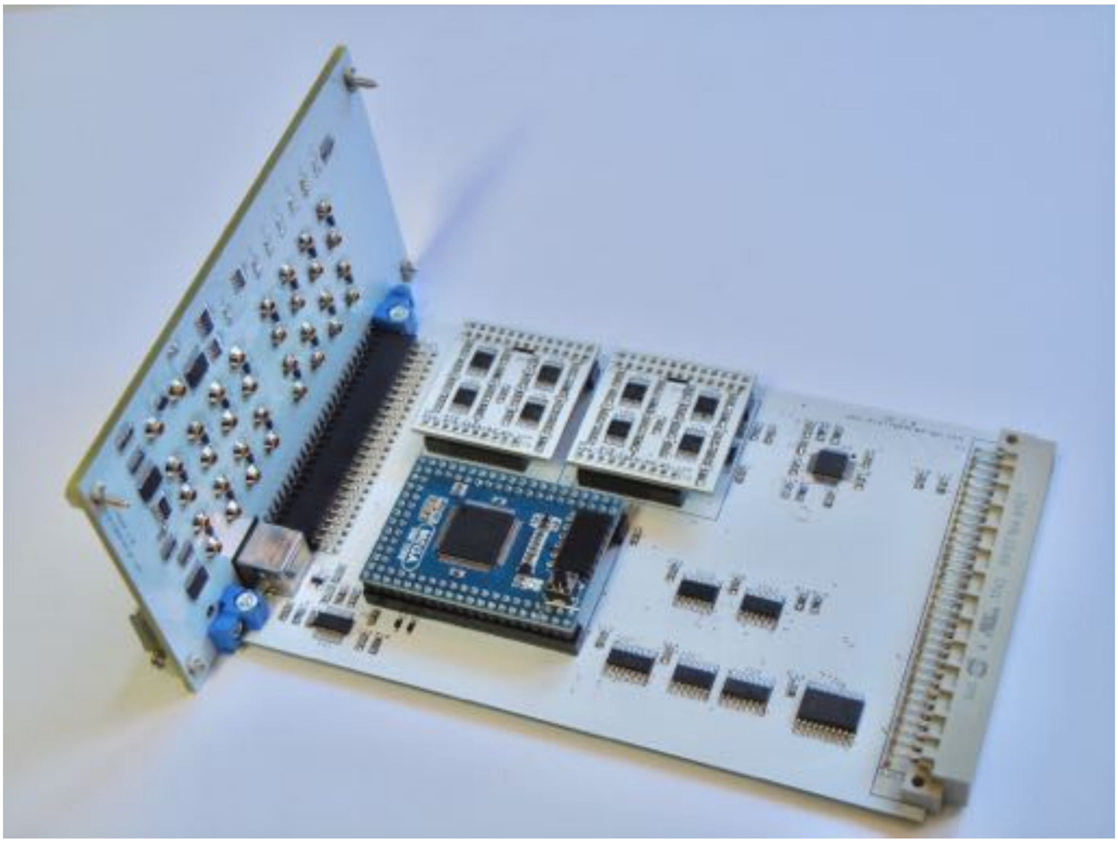
Hybrid Controller[1]

User manual



[1]The author would like to the Mrs. Rikka Mitsam for proofreading and numerous corrections and improvements of the text.

Prof. Dr. Bernd Ulmann, 01.09.2019, Version 0.1

1 Introduction

The *hybrid controller* described in the following allows a digital *host*computer to take total control of an Analog Paradigm Model-1 analog computer by means of a USB interface. The controller itself is based on an AVR processor and not only allows full mode control (initial condition, operate, halt) of the analog computer but also contains eight digitally controlled potentiometers with a 10 bit resolution, and makes it possible to address and read out every element of the analog computer in any mode of operation.

Figure 1.1 shows the front panel of the hybrid controller. The sixteen jacks grouped together in the upper half are the inputs and outputs of eight digitally controller potentiometers, while the sixteen jacks in the lower half are eight digital inputs and eight digital outputs. These digital I/O lines are typically connected to the outputs of comparators or to the inputs of the electronic switches of a comparator module.

The USB connector is visible on the lower far left, while the two connectors on the lower right can be used to apply an external halt signal (typically derived from a comparator output in an analog computer setup) and as a trigger output which can be used to trigger the x-deflection of an oscilloscope etc. Furthermore, there are seven LEDs on the right hand side of the module which display the current mode of operation (`INITIAL`, `OPERATE`, `HALT`, `POTSET`) as well as an `OVERLOAD` condition or any other `ERROR` which typically results from a communication problem with the host computer. The LED labeled `USB` is lit green when the USB port is connected to a host computer and flickers during an ongoing communication.

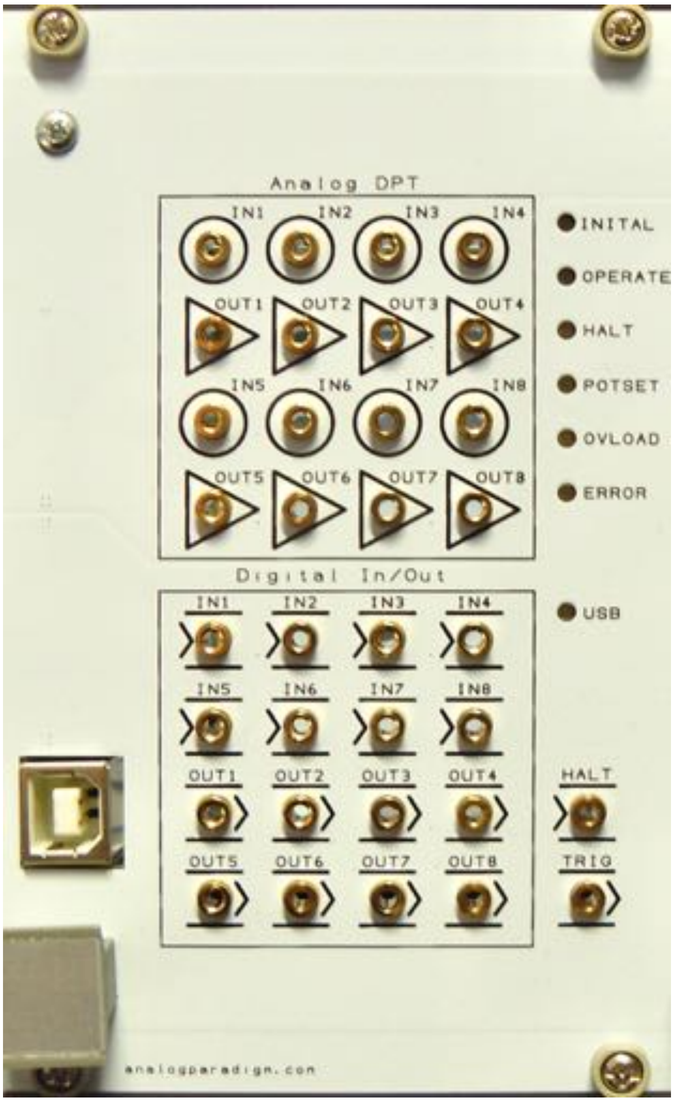


Figure 1.1: Front panel of the hybrid controller

2 Direct interaction

The simplest mode of operation is direct interaction with the hybrid controller by means of a suitable terminal emulation such as the *Serial Monitor* which is part of the *Arduino IDE*.[1] The communication speed of the hybrid controller is set to 250000 Baud, and the interface name depends on the host computer as well as on the particular hybrid controller. A typical device name under LINUX/Mac OS C looks like `/dev/cu.usbserial-DN050L2C`.

