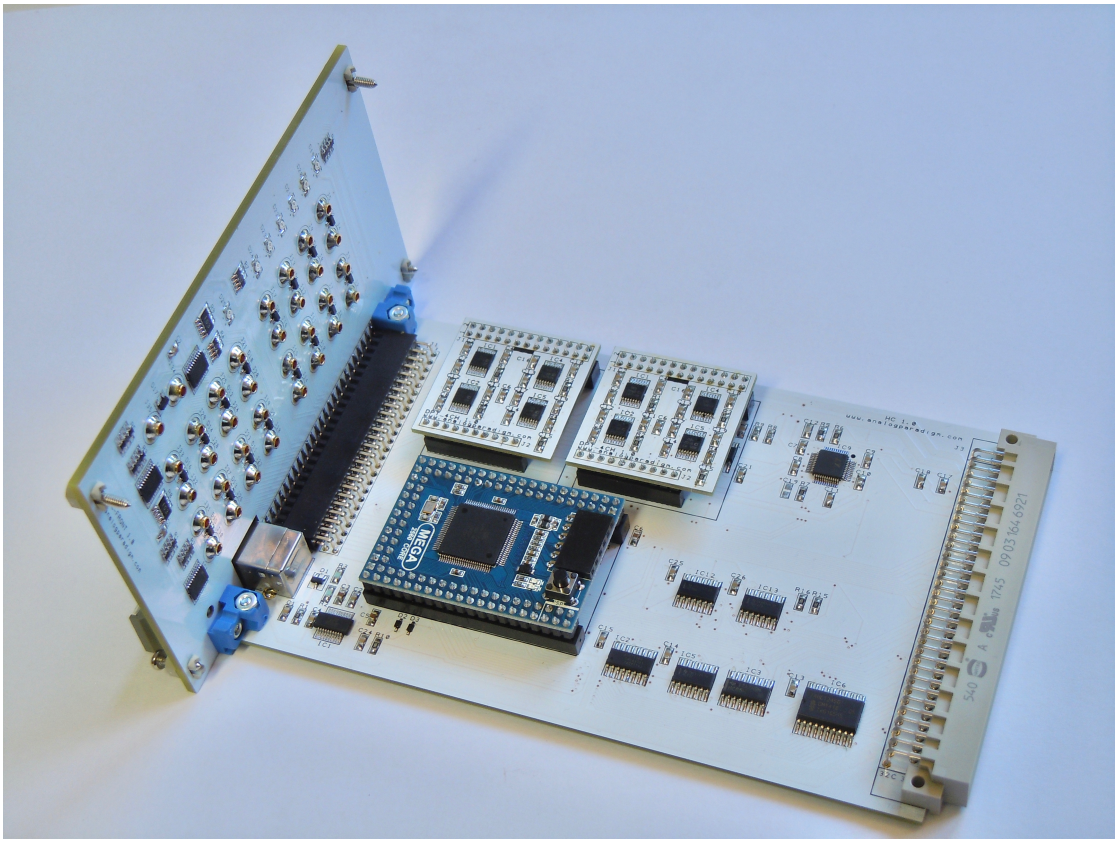
Hybrid Controller[1]

User manual



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

1 Introduction

Important

All documentation, software, and application notes regarding the Model-1 analog and hybrid computer can be found online at `https://github.com/anabrid`.

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 controlled 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 respectively.

The USB connector is visible on the lower far left. On the lower right is an input jack for an external HALT-signal (typically derived from a comparator output in an analog computer setup or from an external device) and a trigger output jack. This trigger signal is typically used in conjunction with an oscilloscope to trigger its x-deflection circuit.

The seven LEDs on the right hand side of the module display the current mode of operation (`INITIAL`, `OPERATE`, `HALT`, `POTSET`) as well as an `OVERLOAD` condition or any other `ERROR` condition, which typically results from a communication problem with the host computer. The LED labeled

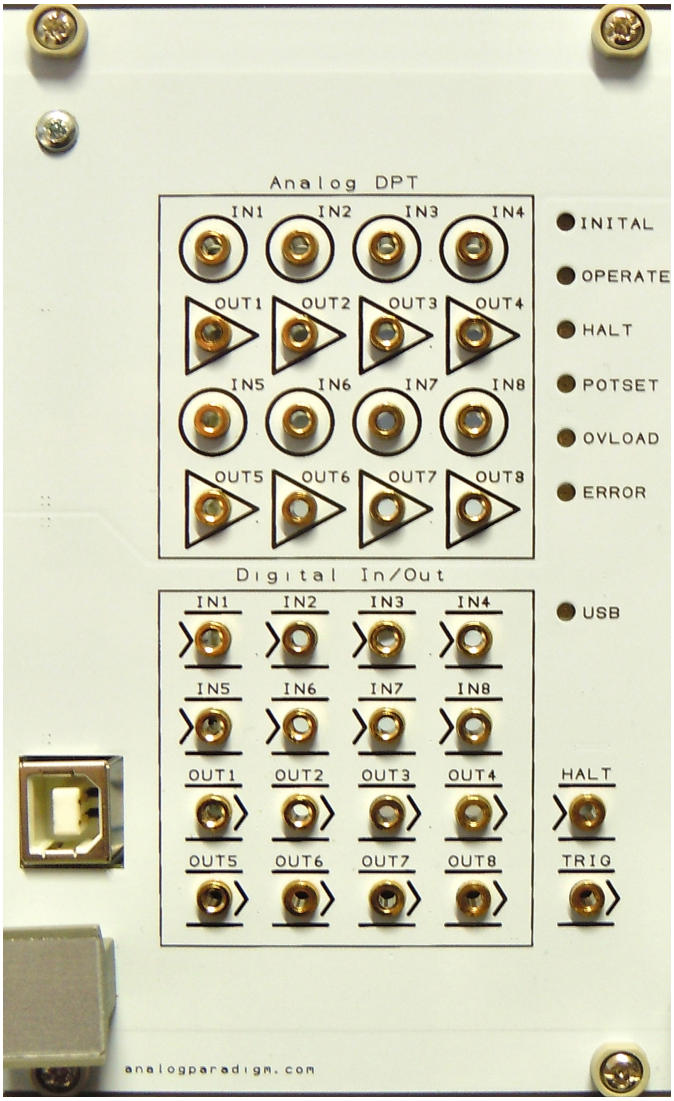


Figure 1.1: Front panel of the hybrid controller

USB is lit green when the USB port is connected to a host computer and flickers during an ongoing communication.

