one message

Again, there is more than one solution to this problem.

### A) All values have the same type and are separated by the same string

Use array records (e.g. [waveform](#), [aai](#)).

```
array_in {separator=","; in "array = (%f)";}
```

The format %f is repeated for each element of the array. A "," is expected between element. Input may look like this:

```
array = (3.14, 17.30, -12.34)
```

### B) The message and the values in it can be filtered easily

Use I/O Intr [processing](#) and [value skipping](#) (%\*)

```
read_A {out "GET A,B"; in "A=%f, B=%*f";}
read_B {in "A=%*f, B=%f";}
record (ai, "$(DEVICE):A") {
  field (DTYP, "stream")
  field (INP, "@$(DEVICETYPE).proto read_A $(BUS)")
  field (SCAN, "1 second")
}
record (ai, "$(DEVICE):B") {
  field (DTYP, "stream")
  field (INP, "@$(DEVICETYPE).proto read_B $(BUS)")
  field (SCAN, "I/O Intr")
}
```

Record A actively requests values every second. The reply contains values A and B. Record A filters only value A from the input and ignores value B by using the \* flag. Nevertheless, a complete syntax check is performed: B must be a valid floating point number. Record B is I/O Intr and gets (a copy of) any input, including input that was directed to record A. If it finds a matching string it ignores value A, reads value B and then processes. Any non-matching input is ignored by record B.

## C) Values should be stored in other records on the same IOC

Use [redirection to records](#). To avoid record names in protocol files, use [protocol arguments](#).

```
read_AB {out "GET A,B"; in "A=%f, B=%(\$1)f";}
record (ai, "$(DEVICE):A") {
  field (DTYP, "stream")
  field (INP, "@$(DEVICETYPE).proto read_AB($(DEVICE):B) $(BUS)")
  field (SCAN, "1 second")
}
record (ai, "$(DEVICE):B") {
}
```

Whenever record A reads input, it stores the first value in its own VAL field as usual and the second in the VAL field of record B. Because the VAL field of record B has the PP attribute, this automatically processes record B.

## I have a device that sends mixed data types: numbers or strings

Use a [@mismatch](#) exception handler and [redirection to records](#). To avoid record names in protocol files, use [protocol arguments](#).

### Example

When asked "CURRENT?", the device send something like "CURRENT 3.24 A" or a message like "device switched off".

```
read_current {out "CURRENT?"; in "CURRENT %f A"; @mismatch {in "%(\$1)39c";}}
record (ai, "$(DEVICE):readcurrent") {
  field (DTYP, "stream")
  field (INP, "@$(DEVICETYPE).proto read_current($(DEVICE):message) $(BUS)")
}
record (stringin, "$(DEVICE):message") {
}
```

After [processing](#) the readcurrent record, you can see from SEVR/STAT if the read was successful or not. With some more records, you can clean the message record if SEVR is not INVALID.

```
record (calcout, "$(DEVICE):clean_1") {
  field (INPA, "$(DEVICE):readcurrent.SEVR CP")
  field (CALC, "A#3")
  field (OOPT, "When Non-zero")
  field (OUT, "$(DEVICE):clean_2.PROC")
}
record (stringout, "$(DEVICE):clean_2") {
  field (VAL, "OK")
  field (OUT, "$(DEVICE):message PP")
}
```

## I need to read a web page

First you have to send a correctly formatted HTML request. Note that this request must contain the full URL like "http://server/page" and must be terminated with [two](#) newlines. The server should be the same as in the [drvAsynIPPortConfigure](#) command (if not using a http proxy). The web page you get often contains much more information than you need. [Regular expressions](#) are great to find what you are looking for.

### Example 1

Read the title of a web page.

```
get_title {
  extrainput = ignore;
  replyTimeout = 1000;
  out "GET http://\$1\n\n";
  in "%+.1/(?im)<title>(.*?)</title>/";
}
```

Terminate the request with two newlines, either explicit like here [or](#) using an [outTerminator](#). The URI (without http:// but including the web server host name) is passed as [argument 1](#) to [\\\$1](#). Note that web servers may be slow, so allow some [replyTimeout](#).

If you don't use an [inTerminator](#) then the whole page is read as one "line" to the [in](#) command and can be parsed easily with a regular expression. We want to see the string between [<title>](#) and [</title>](#), so we put it into a subexpression in [\(\)](#) and request the first subexpression with [.1](#). Note that the [/](#) in the closing tag has be be escaped to avoid a misinterpretation as the closing [/](#) of the regular expression.