The commands shown in table 2.1 are sent to the hybrid controller after entering them, followed by a return-key-press if the serial monitor of the Arduino IDE is used. Please note that this line terminator (actually a carriage return or the like, depending on the host operating system) is not transmitted to the hybrid controller. It merely denotes the end of an input to the serial monitor. Entering the command `?`, followed by pressing the return-key yields a short help text.

2.1 A simple example

To perform a computation run of the analog computer under manual control, just enter `i` (followed by pressing the return-key, as always) to set all integrators (as long as these are not externally controlled) to their respective initial conditions. This is then followed by the command `o` to enter the operate mode. This mode can then be left by either going back to initial condition or by entering `h` to set the analog computer to halt mode.

Like the traditional control unit CU, the hybrid controller also allows repetitive or single operation of the analog computer with configurable times for the initial condition and operate modes. Assume that a given computer setup yields one solution of a simulation in 10 milliseconds and requires a further

[1] This can be downloaded from `https://www.arduino.cc/en/Main/Software`.

a	Disable halt-on-overflow
A	Enable halt-on-overflow
b	Disable external halt
B	Enable external halt
c\d{6}	Set OP time for repetitive/single operation
C\d{6}	Set IC time for repetitive/single operation
d[0-7]	Clear digital output
D[0-7]	Set digital output
e	Start repetitive operation
E	Start single IC/OP-cycle
F	Start single IC/OP-cycle with completion message (for sync operation)
g\x{4}	Set address of computing element and return its ID and value
h	Halt
i	Initial condition
m\x{4}	Set the address of the HC module, default is 0x0090
o	Operate
P\d\d{4}	Set the builtin potentiometer to value \d{4}
q	Dump digital potentiometer settings
R	Read digital inputs
s	Print status
S	Switch to PotSet-mode
t	Print elapsed OP-time
x	Reset
?	Print Help

Table 2.1: Commands supported by the hybrid controller (data formats are specified by regular expressions, \d{4} denoting four decimal digits, \x{4} representing four hexadecimal digits etc.)

10 ms for the initial condition phase. To get a more or less flicker free display on an oscilloscope, the computer should be operated in repetitive mode which can be accomplished by the following sequence of commands (comments are in red):

```
C000010 Set the initial condition time to 10 ms
c000010 Set the operate time to 10 ms
e       Start repetitive operation
```

These commands are echoed by the hybrid controller with `T_IC=10`, `T_OP=10`, and `REP-MODE` respectively (these replies are evaluated by the Perl library described in the following but are easily readable for a human operator as well). To end the repetitive operation, either `i` or `h` can be sent to the hybrid controller to set the analog computer to initial condition or to halt.

It is important that the times specified above are given in microseconds in a strict six-digit format! Entering `C10` would result in an error! The same holds true for other commands expecting parameters.

If the example shown above would have contained the command `A` prior to the start of the repetitive operation (`e`), any overload condition occurring during one of the operate cycles would cause the repetitive operation to stop immediately. In this case, the analog computer is placed into halt mode so that the computing element being overloaded can be easily identified.

2.2 Controlling potentiometers

As already mentioned, the hybrid controller contains eight digitally controlled coefficient potentiometers with a resolution of 10 bits each. These are controlled by the `P`-command which expects the number of the potentiometer to set (0 to 7) immediately followed by a four digit decimal value in the range of 0 to $2^{10} - 1 = 1023$. To set potentiometer 0 to its mid-scale value, the command `P00511` must be issued.

At power-on, the hybrid controller performs an intialization routine that sets all potentiometers to the default value 0. It should be noted that the hybrid controller must be the last module on the first backplane of the analog computer, so it will typically sit right next to the power supply which occupies the rightmost position in the main chassis. This is necessary to ensure that the hybrid controller is at its default address on the bus which is required to set the potentiometers.[2]

2.3 Read out operation

The hybrid controller can be used to read out the value of every computing element of the analog computer with 16 bits of precision. All values are refered to the machine units of ± 10 V, so a voltage

[2] If the hybrid controller is located at another slot, the `m`-command can be used to set a different address for it, but this is typically not recommended for normal operation.

Module	ID
PS	0
SUM8	1
INT4	2
PT8	3
CU	4
MLT8	5
MDS2	6
CMP4	7
HC	8

Table 2.2: Module types

of $+5$ V would be displayed as the value 0.5000. To read out the value of the first (subaddress 0) element of the sixth (element address 5) module in the first chassis (chassis address 0) of the first rack (rack address 0), the command $g0050$ would have to be issued. The first hex-nibble contains the rack address, the second one the chassis address, the third one the slot address, and the last one the address of the computing element on that particular module.

The output of the g-command consists of the ASCII-representation of a floating point value, followed by a number denoting the type of the element read out. Assuming that the element on address 0050 is a summer, having an output value of 5 V, the command above would yields the output `0.5000 1`. Table 2.2 shows all currently supported module types.

3 HyCon.pm

As useful and simple the manual operation described before is, a real hybrid computer setup requires the digital computer to fully take control of the analog computer by means of some control program. This is done by means of a Perl library called `HyCon.pm` which implements a hybrid controller class as described in the following.

3.1 Configuration files

Every program using this library requires a configuration file in YAML-format that has the same name as the Perl program but with extension `.yml` instead of `.pl`. A typical, yet simple configuration file is shown in figure 3.1.

The first section contains the communications parameters of which only the name of the device (port) should be changed as it depends on the actual setup of the host computer.

The next section, `builtin_dpt`, specifies default values for the builtin digitally controlled potentiometers. If it is missing, all eight potentiometers are initialized to 0, otherwise the eight values in the comma separated list following `values:` will be used for their initialization.

The `types`-section maps the numeric id values returned from the various computing elements upon readout to clear text descriptions.

