EPICS StreamDevice

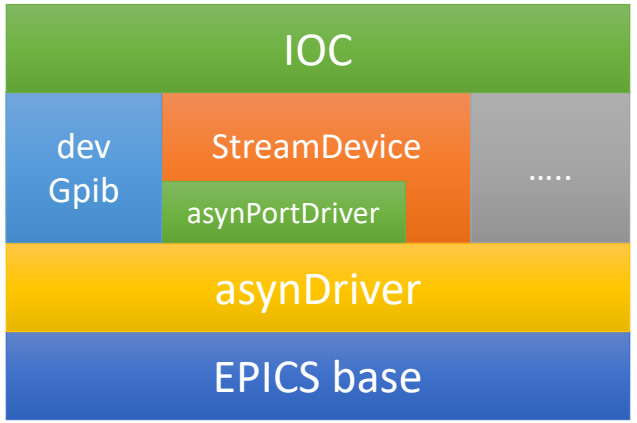
路川徹也　(東日技研)

StreamDeviceとは?

- asynDriverを使った、RS-232C,GPIB,socke通信等のコマンド文字列 (バイナリを含む) で制御するデバイス用IOCを簡単に作成するための中間ライブラリ。
- asynDriverを使いやすくするためのコマンドフォーマットコンバータ。
- コマンド文字列をprotocolファイルで設定する。
- フォーマット形式は、C言語のformatterの書式に酷似している。

protocol の例

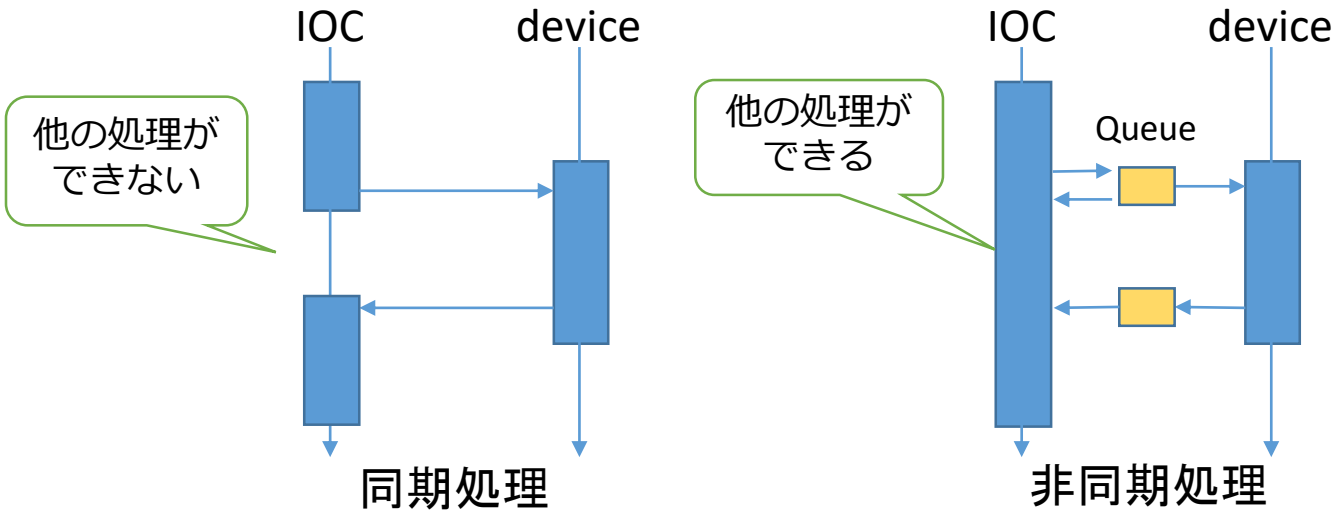
```
Terminator=CR LF;
protocol {
  out "*IDN?";
  in "%39c";
}
```



詳細は、StreamDevice:　https://paulscherrerinstitute.github.io/StreamDevice/index.html

なぜasynDriverが必要？

- asynDriverは、デバイスとの通信を非同期化するためのモジュール。
- 同期通信では、デバイスの応答があるまでIOC(プログラム)が停止してしまうため、その間処理が行われなくなるため非効率。
  - 通信時間がus程度なら問題ないが、ms以上かかるなら非同期にしないと問題になることがある。
- 非同期化することで、デバイスとの通信中でも他の処理を行うことが可能となる。