The tags may be upper or lower case like [<TITLE>](#) or [<Title>](#), so we ask for case insensitive matching with [\(?i\)](#).

The string should be terminated with the first closing [</title>](#), not the last one in the file. (There should not be more than one title but you never know.) Thus we ask not to be greedy with [\(?m\)](#). [\(?i\)](#) and [\(?m\)](#) can be combined to [\(?im\)](#). See the PCRE documentation for more regexp syntax.

The regular expression matcher ignores and discards any content before the matching section. Content after the match is discarded with [extrainput = ignore](#) so that it does not trigger errors reporting "surplus input".

Finally, the title may be too long for the record. The [+](#) tells the format matcher not to fail in this case but to truncate the string instead. You can read the string with a stringin record or for longer strings with a waveform record with data

type CHAR.

```
record (stringin, "$(DEVICE):title") {
    field (DTYP, "stream")
    field (INP, "@$(DEVICETYPE).proto get_title($(PAGE)) $(BUS)")
}
record (waveform, "$(DEVICE):longtitle") {
    field (DTYP, "stream")
    field (INP, "@$(DEVICETYPE).proto get_title($(PAGE)) $(BUS)")
    field (FTVL, "CHAR")
    field (NELM, "100")
}
```

## Example 2

Read a number from a web page. First we have to locate the number. For that we match against any known string right before the number (and [discard the match](#) with `*`). Then we read the number.

```
get_title {
    extrainput = ignore;
    replyTimeout = 1000;
    out "GET http://\n\n";
    in "%*/Interesting value:/%f more text";
}
```

When using `extrainput = ignore;`, it is always a good idea to match a few bytes after the value, too. This catches errors where loading of the page is interrupted in the middle of the number. (You don't want to miss the exponent from something like 1.23E-14).

You can read more than one value from a file with successive regular expressions and [redirections](#). But this only works if the order of the values is predictable. *StreamDevice* is not an XML parser! It always reads sequentially.



# ***Record API***

---

**Sorry, this documentation is still missing.**

---

Dirk Zimoch, 2006

## Bus Interface Class

*StreamDevice* already comes with an interface to *asynDriver* [<http://www.aps.anl.gov/epics/modules/soft/asyn/>]. You should first try to implement your bus driver compatible to *asynDriver*. Then it can be used by *StreamDevice* automatically. Only if that does not work, write your own bus interface.

A bus interface is a C++ class that inherits from *StreamBusInterface*. Its purpose is to provide an interface to *StreamDevice* for a low-level I/O bus driver. *StreamDevice* acts as a client of the interface, calling interface methods and receiving replies via callbacks. Since the internal details of *StreamDevice* are not of interest to a bus interface, I will reference it simply as *client* in this chapter. The interface class must be registered via a call to `RegisterStreamBusInterface()` in the global context of the C++ file (not in a header file).

Interface methods called by the client must not block for arbitrary long times. That means the interface is allowed to take mutex semaphores to protect its internal data structures but it must not take event semaphores to wait for external I/O or similar.

It is assumed that the interface creates a separate thread to handle blocking I/O and to call the callback methods in the context of that thread when I/O has completed or timed out. The callback methods don't block but may in turn call interface methods. Much of the actual work will be done in the context of those callbacks, i.e. in the interface thread, thus be generous with stack.

### Example bus interface class declaration

```
#include <StreamBusInterface.h>

class MyInterface : StreamBusInterface
{
    // ... (internally used attributes and methods)

    MyInterface(Client* client);
    ~MyInterface();

    // StreamBusInterface virtual methods
    bool lockRequest(unsigned long lockTimeout_ms);
    bool unlock();
    bool writeRequest(const void* output, size_t size,
        unsigned long writeTimeout_ms);
    bool readRequest(unsigned long replyTimeout_ms,
        unsigned long readTimeout_ms,
        long expectedLength, bool async);
    bool supportsAsyncRead();
    bool supportsEvent();
    bool acceptEvent(unsigned long mask,
        unsigned long replytimeout_ms);
    bool connectRequest(unsigned long connecttimeout_ms);
    bool disconnect();
    void finish();

public:
    // creator method
    static StreamBusInterface* getBusInterface(
        Client* client, const char* busname,
        int addr, const char* param);
};

RegisterStreamBusInterface(MyInterface);

// ... (implementation)
```

