PAUL SCHERRER INSTITUT

PSI

Dirk Zimoch :: Paul Scherrer Institut

StreamDevice Introduction

EPICS Collaboration Meeting October 2020 Fritz-Haber-Institut

What is StreamDevice?

How can a Device Driver be Generic?

Separate similarities from differences

What do many devices have in common?

Command strings

- Human readable Text
- Fixed size (binary) “telegrams”

Limited set of buses

- Serial (RS232, RS485, USB)
- GPIB
- Ethernet (TCP, UDP)

→ Programming

What is different between those devices?

Command sets

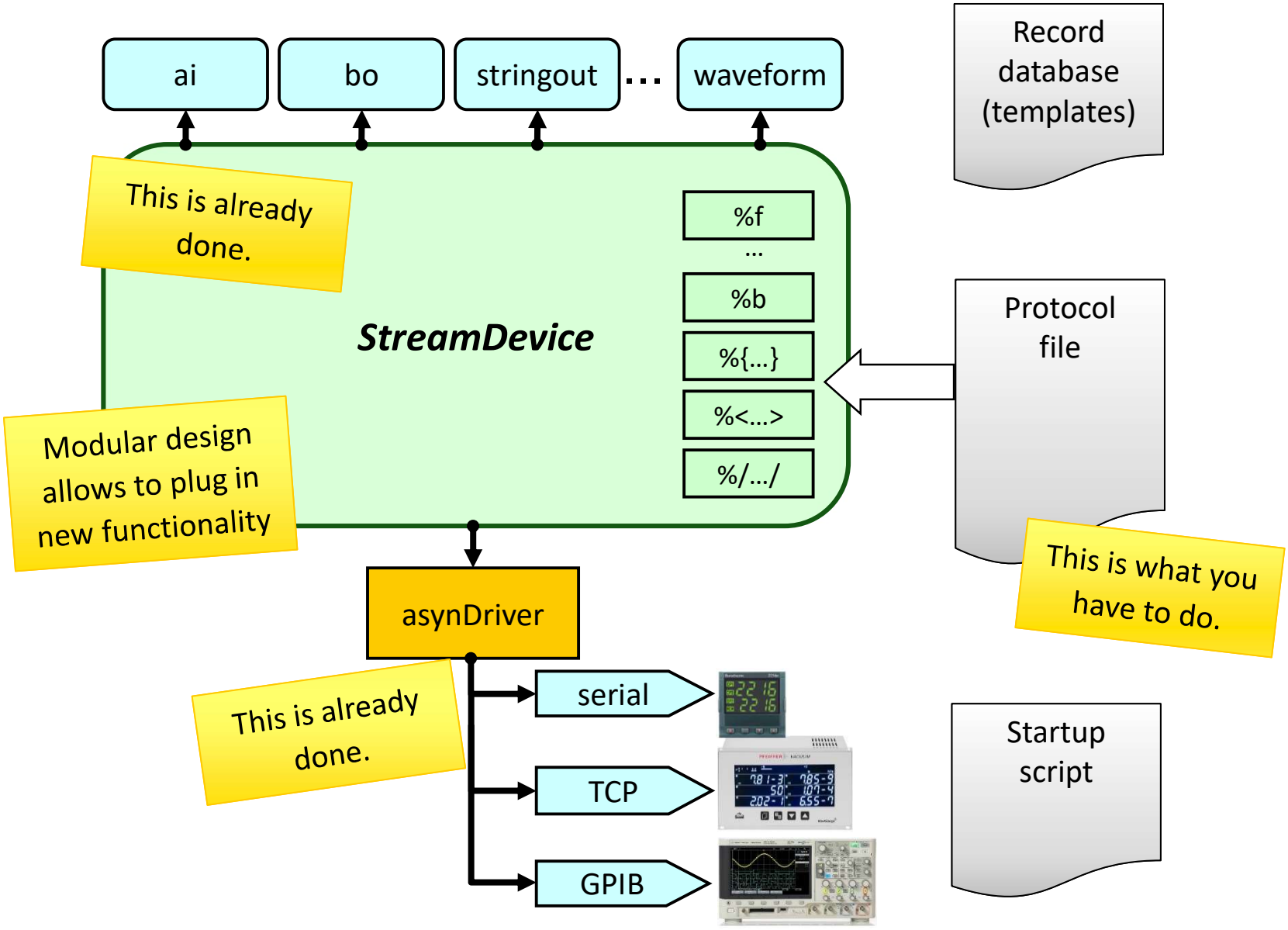
- Command words
- Terminators

Value encoding

- Formatting
- Position in the command

→ Configuration

StreamDevice Architecture

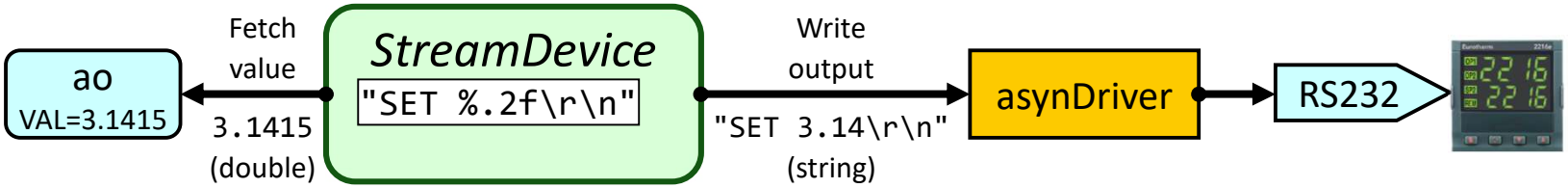


StreamDevice and asynDriver

- Is StreamDevice a replacement for asynDriver?
  - No. In fact it uses asynDriver to talk to the hardware.
  - To be exact: It uses the asynOctet interface.
- But doesn't asynDriver already come with device support?
  - Yes, but for string based devices only stringin, stringout and char waveform.

- **StreamDevice does the string formatting and parsing and asynDriver does the hardware access.**

Example:



Example: Read a Device in 3 Easy Steps

Read Kelvin temperature #1 from a LakeShore 218 via RS232

- *Command: "KRDG? 1" CR LF*
- *Reply: "+273.15" CR LF*

1. Configure asynDriver **port** in startup script

```
drvAsynSerialPortConfigure "LakeShore1","/dev/ttyS0"
asynSetOption "LakeShore1", 0, "baud", "9600"
asynSetOption "LakeShore1", 0, …
```

2. Write a **protocol file**, e.g. file LS218.prot

```
Terminator = CR LF;
temp_K_1 {
    out "KRDG? 1";
    in "%f";
}
```

File name and extension does not matter but I recommend to name it after the device type.

