StreamDevice: Protocol Files

1. General Information

A protocol file describes the communication with one device type. It contains *protocols* for each function of the device type and *variables* which affect how the *commands* in a protocol work. It does not contain information about the individual device or the used communication bus.

Each device type should have its own protocol file. I suggest to choose a file name that contains the name of the device type. Don't use spaces in the file name and keep it short. The file will be referenced by its name in the `INP` or `OUT` link of the records which use it. The protocol file must be stored in one of the directories listed in the environment variable `STREAM_PROTOCOL_PATH` (see chapter Setup).

The protocol file is a plain text file. Everything not enclosed in quotes (single `'` or double `"`) is not case sensitive. This includes the names of commands, protocols and variables. There may be any amount of whitespaces (space, tab, newline, ...) or comments between names, quoted strings and special characters, such as `={};`. A comment is everything starting from an unquoted # until the end of the line.

Example Protocol File:

```
# 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"
}
```

2. Protocols

For each function of the device type, define one protocol. A protocol consists of a name followed by a body in braces `{}`. The name must be unique within the protocol file. It is used to reference the protocol in the `INP` or `OUT` link of the record, thus keep it short. It should describe the function of the protocol. It must not contain spaces or any of the characters `,;={}()$'"\#`.

The protocol body contains a sequence of commands and optionally variable assignments separated by `;`.

Referencing other protocols

To save some typing, a previously defined protocol can be called inside another protocol like a command without parameters. The protocol name is replaced by the commands in the referenced protocol. However, this does not include any variable assignments or exception handlers from the referenced protocol. See the `@init` handlers in the above example.

Limitations

The *StreamDevice* protocol is not a programming language. It has neither loops nor conditionals (in this version of *StreamDevice*). However, if an error occurs, e.g. a timeout or a mismatch in input parsing, an exception handler can be called to clean up.

3. Commands

Seven different commands can be used in a protocol: `out`, `in`, `wait`, `event`, `exec`, `disconnect`, and `connect`. Most protocols will consist only of a single `out` command to write some value, or an `out` command followed by an `in` command to read a value. But there can be any number of commands in a protocol.

`out` *`string`*`;`
: Write output to the device. The argument string may contain format converters which are replaced by the formatted value of the record before sending.

`in` *`string`*`;`
: Read and parse input from the device. The argument string may contain format converters which specify how to interpret data to be put into the record. Input must match the argument string. Any input from the device should be consumed with an `in` command. If a device, for example, acknowledges a setting, use an `in` command to check the acknowledge, even though it contains no user data.

`wait` *`milliseconds`*`;`
: Just wait for some milliseconds. Depending on the resolution of the timer system, the actual delay can be slightly longer than specified.

`event(`*`eventcode`*`)` *`milliseconds`*`;`
: Wait for event *`eventcode`* with some timeout. What an event actually means depends on the used bus. Some buses do not support events at all, some provide many different events. If the bus supports only one event, `(`*`eventcode`*`)` is dispensable.

`exec` *`string`*`;`
: The argument string is passed to the IOC shell as a command to execute.

`disconnect;`
: Disconnect from the hardware. This is probably not supported by all busses. Any `in` or `out` command will automatically reconnect. Only records reading in "I/O Intr" mode will not cause a reconnect.

`connect` *`milliseconds`*`;`
: Expliciteely connect to the hardware with *`milliseconds`* timeout. Since connection is handled automatically, this command is normally not needed. It may be useful after a `disconnect`.

4. Strings

In a *StreamDevice* protocol file, strings can be written as quoted literals (single quotes or double quotes), as a sequence of bytes values, or as a combination of both.

Examples for quoted literals are:

```
"That's a string."
'Say "Hello"'
```

There is no difference between double quoted and single quoted literals, it just makes it easier to use quotes of the other type in a string. To break long strings into multiple lines of the protocol file, close the quotes before the line break and reopen them in the next line. Don't use a line break inside quotes.

As arguments of `out` or `in` commands, string literals can contain format converters. A format converter starts with `%` and works similar to formats in the C functions *printf()* and *scanf()*.