The section labeled `manual_potentiometers` lists all manual potentiometers used in a computer setup. This is useful as the values of all these potentiometers can be read out at once using the `read_mpts()`-method. This functionality is typically used when some simulation required manual changes to the coefficient potentiometers of the analog computers. These values can then be read out by the host computer to persist them for later analysis or the like. Please note that all elements

```
serial:
    port: /dev/cu.usbserial-DN050L2C
    bits: 8
    baud: 250000
    parity: none
    stopbits: 1
    poll_interval: 1000
    poll_attempts: 200
builtin_dpt:
    values: .1, .2, .3, .4, .5, .6, .7, .8
types:
    0: PS
    1: SUM8
    2: INT4
    3: PT8
    4: CU
    5: MLT8
    6: MDS2
    7: CMP4
    8: HC
manual_potentiometers:
    PT_8-0, PT_8-1, PT_8-2, PT_8-3: 0x0223
elements:
    MUP:    0x0000
    MUN:    0x0001
    PT_8-0: 0x0220
    PT_8-1: 0x0221
    PT_8-2: 0x0222
    PT_8-3: 0x0223
    SUM8-0: 0x0050
    SUM8-1: 0x0051
    SUM8-2: 0x0052
    SUM8-3: 0x0053
    SUM8-4: 0x0054
    SUM8-5: 0x0055
    SUM8-6: 0x0056
    SUM8-7: 0x0057
```

Figure 3.1: Example configuration file for a hybrid control program

```
1  use strict;
2  use warnings;
3
4  use lib '../..';     # Path the HyCon.pm
5  use File::Basename;
6  use HyCon;
7
8  (my $config_filename = basename($0)) =~ s/\.pl$//;
9  my $ac = HyCon->new("$config_filename.yml");     # Create object
10
11 $ac->set_ic_time(500);         # Set IC-time to 500 ms
12 $ac->set_op_time(1000);        # Set OP-Time to 1000 ms
13 $ac->single_run();             # Perform a single computation run
14
15 # Read a value from a computing element addressed via the central bus:
16 my $element_name = 'SUM8-0';
17 my $value = $ac->read_element($element_name);
18 print "Value = $value->{value}, Type = $value->{id}\n";
```

Figure 3.2: Simple test program

of the comma separated list following the `manual_potentiometers` entry must be defined in the `elements`-section of the configuration file!

The last and typically largest section is labeled `elements` and contains all available computing elements (ideally only those which are actually used in some particular computer setup). The names defined here are arbitrary and will be typically not be `SUM8-0` but something like `VELOCITY` to enhance readability of the control program.

3.2 A simple test program

Figure 3.2 shows a very simple test program which uses the aforementioned configuration file. Line 4 is only necessary if the `HyCon.pm` module is not installed in a standard location contained in `@INC` of the Perl interpreter. In line 8, the name of the configuration file is derived from the current program's name which is then used to instantiate a `HyCon`-object `$ac`. This object, which is actually a singleton, is used in the following to control the analog computer.

In this case, a single computation with an initial condition time of 500 ms and an operation time of 1 second is to be performed. These times are set in lines 11 and 12, followed by the invocation of the `single_run()`-method. After completion of this computer run, the analog computer is automatically set to halt mode, so that results of the simulation may be read out. In this case, the value of the element named `SUM8-0` as specified in the `elements`-section of the corresponding configuration file is read out in line 17. Each readout operation yields a reference to a hash which contains a `value` and an `id`, each of which contains the actual numeric value and the element's identification code.

4 Examples

4.1 Trajectory optimization

The following example is an extremely simple trajectory optimization problem. Here, the trajectory of an idealized shell experiencing neither drag nor any other influences is to be parameterized by varying its initial velocity v_0 so that it hits a defined target position. The elevation angle of the cannon is fixed.

The x- and y-components of the velocity of the shell are thus

$$\dot{x} = v_0 \sin(\alpha) \text{ and}$$
$$\dot{y} = \cos(\alpha) - gt.$$

with g and t denoting the gravitational acceleration and time. These two variables readily yield the x- and y-components of the shell's position by integration. The resulting setup of the analog computer is pretty straightforward and shown in figures 4.1 and 4.2. Here, the coefficient potentiometer labeled `DPT0` denotes the first of the eight digitally controlled potentiometers of the hybrid controller.

The time-scale factors of all three integrators are set to $k_0 = 10^3$. The outputs of this circuit are as follows:

x **and** y**:** Position of the shell – these two outputs can be used to control an oscilloscope set to xy-mode.

Δx**:** This is the x distance between shell and target and is used in the digital portion of the hybrid computer program to change the initial velocity v_0 of the shell in order to minimize the target miss distance.