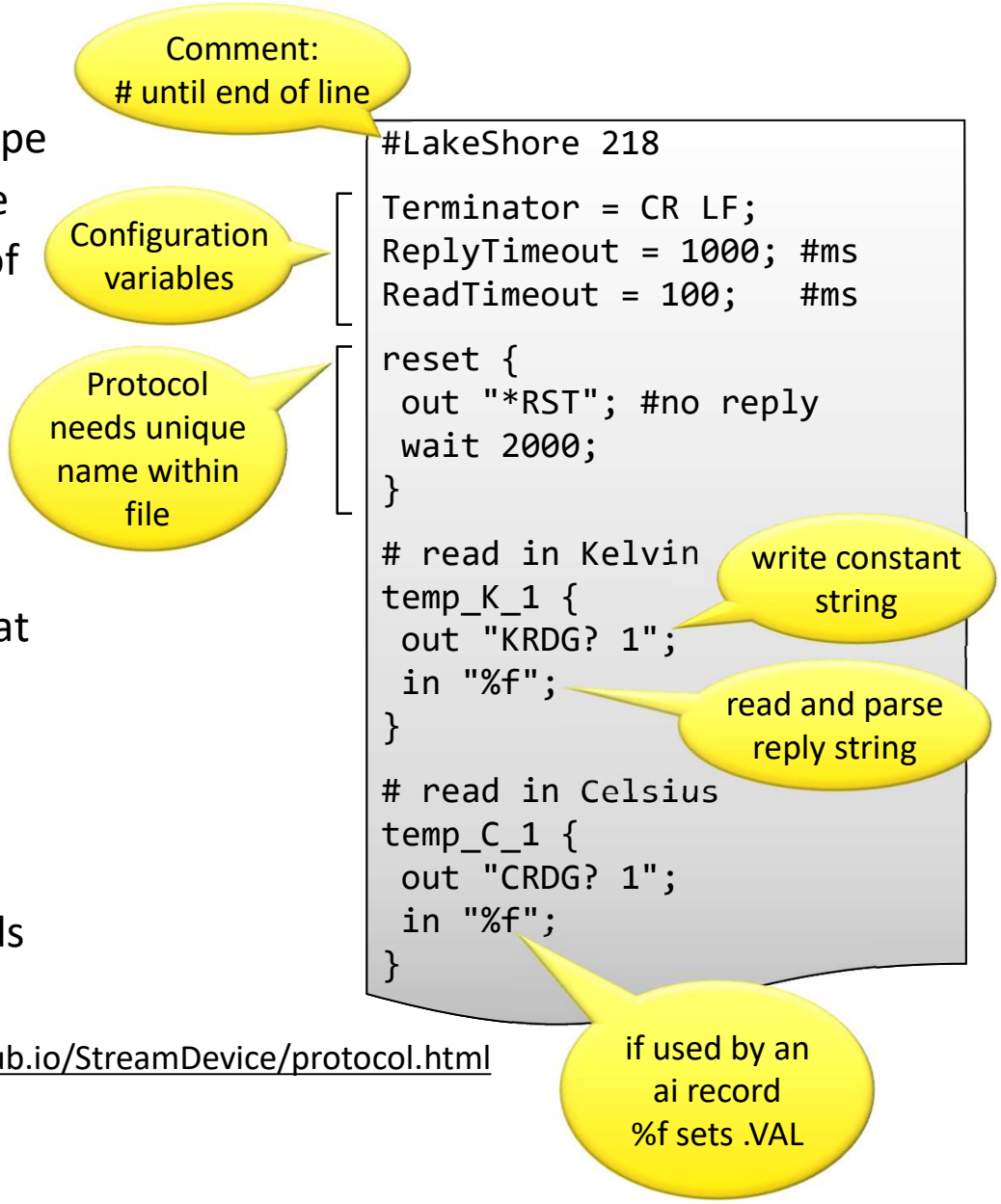
3. Write a **record** connecting to protocol file, protocol, and bus

```
record (ai, "$(DEVICE):TEMP1") {
    field (DTYP, "stream")
    field (INP,  "@LS218.prot temp_K_1 LakeShore1")
    field (EGU,  "K")
    field (PREC, "2")
}
```

Protocol files are searched in ${STREAM_PROTOCOL_PATH}.

Protocol Files

- Have one file for each device type
  - Best name it after device type
- Set some global configuration of communication properties
- Write one protocol for each function/command of device
- Use `out` and `in` commands send and receive strings
- Format/parse values with format converters like `%f`.
  - Format converters work on implicit variables, typically VAL or RVAL fields.
  - You can redirect to other fields or even other records
- See: https://paulscherrerinstitute.github.io/StreamDevice/protocol.html



Strings in StreamDevice

- Strings can be single or double quoted.
  - `"That's a string"`
  - `'This one too'`

Only difference: no need to escape the other quote type

- Strings can be written in chunks, with or without comma
  - `"This is", ' one'`
    `" big"  ' string'`

This is the way to break long strings into multiple lines.

- Non-printable characters can be written in many ways:
  hex, oct, or dec number, escape sequence, symbolic name
  - `0x0a 012 10 "\x0a" "\012" "\10" "\n" LF NL`
  - `0x00 0 "\x00" "\0" NUL`
- Alternatives can be freely mixed
  - `0x1b "00", 'PR3' CR`

For a list of symbolic names and escape sequences see the documentation.