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] or any other terminal software.[2] The communication speed of the hybrid controller is set to 250000 Baud by default,[3] and the device name of the USB-interface 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`.

Figure 2.1 lists all commands the hybrid controller supports in firmware version 2.1. Each command consists of a single letter followed by a varying number of parameters depending on the command.

Caution

A HC command must not be followed by a line termination character of any sort! To get status information, just the letter `s` has to be sent to the HC. No control characters such as line feed or carriage return are allowed in communication with the HC!

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

[2] This direct mode of communication is typically not suitable for larger problems and is normally only used for debugging or testing purposes. Normally, the Perl module `IO::HyCon`, described in more detail in section 3, is used to control the analog computer being part of a hybrid computer setup.

[3] This can be changed in the source code if necessary.

```
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           Get (fetch) data for a readout group defined by an address list
            via G
F           Start single IC/OP-cycle with completion message (for
            synchronous operation)
g\x{4}      Set address of computing element and return its ID and value
G\w         Set addresses \x;\x. for group readout, '.' terminates the input
h           Halt
i           Initial condition
I           Get information about the computer configuration, see
            documentation for parameters
l           Get data from last logging operation
L\x{4}      Locate a computing element by its 4 digit address by turning on
            the read LED
            An address of ffff will deassert read.
o           Operate
P\x{4}\x{2}\d{4} Set the digital potentiometer number \x{2} on the card with
            address \x{4} 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
X\x{4}\x{20} Send a configuration bitstream (40 hex nibbles) to the
            XBAR-module at address \x{4}
?           Print Help
```

Figure 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.)

2.1 A simple example

To perform a computation run of the analog computer under manual control, just send the `i` command to the HC to set all integrators (as long as these are not externally controlled by means of their `ModIC` and `ModOP` input jacks) to their respective initial conditions.

This can then be followed by the command `o` to enter the operate mode. This mode is either left by going back to initial condition or by sending `h` to set the analog computer to halt mode as shown in this simple example:[4]

Example

```
i       Set the analog computer to IC mode
IC
o       Set the mode to operate
OP
h       And finally halt the analog computer
HALT
```

Like the manuel 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 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:

Example

```
C000010 Set the initial condition time to 10 ms
T_IC=10
c000010 Set the operate time to 10 ms
T_OP=10
e       Start repetitive operation
REP-MODE
```

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. The parameters are typically fixed length character sequences.

[4] Commands sent to the hybrid controller are written in black, comments are blue, and responses from the HC are printed in red. The responses are evaluated by the Perl module `IO::HyCon` described in section 3.

		Module address									
		0	1	2	3	4	5	6	7	8	9
Chassis 0	PS	HC		PT8				INT4			
Chassis 1	XIBNC	MLT8		SUM8				INT4			

Figure 2.2: Typical configuration of a small Model-1 analog/hybrid computer system

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 and proper actions can be taken to rectify the problem.

2.2 Addresses

Many commands require a module address as a parameter in order to perform some action. Each module within a Model-1 analog/hybrid computer system has a unique address defined by its position in the system. An address consists of four 4 Bit parts:[5]

- The rack address which is normally `0`.[6]
- The chassis address. Each system consists of a main chassis containing the power supply as well as the PS module. The main chassis always has address `0`. Up to four additional chassis can be connected to a single main chassis in a single system installation. These chassis have addresses `1, 2, 3, 4` respectively.
- Each chassis has a backplane with ten slots. The leftmost slot has address `0`, the rightmost one is at address `9`.[7]
- The element address. Each computer module can hold up to 16 addressable computing elements which are numbered from `0` to `f`.

A typical small Model-1 analog/hybrid computer system consists of two chassis with the module complement as shown in figure 2.2.

[5] Each of these four 4 bit parts is denoted by a single hexadecimal digit `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f`.

[6] A typical Model-1 analog/hybrid computer system consists of up to five chassis which form rack `0`. If this is not sufficient, several such systems can be connected together forming a larger installation.

[7] The PS module is an exception as it is located left from slot `0` and has address `f`.

The various computing elements are now located at the following addresses:

PS	00f0 (fixed address)
HC	0000
PT8	0020
INT4 (chassis 0)	0060
MLT8	0100
SUM8	0120
INT4 (chassis 1)	0160

Accordingly, to read the value of the second integrator of the INT4 module in chassis 1, the address 0161 must be used.[8]

2.3 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 four digit hexadecimal address of the module containins the potentiometer to be set[9] followed by the two digit hexadecimal number of the potentiometer on the module to be set, followed by a four digit decimal (sic!) value in the range of 0000 to 1023.

The following example sets the fourth potentiometer on the HC module in slot 0 of chassis 0 in rack 0 to the value $\frac{1}{2}$:

Example

```
P0000030512 Module address 0000, potentiometer 03, value 0512
P0.3=512
```

At power-on, the hybrid controller performs an intialization routine that sets all potentiometers to the default value 0.

2.4 Read out operation

The hybrid controller can be used to read out the value of each computing element of the analog computer with high precision. All values are read with respect to the machine units of ± 10 V which are interpreted as the scaled values ± 1, so a voltage of $+5$ V would be returned as the value 0.5000.[10]

[8] Note that all addresses start with 0 from a hardware perspective while the numbering scheme on the front panels start with 1 instead!

[9] Apart from the HC there are also DPT24 modules which contain 24 digitally controlled potentiometers each.

[10] The last digit is not certain.

Module	ID
PS	0
SUM8	1
INT4	2
PT8	3
CU	4
MLT8	5
MDS2	6
CMP4	7
HC	8
DPT24	9

Table 2.1: Module types

To read the value of the second integrator of the INT4 module in chassis 1 in the example shown in section 2.2, the following command must be sent to the HC:

Example

```
g0161 Read the value from the element at address 0x0161
-0.3511 2
```

The string returned consists of two parts: The actual value read, -0.3511 in this case, and the module ID (2 denoting an integrator).

Table 2.1 shows all currently supported module types.

In addition to this single readout capability, it is also possible to define so-called *readout groups* i.e. sets of computing element addresses which can be read out with a single command after the readout group has been defined as shown in the following example:[11]

Example

```
G0100;0103;0063.     Define a readout group consisting of three elements at
                     addresses 0100, 0103, and 0063. Please note that the
                     semicolons and the final dot are mandatory!
0.1940;0.2364;0.0050
```

[11] This functionality is typically used by the Perl module `IO::Hycon`, see section 3 and not for manual readout.