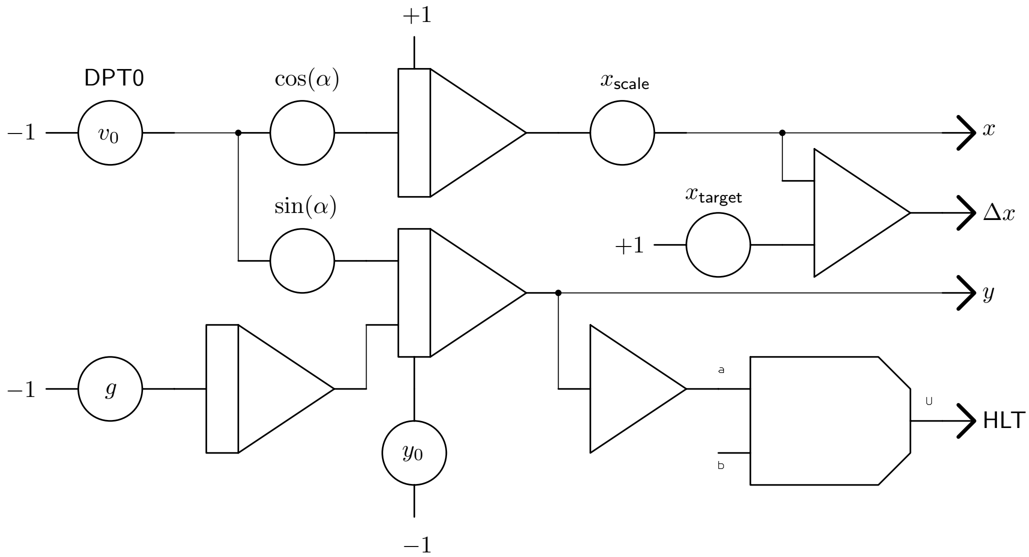


Figure 4.1: Setup of the analog computer for the basic trajectory problem

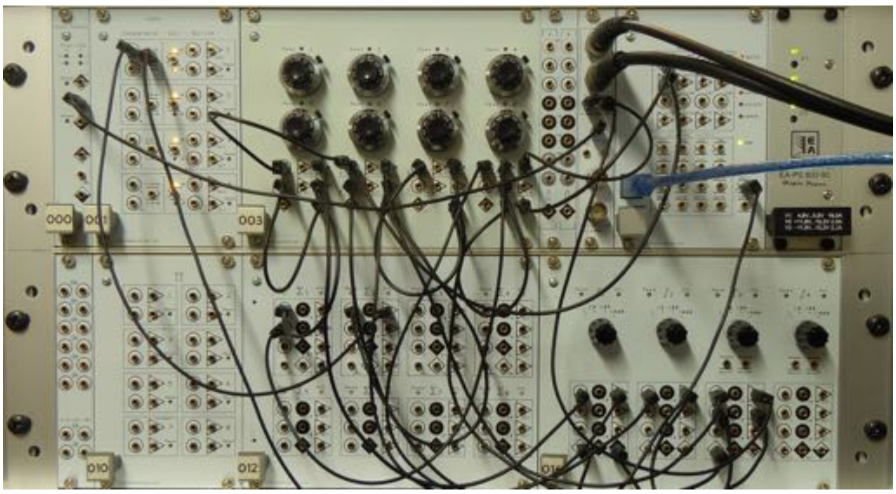


Figure 4.2: Analog portion of the trajectory optimization program

HLT: This logical output from a comparator is used to trigger the external halt input of the hybrid controller when the shell hits ground level. Therefore it is necessary that the height of the cannon satisfies $y_0 > 0$. Otherwise the comparator would trigger halt before the actual flight of the shell begins.

The digital portion of the hybrid computer setup consists of the YAML configuration file shown in figure 4.3 and its accompanying Perl program shown in figure 4.4. The sections `serial` and `builtin_dpt` are standard. The `manual_potentiometers`- and `elements`-sections only list those computing elements which are used in this setup. Note that all elements have been given symbolic names like `delta_x` instead of `S0120`, `SUM8-0` etc.

The digital computer program as shown in figure 4.4 is pretty straight-forward: Since the `HyCon.pm` module was not installed in a standard location for this demonstration, the Perl interpreter has to be notified to include an additional directory into its `@INC`-variable (line 4). This program displays a status line in the first line of the screen which is getting updated during the simulation run. Therefore, the `Term::ANSIScreen` package is used which implements (among many other useful things) a clear screen routine. Setting the special variable `$|` to 1 disables output buffering of stdout which allows us to update this status line periodically. All of this is done in lines 6 and 9–11.

The analog computer is setup in lines 16–19. Changing the address of the hybrid controller to hexadecimal 0080 is typically not necessary and results from the fact that this example has been implemented on a very early prototype of the Analog Paradigm Model-1 analog computer which has some quirks regarding the addressing scheme. In line 17 the external halt input of the hybrid controller is enabled. The idea is that a simulation runs as long as the shell requires to hit "ground" which is detected by a comparator which is connected to the `HALT` input of the controller.

Since $k_0 = 10^3$, an initial condition time of 1 ms as set in line 18 is more than sufficient. The operating time of 1000 ms is unrealistically high but such a large value won't do any harm in this context since one simulation run ends as soon as the comparator detects the shell hitting ground level. In fact, any value for this time frame (line 19) that exceeds the maximum run time of a single run in the worst case would be sufficient.

The idea behind the parameter variation loop spanning lines 22–35 is extremely simple: If the shell's flight path is too short, v_0 is increased, if it is too far, v_0 is decreased accordingly. This change in v_0 is made a bit dynamical by taking the hit miss distance into account. If the distance to the target is less than the value of `$epsilon`, the loop is terminated, and the current potentiometer settings are dumped to the screen.

Figure 4.5 shows a typical output of the digital portion of the hybrid computer setup. It took 41 one trials to determine a suitable v_0 which allows the shell to miss the target by only 0.0008 (arbitrary units). Following this status line the values of all relevant manual potentiometers are shown.

Figure 4.6 shows a long-term exposure photograph of the output on the attached oscilloscope screen. The varying step size by which v_0 is changed is clearly visible. Please note that this hybrid computer setup is neither too realistic nor too astute, it's only purpose is to serve as an introductory

```
serial:
    port: /dev/cu.usbserial-DN050L2C
    bits: 8
    baud: 250000
    parity: none
    stopbits: 1
    poll_interval: 10
    poll_attempts: 20000
builtin_dpt:
    values: 0, 0, 0, 0, 0, 0, 0, 0
types:
    0: PS
    1: SUM8
    2: INT4
    3: PT8
    4: CU
    5: MLT8
    6: MDS2
    7: CMP4
    8: HC
manual_potentiometers: cos_alpha, sin_alpha, PT_y0, PT_x_scale, PT_x_target, g
elements:
    cos_alpha:   0x0030
    sin_alpha:   0x0031
    PT_y0:       0x0032
    PT_x_scale:  0x0033
    PT_x_target: 0x0034
    g:           0x0035

    delta_x: 0x0120
    minus_y: 0x0121

    x:      0x0160
    int_g:  0x0161
    y:      0x0162
```

Figure 4.3: Configuration file for the simple trajectory optimization

```perl
use strict;
use warnings;

use lib '../..'; # Path the HyCon.pm
use File::Basename;
use Term::ANSIScreen;
use HyCon;

$| = 1;
my $console = Term::ANSIScreen->new();
$console->Cls();

(my $config_filename = basename($0)) =~ s/\.pl$//;
my $ac = HyCon->new("$config_filename.yml"); # Create object

$ac->set_address('0080'); # Just for the prototype system
$ac->enable_ext_halt();   # This is essential
$ac->set_ic_time(1);
$ac->set_op_time(1000);    # This is a limit that will never be reached

my ($counter, $v0, $delta_v, $epsilon) = (1, 0, .1, .001);
while (1) {
    $ac->set_pt(0, $v0);

    my $halt = $ac->single_run_sync();
    $halt = 0 unless defined($halt);
    my $delta_x = $ac->read_element('delta_x')->{value};

    printf("H: $halt\tTrial: %06d\tv0 = %+0.4f\tDelta x = %+0.4f\r",
        $counter++, $v0, $delta_x);

    last if abs($delta_x) < $epsilon;

    $v0 += $delta_x * $delta_v;
}
print "\n\n";
my $pot_settings = $ac->read_mpts();
print "$_:\t$pot_settings->{$_}{value}\n" for sort(keys(%$pot_settings));
```