- See: https://paulscherrerinstitute.github.io/StreamDevice/protocol.html#str

Formats as Known From `printf/scanf`

- Standard formats
  - `%f, %e, %E, %g, %G, %d, %i, %x, %o, %s, %c, %[…]`
  - standard flags `%0 -+#` to modify format and `%*` to skip input
  - new flags `%?` for optional input and `%=` for input comparison
  - but no type modifiers like `%hi` or `%lf`
- Additional formats and features
  - `%b` (bit strings like "10010001")
  - `%r` (raw machine format numbers)
  - `%{…|…|…}` (enumerations)
  - `%<…>` (checksums)
  - `%/…/` (regular expressions)

These are just some examples. More formats can be added.

- Variable is implicit, usually VAL or RVAL field
  - Depends on converter type and record type
  - Can be redirected like `%(EGU)s` or `%(record)f`

"redirection" is explained later

- See: https://paulscherrerinstitute.github.io/StreamDevice/formats.html

Enum Format: %{...|...|...}

- Translates integers to enumerated strings and back
  Example: Device can be switched with "STOP" *and* "GO" *and can run forward or backward with different speeds (e.g. 1 or 5 mm/s) and reports this as* "<<", "<", "=", ">", *or* ">>".

```
record(bo,"$(DEV):SWITCH") {
 field(DTYP,"stream")
 field(OUT, "@$(PROTO) switch $(PORT)")
 field(ZNAM,"off")
 field(ONAM,"on")
}

record(longin,"$(DEV):SPEED") {
 field(DTYP,"stream")
 field(INP, "@$(PROTO) speed $(PORT)")
 field(EGU, "mm/s")
}
```

```
switch {
 out "%{STOP|GO}";
}

speed {
 out "DIR?";
 in  "DIR=%#{<<=-5|<=-1|\=|>>=5|>=1}"
}
```

0 = "STOP"
1 = "GO"

%#{ allows assignments:
"<<" = -5
"<" = -1
"=" = 0
">" = 1
">>" = 5

Escape any literal | or } or = or \ in enum strings as \| or \} or \= or \\

>> before > to avoid wrong match

Input not matching any string raises INVALID/CALC alarm

- See: https://paulscherrerinstitute.github.io/StreamDevice/formats.html#enum

Checksums: %<…>

- Many types of checksums supported
  - sum, xor, crc (8,16,32 bit), adler32, …
- Output: checksum is appended to output
- Input: checksum is compared with input

Checksum mismatch raises INVALID/CALC alarm

- Range selection: %<*start*>.<*end*><*checksum*>
  - <*start*>: number of bytes to skip from beginning of string (default 0)
  - <*end*>: number of bytes to skip just before %<…> (default 0)