3 HyCon.pm

3.1 Introductory example

The best way to introduce the Perl module `IO::HyCon` will be a simple demonstration program that will just run the analog computer for a predetermined time and then readout all computing elements and display their respective values as shown in the following.

`IO::HyCon` can be found at `https://metacpan.org/pod/IO::HyCon` and features an object oriented interface to the hybrid controller.[1] A program using `IO::HyCon` must have an associated YML[2] configuration file with the same filename in front of the extension as the Perl main program.

In the following example, the Perl program will be called `read_all.pl`. Accordingly, the corresponding configuration file must be named `read_all.yml`. The system used as an example in the following and shown in figure 3.1 has the complement of computing elements shown in table 3.1.

The configuration file `read_all.yml` shown in the following example is pretty minimal and consists of several sections:

`serial`: Here, the serial line configuration is stored. The hybrid controller expects a Baud rate of 250000 Baud, the Devicename (`port`) depends on the hybrid controller and on the operating system used. Here, a UNIX-like operating system is assumed.[3] `poll_interval` and `poll_attempts` are good defaults and should not be changed without reason.

[1] It is perfectly feasible to not use this module and instead write another interface which just has to send the proper short commands to the hybrid controller and interpret the return values.

[2] C. f. `https://en.wikipedia.org/wiki/YAML`.

[3] Mac OS-X

Module	Description	Address
PS	Machine unit reference	0x00f0, 0x00f1
HC	Hybrid controller	0x0000
PT8	Eight manual poteniometers	0x0020...0x0027
INT4	Four integrators	0060...0063
XIBNC	Four BNC adapters	N/A
MLT8	Eight multipliers	0x0100...0x0107
SUM8	Eight Summers	0x0120...0x0127
INT4	Four integrators	0x0160...0x0163

Table 3.1: Typical module complement of a hybrid system

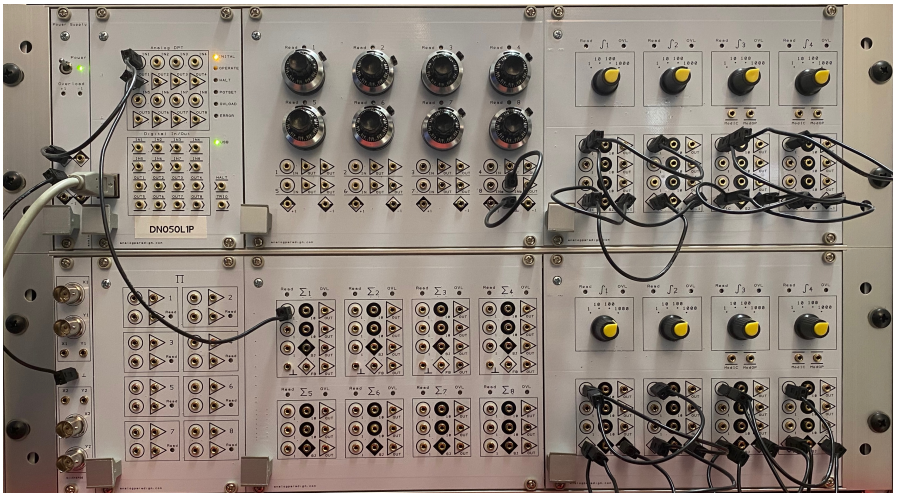


Figure 3.1: Typical hybrid computing system

`types`: Each computing element returns a module ID when it is read out. Here the mapping is done between module IDs and cleartext names.

`elements`: Here all computing elements within a system are defined.[4]

Example

```
serial:
    port: /dev/cu.usbserial-DN050L1P
    bits: 8
    baud: 250000
    parity: none
    stopbits: 1
    poll_interval: 10
    poll_attempts: 20000
types:
    0: PS
    1: SUM8
    2: INT4
    3: PT8
    4: CU
    5: MLT8
    6: MDS2
    7: CMP4
    8: HC
elements:
    PT0: 0x0020
    PT1: 0x0021
    PT2: 0x0022
    PT3: 0x0023
    PT4: 0x0024
    PT5: 0x0025
    PT6: 0x0026
    PT7: 0x0027
    INT0-0: 0x0060
    INT0-1: 0x0061
    INT0-2: 0x0062
```

[4]Typically, only those computing elements should be defined in this section which are to be read out during a computer run.

```
INT0-3: 0x0063
MLT0: 0x0100
MLT1: 0x0101
MLT2: 0x0102
MLT3: 0x0103
MLT4: 0x0104
MLT5: 0x0105
MLT6: 0x0106
MLT7: 0x0107
SUM0: 0x0120
SUM1: 0x0121
SUM2: 0x0122
SUM3: 0x0123
SUM4: 0x0124
SUM5: 0x0125
SUM6: 0x0126
SUM7: 0x0127
INT1-0: 0x0160
INT1-1: 0x0161
INT1-2: 0x0162
INT1-3: 0x0163
```

The associated Perl program is shown in the following listing. First some modules are imported, then a single numerical parameter is read from the command line which specifies how long an OP-cycle should be. The actual object oriented interface to the hybrid computer is generated by `$ac = IO::HyCon->new();`. All functionality is then implementes as methods accessible through `$ac`.

Performing a single run is done by calling `$ac->single_run_sync()` (at the end of the program). Calling `$ac->read_all_elements()` reads all elements defined in the `elements`-section in the YML-configuration file.[5]

Example

```
use strict;
use warnings;

use File::Basename;
use Time::HiRes qw(usleep);
```

[5]This is by far not the fasted way to read many elements but the simplest. A more sophisticated technique using *readout groups* is described later.

```
use Term::ANSIScreen;
use IO::HyCon;

$| = 1;

die "Usage: $0 <op-time in ms>\n" unless @ARGV == 1;
my ($tc) = @ARGV;

my $console = Term::ANSIScreen->new();  # This is required for cursor control
my $ac = IO::HyCon->new();

my $op_time = ($tc);                    # Milliseconds, each
my $ic_time = $op_time;

$ac->reset();
$ac->set_ic_time($ic_time);
$ac->set_op_time($op_time);

$console->Cls();
while (1)
{
    $console->Cursor(0, 0);
    my $result = $ac->read_all_elements();
    my $counter = 0;
    my $last_key = '';
    for my $key (sort(keys(%$result)))
    {
        if (substr($key, 0, 2) ne $last_key) {
            $last_key = substr($key, 0, 2);
            print "\n";
            print "\n" if $counter % 4;
            $counter = 0;
        }
        print "$key: $result->{$key}{value}\t";
        print "\n" if !(++$counter % 4);
    }
    print "\nOP-TIME: $op_time ms\n";
```