詳細は、asynDriver: https://epics-modules.github.io/asyn/

なぜStreamDeviceなのか

- 簡単なEPICS IOCでも新規に作るのは、初心者にはハードルが高い。
  - C言語でのプログラミング
  - 開発環境の知識
  - EPICSライブラリ関数
- EPICS IOCの作成は、経験者でもある程度時間がかかり、問題が発生した場合には、debugも難しい。
- 非同期処理を実現するには、asynDriverを直接使ったり、devGpibやasynPortDriver等の手段もあるが、わかり辛いのにはあまり変わりない。
- 測定器とのコマンド通信は、シリアルやGPIB(VXI-11)、ソケット通信のASCII文字列が多いので、単純なコマンドなら短時間で作れる。

StreamDeviceの利点と欠点

- 利点
  - 導入のためのハードルは、**比較的**低い。
  - 書式が単純なので、プロトタイピングにも使用できる。
  - protocolファイルの修正だけなら、再コンパイル不要。
  - StreamDeviceのみなら、C言語の知識はあまり必要ない。(全くではない)

- 欠点
  - ASCII文字列でも複雑なデータを扱うのは困難。
  - １つのコマンドから、複数のデータに分割するのは難しい。
    - 分割する方法は、後述する。
  - 文字列のパースが遅い。
    - 処理速度の速いサーバーなら問題ないが、組み込みCPUでは問題になることがある。
  - binaryも単純なフォーマットなら扱えるが、複雑なものは別の手段を考えたほうがいい。

protocol ファイル 基本的な記述

```
# 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"
}
```

コメント設定
- #以降はコメント

ターミネータ設定
- in/outの終端文字列
  - Terminator:　　in/out共通

関数設定
- レコードのINP/OUTに設定する
- { } で囲んだ部分が対象

送受信文字列設定
- 実際に送受信する際には、Teminator文字が後ろについて送受信される。
- out：　送信文字列
- in：　　受信文字列

初期化関数設定(@init)
- iocInit時に実行する関数

余剰入力設定(ExtraInput)
- inで余計な入力文字があった場合の処置
  - Ignore：　無視
  - Error:　　エラー

dbファイルとprotocolファイルの関係

stream.db

```
record(ai, "$(user):freq:get") {
    field(DTYP, "stream")
    field(INP,  "@stream.proto getFrequency $(dev)")
}

record(ao, "$(user):freq:set") {
    field(DTYP, "stream")
    field(OUT,  "@stream.proto setFrequency $(dev)")
}
```

stream.proto

```
Terminator = CR LF;
getFrequency {
    out "FREQ?";
    in "%f";
}

setFrequency {
    out "FREQ %f";
}
```

dbでのStreamDeviceの設定

- DTYPは“stream”
- INP/OUTは “@filename function device”

protocolファイルの設定

- function {
  command “string”;
  }
- “%～”がレコードのVALに対応する。

一般的なprotocolファイル設定

Command

- in　　入力
- out　　出力
- wait　　待ち

format

- double(%f, %e,%g)
- long(%d, %u, %i)
  - 8進数(%o)
  - 16進数(%x, %X)
- string,char(%s,%c)
- enum(%{str0|str1|..})
- raw long(%r)
- etc…

System

- Terminator　　入出力ターミネータ
- OutTerminator　　出力ターミネータ
- InTerminator　　入力ターミネータ
- WriteTimeout　　書込タイムアウト
- ReadTimeout　　読込タイムアウト
- ReplyTimeout　　応答タイムアウト
- MaxInput　　入力最大長
- Separator　　分割文字列
- ExtraInput　　Error/Ignore

例外

- @mismatch　　フォーマットエラー
- @writetimeout　　書込タイムアウト
- @replytimeout　　応答タイムアウト
- @readtimeout　　読込タイムアウト
- @init　　初期化

主なprotocolファイルの特殊文字、シンボル

主な特殊文字

- **¥ % ' "**
  システムで使用する特殊文字
  ¥はエスケープ文字として使用
- **¥$**
  inの関数引き数を指定
- **¥_**
  スペース

主なシンボル

- NUL 0x00
- SOH 0x01
- STX 0x02
- ETX 0x03
- EOT 0x04
- ACK 0x06
- LF(¥n) 0x0A
- CR(¥r) 0x0D
- NAK 0x15
- ESC 0x1B

protocol ファイル(1) 主なコマンド

基本フォーマット

```
Terminator = CR LF;

function{
  out "string";
  in "string";
}
```