## Methods to implement

The interface class must implement a public static creator method:

```
static StreamBusInterface* getBusInterface(Client* client, const char* busname, int addr,
    const char* param);
```

And it must implement the following pure virtual methods:

```
bool lockRequest(unsigned long lockTimeout_ms);
bool unlock();
```

It may implement additional virtual methods if the bus supports it:

```
bool writeRequest(const void* output, size_t size, unsigned long writeTimeout_ms);
bool readRequest(unsigned long replyTimeout_ms, unsigned long readTimeout_ms,
    long expectedLength, bool async);
bool supportsAsyncRead();
bool supportsEvent();
bool acceptEvent(unsigned long mask, unsigned long replytimeout_ms);
bool connectRequest(unsigned long connecttimeout_ms);
bool disconnect();
void finish();
```

It also may override the following virtual method:

```
void release();
```

## Callback methods provided

The base class *StreamBusInterface* implements a set of protected callback methods which must be called in response to the above request methods (most probably from another thread):

```
void lockCallback(StreamIoStatus status);
void writeCallback(StreamIoStatus status);
long readCallback(StreamIoStatus status, const void* input = NULL, long size = 0);
void eventCallback(StreamIoStatus status);
void connectCallback(StreamIoStatus status);
void disconnectCallback(StreamIoStatus status);
```

## Other provided methods, attributes, and types

```
StreamBusInterface(Client* client);
long priority();
const char* clientName();
const char* getOutTerminator(size_t& length);
const char* getInTerminator(size_t& length);
enum StreamIoStatus {StreamIoSuccess, StreamIoTimeout, StreamIoNoReply, StreamIoEnd,
    StreamIoFault};
```

## Theory of Operation

### Registration

```
RegisterStreamBusInterface(interfaceClass);
```

During initialization, the macro `RegisterStreamBusInterface()` registers the bus interface. It must be called exactly once for each bus interface class in global file context.

### Creation and deletion

```
static StreamBusInterface* getBusInterface(Client* client, const char* busname, int addr,
    const char* param);
StreamBusInterface(Client* client);
void release();
const char* clientName();
```

During startup, each client instance searches for its bus interface by name. It does so by calling the static

`getBusInterface()` method of every registered interface class. This method should check by `busname` if its interface class is responsible for that bus. If yes, it should check if the address `addr` is valid and associate a *device* with `busname/addr`. Some busses do not have addresses and allow only one device (e.g. RS232). Interfaces to such busses can ignore `addr`. The bus interface may then try to connect to the device, but it should allow it to be disconnected or switched off at that time. If the bus interface requires additional parameters, parse the `param` string. Your constructor should pass `client` to the base class constructor `StreamBusInterface(Client* client)`.

On success, `getBusInterface` should then return a pointer to a bus interface instance. Note that many client instances may want to connect to the same device. Each needs its own bus interface instance. The bus interface can get a string containing the name of the client instance from `clientName()`. This name is for use in error and log messages.

On failure, or if this interface class is not responsible for that bus, `getBusInterface` should return `NULL`. The client will then try other bus interface classes.

When the client does not need the interface any more, it calls `release()`. The default implementation of `release()` assumes that `getBusInterface()` has allocated a new bus interface and just calls `delete`. You should change `release()` if that assumption is not correct.

## Connecting and disconnecting

```
bool connectRequest(unsigned long connecttimeout_ms);
bool disconnect();
void connectCallback(IoStatus status);
void disconnectCallback(IoStatus status);
```

Connection should be handled automatically. If the device is disconnected, each attempt to access the device should try to (re-)connect. Normally, the interface should not try to disconnect unless the device does so.

However, sometimes the client wants to connect or disconnect explicitly. To connect, the client calls `connectRequest()`. This function should return `true` immediately or `false` if the request cannot be accepted or connection handling is not supported. The interface should call `connectCallback(StreamIoSuccess)` once the bus could be connected. If the bus cannot be connected within `connecttimeout_ms` milliseconds, the bus interface should call `connectCallback(StreamIoTimeout)`.

If a device cannot be connected, for example because there is something wrong with the I/O hardware, `connectCallback(StreamIoFault)` may be called.

To disconnect, the client calls `disconnectRequest()`; This function should return `true` immediately or `false` if the request cannot be accepted or connection handling is not supported. The interface should call `connectCallback(StreamIoSuccess)` once the bus is disconnected. There is no timeout for this operation. If disconnecting is impossible, the interface should call `connectCallback(StreamIoFault)`.