StreamDevice uses the backslash character `\` to define some escape sequences in quoted string literals:
`\"`, `\'`, `\%`, and `\\` mean literal `"`, `'`, `%`, and `\`.
`\a` means *alarm bell* (ASCII code 7).
`\b` means *backspace* (ASCII code 8).
`\t` means *tab* (ASCII code 9).
`\n` means *new line* (ASCII code 10).
`\r` means *carriage return* (ASCII code 13).
`\e` means *escape* (ASCII code 27).
`\x` followed by up to two hexadecimal digits means a byte with that hex value.
`\0` followed by up to three octal digits means a byte with that octal value.
`\1` to `\9` followed by up to two more decimal digits means a byte with that decimal value.
`\?` in the argument string of an `in` command matches any input byte
`\$` followed by the name of a protocol varible is replaced by the contents of that variable.

For non-printable characters, it is often easier to write sequences of byte values instead of escaped quoted string literals. A byte is written as an unquoted decimal, hexadecimal, or octal number in the range of -128 to 255 (-0x80 to 0xff, -0200 to 0377). *StreamDevice* also defines some symbolic names for frequently used byte codes as aliases for the numeric byte value:
`EOT` means *end of transmission* (ASCII code 4).
`ACK` means *acknowledge* (ASCII code 6).
`BEL` means *bell* (ASCII code 7).
`BS` means *backspace* (ASCII code 8).
`HT` or `TAB` mean *horizontal tabulator* (ASCII code 9).
`LF` or `NL` mean *line feed* / *new line* (ASCII code 10).
`CR` means *carriage return* (ASCII code 13).
`ESC` means *escape* (ASCII code 27).
`DEL` means *delete* (ASCII code 127).
`SKIP` in the argument string of an `in` command matches any input byte.

A single string can be built from several quoted literals and byte values by writing them separated by whitespaces or comma.

Example:

The following lines represent the same string:

```
"Hello world\r\n"
'Hello',0x20,"world",CR,LF
72 101 108 108 111 32 119 111 114 108 100 13 10
```

5. Protocol Variables

StreamDevice uses three types of variables in a protocol file. *System variables* influence the behavior of `in` and `out` commands. *Protocol arguments* work like function arguments and can be specified in the `INP` or `OUT` link of the record. *User variables* can be defined and used in the protocol as abbreviations for often used values.

System and user variables can be set in the global context of the protocol file or locally inside protocols. When set globally, a variable keeps its value until overwritten. When set locally, a variable is valid inside the protocol only. To set a variable use the syntax:

`variable = value;`

Set variables can be referenced outside of quoted strings by `$variable` or `${variable}` and inside quoted strings by `\$variable` or `\${variable}`. The reference will be replaced by the value of the variable at this point.

System variables

This is a list of system variables, their default settings and what they influence.

`LockTimeout = 5000;`
: Integer. Affects first `out` command in a protocol.
If other records currently use the device, how many milliseconds to wait for exclusive access to the device before giving up?

`WriteTimeout = 100;`
: Integer. Affects `out` commands.
If we have access to the device but output cannot be written immediately, how many milliseconds to wait before giving up?

`ReplyTimeout = 1000;`
: Integer. Affects `in` commands.
Different devices need different times to calculate a reply and start sending it. How many milliseconds to wait for the first byte of the input from the device? Since several other records may be waiting to access the device during this time, `LockTimeout` should be larger than `ReplyTimeout`.

`ReadTimeout = 100;`
: Integer. Affects `in` commands.
The device may send input in pieces (e.g. bytes). When it stops sending, how many milliseconds to wait for more input bytes before giving up? If `InTerminator = ""`, a read timeout is not an error but a valid input termination.

`PollPeriod = $ReplyTimeout;`
: Integer. Affects first `in` command in `I/O Intr` mode (see chapter Record Processing).
In that mode, some buses require periodic polling to get asynchronous input if no other record executes an `in` command at the moment. How many milliseconds to wait after last poll or last received input before polling again? If not set the same value as for `ReplyTimeout` is used.

`Terminator`
: String. Affects `out` and `in` commands.
Most devices send and expect terminators after each message, e.g. `CR LF`. The value of the `Terminator` variable is automatically appended to any output. It is also used to find the end of input. It is removed before the input is passed to the `in` command. If no `Terminator` or `InTerminator` is defined, the underlying driver may use its own terminator settings. For example, *asynDriver* defines its own terminator settings.