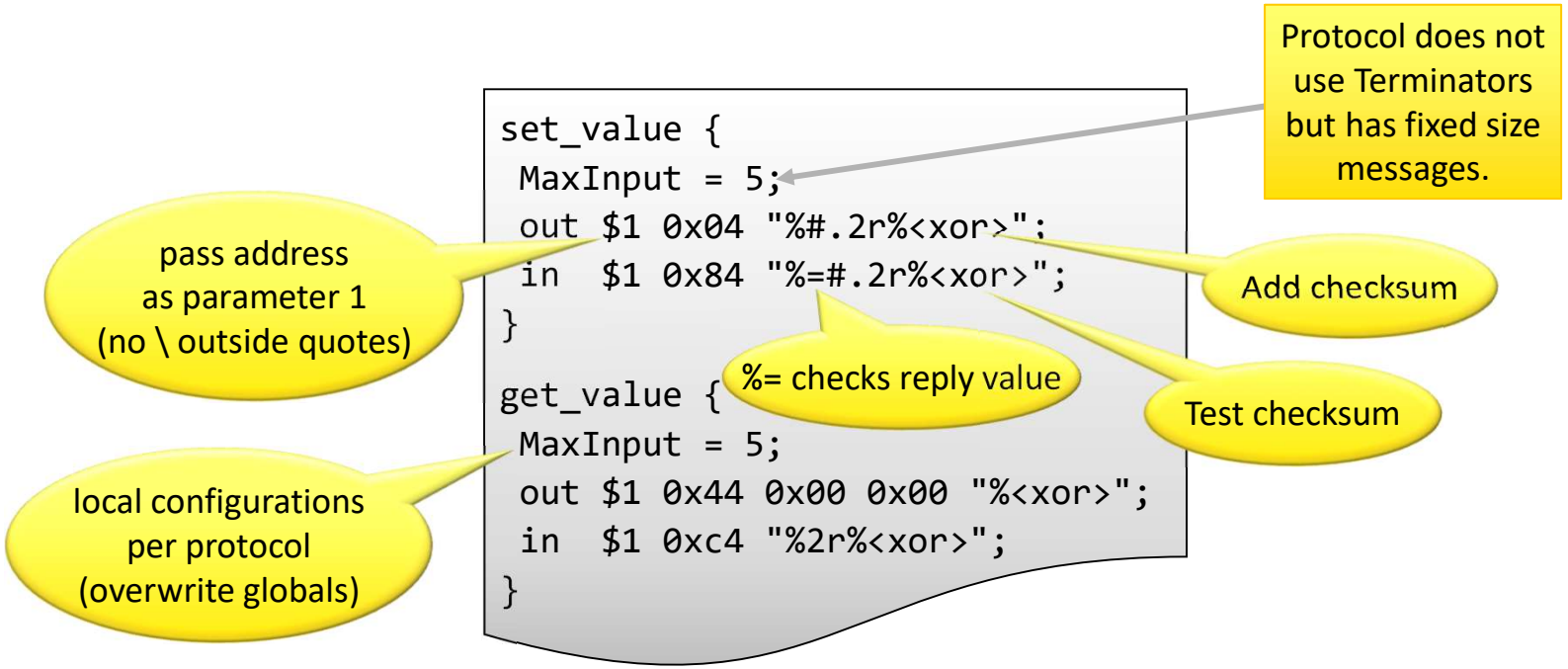
  Example: `out "Exa**mple stri**ng%3.2<xor>";`
- Flag %0<…> switches to (upper case) hex digits instead of raw bytes
- Flag %#<…> switches to little endian for multi-byte checksums
- See: https://paulscherrerinstitute.github.io/StreamDevice/formats.html#chksum

Binary Protocols (aka “Telegrams”)

Example: Send/receive 5 byte telegrams:

- *Address, command, 2 byte value (little endian), xor-checksum*
- *Confirmation is the same with bit 8 set in command byte*

- Use %r to send/receive integers in "raw" machine code format



Raw Format: %r and %R

- Output format: %<*flag*><*width*>.<*precision*>r
  - <*flag*> #: little endian (default is big endian)
  - <*width*>: number of bytes printed (sign extended, default 1)
  - <*precision*>: number of bytes taken from variable (default 1)

  Example: value = 0xdeadbeef

```
out "%2r";     0xff 0xef             1 byte sign extended to 2 bytes
out "%.2r";    0xbe 0xef             2 bytes
out "%#.4r";   0xef 0xbe 0xad 0xde   4 bytes little endian
out "%#4.2r"; 0xef 0xbe 0xff 0xff   2 bytes little endian sign extended to 4 bytes
```

Pitfall !

- Input format: %<*flag*><*width*>r
  - <*flag*> #: little endian
  - <*flag*> 0: zero extend to long (default is sign extend)
  - <*width*>: number of bytes to read

sorry for the mess
out "%.2r"
versus
in "%2r"

- %4R is float and %8R is double
- See: https://paulscherrerinstitute.github.io/StreamDevice/formats.html#raw

Regular expressions: %/…/ and %#/…/…/

- Uses Perl Compatible Regular Expressions (PCRE)
- Normal regex: %/*expression*/ returns a string
  - E.g. find keyword in long text: `in "%.1/(?im)<title>(.*?)<\/title>/";`

%/…/ skips all input before match if expression does not start with ^

Get 1st sub expression here: (.*?)