主なコマンド

- **out *string*;**
  デバイスに出力する文字列を設定
- **in *string*;**
  デバイスから受け取る文字列を設定
- **wait milliseconds;**
  待ち時間をミリ秒で設定

主な特殊文字

- **¥ % ' "**
  システムで使用する特殊文字
  ¥はエスケープ文字として使用
- **¥$**
  inの関数引き数を指定
- **¥_**
  スペース

主なシステム設定

- **WriteTimeout=100;**
  デバイス出力タイムアウト(ミリ秒)
- **ReadTimeout=100;**
  デバイス入力タイムアウト(ミリ秒)
  InTerminatorが設定されていない("")場合には、
  タイムアウトは発生しない。
- **Terminator**
  in/outの終端文字を設定する。
- **OutTerminator=$Terminator;**
  outの終端文字を設定する。
- **InTerminator=$Terminator;**
  inの終端文字を設定する。
  この設定を行わないと、入力が完了しないので、ReadTimeoutが発生する。
  但し、MaxInputが設定されている場合には無視される。
- **MaxInput=0;**
  入力文字列長を設定する。固定長メッセージを受信する場合に設定する。
- **Separator="";**
  waveformやaai/aaoレコードで、複数のデータを連結/分割する文字を設定する。
- **ExtraInput=Error;**
  inでメッセージ処理が終わった後で受け取ったメッセージの取り扱いを設定する。Errorは処理異常になり、Ignoreは無視する。

protocol ファイル(2) 主なコマンド

プロトコル引数

- レコードに引数を設定することで、プロトコル定義の共有化が可能になる。
- 詳細と例は後述する。

```
field(INP, "@proto func(x) dev")

func {
  out "get¥$1";
  in  "%f";
}
```

ユーザー定義変数

- ユーザー定義変数が設定可能
- 固定コマンドヘッダ等を設定しておくと便利

```
f = "FREQ";
fq = $f "?";

setFreq{out "$f %f";}
getFreq{out "$fq"; in "%f";}
```

例外ハンドラ

例外(エラー)発生時に実行するコマンドを設定する。

```
function{
  out "data?";
  in "%f";
  @mismatch {}
  @init {out "Init"; in "%f";}
}
```

- **@mismatch**
  inでフォーマットと違うメッセージを受け取った場合に発生する
  エラー時に文字列を返すようなデバイスで使用する。
- **@writetimeout**
  デバイス書き込みタイムアウト時に発生する。
- **@replytimeout**
  デバイスのリプライタイムアウト時に発生する。
- **@readtimeout**
  デバイスの読み込みタイムアウト時に発生する。
- **@init**
  iocInit時に発生する。
  データの初期値をデバイスから取得しておきたい場合に使用する。

Format Converter(1) 書式

フォーマット変換書式

- “%”が変換する文字列の先頭文字
- “()”で囲むとレコード名やフィールドを指定する
- “*# +0-?=!”は特殊文字として扱う
- 数字は文字列長

```
in “%f”;                固定少数(小数点以下6桁)
out “%(HOPR)7.4f”;      HOPRフィールドの値を7桁小数点以下4桁の少数に変換
out “%#010x”;           0詰めした10桁の16進数(0x0000012345)
in  “%*i”;              整数値を変換しない
in  “%?d”;              整数値が無ければ、0
in  “%=.3f”;            現在値と変換値があっているか
```

フォーマット変換識別子

- “*”は、inで使用し、指定したデータを変換しない時に使用する。
  - “1.23 2.34”を“%f %*f”で変換した場合には、”1.23”のみが変換され、レコードのVALに設定される。
  - 但し、変換はされないがフォーマット自体はチェックされるので、型が一致しなければエラーになる。
- “#”は、レコードの種類によって処理内容が異なる。
- “ ”(スペース)と“+”は、その後ろにあるデータが正の値であることを示し、“-”は負の値であることを示す。
- “0”は、その後にある数字で“0”埋めするフラグ
- “-”は、outでは左詰め
- “?”は、変換する値が無ければ”0”とする
- “=”は、現在値と変換値があっているかをチェックする
- “!”は、厳密に桁数をチェックする

Format Converter(2) 型変換

double(%f,%e,%E,%g,%G)