Figure 4.4: Digital portion of the simple trajectory optimization

```
H: 1    Trial: 000034   v0 = +0.4204    Delta x = +0.0007

PT_x_scale: 0.2186
PT_x_target:    -0.2191
PT_y0:  -0.5250
cos_alpha:  -0.2975
g:  -0.1074
sin_alpha:  -0.2322
```

Figure 4.5: Typical output of a simulation run on the digital computer

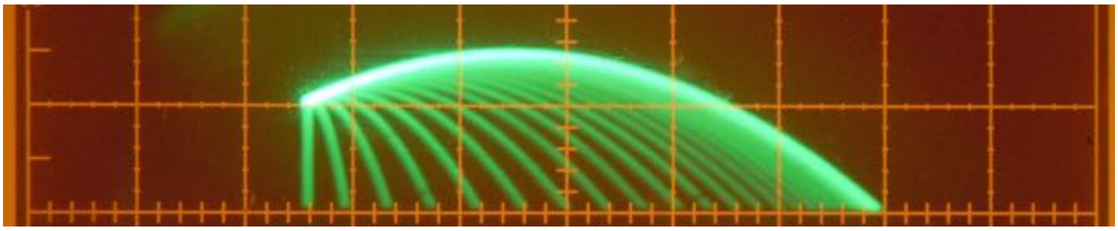


Figure 4.6: Screenshot from the running trajectory optimization program

example to hybrid computer programming. A more sophisticated implementation might take the velocity dependent drag into account.[1] Furthermore, it is not realistic to change the muzzle velocity v_0 of a shell as the only parameter of a simulation. Changing the elevation angle of the cannon, and thus changing the values of $\cos(\alpha)$ and $\sin(\alpha)$ would be more realistic. This is left as an exercise to the reader.[2] :-)

[1]See http://analogparadigm.com/downloads/alpaca_9.pdf for an example of such a computation.
[2]Or to a later application note...

A

HyCon.pm

HyCon.pm

```
#
#  The HyCon-package provides an object oriented interface to the HYCON
# hybrid controller for the Analogparadigm Model-1 analog computer.
#
# 06-AUG-2016   B. Ulmann   Initial version
# 07-AUG-2016   B. Ulmann   Added extensive error checking, changed
#                           c-/C-commands for easier interfacing
# 08-AUG-2016   B. Ulmann   Analog calibration capability added
# 31-AUG-2016   B. Ulmann   Support of digital potentiometers
# 01-SEP-2016   B. Ulmann   Initial potentiometer setting based on
#                           configuration file etc.
# 13-MAY-2017   B. Ulmann   Start adaptation to new, AVR2560-based hybrid
#                           controller with lots of new features
# 16-MAY-2017   B. Ulmann   single_run_sync() implemented
# 08-FEB-2018   B. Ulmann   Changed read_element to expect the name of a
#                           computing element instead of its address
# 01-SEP-2018   B. Ulmann   Adapted to the final implementation of the
#                           hybrid controller (version 0.4)
# 02-SEP-2018   B. Ulmann   Bug fixes, get_response wasn't implemented too
#                           cleverly, it is now much faster than before :-)
#

```

```
package HyCon;

=pod

=head1 NAME

HyCon - Hybrid controller for analog computers

=head1 VERSION

This document refers to version 0.4 of HyCon

=head1 SYNOPSIS

    use strict;
    use warnings;

    use File::Basename;
    use HyCon;

    (my $config_filename = basename($0)) =~ s/\.pl$//;
    print "Create object...\n";
    my $ac = HyCon->new("$config_filename.yml");

    $ac->set_ic_time(500);              # Set IC-time to 500 ms
    $ac->set_op_time(1000);             # Set OP-Time to 1000 ms
    $ac->single_run();                  # Perform a single computation run

    # Read a value from a computing element addressed via the central bus:
    my $element_name = 'SUM8-0';
    my $value = $ac->read_element($element_name);

=head1 DESCRIPTION

This module implements a simple interface to the a hybrid controller which
interfaces an analog computer to a digital computer and thus allows true
hybrid computation.

=cut
```

```
use strict;
use warnings;

our $version = 0.4;

use YAML qw(LoadFile);
use Carp qw(confess cluck);
use Device::SerialPort;
use Time::HiRes qw(usleep);

use Data::Dumper;

use constant {
    DIGITAL_OUTPUT_PORTS   => 8,
    DIGITAL_INPUT_PORTS    => 8,
    BUILTIN_DPT            => 8,
    BUILTIN_DPT_RESOLUTION => 10,
};

my $instance;

=head1 Functions and methods

=head2 new($filename)

This function generates a HyCon-object. Currently there is only one hybrid
controller supported, so this is, in fact, a singleton and every subsequent
invocation will cause a fatal error. This function expects a path to a YAML
configuration file of the following structure:

config.yml:
    serial:
        port: /dev/tty.usbmodem621
        bits: 8
        baud: 250000
        parity: none
        stopbits: 1
        poll_interval: 1000
```

```
        poll_attempts: 200
builtin_dpt:
    values: .1, .2, .3, .4, .5, .6, .7, .8
types:
    0: PS
    1: SUM8
    2: INT4
    3: PT8
    4: CU
    5: MLT8
    6: MDS2
    7: CMP4
    8: HC
elements:
    Y_DDOT: 0x0100
    Y_DOT:  0x0101
    PT_8-0: 0x0220
    PT_8-1: 0x0221
    PT_8-2: 0x0222
    PT_8-3: 0x0223
    PT_8-4: 0x0224
    PT_8-5: 0x0225
    PT_8-6: 0x0226
    PT_8-7: 0x0227
manual_potentiometers:
    PT_8-0, PT_8-1, PT_8-2, PT_8-3, PT_8-4, PT_8-5, PT_8-6, PT_8-7
```

The setup shown above will not fit your particular configuration, it just serves as an example. The remaining parameters nevertheless apply in general and are mostly self-explanatory. poll_interval and poll_attempts control how often this interface will poll the hybrid controller to get a response to a command. The values shown above are overly pessimistic but won't hurt during normal operation.