Case insensitive multi-line match

Escape the /

- Regsub: %#/*expression*/*substitution*/ modifies input/output
  - Make strange input readable:
    - 1.23- → -1.23 `in "%#/([^+-])*([+-])/\2\1/%f";`
    - 865+3 → 8.65e+3 `in "%#/([0-9])([0-9]*)([-+][0-9]+)/\1.\2e\3/%f";`
  - Convert output to upper or lower case (and similar things)
    - \path\x\y → /path/x/y `out "%s%/\\/\//";`
    - Lower case hex checksum `out "%s%0<xor>%#-2/.*/\L&/";`

Last 2 bytes to lower case

- See: https://paulscherrerinstitute.github.io/StreamDevice/formats.html#regex

Protocol Parameters

- Passing parameters from record to protocol can save typing.
  Example: LakeShore218 has 8 temperatures

```
Terminator = CR LF;
temp_K_1 {out "KRDG? 1"; in "%f";}
temp_K_2 {out "KRDG? 2"; in "%f";}
temp_K_3 {out "KRDG? 3"; in "%f";}
temp_K_4 {out "KRDG? 4"; in "%f";}
temp_K_5 {out "KRDG? 5"; in "%f";}
temp_K_6 {out "KRDG? 6"; in "%f";}
temp_K_7 {out "KRDG? 7"; in "%f";}
temp_K_8 {out "KRDG? 8"; in "%f";}
```

```
record(ai,"$(DEVICE):TEMP2") {
 field(DTYP,"stream")
 field(INP,"@LS218.prot temp_K_2 $(PORT)")
 field(EGU,"K")
 field(PREC,"2")
}
```

```
Terminator = CR LF;
temp_K {out "KRDG? \$1"; in "%f";}
```

Use parameter instead:
"\$1" is 1st parameter.

Pass value
for 1st parameter.

```
record(ai,"$(DEVICE):TEMP2") {
 field(DTYP,"stream")
 field(INP,"@LS218.prot temp_K(2) $(PORT)")
 field(EGU,"K")
 field(PREC,"2")
}
```

Up to 9 parameters possible.

Redirections

- Normally format converters choose field automatically
  - VAL or RVAL, depending on format data type
  - Scaling (ASLO, AOFF) and smoothing (SMOO) are supported
- But sometimes other fields or even other records are needed
  - e.g. when one message contains more than one value.

  Examples:

  Write value and units: `out "%f %(EGU)s";`

  Read status and value: `in "%(otherrecord.FIELD)d,%f";`
- Other records must be on the same IOC.
- Use parameters to avoid record names in protocols.
  - Like `"%(\$1)d"` or `"%(\$1:STATUS)d"`

If no field is given .VAL is assumed.

Pass full record name as parameter.

Pass only part of record name as parameter.

Redirection Example

Example: `"position?"` *request gives reply with 6 coordinates.*

```
"position( 1.2    , 2.1234, 23.12  ,-2.312 ,-12.33  ,24.4321)"
```

```
record(ai,"$(DEV):POS-X") {
 field(DTYP,"stream")
 field(INP, "@$(PROT) pos($(DEV)) $(PORT)")
 field(EGU, "mm")
 field(PREC,"2")
 field(SCAN,"5 second")
 field(FLNK,"…")
}
record(ai,$(DEV):POS-Y) {
 field(EGU, "mm")
 field(PREC,"2")
}
…
record(ai,$(DEV):POS-RZ) {
 field(EGU, "deg")
 field(PREC,"3")
}
```

When FLNK is processed, the other records have already got their values.

These records have SCAN="Passive" and DTYP="Soft Channel"

When redirecting to .RVAL use DTYP="Raw Soft Channel"

```
pos {
  out "position?";
  in "position(%f\_,"
     "%(\$1:POS-Y)f\_,"
     "%(\$1:POS-Z)f\_,"
     "%(\$1:POS-RX)f\_,"
     "%(\$1:POS-RY)f\_,"
     "%(\$1:POS-RZ)f\_)";
}
```

Spaces around numbers:

%f accepts leading whitespace.
Any amount of trailing whitespace is consumed by _.

