EPICS StreamDevice Documentation

What is *StreamDevice*?

StreamDevice is a generic 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 TCP/IP. *StreamDevice* comes with an interface to *asynDriver* but can be extended to support other bus drivers.

If the device can be controlled with strings like "`RF:FREQ 499.655 MHZ`", *StreamDevice* can be used. How the strings exactly look like is defined in *protocols*. Formatting and interpretation 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* to read or write its value. All *protocols* are defined in *protocol files* in plain ASCII text. No compiling is necessary to change a protocol or to support new devices. *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 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 (PDF)

EPICS Record Reference Manual

Next: Setup

Dirk Zimoch, 2006

StreamDevice: Setup

1. Prerequisites

StreamDevice requires either EPICS base R3.14.6 or higher or EPICS base R3.13.7 or higher. How to use *StreamDevice* on EPICS R3.13 is described on a separate page. Because *StreamDevice* comes with an interface to *asynDriver* version R4-3 or higher as the underlying driver layer, you should have *asynDriver* installed first.

StreamDevice has support for the *scalcout* record from the *calc* module of *synApps*. Up to *calc* release R2-6 (*synApps* release R5_1), the *scalcout* record needs a fix. (See separate *scalcout* page.) Support for the scalcout is optional. *StreamDevice* works as well without scalcout or SynApps.

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

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

Make sure that the *asyn* library (and the *calc* module of *synApps*, if desired) can be found, e.g. by adding `ASYN` and (if installed) `CALC` or `SYNAPPS` to your `<top>/configure/RELEASE` file:

```
ASYN=/home/epics/asyn/4-5
CALC=/home/epics/synApps/calc/2-7
```

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 of the *PCRE* header and library to your `RELEASE` file:

```
PCRE_INCLUDE=/usr/include/pcre
PCRE_LIB=/usr/lib
```

If you want to build *StreamDevice* for platforms without *PCRE* support, it is the easiest to build *PCRE* as an EPICS application. Download the *PCRE* package from www.pcre.org and compile it with my EPICS compatible Makefile. Then define the location of the application in your RELEASE file.

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

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

For details on `<top>` directories and RELEASE files, please refer to the *IOC Application Developer's Guide* chapter 4: EPICS Build Facility.

2. Build the *StreamDevice* Library

Unpack the *StreamDevice* package in a `<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 and IP sockets:

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

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, "crtscts", "N")
```

If the power supply was connected via telnet-style TCP/IP at address 192.168.164.10 on port 23, the startupscript 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 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. If the protocol requires arguments, specify them enclosed in parentheses: *`protocol`*`(`*`arg1,arg2,...`*`)`.

The communication channel is specified with *`bus`* 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

Dirk Zimoch, 2007

StreamDevice: Protocol Files

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`*`;`
: Explicitely 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 the argument string of an `in` command matches any input byte
`\$` followed by the name of a protocol varible 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, -0200 to 0377). *StreamDevice* also defines some symbolic names for frequently used byte codes as aliases for the numeric byte value:
`EOT` means *end of transmission* (ASCII code 4).
`ACK` means *acknowledge* (ASCII code 6).
`BEL` means *bell* (ASCII code 7).
`BS` means *backspace* (ASCII code 8).
`HT` or `TAB` mean *horizontal tabulator* (ASCII code 9).
`LF` or `NL` mean *line feed* / *new line* (ASCII code 10).
`CR` means *carriage return* (ASCII code 13).
`ESC` means *escape* (ASCII code 27).
`DEL` means *delete* (ASCII code 127).
`SKIP` in the argument string of an `in` command 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.

Example:

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.

`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

Dirk Zimoch, 2006

StreamDevice: 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 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 ony for most formats)
- A conversion character
- Additional information required by some converters

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.

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

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.

Accessing record fields directly

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`*`.VAL)f";` to write the floating point input value into the `VAL` field of *`otherrecord`*. (You can't skip `.VAL` here.) This 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), 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. Note that using this syntax is by far not as efficient as using the default field.

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`)

In 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.

In input, all these formats are equivalent. Leading whitespaces are skipped.

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

In 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`.

In 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.

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

In 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!

In input, `%s` matches a sequence of non-whitespace characters and `%c` 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` but not with `%c`. The empty string matches.

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 `|`. If one of the strings contains `|` or `}`, a `\` must be used to escape the character.

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

In output, depending on the value, one of the strings is printed.

In input, if any of the strings matches the value is set accordingly.

8. Binary LONG Converter (`%b`, `%B`*zo*)

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 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 *width* least significant bytes of the value are written. If *width* is larger than the size of a `long`, the value is sign extended or zero extended, depending on the `0` flag.

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.

10. 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.

11. 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 *prec* 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 from binary to hexadecimal ASCII (2 bytes per checksum byte).

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).

`%<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).

`%<ccitt16>`
: Two bytes. The usual (but wrong?) implementation of the CCITT standard 16 bit crc checksum (poly=0x1021, init=0xFFFF, xorout=0x0000).

`%<ccitt16a>`
: Two bytes. The unusual (but correct?) implementation of the CCITT standard 16 bit crc checksum with augment. (poly=0x1021, init=0x1D0F, 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).

`%<jamcrc>`
: Four bytes. Another reflected 32 bit crc checksum. (poly=0x04C11DB7, init=0xFFFFFFFF, xorout=0x00000000).

`%<adler32>`
: Four bytes. The Adler32 checksum according to RFC 1950.

`%<hexsum8>`
: One byte. The sum of all hex digits. (Other characters are ignored.)

12. Regular Expresion STRING Converter (`%/`*regex*`/`)

This input-only format matches Perl compatible regular expressions (PCRE). 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 and try my EPICS compatible Makefile to compile it like a normal EPICS application. 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 your system, use the variables `PCRE_INCLUDE` and `PCRE_LIB` instead to provide the install directories of `pcre.h` and the library.

If you have PCRE installed in different locations for different (cross) architectures, define the variables in RELEASE.Common.<architecture> instead of the global RELEASE file.

If the regular expression is not anchored, i.e. does not start with `^`, leading non-matching input is skipped. A

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

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

Next: Record Processing

Dirk Zimoch, 2007

StreamDevice: Record Processing

1. Normal Processing

StreamDevice is an asynchronous device support (see IOC Application Developer's Guide 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 `@init`handler 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

Dirk Zimoch, 2005

StreamDevice: aai Records

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

Dirk Zimoch, 2006

StreamDevice: aao Records

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
Dirk Zimoch, 2006

StreamDevice: ai Records

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.

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, 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

Dirk Zimoch, 2005

StreamDevice: ao Records

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*=(OVAL-AOFF)/ASLO
- Input: VAL=*x**ASLO+AOFF
- In both cases, if ASLO==0.0, it is treated as 1.0. Default values are ASLO=1.0, AOFF=0.0.
- Note that OVAL is not necessarily equal to VAL if OROC!=0.0.

LONG format (e.g. `%i`):
- Output: *x*=RVAL
- Input: RBV=*x*
- Note that the record calculates RVAL=(((OVAL-EOFF)/ESLO)-AOFF)/ASLO 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 -10.0 to 0x0000, 0.0 to 0x7FFF and 10.0 to 0xFFFF.

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, output in DOUBLE format uses VAL instead of OVAL. Note that the record initializes VAL from DOL if that is a constant. LONG input is put to RVAL as well as to RBV and converted by the record.

---

aai aao ai bi bo mbbi mbbo mbbiDirect mbboDirect longin longout stringin stringout waveform calcout scalcout

Dirk Zimoch, 2005

StreamDevice: bi Records

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 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

Dirk Zimoch, 2005

StreamDevice: bo Records

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 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

Dirk Zimoch, 2005

StreamDevice: mbbi Records

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

Dirk Zimoch, 2005

StreamDevice: mbbo Records

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 mbboDirect longin longout stringin stringout waveform calcout scalcout

Dirk Zimoch, 2005

StreamDevice: mbbiDirect Records

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

Dirk Zimoch, 2005

StreamDevice: mbboDirect Records

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

Dirk Zimoch, 2005

StreamDevice: longin Records

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*

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

StreamDevice: longout Records

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*

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

StreamDevice: stringin Records

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

StreamDevice: stringout Records

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

StreamDevice: waveform Records

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]*=`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 waveform 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 aao ai ao bi bo mbbi mbbo mbbiDirect mbboDirect longin longout stringin stringout calcout scalcout

Dirk Zimoch, 2005

StreamDevice: calcout Records

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

Dirk Zimoch, 2005

StreamDevice: scalcout Records

Note: The scalcout record is part of the *calc* module of the *synApps* 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. `%{`):
- 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

Dirk Zimoch, 2005