The section labeled 'builtin_dpt' contains data to setup and control the digital potentiometers of the hybrid controller. The initial values of these potentiometers can be specified by a list containing eight entries following 'values'. If this part is missing, initial values of 0 are assumed. The new() function will set all digitally controlled

```
potentiometers of the hybrid computer according to these data upon
invocation.

If the number of values specified in the array 'values' does not match the
number of configured potentiometers, the function will abort.

The types-section contains the mapping of the devices types as returned by
the analog computer's readout system to their module names.

The elements-section contains a list of computing elements defined by an
arbitrary name and their respective address in the computer system. Calling
read_all_elements() will switch the computer into HALT-mode, read the
values of all elements in this list and return a reference to a hash
containing all values and IDs of the elements read.

Ideally, all manual potentiometers are listed following
manual_potentiometers which is used for automatic readout of the settings
of these potentiometers by calling read_mpts(). This is useful if a
simulation has been parameterized manually and these parameters are then
required for documentation purposes or the like. Caution: All
potentiometers to be read out by read_mpts() must be defined in the
elements-section!

The new() function will clear the communication buffer of the hybrid
controller by reading and discarding and data until a timeout will be
reached. This is currently as long as the product of poll_interval and
poll_attempts.

=cut

sub new {
    my ($class, $config_filename) = @_;

    confess "Only one instance of a HyCon-object at a time is supported!"
        if $instance++;

    my $config = LoadFile($config_filename) or
        confess "Could not read configuration YAML-file: $!";

```

```
    my $port = Device::SerialPort->new($config->{serial}{port}) or
        confess "Unable to open USB-port: $!\n";
    $port->databits($config->{serial}{bits});
    $port->baudrate($config->{serial}{baud});
    $port->parity($config->{serial}{parity});
    $port->stopbits($config->{serial}{stopbits});

    # If no poll-interval is specified, use 1000 microseconds
    $config->{serial}{poll_interval} //= 1000;
    $config->{serial}{poll_attempts} //= 200;   # and 200 such intervals.

    # Get rid of any data which might still be in the serial line buffer
    my $attempt;
    $port->write('x'); # Reset the hybrid controller
    while (++$attempt < $config->{serial}{poll_attempts}) {
        my $data = $port->lookfor();
        last if $data eq 'RESET';
        usleep($config->{serial}{poll_interval});
    }

    #  Create array of initial potentiometer values - if there is nothing
    # specified assume zero:
    my $pv = defined($config->{builtin_dpt}{values})
           ? [ split(/\s*,\s*/, $config->{builtin_dpt}{values}) ]
           : [ map{0}(1 .. BUILTIN_DPT) ];

    # Create the actual object
    my $object = bless(my $self = {
        port => $port,
        poll_interval => $config->{serial}{poll_interval},
        poll_attempts => $config->{serial}{poll_attempts},
        builtin_dpt => {
            number => BUILTIN_DPT,
            resolution => BUILTIN_DPT_RESOLUTION,
            values => $pv,
        },
        elements => $config->{elements},
        types    => $config->{types},
        times    => {
```

```
            ic_time => -1,
            op_time => -1,
        },
        manual_potentiometers =>
            [ split(/\s*,\s*/, $config->{manual_potentiometers}) ],
    }, $class);

    # Initial potentiometer setup
    set_pt($object, $_, $pv->[$_]) for (0 .. BUILTIN_DPT - 1);

    return $object;
}

=head2 get_response()

In some cases, e.g. external HALT conditions, it is necessary to query the
hybrid controller for any messages which may have occured since the last
command. This can be done with this method - it will poll the controller
for a period of poll_interval times poll_attemps microseconds. If this
time-out value is not suitable, a different value in milliseconds can be
supplied as first argument of this method. If this argument is zero or
negative, get_response will wait indefinitely for a response from the
hybrid controller.

=cut

sub get_response {
    my ($self, $timeout) = @_;
    $timeout = $self->{poll_interval} unless defined($timeout);

    my $attempt;
    do {
        my $response = $self->{port}->lookfor();
        return $response if $response;
        # If we poll indefinitely, there is no need to wait at all
        usleep($timeout) if $timeout > 0;
    } while ($timeout < 1 or ++$attempt < $self->{poll_attempts});
}