Redirection to other records uses `dbPutField()` and thus processes the target if the field (here .VAL) is PP

Complex Protocol: Pfeiffer MaxiGauge

1. Send device address (on RS485)
   ESC *(=0x1b)* "*aa*"
   aa is address 00, 01, …
2. Send pressure read command:
   "PR*n*" CR
   n is sensor number 1 … 6
3. Each command is acknowledged with
   ACK *(=0x06)* CR LF
4. After acknowledge request value with
   ENQ *(=0x05)*
5. Receive status and pressure:
   "0,1.234E-5" CR LF

```
InTerminator = CR LF;
# no OutTerminator because of ENQ
pressure {
 out ESC "0\$1"
     "PR\$2" CR;
 in  ACK;
 out ENQ;
 in "%(\$3:STATUS)d,%f";
}
```

address

sensor

Redirect status to other record.

Keep pressure in active record.

Pass address, sensor and device name as 3 parameters.

```
record(ai,"$(D):PRESSURE") {
 field(DTYP,"stream")
 field(INP, "@%(PROT) pressure($(A),$(S),$(D)) $(B)")
}

record(mbbi,"$(D):STATUS") {
 field(ZRST,"OK")
 field(ONST,"Underrange")
 field(TWST,"Overrange")
 …
}
```

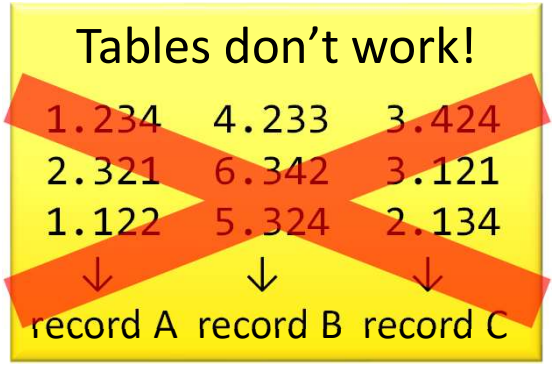
Arrays

- For arrays the format is repeated for each element with optional separator string in between.
  Example: "VALUE=[10.2,345.3,23.3,10.0]"

```
readArray {
 Separator = ",";
 in "VALUE=[%f]";
}
```

```
record(aai,"$(DEVICE):Array") {
 field(FTVL,"FLOAT")
 field(NELM,"100")
 field(DTYP,"stream")
 field(INP, "@$(PROT) readArray $(PORT)")
}
```

- Input
  - reads a maximum of NELM elements
  - at least one element must be found
  - sets NORD to number of elements found
- Output
  - writes NORD elements



Asynchronous input: I/O Intr

- Some devices send unsolicited data

  Example: Temperature sensor sends periodically: `"+27.3 C" CR LF`

```
record(ai,"$(DEVICE):TEMP") {
 field(DTYP,"stream")
 field(INP, "@$(PROT) temperature $(PORT)")
 field(EGU, "C")
 field(PREC,"1")
 field(SCAN,"I/O Intr")
}
```

```
Terminator = CR LF;

temperature {
 in "%f C";
}
```

- Asynchronous records have SCAN=`"I/O Intr"`.
- Asynchronous protocols typically have only an `in` command.
- Any input from that device is checked.
  - Including input going to other protocols
  - Record is processed whenever matching input is found.
  - Non-matching input is silently ignored (no error).

I/O Intr vs. Redirection

Example: Read input line with two values x and y every second

```
record(ai,"$(D):x") {
 field(SCAN,"1 second")
 field(DTYP,"stream")
 field(INP, "@$(PROT) x $(PORT)")
}
record(ai,"$(D):y") {
 field(SCAN,"I/O Intr")
 field(DTYP,"stream")
 field(INP, "@$(PROT) y $(PORT)")
}
```

```
x {
 out "xy?";
 in "%f %*f";
}
y {
 in "%*f %f";
}
```

y is processed when **anyone** receives input that matches.

