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