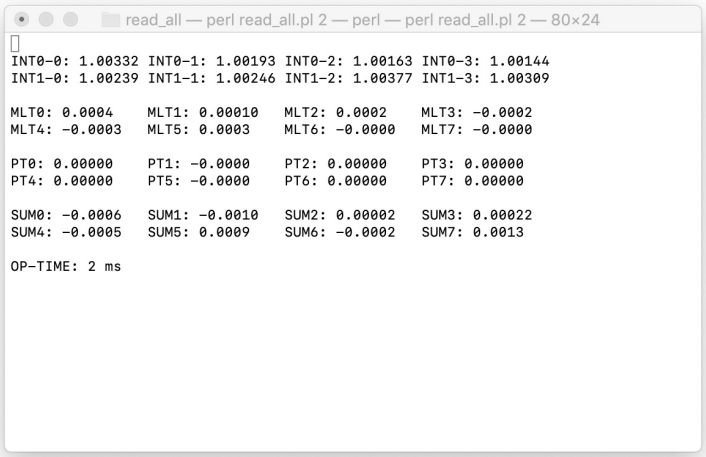


Figure 3.2: Typical output of the `read_all.pl` example program

```
    $ac->single_run_sync();
}
```

A typical output of this simple test program is shown in figure 3.2.

3.2 Varying parameters and gathering data

The example in the preceding section showed how to read the values of all elements defined in the `elements`-section of the configuration file by calling `$ac->read_all_elements()`. This process is pretty slow since the elements are read sequentially under control of the Perl program involving quite some communication between the digital computer and the HC.

The same restriction is true for the method `$ac->read_element()` which can be used to read the value of a single computing element defined in the configuration file. This, too, is a single, explicit read operation and thus quite time consuming so that these functions should only be used after a completed analog computer run when the computer is in HALT-mode and only stationary values[6] are to be read.

[6]Please note that there is some unavoidable drift in the integrators in HALT-mode! Any readout done in this mode should be done quickly to obtain as precise values as possible.

The hybrid controller allows a different way to read values from computing element by defining a *readout group*.[7] Such a readout group can contain the addresses of one or more computing elements. During an analog computer run (OP-mode), the HC automatically reads out the elements defined in the readout group and stores the values in its internal memory.

Important

There is no way to externally control the sampling interval at which the elements defined in a readout group are read. The HC computes the shortest possible sampling interval depending on its amount of free memory, the number of elements contained in the group, and the duration for which the analog computer is in OP-mode.

Thus it is a good idea to have as few elements as possible in a readout group to allow for more samples during the OP-mode interval.

Nevertheless, there is a minimum sample interval due to the time it takes to perform an actual read operation.[a] Accordingly it is typically a good idea to select a slower time constant on the integrators and read as few computing elements as possible to get as many samples as possible.

[a] This involves placing the address of the computing element on the bus, waiting for the readout line to settle, triggering the ADC, waiting for the data conversion process to finish, and finally reading the value from the ADC and storing it in the HC memory.

The following example which is based on [Ulmann(2020), pp. 116] shows how to use this feature: Mathieu's equation,

$$\ddot{y} + (a - 2q\cos(2t))\, y = 0,$$

is to be solved for various parameter values a. The computer setup is shown in figure 3.3. Of interest here is the parameter $\frac{a}{10}$ shown on the lower right of this figure. Using the first digital potentiometer of the HC this value can be varied under program control by the digital computer.

The computer setup is shown in figure 3.4. All parameters except $\frac{a}{10}$ are set by the manual coefficient potentiometers of the PT8 module in the top chassis while $\frac{a}{10}$ is controlled by DPT0 of the HC.

The following listing shows the configuration file for this setup:

Example

```
serial:
    port: /dev/cu.usbserial-DN050L1P
    bits: 8
    baud: 250000
```

[7] This is done with the g-command when communicating directly with the HC.

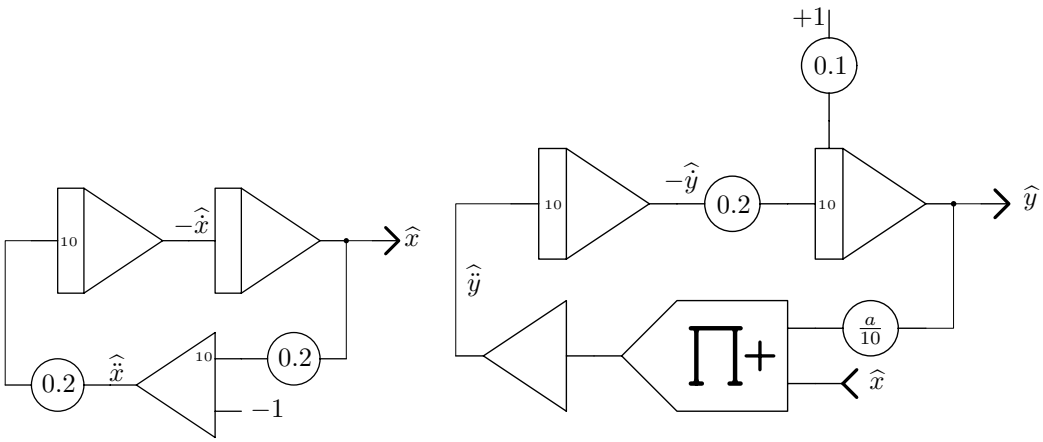


Figure 3.3: Scaled setup for MATHIEU's equation, part II

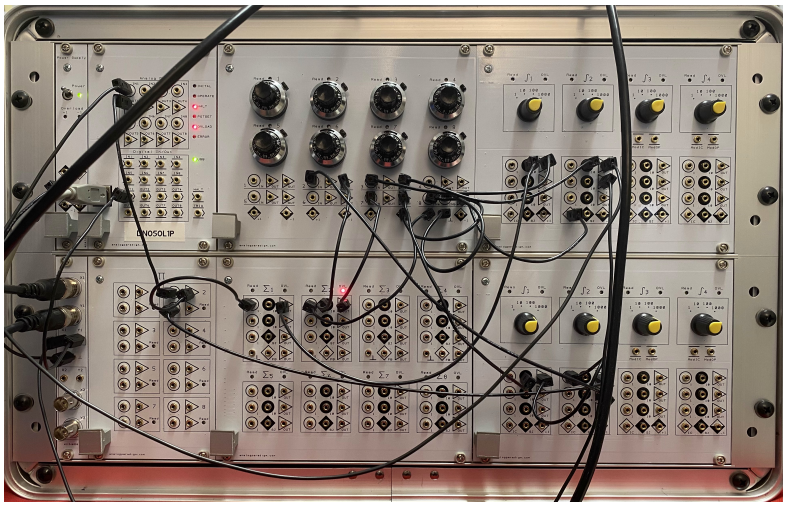


Figure 3.4: Setup of MATHIEU's equation on a Model-1 analog/hybrid computer system

```
    parity: none
    stopbits: 1
    poll_interval: 10
    poll_attempts: 20000
types:
    0: PS
    1: SUM8
    2: INT4
    3: PT8
    4: CU
    5: MLT8
    6: MDS2
    7: CMP4
    8: HC
elements:
    y: 0061
    a: 0000/0
problem:
    times:
        ic: 10
        op: 50
    ro-group:
        - y
```