Ignore second value.

Ignore first value.

Be careful:

Any input with two numbers matches.

```
record(ai,"$(D):x") {
 field(SCAN,"1 second")
 field(DTYP,"stream")
 field(INP, "@$(PROT) x($(D)) $(PORT)")
}
record(ai,"$(D):y") {
 field(SCAN,"Passive")
 field(DTYP,"Soft Channel")
}
```

y is processed when x **writes** value to it.

```
x {
 out "xy?";
 in "%f %(\$1:y)f";
}
```

Redirect second value.

Redirection is more selective.

Only input from x goes to y.

Error Handling

- All errors set SEVR to INVALID.
  - STAT depends on error type:
    - Reply timeout: Device does not reply. → TIMEOUT
    - Read timeout: Device stops sending unexpectedly. → READ
    - Write timeout: Writing failed. → WRITE
    - Lock timeout: Bus is blocked by other records. → TIMEOUT
    - Mismatch: Input does not match format string. → CALC
- Errors abort the protocol
  - but may trigger exception handlers .
- Errors are not propagated along redirections.
  - Redirections before mismatch are processed normally.
  - Redirections after mismatch are not processed at all.
  - Only active record gets error status.
- Mismatch in "I/O Intr" records is silently ignored.

I am planning to change that.

Exception Handlers

- Jump to different protocol if something goes wrong
  - @mismatch, @replyTimeout, @readTimeout, @writeTimeout

  Example: Device replies with value or error message

```
readValue {
 out "value?";
 in  "%f";

 @mismatch {
  in  "%(\$1:ErrorString)39c";
 }
}
```

If input does not match "%f" jump to @mismatch.

Reparse input and redirect.

If input does not match and the @mismatch handler starts with in, then **the same** input is parsed a second time.

Exception handlers can be defined **before** or **inside** normal protocol.

```
record(ai,"$(DEVICE):Value") {
 field(DTYP,"stream")
 field(INP, "@$(PROT) readValue($(DEVICE)) $(PORT)")
}
record(stringin,"$(DEVICE):ErrorString") {
}
```

Record Initialization

- Output records can be initialized from hardware
  - Initialization is formally handled as an exception: `@init`

```
getVolt {
 out "VOLT?"; in "VOLT %f";
}

setVolt {
 out "VOLT %.2f";
 @init {getVolt;}
}

getFreq {
 out "FREQ?"; in "FREQ %f MHZ";
}

setFreq {
 out "FREQ %.6f MHZ";
 @init {getFreq;}
}
```

Reference other protocol to insert all its commands.

@init is executed in iocInit before scans are started, when ioc is un-paused and when device re-connects.

```
record(ai,"$(D):ReadVoltage") {
 field(DTYP,"stream")
 field(INP, "@$(PROT) getVolt $(PORT)")
}
record(ao,$(D):SetVoltage") {
 field(DTYP,"stream")
 field(OUT, "@$(PROT) setVolt $(PORT)")
}
record(ai,"$(D):ReadFrequency") {
 field(DTYP,"stream")
 field(INP, "@$(PROT) getFreq $(PORT)")
}
record(ao,$(D):SetFrequency") {
 field(DTYP,"stream")
 field(OUT, "@$(PROT) setFreq $(PORT)")
}
```

Recommended Reading

- StreamDevice documentation
  https://paulscherrerinstitute.github.io/StreamDevice
- AsynDriver documentation:
  https://epics-modules.github.io/master/asyn
- Record Reference Manual
  https://wiki-ext.aps.anl.gov/epics/index.php/RRM_3-14
  https://epics.anl.gov/base/R3-15/8-docs/RecordReference.html
  https://epics.anl.gov/base/R7-0/4-docs/RecordReference.html
- EPICS How-To pages "Getting started"
  - https://docs.epics-controls.org/projects/how-tos
- Perl Compatible Regular Expressions (PCRE)
  - https://www.pcre.org/current/doc/html/pcre2syntax.html