固定小数点や対数と文字列を変換する。

- **Out**
  - %f: 固定小数点
  - %e,%E: 対数
  - %g,%G: 固定小数点か対数
- **in**
  - %f: 固定小数点
  - %e,%E: 対数
  - %g,%G: 固定小数点か対数
  - #: 数値と符合"+,-"の間のスペースを無視

```
out "%.3f";       1.23      ->  "1.230"
out "%f";         1e-7      ->  "0.000000"
out "%e";         1.5e-4    ->  "1.500000e-04"
out "%.2e";       1.5e-4    ->  "1.50e-04"
out "%g"          1e-4      ->  "0.0001"
out "%g"          1.5e-6    ->  "1.5e-06"

in "%f";          "1.23"    ->  1.23
in "%e";          "1.5e-5"  ->  1.5e-05
in "%e";          "0.0001"  ->  1.0e-04
in "%#f";         "-   12.3"->  -12.3
```

Format Converter(3) 型変換

long(%d,%i,%u,%o,%x,%X)

整数と文字列を変換する。

- **Out**
  - %i,%d: 符号付整数
  - %u: 符号なし整数
  - %o: 8進数
  - %x,%X: 16進数
  - #: 8進数は0付加、16進数は0x,0Xを付加
    桁数が表示桁以上なら右詰
- **in**
  - %d: 符号付整数
  - %i: 符号付整数
  - %u: 符号なし整数
  - %o: 符号なし8進数
  - %x,%X:符号なし16進数
  - #:数値と符合"+,-"の間のスペースを無視
  - -:負値の8進数、16進数

```
out "%d";          123      ->  "123"
out "%04d";        123      ->  "0123"
out "%#6d";        123      ->  "   123"
out "%o";          123      ->  "173"
out "%#6o";        123      ->  "  0173"
out "%x";          123      ->  "7b"
out "%#6x"         123      ->  "  0x7b"

in "%d";           "123"    ->  123
in "%#d";          "-  123" ->  -123
in "%o";           "123"    ->  173
in "%x";           "123"    ->  0x7b
```

Format Converter(4) 型変換

string(%s,%c)

- **Out**
  - %s: 文字列
  - %c: 1文字
- **in**
  - %s: 文字列
  - %c: 1文字
  - #: 数値と符合"+,-"の間のスペースを無視

```
out "%c";          "abc"     ->  "a"
out "%s";          "abc"     ->  "abc"
out "%39c";        "abc"     ->  "abc"

in "%c";           "abc"     ->  ERROR
in "%s";           "abc"     ->  "abc"
```

enum(%{string0|string1|..})

"|"で区切った文字列に合致した番号をVALに設定する。
一番初めに設定した文字列は"0"になり、それ以降は、
1,2,3…の値に変換される。
string=Nで文字列に対応する番号が設定された場合には、
その値になる。

- **Out**
  文字列を出力する。
- **in**
  入力文字列と合致する文字列の番号を設定する。

```
out "%{OFF|ON}";           0 ->  "OFF"
out "%{OFF=1|ON=0}";       0 ->  "ON"

in "%{OFF|ON}";            "ON"   ->  1
in "%{OFF|ON|BOTH}";       "BOTH" ->  2
```

Format Converter(5) 型変換

raw long(`%r`)

binaryをそのまま取り扱う。
通常はlong型(4byte,big endian)として取り扱う。”#”でlittle endianに変換。

- **Out**
  - `%r`: long型(4byte)で出力
  - `%.2r`: short型(2byte)で出力
- **in**
  - `%r`: long型(4byte)で入力
  - `%02r`: short型(2byte)で入力

raw double(`%R`)

IEEE形式のbinaryをそのまま取り扱う。
通常はfloat型(4byte,big endian)として取り扱う。”#”でlittle endianに変換。

- **Out**
  - `%R`: float型(4byte)で出力
  - `%.8R`: double型(8byte)で出力
- **in**
  - `%R`: long型(4byte)で出力
  - `%08R`: double型(8byte)で出力

Format Converter(6) 型変換

checksum(`%<checksum>`)

文字列のチェックサムを作成/チェックする。
色々なチェックサム関数が実装されているが、
数が多いので今回は割愛する。

- **`Out`**
  送信する文字列の終端に付加する。
- **`in`**
  受信文字列のチェックサムをチェックする。

他にも型変換の定義はたくさんあるが、今回は割愛する。

binary long(`%b`,`%B`)