```

```
=head2 ic()

This method switches the analog computer to IC (initial condition) mode
during which the integrators are (re)set to their respective initial value.
Since this involves charging a capacitor to a given value, this mode should
be activated for the right duration as required by the analog computer being
controlled. Especially on historic machines it is not uncommon to have
IC-durations of up to a second when a given computer setup includes
integrators set to time constant 1. Refer to the analog computer's
documentation for detailed information on setup times.

ic() and the two following methods should not be used normally when timing
is critical. Instead, IC- and OP-times should be setup explicitly (see
below) and then a single-run should be initiated which will be under
control of the hybrid controller which takes care for sub-millisecond
precision with respect to timing issues.

=head2 op()

This method switches the analog computer to OPerating-mode.

=head2 halt()

Calling this method causes the analog computer to switch to HALT-mode. In
this mode the integrators are halted and store their last value. After
calling halt() it is possible to return to OP-mode by calling op() again.
Depending on the analog computer being controlled, there will be a more or
less substantial drift of the integrators in HALT-mode, so it is advisable
to keep the HALT-periods as short as possible to minimize errors.

A typical operation cycle may look like this: IC-OP-HALT-OP-HALT-OP-HALT.
This would start a single computation with the possibility of reading
values from the analog computer during the HALT-intervals.

Another typical cycle is called "repetitive operation" and looks like this:
IC-OP-IC-OP-IC-OP... This is normally used with the integrators set to fast
time-constants and allows to display a solution as a more or less flicker
free curve on an oscilloscope for example.
```

```
=head2 enable_ovl_halt()

During a normal computation on an analog computation there should be no
overloads of summers or integrators. Such overload conditions are either a
sign of a machine failure or (more common) an erroneous computer setup
(normally caused by wrong scaling of the underlying equations). To catch
such problems it is usually a good idea to switch the analog computer
automatically to HALT-mode when an overload occurs. This is done by calling
this method during the setup phase.

=head2 disable_ovl_halt()

Calling this method will disable the automatic halt-on-overload
functionality of the hybrid controller.

=head2 enable_ext_halt()

Sometimes it is necessary to halt a computation when some condition is
fulfilled (some value reached etc.). This is normally detected by a
comparator used in the analog computer setup. The hybrid controller
features an EXT-HALT input jack that can be connected to such a comparator.
After calling this method, the hybrid controller will switch the analog
computer from OP-mode to HALT as soon as the input signal patched to this
input jack goes high.

=head2 disable_ext_halt()

This method disables the HALT-on-overflow feature of the hybrid controller.

=head2 single_run()

Calling this method will initiate a so-called "single-run" on the analog
computer which automatically performs the sequence IC-OP-HALT. The times
spent in IC- and OP-mode are specified with the methods set_ic_time() and
set_op_time() (see below).

It should be noted that the hybrid controller will not be blocked during
such a single-run - it is still possible to issue other commands to read or
set ports etc.
```

```
=head2 single_run_sync()

This function behaves quite like single_run() but waits for the termination
of the single run, thus blocking any further program execution. This method
returns true if the single-run mode was terminated by an external halt
condition. Otherwise undef is returned.

=head2 repetitive_run()

This initiates repetitive operation, i.e. the analog computer is commanded
to perform an IC-OP-IC-OP-... sequence. The hybrid controller will also not
block during this run. To terminate a repetitive run, either ic() or halt()
may be called, depending on the mode the analog computer should stop. Note
that these methods act immediately and will interrupt any ongoing IC- or
OP-period of the analog computer.

=head2 pot_set()

This function switches the analog computer to POTSET-mode i.e. the
integrators are set implicitly to HALT, while all (manual) potentiometers
are connected to +1 on their respective input side. This mode can be used
to readout the current settings of the potentiometers.

=cut

# Create basic methods
my %methods = (
    ic               => ['i', '^IC'],            # Switch AC to IC-mode
    op               => ['o', '^OP'],            # Switch AC to OP-mode
    halt             => ['h', '^HALT'],          # Switch AC to HALT-mode
    disable_ovl_halt => ['a', '^OVLH=DISABLED'], # Disable HALT-on-overflow
    enable_ovl_halt  => ['A', '^OVLH=ENABLED'],  # Enable HALT-on-overflow
    disable_ext_halt => ['b', '^EXTH=DISABLED'], # Disable external HALT
    enable_ext_halt  => ['B', '^EXTH=ENABLED'],  # Enable external HALT
    repetitive_run   => ['e', '^REP-MODE'],      # Switch to RepOp
    single_run       => ['E', '^SINGLE-RUN'],    # One IC-OP-HALT-cycle
    pot_set          => ['S', '^PS'],            # Activate POTSET-mode
);
```

```

eval ('
    sub ' . $_ . ' {
        my ($self) = @_;
        $self->{port}->write("' . $methods{$_}[0] . '");
        my $response = get_response($self);
        confess "No response from hybrid controller! Command was \'' .
                $methods{$_}[0] . '\'." unless $response;
        confess "Unexpected response from hybrid controller:\\n\\tCOMMAND=\'' .
                $methods{$_}[0] . '\', RESPONSE=\'$response\', PATTERN=\'' .
                $methods{$_}[1] . '\'\\n"
            if $response !~ /' . $methods{$_}[1] . '/;
    }
') for keys(%methods);

sub single_run_sync() {
    my ($self) = @_;
    $self->{port}->write('F');
    my $response = get_response($self);
    confess "No Response from hybrid controller! Command was 'F'"
        unless $response;
    confess "Unexpected response:\n\tCOMMAND='F', RESPONSE='$response'\n"
        if $response !~ /^SINGLE-RUN/;
    my $timeout = 1.1 * ($self->{times}{ic_time} + $self->{times}{op_time});
    $response = get_response($self);
    confess "No Response during single_run_sync within $timeout ms"
        unless $response;
    confess "Unexpected response after single_run_sync: '$response'\n"
        if $response !~ /^EOSR/ and $response !~ /^EOSRHLT/;
    # Return true if the run was terminated by an external halt condition
    return $response =~ /^EOSRHLT/;
}

=head2 set_ic_time($milliseconds)

It is normally advisable to let the hybrid controller take care of timing the
analog computer modes of operation as the communication with the digital host
introduces quite some jitter. This method sets the time the analog computer
will spend in IC-mode during a single- or repetitive run. The time is
```

```
specified in milliseconds and must be positive and can not exceed 999999
milliseconds due to limitations of the hybrid controller itself.

=cut

# Set IC-time
sub set_ic_time {
    my ($self, $ic_time) = @_;
    confess 'IC-time out of range - must be >= 0 and <= 999999!'
        if $ic_time < 0 or $ic_time > 999999;
    my $pattern = "^T_IC=$ic_time\$";
    my $command = sprintf("C%06d", $ic_time);
    $self->{port}->write($command);
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    confess "Unexpected response: '$response', expected: '$pattern'"
        if $response !~ /$pattern/;
    $self->{times}{ic_time} = $ic_time;
}

=head2 set_op_time($milliseconds)

This method specifies the duration of the OP-cycle(s) during a single- or
repetitive analog computer run. The same limitations hold with respect to the
time specified as for the set_ic_time() method.

=cut

# Set OP-time
sub set_op_time {
    my ($self, $op_time) = @_;
    confess 'OP-time out of range - must be >= 0 and <= 999999!'
        if $op_time < 0 or $op_time > 999999;
    my $pattern = "^T_OP=$op_time\$";
    my $command = sprintf("c%06d", $op_time);
    $self->{port}->write($command);
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    confess "Unexpected response: '$response', expected: '$pattern'"
```

```
        if $response !~ /$pattern/;
    $self->{times}{op_time} = $op_time;
}

=head2 read_element($name)

This function expects the name of a computing element specified in the
configuation YML-file and applies the corresponding 16 bit value $address to
the address lines of the analog computer's bus system, asserts the active-low
/READ-line, reads one value from the READOUT-line, and de-asserts /READ again.
read_element(...) returns a reference to a hash with the keys 'value' and
'id'.

=cut

sub read_element {
    my ($self, $name) = @_;
    my $address = hex($self->{elements}{$name});
    confess "Computing element $name not configured!\n"