New is the `problem`-section which is used to describe a "problem" to be solved.[8] In this example the times for the IC- and OP-intervals are defined as 10 ms and 50 ms respectively, and a readout group consisting of only one computing element named `y` is defined. This element is mapped to address `0x0061` i. e. the second integrator on the INT4 module on the right of the top chassis.

The corresponding Perl control program looks like this:

Example

```
use strict;
use warnings;

use IO::HyCon;
```

[8] On systems equipped with an XBAR module even the interconnection between computing elements can be changed under program control etc.

```
my $ac = IO::HyCon->new();
$ac->setup();

for my $a (0 .. 10) {
    print "Running with a = $a...\n";
    $ac->set_pt('a', $a / 10);
    $ac->single_run_sync(); # Run and collect data as defined in the ro-group
    $ac->get_data();
    $ac->plot();
}
```

Calling `$ac->setup()` configures the analog computer according to the definitions in the `problem`-section. In this case the IC- and OP-times are defined and a readout group containing only the one element at address `0x0061` is created.

The value a is varied from 0 to 10 in the central `for`-loop of the program and set by calling `$ac->set_pt('a', $a / 10)`. After setting this digital potentiometer, potentiometer 0 on the module at address `0x0000` i. e. the HC module, a single IC/OP-cycle of the analog computer is initiated by calling `$ac->single_run_sync()`. This call blocks further execution of the Perl-program until the cycle has been completed.

Since a readout group has been defined, the HC automatically collected data from the element at address `0x0061` during the OP-phase. This data is then read from the HC by calling `$ac->get_data()` which stores the data in the $ac-object. The data collected can then be plotted by invoking `$ac->plot()`.[9]

In this case 11 graphs are produced as shown in figure 3.5. If the raw values read from the HC are required instead of a graphical output, the method `store_data()` could be invoked like this `$ac->store_data(filename => "result_$a.dat")` instead of (or in addition to) the `$ac->plot()`-call.

[9] Please note that this feature requires that `gnuplot` is installed on the digital computer and assumes that a UNIX-like operating system is used.

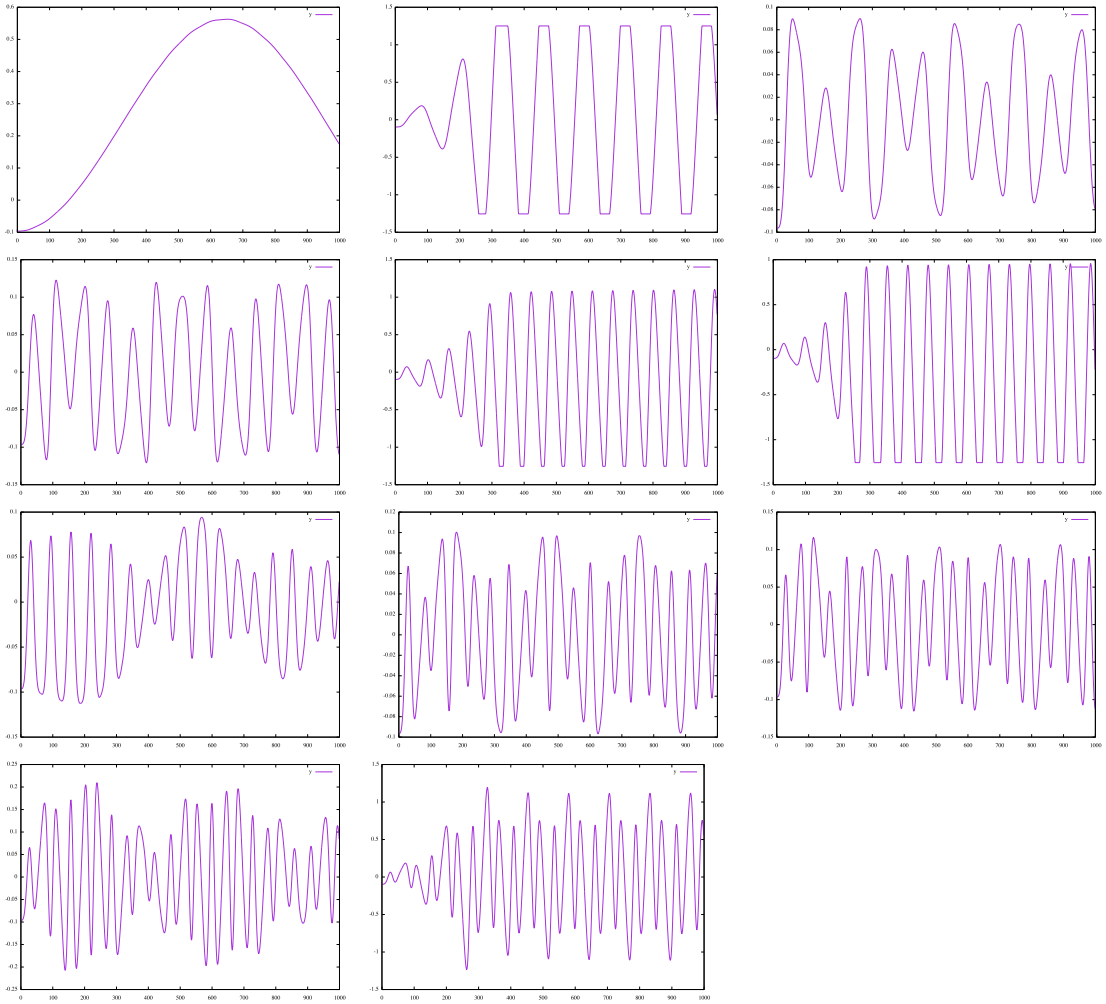


Figure 3.5: behaviour of Mathieu's equation for a varying between 0 and 10 in unit steps

4 The GUI

The hybrid controller is also supported by a simple Perl based GUI based on `HyCon.pm` that allows easy control of the analog computer without the necessity of manually sending commands to the controller by means of a terminal emulator or requiring the user to write Perl control program as shown before. It should be noted that this GUI mainly serves as an example and a platform on which further extensions may be programmed by users.