Packed BCD(`%D`)

Regular Expresion String(`%/regex/`)

Regular Expresion Substitution(`%/regex/subst/`)

MantissaExponent double(`%m`)

Timestamp double(`%T(timeformat)`)

出力設定例(1) 引き数設定(1)

- **似たようなコマンドが複数ある場合には、引数を設定**
  - moveX,moveY,moveZのようなコマンドがあるデバイスの場合、引数を設定することで、プロトコル定義は1つで済む。

```
• moveX
• moveY
• moveZ
```

```
field(OUT, "@motor.proto moveaxis(X) motor1")
field(OUT, "@motor.proto moveaxis(Y) motor1")
field(OUT, "@motor.proto moveaxis(Z) motor1")
```

```
moveaxis {
  out "move¥$1 %.6f";
}
```

- **引数は複数設定が可能(最大9個)**

```
reocrd(ao, "$(RECORD):ex02") {
  field(OUT,  "@XXXX.proto FUNCTION(arg1,arg2,arg3) BUS")
  field(DTYP, "stream")
}
```

出力設定例(1) 引き数設定(2)

例1: 文字列を引数として設定

```
reocrd(ao, "$(RECORD):ex01") {
  field(OUT, "@motor.proto moveaxis(X) motor1")
  field(DTYP, "stream")
}
```

```
moveaxis {
  out "move¥$1 %.6f";
}
```

```
moveaxis {
  out "moveX %.6f";
}
```

- “¥$1”が文字列”X”に変換される。
- “$”が文字として認識されるので”¥”でエスケープ

例2: 数値を引数として設定

```
reocrd(ao, "$(RECORD):ex02") {
  field(INP, "@vac.proto readpressure(0x84) gauge3")
  field(DTYP, "stream")
}
```

```
readpressure {
    out 0x02 0x00 $1;
    in 0x82 0x00 $1 "%2r";
}
```

```
readpressure {
    out 0x02 0x00 0x84;
    in 0x82 0x00 0x84 "%2r";
}
```

- “$1”が数値”0x84”に変換される。

出力設定例(2) 複数データ出力(1)

- **同じ型のデータを定型文字で連結**
  - waveform,aaoレコードを使用する方法

```
record(aao, "$(RECORD):ex05") {
    field(OUT, "@$(DEVICETYPE).proto array_out $(BUS)")
    field(DTYP, "stream")
    field(FTVL, "DOUBLE")
    field(NORD, "4")
}
```

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

この記述子が
配列数にあわせて
変換される

  - "%.2f"のフォーマット文字列が配列数にあわせて、", "で連結される。

```
an array: (1.23, 2.34, 3.45, 4.56)
```

出力設定例(2) 複数データ出力(2)

- **最大12個のデータを連結**
  - calcout のINPA,INPB,...にレコード名を設定する。
  - INPA,INPB,... のVALが%(A),%(B),...に設定される。
  - CALC fieldには計算をしなくても何かしらの設定は必須。