        unless defined($address);
    $self->{port}->write('g' . sprintf("%04X", $address & 0xffff));
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    my ($value, $id) = split(/\s+/, $response);
    $id = $self->{types}{$id & 0xf} || 'UNKNOWN';
    return { value => $value, id => $id};
}

=head2 read_all_elements()

The routine read_all_elements() switches the computer to HALT and reads the
current values from all elements listed in the elements-section of the
configuration file. It returns a reference to a hash containing all elements
with their associated values and IDs.

=cut

sub read_all_elements {
    my ($self) = @_;
```

```
    $self->halt();
    my %result;
    for my $key (sort(keys(%{$self->{elements}})))
    {
        my $result = $self->read_element($key);
        $result{$key} = { value => $result->{value}, id => $result->{id} };
    }
    return \%result;
}

=head2 read_digital()

In addition to the analog input channels mentioned above, the hybrid
controller also features digital inputs (two) which can be used to read out
the state of comparators or other logic elements of the analog computer being
controlled. This method also returns an array-reference containing values of
0 or 1.

=cut

# Read digital inputs
sub read_digital {
    my ($self) = @_;
    $self->{port}->write('R');
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    my $pattern = '^' . '\d+\s+' x (DIGITAL_INPUT_PORTS - 1) . '\d+';
    confess "Unexpected response: '$response', expected: '$pattern'"
        if $response !~ /$pattern/;
    return [ split(/\s+/, $response) ];
}

=head2 digital_output($port, $value)

The hybrid controller also features digital outputs (two) which can be used to
control electronic/relay switches in the analog computer being controlled.
Calling digital_output(0, 1) will set the first (0) digital output to 1 etc.

=cut
```

```

# Set/reset digital outputs
sub digital_output {
    my ($self, $port, $state) = @_;
    confess '$port must be >= 0 and < ' . DIGITAL_OUTPUT_PORTS
        if $port < 0 or $port > DIGITAL_OUTPUT_PORTS;
    $self->{port}->write(($state ? 'D' : 'd') . $port);
}

=head2 read_mpts()

Calling read_mpts() returns a reference to a hash containing the current
settings of all manual potentiometers listed in the
manual_potentiometers-section in the configuration file. To accomplish this,
the analog computer is switched to POTSET-mode (implying HALT for the
integrators). In this mode, all inputs of potentiometers (apart from "free"
potentiometers unless their second input is patched to AGND) are connected to
+1, so that their current setting can be read out.

=cut

sub read_mpts {
    my ($self) = @_;
    $self->pot_set();
    my %result;
    for my $key (@{$self->{manual_potentiometers}}) {
        my $result = $self->read_element($key);
        $result{$key} = { value => $result->{value}, id => $result->{id} };
    }
    return \%result;
}

=head2 set_pt($address, $value)

To set a digital potentiometer, set_pt() is called. The first argument is the
address of the potentiometer to be set (0 <= number < number-of-potentiometers
as specified in the potentiometers section in the configuration YML-file), the
second argument is a floatingpoint value 0 <= v <= 1. If either the address or
the value is out of bounds, the function will die.
```

```
=cut

sub set_pt {
    my ($self, $address, $value) = @_;
    confess 'Addr must be >= 0 and < ' . $self->{builtin_dpt}{number} .
            " it is $address"
        if $address < 0 or $address >= $self->{builtin_dpt}{number};
    confess '$value must be >= 0 and <= 1' if $value < 0 or $value > 1;

    #  Convert value to an integer suitable to setting the potentiometer and
    # generate fixed length strings for the parameters address and value:
    $value = sprintf('%04d',
        int($value * (2 ** $self->{builtin_dpt}{resolution} - 1)));
    $address = sprintf('%d', $address);
    $self->{port}->write("P$address$value"); # Send command
    my $response = get_response($self);      # Get response
    confess 'No response from hybrid controller!' unless $response;
    my ($raddress, $rvalue) = $response =~ /^P(\d+)=(\d+)$/;
    confess "set_pt failed! $address vs. $raddress, $value vs. $rvalue"
        if $address != $raddress or $value != $rvalue;
}

=head2 get_status()

Calling get_status() yields a reference to a hash containing all current
status information of the hybrid controller. A typical hash structure
returned may look like this:

    $VAR1 = {
          'IC-time' => '500',
          'MODE' => 'HALT',
          'OP-time' => '1000',
          'STATE' => 'NORM',
          'OVLH' => 'DIS',
          'EXTH' => 'DIS'
        };

In this case the IC-time has been set to 500 ms while the OP-time is set to
```

```
one second. The analog computer is currently in HALT-mode, and the hybrid
controller is in its normal state, i.e. it is not currently performing a
single- or repetitive-run. HALT on overload and external HALT are both
disabled.

=cut

# Get status returns a hash-reference
sub get_status {
    my ($self) = @_;
    $self->{port}->write('s');
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    my %state;
    for my $entry (split(/\s*,\s*/, $response)) {
        my ($parameter, $value) = split(/\s*=\s*/, $entry);
        $state{$parameter} = $value;
    }
    return \%state;
}

=head2 get_op_time()

In some applications it is useful to be able to determine how long the analog
computer has been in OP-mode. As time as such is the only so-called free
variable of integration in an analog-electronic analog computer, it is a
central parameter to know. Imagine that some integration is being performed by
the analog computer and the time which it took to reach some threshold value
is being investigated. In this case, the hybrid controller would be configured
so that external-HALT is enabled. Then the analog computer would be placed to
IC-mode and then to OP-mode. After an external HALT has been triggered by some
comparator of the analog commputer, the hybrid controller will switch the
analog computer to HALT-mode immediately. Afterwards, the time the analog
computer spent in OP-mode can be determined by calling this method. The time
will be returned in microseconds (the resolution should be +/- 3 to 4
microseconds).

=cut
```

```
# Get current time the AC spent in OP-mode
sub get_op_time {
    my ($self) = @_;
    $self->{port}->write('t');
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    my $pattern = 't_OP=\-?\d*';
    confess "Unexpected response: '$response', expected: '$pattern'"
        if $response !~ /$pattern/;
    my ($time) = $response =~ /=\s*(\-?\d+)$/;
    return $time ? $time : -1;
}

=head2 reset()

The reset() method resets the hybrid controller to its initial setup. This
will also reset all digital potentiometer settings including their number!
During normal operations it should not be necessary to call this method which
was included primarily to aid debugging.

=cut

sub reset {
    my ($self) = @_;
    $self->{port}->write('x');
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    confess "Unexpected response: '$response', expected: 'RESET'"
        if $response ne 'RESET';
}

=head2 set_address(address)

set_address() is used to set the hybrid controller to a different address than
its default address of 0x0090. The hybrid controller requires its own address
on the backplane in order to set the builtin digital potentiometers. If the
controller is placed into another slot than the last one of the main backplane
(which is not recommended), then this method has to be called before any
changes to the builtin digitally controlled potentiometers are made. Caution:
```

```
In this case, setting these potentiometers to their default values as
specified in the corresponding configuration YML-file will not succeed! The
address has to be specified in hexadecimal notation with four digits (padded
on the left with zeros if necessary).

=cut

sub set_address() {
    my ($self, $address) = @_;
    $self->{port}->write("m$address");
    my $response = get_response($self);
    confess 'No response from hybrid controller!' unless $response;
    my ($value) = $response =~ /^MY_ADDR=(.+)$/;
    confess "Unexpected response: '$response', expected: 'MY_ADDR=...'"
        unless defined($value);
    $_ =~ s/^0+// for $address, $value;
    confess "Address returned ($value) differs from address sent ($address)!"
        unless $address == $value;
}

=head1 Examples

The following example initates a repetitive run of the analog computer with 20
ms of operating time and 10 ms IC time:

    use strict;
    use warnings;

    use File::Basename;
    use HyCon;

    (my $config_filename = basename($0)) =~ s/\.pl$//;
    my $ac = HyCon->new("$config_filename.yml");

    $ac->set_op_time(20);
    $ac->set_ic_time(10);

    $ac->repetitive_run();
=cut
```

```

=head1 AUTHOR

Dr. Bernd Ulmann <lt>ulmann@analogparadigm.com<gt>

=cut

return 1;
```

HyCon.pm