## Bus locking

```
bool lockRequest(unsigned long lockTimeout_ms);
void lockCallback(IoStatus status);
bool unlock();
long priority();
void finish();
```

Before doing output, the client calls `lockRequest()` to get exclusive access to the device. This function should return `true` immediately or `false` if the request cannot be accepted. If the device is already locked, the bus interface should add itself to a queue, sorted by `priority()`. As soon as the device is available, the bus interface should call `lockCallback(StreamIoSuccess)`. If the bus cannot be locked within `lockTimeout_ms` milliseconds, the bus interface should call `lockCallback(StreamIoTimeout)`.

If a device cannot be locked, for example because there is something wrong with the I/O hardware, `lockCallback(StreamIoFault)` may be called.

Normally, it is not necessary to lock the complete bus but only one device (i.e. one address). Other clients should still be able to talk to other devices on the same bus.

The client may perform several read and write operations when it has locked the device. When the protocol ends and the device is locked, the client calls `unlock()`. If other bus interfaces are in the lock queue, the next one should call `lockCallback(StreamIoSuccess)` now.

The client calls `finish()` when the protocol ends. This allows the bus interface to clean up. The bus interface should also cancel any outstanding requests of this client.

## Writing output

```
bool writeRequest(const void* output, size_t size, unsigned long writeTimeout_ms);
void writeCallback(IoStatus status);
const char* getOutTerminator(size_t& length);
```

To start output, the client calls `writeRequest()`. You can safely assume that the device has already been locked at this time. That means, no other client will call `writeRequest()` for this device and no other output is currently active for this device until it has been unlocked.

The function should arrange transmission of `size` bytes of `output` but return `true` immediately or `false` if the request cannot be accepted. It must not block until output has completed. After all output has been successfully transmitted, but not earlier, the interface should call `writeCallback(StreamIoSuccess)`.

If output blocks for `writeTimeout_ms` milliseconds, the interface should abort the transmission and call `writeCallback(StreamIoTimeout)`.

If output is impossible, for example because there is something wrong with the I/O hardware, `writeCallback(StreamIoFault)` may be called.

The interface must transmit exactly the `size` bytes from `output`. It must not change anything and it should not assume that any bytes have a special meaning. In particular, a null byte does not terminate `output`.

A call to `getOutTerminator` tells the interface which terminator has already been added to the output. If `NULL` was returned, the client is not aware of a terminator (no `outTerminator` was defined in the protocol). In this case, the interface may add a terminator which it knows from other sources. An interface is not required to support `NULL` results and may not add any terminator in this case.

The buffer referenced by `output` stays valid until `writeCallback()` is called.

The client may request more I/O or call `unlock()` after `writeCallback()` has been called.

## Reading input

```
bool readRequest(unsigned long replyTimeout_ms, unsigned long readTimeout_ms,
                long expectedLength, bool async);
long readCallback(IoStatus status, const void* input = NULL, long size = 0);
const char* getInTerminator(size_t& length);
bool supportsAsyncRead();
```

The client calls `readRequest()` to tell the bus interface that it expects input. Depending on the bus, this function might have to set the bus hardware into receive mode. If `expectedLength>0`, the the bus interface should stop input after this number of bytes have been received. In opposite to writing, the device may be in a non-locked status when `readRequest()` is called.

This function must not block until input is available. Instead, it should arrange for `readCallback(StreamIoSuccess, buffer, size)` to be called when input has been received and return `true` immediately or `false` if the request cannot be accepted.

Here, `buffer` is a pointer to `size` input bytes. The bus interface is responsible for the buffer. The client copies its contents. It does not modify or free it.

It is not necessary to wait until all data has been received. The bus interface can call `n=readCallback()` after any amount of input has been received. If the client needs more input, `readCallback()` returns a non-zero value. A positive `n` means, the client needs another `n` bytes of input. A negative `n` means, the client needs an unspecified amount of additional input.

With some bus interfaces, `readRequest()` might not have to do anything because the bus is always receiving. It might also be that the bus has no local buffer associated to store input before it is fetched with some `read()` call. In this case, a race condition between device and client can occur. To avoid loss of data, `readCallback(StreamIoSuccess, buffer, size)` may be called in this case even before `readRequest()`. If the client is expecting input in the next future, it will store it. Otherwise the input is dropped.