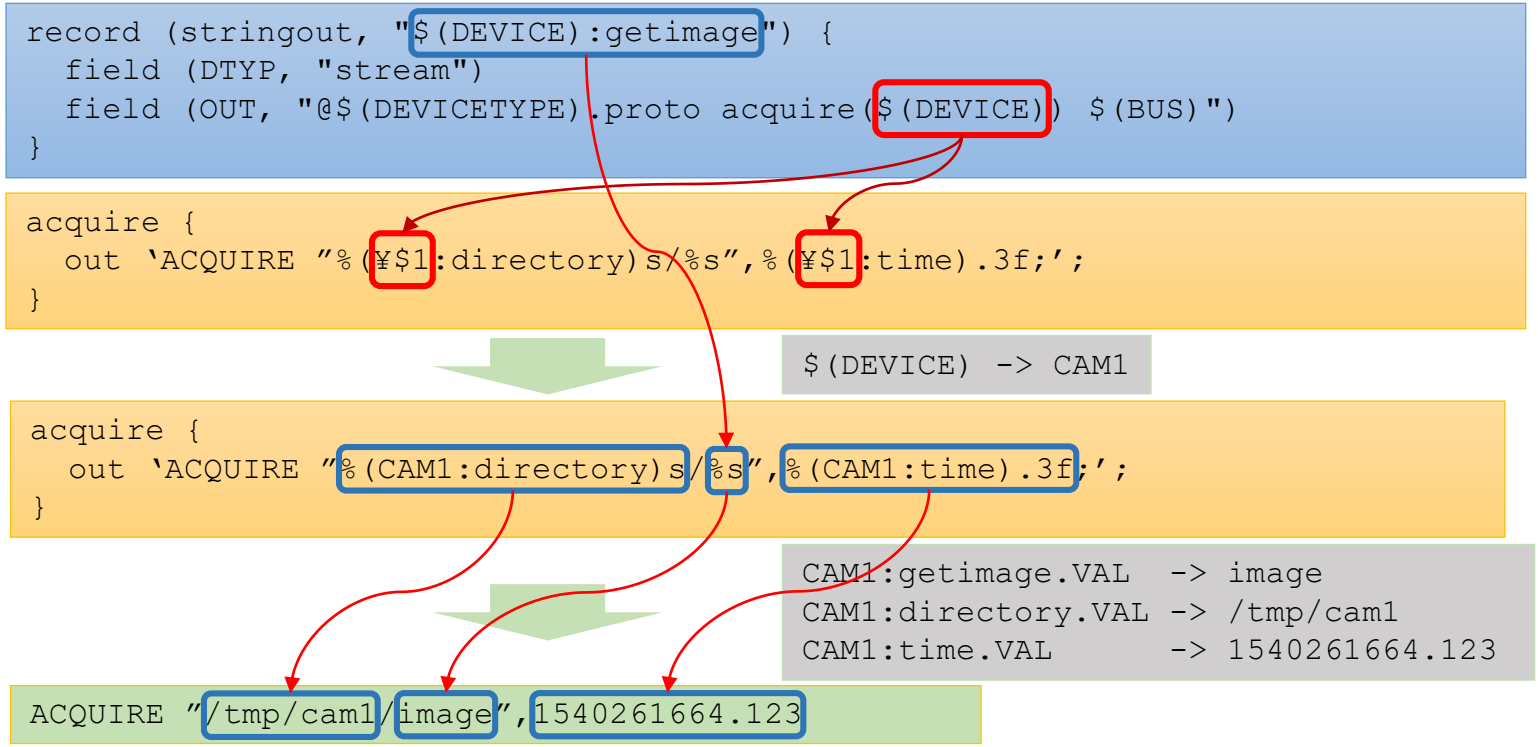
```
record (calcout, "$(RECORD):ex06") {
  field (INPA, "$(A_RECORD)")
  field (INPB, "$(B_RECORD)")
  field (INPC, "$(C_RECORD)")
  field (CALC, "0")
  field (DTYP, "stream")
  field (OUT, "@$(DEVICETYPE).proto write_ABC $(BUS)")
}
```

```
write_ABC {
    out "A=%(A).2f B=%(B).6f C=%(C).1f";
}
```

```
A=1.12 B=2.123456 C=3.1
```

出力設定例(2) 複数データ出力(3)

- **違うレコードの値を出力**
  - プロトコルファイルの引数には文字列が設定できるので、レコード名(その一部)の設定も可能



入力設定例(1) 複数データ入力(1)

- **同じ型のデータが定型文字で連結されている**
  - waveform,aaiレコードを使用する方法

```
record(aai, "$(RECORD):ex05") {
    field(INP, "@$(DEVICETYPE).proto array_in $(BUS)")
    field(DTYP, "stream")
    field(FTVL, "DOUBLE")
    field(NELM, "4")
}
```

