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