The `replyTimeout_ms` parameter defines how many milliseconds to wait for the first byte of a reply before the device is considered offline. If no input has been received after `replyTimeout_ms` milliseconds, the bus interface should call `readCallback(StreamIoNoReply)`.

The `readTimeout_ms` parameter is the maximum time to wait for further input. If input stops for longer than `readTimeout_ms` milliseconds the bus interface should call `readCallback(StreamIoTimeout,buffer,size)`. The client decides if this timeout is an error or a legal termination. Thus, pass all input received so far.

A call to `getInTerminator(length)` tells the interface which terminator is expected for input and `length` is set to



the number of bytes of the terminator. The result is a hint to the bus interface to recognize the end of an input. Once the terminator string is found, the bus interface should stop receiving input and call `readCallback(StreamIoSuccess, buffer, size)`. It is not necessary to remove the terminator string from the received input. An empty terminator string (`length==0`) means: Don't look for terminators.

If `NULL` was returned, the client is not aware of a terminator (no `InTerminator` was defined in the protocol). In this case, the interface may look for a terminator which it knows from other sources, reduce size by the terminator length and call `readCallback(StreamIoEnd, buffer, size)`. A bus interface is not required to support `NULL` results and may treat them as empty terminator (see above).

Some busses (e.g. GPIB) support special "end of message" signals. If such a signal is received, the bus interface should call `readCallback(StreamIoEnd, buffer, size)`. Use it to indicate a special "end of message" signal which is not visible in the normal byte data stream. If `getInTerminator()` has not returned `NULL` it is not necessary to remove a terminator which may come in addition to the "end of message" signal.

If input is impossible, for example because there is something wrong with the I/O hardware, `readCallback(StreamIoFault)` may be called.

If the `async` flag is `true`, the client wants to read input asynchronously without any timeout. That means, the bus interface should call `readCallback()` even if the input was requested by another client.

If a client wishes to receive asynchronous input, it first calls `supportsAsyncRead()`. The default implementation of this method always returns `false`. A bus interface may overwrite this method to return `true` and eventually prepare for asynchronous input. The client is then allowed to call `readRequest()` with the `async==true`. This means that the client is interested in any input. It should receive a `readCallback()` of all input which came in response to a synchronous (`async==false`) request from another client (which of course should receive the input, too). The interface should also receive asynchronous input when no synchronous client is active at the moment. Many asynchronous `readRequest()` calls from different clients may be active at the same time. All of them should receive the same input.

For asynchronous requests, `replyTimeout_ms` has a different meaning: If the bus interface has to poll the bus for input, it may take `replyTimeout_ms` as a hint for the poll period. If many asynchronous requests are active at the same time, it should poll with the shortest period of all clients. An asynchronous request does not time out. It stays active until the next input arrives. The client may reissue the asynchronous `readRequest()` from within the `readCallback()` if it wants to continue receiving asynchronous input.

If the client calls `finish()` at any time, the bus interface should cancel all outstanding requests, including asynchronous read requests.

## Handling events

```
bool supportsEvent();
bool acceptEvent(unsigned long mask, unsigned long replytimeout_ms);
void eventCallback(StreamIoStatus status);
```

An event is a sort of input from a device which is not part of the normal byte stream. One example is the SRQ line of GPIB. Not all bus types have events. To support events, the bus interface must overwrite `supportsEvent()` to return `true`. The default implementation always returns `false`.

If `true` is returned, the client is allowed to call `acceptEvent()`, where `mask` defines the (bus dependent) type of event or events to wait for. If `mask` is illegal, `acceptEvent()` should return `false`. The call to `acceptEvent()` must not block. It should arrange to call `eventCallback(StreamIoSuccess)` when the event matching `mask` arrives within `replytimeout_ms` milliseconds. If no such event arrives within this time, the bus interface should call `eventCallback(StreamIoTimeout)`.

To avoid race conditions, the bus interface should buffer events and also report a matching event which occurred before the actual call to `acceptEvent()` but after any previous call of any other request method like `writeRequest()`.

## Converter Class

A user defined converter class inherits public from *StreamFormatConverter* and handles one or more conversion characters. It is not necessary that a given conversion character supports both, printing and scanning. But if it does, both must be handled by the same converter class.

Any conversion corresponds to one data type. The converter class must implement print and/or scan methods for this data type. It must also implement a parse method to analyse the format string.

A converter class must be registered with a call to `RegisterConverter()` in the global file context.