  - "%f"のフォーマット文字列が配列数分、", "で連結されている文字列が入力可能

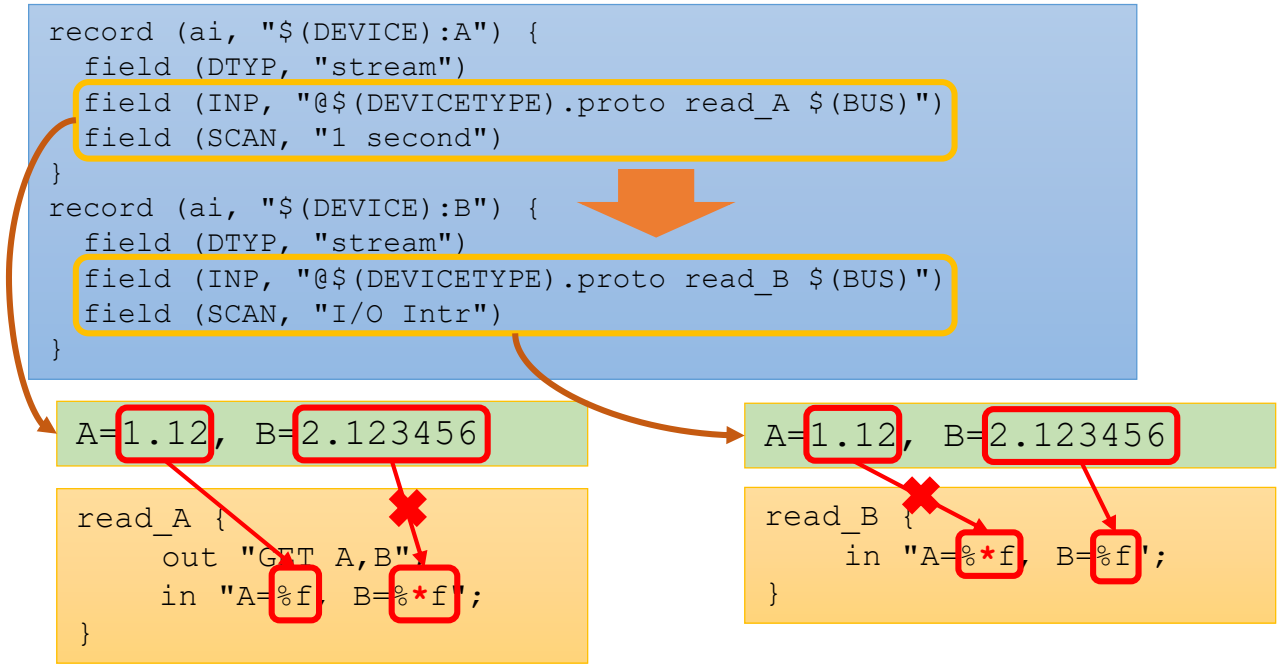
```
an array: (1.23, 2.34, 3.45, -4.56)
```

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

この記述子が
データ数分変換
される

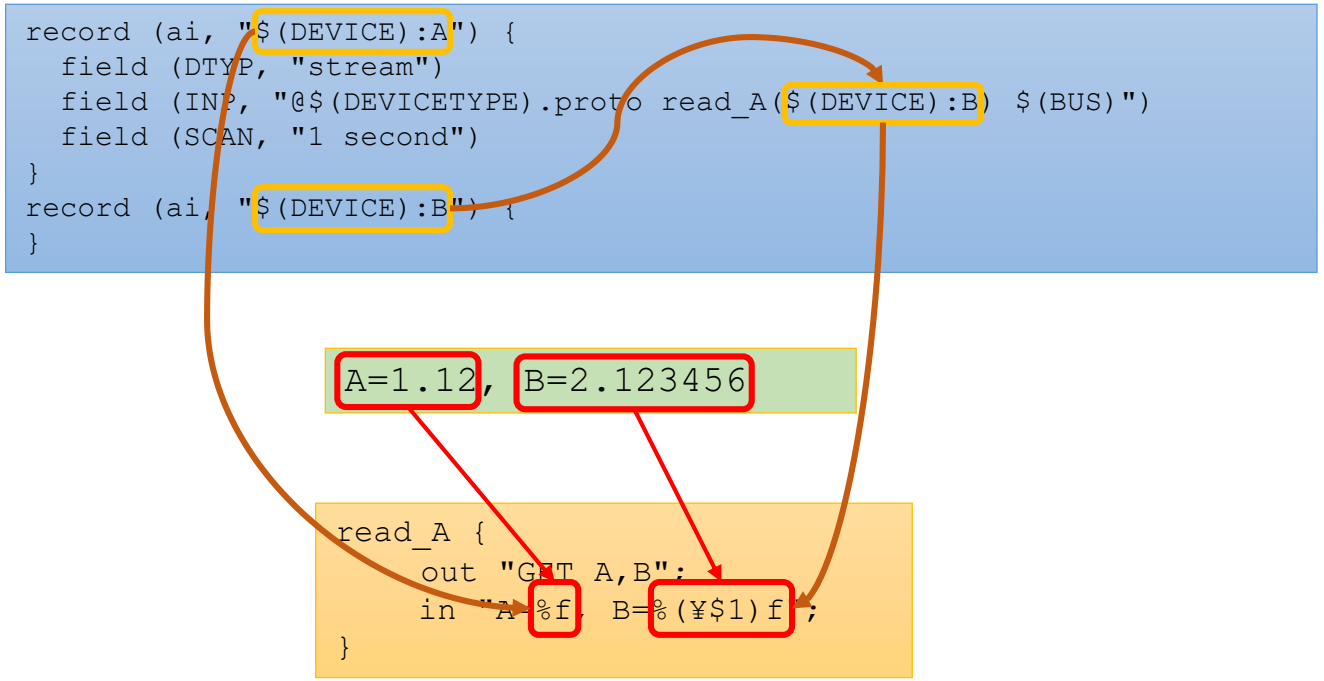
入力設定例(2) 複数データ入力(2)