4.1 Prerequisites

The GUI is based on WxPerl. It is therefore recommended to install *Citrus Perl* which is available for download here: `http://www.citrusperl.com/`.[1]

The GUI (`hc_gui.pl`), like any other program using the module `IO::HyCon`, needs a configuration file as shown in figure 4.1 which is expected to be in the current working directory from which the GUI is invoked.[2]

4.2 Using the GUI

Figure 4.2 shows the GUI which has been invoked by `perl hc_gui.pl` with `HyCon.pm` either in the same directory or in a directory which is part of Perl's `@INC`. Operation is straightforward: The

[1] Make sure after installation that the path to this Perl interpreter is first found in the `$PATH` environment variable!

[2] The `port` setting in the configuration file typically has to be changed to the device used in the current hardware setup!

```
serial:
    port: /dev/cu.usbserial-DN050L1P
    bits: 8
    baud: 250000
    parity: none
    stopbits: 1
    poll_interval: 10
    poll_attempts: 20000
types:
    0: PS
    1: SUM8
    2: INT4
    3: PT8
    4: CU
    5: MLT8
    6: MDS2
    7: CMP4
    8: HC
    9: DPT24
elements:
    DPT0: 0000/0
    DPT1: 0000/1
    DPT2: 0000/2
    DPT3: 0000/3
    DPT4: 0000/4
    DPT5: 0000/5
    DPT6: 0000/6
    DPT7: 0000/7
```

Figure 4.1: Configuration file `hc_gui.yml`

buttons labeled *D0 = 0* etc. control the eight digital output lines of the hybrid controller. These buttons are actually toggle switches – the current status of each output line is shown on the button. The three interlocked buttons on top control the mode of operation (IC, OP, HALT). Single and repetitive runs can also be started based on the times entered in the fields *IC time (ms)* and *OP time (ms)* (in milliseconds with a minimum value of 1 and a maximum of 999999).

Pressing the *Readout* button will read out the computing element addressed by the hexadecimal value entered into the *Address* field. *Digital in* polls the eight digital inputs of the controller and displays their respective values. The eight digital potentiometers can be controlled by the eight sliders in the lower half of the window.

All in all the controls should be quite self-explanatory.

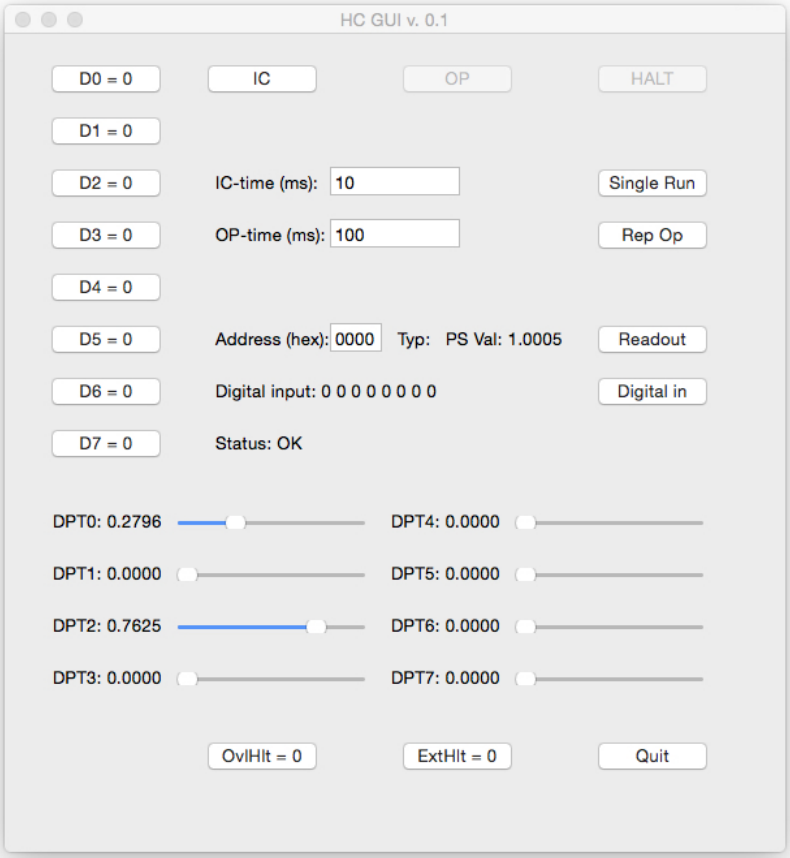


Figure 4.2: Hybrid controll GUI

5 Low level programming

Using `IO::HyCon` from within a Perl control program is quite convenient but not too fast. Especially the communication between the hybrid controller and the attached digital computer is quite time consuming.[1] Another bottleneck is the latency involved in USB data transfers on the digital computer side – incoming data generates an interrupt which must be served which can cost up to several 10 ms of delay on a modern computer system.

In cases where this communication bottleneck is not tolerable, the HC can be directly programmed in C++. A typical application for this is a Monte-Carlo-simulation where one or several parameters are to be varied automatically with each parameter controlling one analog computer run. Here the parameter variation can be done directly within the HC control software while ideally only few result values are sent back to the digital computer.

It is strongly advised to use the existing firmware `HyConAVR.ino`[2] as a basis for customer developments. It is also advised to place all customer logic and code into the central `loop()`-function at the end of the source code.[3] The code basis before `loop()` should be ideally left intact and only the contents of `loop()` changed.

[1] The standard Baud rate can be increased from 250000 Baud to up to 2000000 Baud although not all operating systems support this. 2 MBaud have been successfully tested under Mac OS X and several LINUX distributions. Nevertheless, some LINUX distributions and most Windows systems do not support this rather speed.

[2] See appendix C.

[3] This is an Arduino supplied function which is automaticall run after completing the `setup()` routine etc.

A System information

The hybrid controller features the I-command which returns information on the current system configuration. This command differs from the other commands as its has optional parameters.