The converter must not contain any class variables, because there will be only one global instance for each conversion character - not one for each format string!

### Example: LONG converter for %Q

```
#include "StreamFormatConverter.h"

class MyConverter : public StreamFormatConverter
{
    int parse(const StreamFormat&, StreamBuffer&, const char*&, bool);
    bool printLong(const StreamFormat&, StreamBuffer&, long);
    int scanLong(const StreamFormat&, const char*, long&);
};

RegisterConverter(MyConverter, "Q");

// ... (implementation)
```

## Theroy of Operation

### Registration

```
RegisterConverter(converterClass, "characters");
```

This macro registers the converter class for all given conversion characters. In most cases, you will give only one character. The macro must be called once for each class in the global file context.

HINT: Do not branch depending on the conversion character. Provide multiple classes, that's more efficient.

### Parsing

```
int parse (const StreamFormat& fmt, StreamBuffer& info, const char*& source,
          bool scanFormat);

struct StreamFormat { char conv; StreamFormatType type; unsigned char flags; short prec;
                    unsigned short width; unsigned short infolen; const char* info; };
```

During initialization, `parse()` is called whenever one of the conversion characters handled by your converter class is found in a protocol. The fields `fmt.conv`, `fmt.flags`, `fmt.prec`, and `fmt.width` have already been filled in. If a scan format is parsed, `scanFormat` is `true`. If a print format is parsed, `scanFormat` is `false`.

The `fmt.flags` field is a bitset and can have any of the following flags set:

- `left_flag`: the format contained a `-`. This is normally used to indicate that the value should be printed left-aligned.
- `sign_flag`: the format contained a `+`. This normally requests to print a sign even for positive numbers.
- `space_flag`: the format contained a `' '` (space). This normally requests to print a space instead of a sign for positive numbers.
- `alt_flag`: the format contained a `#`. This indicated the request to use an alternative format. For example in `%#x` the hex number is prefixed with `0x`.
- `zero_flag`: the format contained a `0`. This normally requests to pad a numerical value with leading zeros instead of leading spaces.

- `skip_flag`: the format contained a `*`. The value is parsed and checked but then discarded.

It is not necessary that these flags have exactly the same meaning in your formats, but a similar and intuitive meaning helpful for the user.

There are two additional flags, `default_flag` indicating a `?` and `compare_flag` indicating a `=` in the format, that are handled internally by *StreamDevice* and are not of interest to the converter class.

The `source` pointer points to the character of the format string just after the conversion character. You can parse additional characters if they belong to the format string handled by your class. Move the `source` pointer so that it points to the first character after your format string. This is done for example in the builtin formats `%[charset]` or `{enum0|enum1}`. However, many formats don't need additional characters.

## Example

source	source
before	after
parse()	parse()
"%39[0-9a-zA-Z]constant text"	
conversion	
character	

You can write any data you may need later in `print*()` or `scan*()` to the *Streambuffer info*. This will probably be necessary if you have parsed additional characters from the format string as in the above example

Return `long_format`, `double_format`, `string_format`, or `enum_format` depending on the datatype associated with the conversion character. It is not necessary to return the same value for print and for scan formats. You can even return different values depending on the format string, but I can't imagine why anyone should do that.

If the format is not a real data conversion but does other things with the data (append or check a checksum, encode or decode the data,...), return `pseudo_format`.

Return `false` if there is any parse error or if print or scan is requested but not supported by this conversion.

## Printing and Scanning

Provide a `print[Long|Double|String|Pseudo]()` and/or `scan[Long|Double|String|Pseudo]()` method appropriate for the data type you have returned in the `parse()` method. That method is called whenever the conversion appears in an output or input, respectively. You only need to implement the flavour of print and/or scan suitable for the datatype returned by `parse()`.

Now, `fmt.type` contains the value returned by `parse()`. With `fmt.info()` get access to the string you have written to `info` in `parse()` (null terminated).

The length of the info string can be found in `fmt.infolen`.

In `print*()`, append the converted value to `output`. Do not modify what is already in output (unless you really know what you're doing). Return `true` on success, `false` on failure.

In `scan*()`, read the value from input and return the number of consumed bytes. In the string version, don't write more bytes than `maxlen`! If the `skip_flag` is set, you don't need to write to `value`, since the value will be discarded anyway. Return `-1` on failure.



**Sorry, this documentation is still missing.**

---

Dirk Zimoch, 2006