- **複数データをフィルターで分割**
  - SCANを”I/O Intr”に設定して、”%*”を使ってフィルタリングする。
  - Aレコードを実行後、Bレコードに”I/O Intr”のイベントが発生し、Bレコードの処理が行われる。
  - データを2回取りに行くことはしない。



入力設定例(2) 複数データ入力(3)

- **複数データを引き数設定されたレコードで分割**
  - protocolの引き数にレコード名を設定して、複数のレコードにデータを分割。
  - INPには文字列長制限があるので注意。



StreamDeviceでは扱いにくい例(1)

～ Raymax CLX-1100 Timing stabilizer ～

- Terminatorが遅れて出力される

  ラインプリンタ用のデバッグ出力のようで、1行分のデータを出力後、次のデータを出力する直前にCR+LFが出力される。データの先頭にデリミタがついているような状態のメッセージを出力する。
  また、10行に1回ヘッダ文字列を出力する。

```
2018/11/01 11:11:00 0.111 0.222 3.333
```

← この状態で出力が10秒程度止まる

```
2018/11/01 11:11:00 0.111 0.222 3.333
2018/11/01 11:11:10 0.112 0.223 3.334
```

← 次の行を出力する前にCR+LFを出力

```
2018/11/01 11:11:00 0.111 0.222 3.333
2018/11/01 11:11:10 0.112 0.223 3.334
 DATE         TIME    VAL1   VAL2   VAL3
```

← ヘッダ文字列を出力する

- データ間の区切り文字が無くなる

  ある一定期間を過ぎると、積算動作時間が桁あふれを起こし、前項目のデータにくっついてしまう。

```
2018/11/01 11:11:00 9999 0.111 0.222 3.333
2018/11/01 11:11:1010000 0.112 0.223 3.334
```

StreamDeviceでは扱いにくい例(2)

- データレンジが切り替わると、単位と係数が変わる。 (Fluke 481 Radiation meter)
  データレンジが自動的に切り替わり、その際に係数と単位が変化する。
  また、1行に時系列にデータを出力し、1行分のデータ個数も変化する。

```
1uSv/h
10.11 10.09 10.08
10uSv/h
  0.21   2.51  20.74 106.61 367.88 556.41 676.06 784.12 567.55 459.65
976.62 678.01 995.23
1mSv/h
1.15 1.28
```

- 1回で取得できるデータの数が多い。 (Yokogawa MV2000等)
  1回で取得できるデータ数が、100個以上ある。

StreamDeviceで対応できない場合には、
AsynPortDriverを使ったり、
独自のDeviceSupportを作成したりする。

参考資料

- StreamDevice：
  http://epics.web.psi.ch/software/streamdevice/doc/index.html
- asynDriver：
  https://epics.anl.gov/modules/soft/asyn/
- EPICS Users JP：
  http://cerldev.kek.jp/trac/EpicsUsersJP/wiki/epics/streamdevice
- How to use StreamDevice and ASYN to create EPICS device support for a simple serial, GPIB, or network attached device：
  https://epics.anl.gov/modules/soft/asyn/R4-23/HowToDoSerial/HowToDoSerial_StreamDevice.html
- EPICS on Raspberry-pi：
  http://www.rri.kyoto-u.ac.jp/EPICS/materials/0514-RaspEPICS_training_kumatori.pdf
- EPICS serial communication with Arduino:
  https://www.smolloy.com/2015/12/epics-serial-communication-with-arduino/