If no parameter is specified, the command will scan all racks, all chassis and all cards within the chassis and return a list of cards in the system as shown in the following example:

Example

```
I

system info:
------------------
0000 HC
0020 PT8
0060 INT4
00F0 PS
------------------
0100 MLT8
0120 SUM8
0160 INT4
------------------
```

The I-command can also accept a hexadecimal address which can be from one to four hexadecimal digits in length. The first digit specifies the rack to be scanned, the second the chassis within the

rack, the third the card address and the fourth the computing element on the card selected. The following example shows a scan of all cards and computing elements in chassis 1 in rack 0:

Example

```
I01     01 specifies only the first two address digits

system info:
------------------
0100 MLT8
0120 SUM8
0160 INT4
------------------
```

Appending the character "+" to the (optional) address part of the I-command will also read all elements and print their respective values:

Example

```
I+      Addresses can be used, too, as in I01+ and the like

system info:
------------------
0000 HC
0020 PT8     0.0000
0021 PT8     0.0001
0022 PT8     0.0000
0023 PT8    -0.0001
0024 PT8     0.0000
0025 PT8     0.0000
0026 PT8     0.1994
0027 PT8     0.0953
0060 INT4    0.0000
0061 INT4   -0.0961
0062 INT4    0.0000
0063 INT4   -0.0001
00F0 PS      0.9999
00F1 PS     -1.0009
00F2 PS     -0.1789
00F3 PS     -0.0462
```

```
------------------
0100 MLT8   0.0003
0101 MLT8   0.0001
0102 MLT8   0.0000
0103 MLT8  -0.0004
0104 MLT8  -0.0003
0105 MLT8   0.0002
0106 MLT8  -0.0003
0107 MLT8   0.0001
0120 SUM8  -0.0008
0121 SUM8   0.9966
0122 SUM8  -0.0002
0123 SUM8   0.0007
0124 SUM8   0.0004
0125 SUM8   0.0010
0126 SUM8   0.0011
0127 SUM8   0.0015
0160 INT4   0.0003
0161 INT4   0.0002
0162 INT4   0.0003
0163 INT4  -0.0002
------------------
```

B IO::HyCon

The following sections contain the IO::HyCon module description which was mostly automatically derived from the POD documentation within the module.

B.1 NAME

IO::HyCon - Perl interface to the Analog Paradigm hybrid controller.

B.2 VERSION

This document refers to version 1.5 of HyCon

B.3 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 specific computing element:
    my $element_name = 'SUM8-0';
    my $value = $ac->read_element($element_name);
```

B.4 DESCRIPTION

This module implements a simple object oriented interface to the Arduino® based Analog Paradigm hybrid controller which interfaces an analog computer to a digital computer and thus allows true hybrid computation.

B.5 Functions and methods

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. If no configuration file path is supplied as parameter, new() tries to open a YAML-file with the name of the currently running program but with the extension '.yml' instead of '.pl'. This file is assumed to have the following structure (this example configures a van der Pol oscillator):

```
serial:
    port: /dev/cu.usbserial-DN050L1O
    bits: 8
    baud: 115200
    parity: none
    stopbits: 1
    poll\_interval: 10
    poll\_attempts: 20000
types:
    0: PS
    1: SUM8
    2: INT4
    3: PT8
    4: CU
```

```
    5: MLT8
    6: MDS2
    7: CMP4
    8: HC
elements:
    INT0-: 0160
    INT0+: 0123
    INT0a: 0060/0
    INT0b: 0060/1
    INT0ic: 0080/0

    INT1-: 0161
    INT1+: 0126
    INT1a: 0060/2
    INT1b: 0060/3
    INT1ic: 0080/1
    INT2-: 0162
    INT2a: 0060/4
    INT2b: 0060/5
    INT2ic: 0080/2
    MLT0+: 0100
    MLT0-: 0127
    MLT0a: 0060/6
    MLT0b: 0060/7
    MLT1+: 0101
    MLT1a: 0060/8
    MLT1b: 0060/9
    SUM0-: 0120
    SUM0+: 0124
    SUM0a: 0060/a
    SUM0b: 0060/b
    SUM1-: 0121
    SUM1+: 0125
    SUM1a: 0060/c
    SUM1b: 0060/d
    SUM2-: 0122
    SUM2a: 0060/e
    SUM2b: 0060/f
    XBAR16: 0040
```

```
xbar:
    input:
        - +1
        - -1
        - SUM2-
        - SUM1+
        - SUM1-
        - SUM0+
        - SUM0-
        - MLT1+
        - MLT0+
        - MLT0-
        - INT2-
        - INT1+
        - INT1-
        - INT0+
        - INT0-
    output:
        - INT0a
        - INT0b
        - INT1a
        - INT1b
        - INT2a
        - INT2b
        - MLT0a
        - MLT0b
        - MLT1a
        - MLT1b
        - SUM0a
        - SUM0b
        - SUM1a
        - SUM1b
        - SUM2a
        - SUM2b
problem:
    IC:
        INT1ic: +.1 # Must start with + or -!
    times:
        ic: 20
```

