Sparky Script / CANOpener

This documents defines a specification for a scripting language that is meant to serve as an easy to use and easy to implement general interface between humans and certain kinds of networks like CANOpen, I2C, SPI, etc. It is written specifically with CANOpen in mind, but could in theory be adapted to work with any network that implements a master/slave protocol.

This should be treated as more of a thought experiment than anything formal. It may or may not prove to be a good idea, but is potentially worth exploring. Ideas and feedback would be welcome, especially with regards to the name. The two leading ideas as of now are a pun based on the name of the network, and an homage to the Lafayete FSAE car, named Sparky by the 2020 team

Motivation

The CANOpen network on the Lafayette FSAE car is very powerful, but also very difficult to use. The Motor Controller, for example, has a CANOpen interface that exposes hundreds of registers, each of which represents a different sensor or control interface, and performing basic procedures like booting into a specific mode, setting the throttle, and collecting sensor data often require complex sequences of read and write commands across several distinct registers.

To assist with this, the Motor Controller (EmDrive 500) ships with a proprietary program for performing these procedures over the CAN bus. It is a GUI based program that runs on windows and interfaces with the CAN bus through a USB to CAN adapter. Currently, engineers perform motor tests and configuration by setting up a PC and stool just outside of the dyno room, running a CAN cable from the PC into the test setup, and using this program to send the necessary commands.

While this method works for doing basic operations, it is less than ideal. Some of the drawbacks include:

- The provided program can only be run on windows, meaning it requires a large external PC to be physically wired to the car. This is a mild inconvenience during dyno testing, but could prove to be a constant source of problems and annoyance during any kind of dynamic testing or competition. It would be much more natural for an equivilant piece of software to run on the integrated SCADA computer.
- The provided program only supports the motor controller out of the box. As more and more CANOpen compliant nodes get added to the car, similar configuration and testing software will have to be written for each of them.
- The program is difficult to learn. It consists of a large number of menus and windows which can be daunting to new users.
- There is currently no good, well defined way for recording sequences of commands, sharing them with other engineers, and playing them back at arbitrary times.
- No support for complicated logic or interaction with external systems.

It is my opinion that the solution to all of these problems can be found in the form of a scripting language, whose job it would be to map a human readable syntax into CANOpen read and write commands, send them over the network, and record their responses.

A general purpose scripting language like python or javascript could accomplish this, but being general purpose, the syntax for doing highly specific tasks like this tends to be highly verbose, and include details which are unimportant to the general user.

Instead, a new scripting language is proposed, which is highly specific to the task of sending read and write commands over a Master/Slave network. It aims to be simple enough that basic tasks can be done without much effort or thought, but also flexible enough to allow for very complex behavior.

Syntax

Each CANOpener script is conceptually just a list of instructions. Every instruction takes up exactly one line, it begins with a keyword representing the instruction name and is followed by a series of 0 or more arguments, separated by spaces.

CANOpener supports two kinds of data storage, which are referred to as registers and variables.

Registers always begin with an `@` symbol, and store addresses for certain registers on the network. In the case of CANOpen, this consists of a combination of node id, index, and subindex, but would be different for different networks.

Variables always begin wint a `$` symbol, and represent data stored locally on the master node. Like variables in a traditional scripting language, they can store any kind of data and can be manipulated arbitrarily.

- **read and write**

```
read @<register> $<variable | empty>
write @<register> $<variable | literal>
```

The bread and butter of CANOpener scripts are the read and write commands. Each one takes two arguments, a register and a variable. Data is either copied from the register into the variable (read) or from the variable into the register (write)

In the case of read, the variable argument can be omitted, and the data from the register will instead be piped into stdout. In the case of write, the variable can be replaced with a literal number.

- **define**

```
define @<register> <address>
```

Registers are defined with the define keyword, they accept a register name and an address literal. A sequence of define commands is meant to mimic the purpose of an electronic data sheet, with the idea that importing an EDS be an alternative to defining registers manually.

- **procedure**

```
procedure <name> begin
...
...
end
```

Sequences of read and write commands can be grouped together and named with the procedure block, which consists of a procedure begin instruction and an end instruction. Procedures are named procedures to distinguish them from functions, which they superficially resemble, but are distinct from. The accept no arguments and return no value. Instead, procedures interact with the rest of the program by reading and writing to variables, which are all global in scope.

- **run, runif, repeat**

```
run <procedure>
runif $<variable> <procedure>
repeat <number | inf> <procedure>
```

Procedures can be run with the `run` instruction. They also support some basic flow control with the `runif` command, which will run the procedure only if the given variable is nonzero, and the `repeat` instruction, which runs the procedure a set number of times.

- **using and call**

```
using <function1>, <function1>
call <function> (<arguments>) -> (<variables>)
```

CANOpener discourages using its syntax to write complicated logic. Instead, it recommends exporting this behavior to other, more sophisticated programs via the `call` instruction, which accepts a function name, a list of arguments, (either variables or literals) and a list of variables to write the result to.

The list of supported functions is defined by the parser. The script tells the parser which functions it intends to use with the `using` instruction. These can be almost anything, good examples include stuff like:

- sleep statements
- prompting the user for input
- reading data from a file
- writing csv data
- sending alerts to the user
- assert statements (useful during tests)

These functions could easily be defined by the user and added to the parser as plugins.

- **other**

```
# this is a comment
```

Comments are any line begining with the `#` character

```
include <filename>
```

include statements copy all contents of the given file into the current script.

Example: Soft Throttle

```
# continually polls an external process for input,
# then writes that input to the Motor Controller
# throttle register

using input

define @throttle_address 01:0x3A:00

procedure main begin
    call input () -> ($throttle_value)
    write @throttle_address $throttle_value
end

repeat inf main
```

Example: Motor Efficiency Experiment

```
from csv use record, iterate

define @throttle 01:0x2A:00

# other addresses omitted so I didn't have to make any more up

define @current ...
define @voltage ...
define @torque ...
define @rpm ...

procedure sleep_and_read_values begin
    call sleep(1)

    read @current $current
    read @voltage $voltage
    read @torque $torque
    read @rpm $rpm

    call record_csv ($throttle, $current, $voltage, $torque, $rpm)
end
```

```
procedure main begin
    call next_value ('throttles') -> ($throttle, $has_next)

    call print ('setting throttle to ')
    call print ($throttle)
    call print ('\n')

    write @throttle $throttle
    repeat 10 sleep_and_read_values
    runif $has_next main
end

run main
```