`OutTerminator = $Terminator;`
: String. Affects `out` commands.
If a device has different terminators for input and output, use this for the output terminator.

`InTerminator = $Terminator;`
: String. Affects `in` commands.
If a device has different terminators for input and output, use this for the input terminator. If no `Terminator` or `InTerminator` is defined, the underlying driver may use its own terminator settings.
If `InTerminator = ""`, a read timeout is not an error but a valid input termination.

`MaxInput = 0;`
: Integer. Affects `in` commands.
Some devices don't send terminators but always send a fixed message size. How many bytes to read before terminating input even without input terminator or read timeout? The value `0` means "infinite".

`Separator = "";`
: String. Affects `out` and `in` commands.
When formatting or parsing array values in a format converter (see formats and waveform record), what

string to write or to expect between values? If the first character of the `Separator` is a space, it matches any number of any whitespace characters in an `in` command.

`ExtraInput = Error;`
: `Error` or `Ignore`. Affects `in` commands.
Normally, when input parsing has completed, any bytes left in the input are treated as parse error. If extra input bytes should be ignored, set `ExtraInput = Ignore;`

Protocol arguments

Sometimes, protocols differ only very little. In that case it can be convenient to write only one protocol and use *protocol arguments* for the difference. For example a motor controller for the 3 axes X, Y, Z requires three protocols to set a position.

```
moveX { out "X GOTO %d"; }
moveY { out "Y GOTO %d"; }
moveZ { out "Z GOTO %d"; }
```

It also needs three versions of any other protocol. That means basically writing everything three times. To make this easier, *protocol arguments* can be used:

```
move { out "\$1 GOTO %d"; }
```

Now, the protocol can be references in the `OUT` link of three different records as `move(X)`, `move(Y)` and `move(Z)`. Up to 9 parameters, referenced as `$1` ... `$9` can be specified in parentheses, separated by comma. The variable `$0` is replaced by the name of the protocol.

User variables

User defined variables are just a means to save some typing. Once set, a user variable can be referenced later in the protocol.

```
f = "FREQ";     # sets f to "FREQ" (including the quotes)
f1 = $f " %f";  # sets f1 to "FREQ %f"

getFrequency {
    out $f "?"; # same as: out "FREQ?";
    in $f1;     # same as: in "FREQ %f";
}

setFrequency {
    out $f1;    # same as: out "FREQ %f";
}
```

6. Exception Handlers

When an error happens, an exception handler may be called. Exception handlers are a kind of sub-protocols in a protocol. They consist of the same set of commands and are intended to reset the device or to finish the protocol cleanly in case of communication problems. Like variables, exception handlers can be defined globally or locally. Globally defined handlers are used for all following protocols unless overwritten by a local handler. There is a fixed set of exception handler names starting with `@`.

`@mismatch`
: Called when input does not match in an `in` command.
It means that the device has sent something else than what the protocol expected. If the handler starts

with an `in` command, then this command reparses the old input from the unsuccessful `in`. Error messages from the unsuccessful `in` are suppressed. Nevertheless, the record will end up in `INVALID/CALC` state (see chapter Record Processing).

`@writetimeout`
: Called when a write timeout occurred in an `out` command.
It means that output cannot be written to the device. Note that `out` commands in the handler are also likely to fail in this case.

`@replytimeout`
: Called when a reply timeout occurred in an `in` command.
It means that the device does not send any data. Note that `in` commands in the handler are also likely to fail in this case.

`@readtimeout`
: Called when a read timeout occurred in an `in` command.
It means that the device stopped sending data unexpectedly after sending at least one byte.

`@init`
: Not really an exception but formally specified in the same syntax. This handler is called from `iocInit` during record initialization. It can be used to initialize an output record with a value read from the device. Also see chapter Record Processing.

Example:

```
setPosition {
    out "POS %f";
    @init { out "POS?"; in "POS %f"; }
}
```

After executing the exception handler, the protocol terminates. If any exception occurs within an exception handler, no other handler is called but the protocol terminates immediately. An exception handler uses all system variable settings from the protocol in which the exception occurred.

Next: Format Converters

Dirk Zimoch, 2006