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