```
        op: 400
    coefficients:
        INT1a: .25
        INT2a: .2
        MLT0a: 1
        MLT0b: 1
        MLT1a: 1
        MLT1b: 1
        SUM0a: .02
        SUM0b: .08
        SUM1a: .1
        SUM1b: .25
    circuit:
        INT1a: INT2-
        INT2a: SUM0-
        MLT0a: INT1-
        MLT0b: INT1-
        MLT1a: INT2-
        MLT1b: SUM1-
        SUM0a: INT1-
        SUM0b: MLT1+
        SUM1a: MLT0+
        SUM1b: -1
```

The setup shown above will not fit your particular analog computer 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 issued before. The values shown above are overly pessimistic but this won't matter during normal operation.

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. This should not be changed but will be expanded when new analog computer modules will be developed.

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. (If jitter during readout is to be minimized, a readout-group should be defined instead, see below.)

Ideally, all manual potentiometers are listed under '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 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 currently equals the product of 'poll_interval' and 'poll_attempts' and may take a few seconds during startup.

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 timeout 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.

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 a minimum duration as required by the time scale factors involved.

ic() and the two following methods should not be used 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. This avoids latencies involved with the communication to and from the hybrid controller and allows sub-millisecond resolution.

op()

This method switches the analog computer to operating-mode.

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 time-constants of 100 or 1000 and allows to display a solution as a more or less flicker free curve on an oscilloscope for example.

enable_ovl_halt()

During a normal computation on an analog computation there should be no overloads of summers or integrators. Such overload conditions are typically the result of 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. The computing element(s) causing the overload condition can the easily identified on the analog computer's console and the variables of the computation run can be read out to identify the cause of the problem.

disable_ovl_halt()

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

enable_ext_halt()

Sometimes it is necessary to halt a computation when some condition is satisfied (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.

disable_ext_halt()

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

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.

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. undef is returned otherwise.

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 not block during this sequence. To terminate a repetitive run either ic() or halt() may be called. Note that these methods act immediately and will interrupt any ongoing IC- or OP-period of the analog computer.

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 read the current settings of the potentiometers.

set_ic_time($milliseconds)

It is normally advisable to let the hybrid controller take care of the overall timing of OP and IC operations since 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 firmware.

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 value specified as for the set_ic_time() method.

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 containing the keys 'value' and 'id'.

read_element_by_address($address)

This function expects the 16 bit address of a computing element as parameter and returns a data structure identically to that returned by read_element. This routine should not be used in general as computing elements are better addressed by their name. It is mainly provided for completeness.

locate()

The locate() method allows to switch the read LED of a computing element on in order to locate it in a large installation. It expects either an element's name or address (hexadecimal) or just nothing in order to switch the currently activated read LED off.

get_data()

get_data() reads data from the internal logging facility of the hybrid controller. When a readout group has been defined and a single_run is executed, the hybrid controller will gather data from the readout-group automatically. There are 1024 memory cells for 16 bit data in the hybrid controller. The sample rate is automatically determined.

read_all_elements()

The routine read_all_elements() 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 read with their associated values and IDs. It may be advisable to switch the analog computer to HALT mode before calling read_all_elements() to minimize the effect of jitter. After calling this routine the computer has to be switched back to OP mode again. A better way to readout groups of elements is by means of a readout-group (see below).

set_ro_group()

This function defines a readout group, i.e. a group of computing elements specified by their respective names as defined in the configuration file. All elements of such a readout group can be read by issuing a single call to read_ro_group(), thus reducing the communications overhead between the HC and

digital computer substantially. A typical call would look like this (provided the names are defined in the configuration file):

```
$ac->set_ro_group('INT0_1', 'SUM2_3');
```

read_ro_group()

read_ro_group() reads all elements defined in a readout group. This minimizes the communications overhead between digital and analog computer and reduces the effect of jitter during readout as well as the risk of a serial line buffer overflow on the side of the hybrid controller. The function returns a reference to a hash containing the names of the elements forming the readout group with their associated values.

read_digital()

In addition to these analog readout capabilities, the hybrid controller also features eight digital inputs which can be used to read the state of comparators or other logic elements of the analog computer being controlled. This method returns an array-reference containing values of 0 or 1 for each of the digital input ports.

digital_output($port, $value)

The hybrid controller also features eight digital outputs which can be used to control the electronic switches which are part of the comparator unit. Calling digital_output(0, 1) will set the first (0) digital output to 1 etc.

set_xbar()

set_xbar creates and sends a configuration bitstream to an XBAR-module specified by its name in the elements section of the configuration file. The routine is called like this:

```
xbar(name, config-string);
```

where name is the name of the XBAR-module to be configured and config-string is a string describing the mapping of output lines to input lines at the XBAR. This string consists of 32 single hex digits or '-'. Each digit/'-' denotes one output of the XBAR-module, starting with output 0. An output denoted by '-' is disabled.

To connect output 0 to input B, output 2 to input E and output 1F to input 2 while all other outputs are disabled, the following call would be issued:

```
xbar(name, 'B-E--------------------------2');
```

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 are connected to the positive machine unit +1, so that their current setting can be read out. ("Free" potentiometers will behave erroneously unless their second input is connected to ground, refer to the analog computer manual for more information on that topic.)

set_pt($name, $value)

To set a digital potentiometer, set_pt() is called. The first argument is the name of the the digital potentiometer to be set as specified in the elements section in the configuration YML-file (an entry like 'DPT24-2: 0060/2'). The second argument is a floating point value 0 <= v <= 1. If the potentiometer to be set can not be found in the configuration data or if the value is out of bounds, the function will die.

read_dpts()

Read the current setting of all digital potentiometers. Caution: This does not query the actual potentiometers as there is not readout capability on the modules containing DPTs, instead this function will query the hybrid controller to return the values it has stored when DPTs were set.

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',
       'RO_GROUP' => [..., ..., ...],
       'DPTADDR' => [60 => 9, 80 => 8, ], # hex address and module id
     };
```

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. A readout-group has been defined, too.

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 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 of interest. 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 is about +/- 3 to 4 microseconds).

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.

store_data()

store_data() stores data gathered from an analog computer run into a file. If no arguments are supplied, the data is read from the current object where it has to have been stored by previously invoking get_data().

If external data and/or an external filename should be used these are expected as optional named parameters as in this example:

```
store_data(data =$>$ [...], filename =$>$ 'scratch.dat');
```

plot()

plot() uses gnuplot (which must be installed and be found in PATH) to plot data gathered by get_data(). If no argument is given, it uses the data stored in the ac-object. Otherwise, data can be given as an optional named parameter which consists of a reference to an array which either contains

data values or arrays of data tuples in case multiple variables were logged during an analog computer run:

```
plot(data =$>$ [...]);
```

If the data set to be plotted contains two element tuples, a phase space plot can be created by specifying the named parameter type:

```
plot(type =$>$ phase);
```

Alternatively, a 3D-plot can be created by specifying

```
plot(type =$>$ 3d);
```

This is useful when partial differential equations are solved with a discretized space.
Another optional parameter is used to set an optional title:

```
plot(title =$>$ 'test');
```

setup()

setup() prepares a problem based on the information contained in the problem section of the configuration YAML-file.

B.6 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 $ac = HyCon->new();

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

B.7 AUTHOR

Dr. Bernd Ulmann, ulmann@analogparadigm.com

C

The firmware `HyConAVR.ino`

The firmware for the HC can be downloaded from `https://github.com/anabrid/Model-1/tree/main/software/HC/firmware`.

Caution

Any changes on the firmware are done at the risk of the customer! It is not impossible to damage hardware by configuring read ports for writing etc. Do not attempt to modify this code unless you are a proficient programmer with at least some experience in an Arduino environment!

The firmware is only intended to run on an HC module which features an 2560mega Core Arduino microcontroller board.

The firmware is most easily compiled using the Arduino IDE which is available for download at `https://www.arduino.cc/en/software`. To compile it, two additional modules, namely `TimerThree` and `TimerFive` need to be downloaded and added to the library of the Arduino IDE.

```
/*
   Simple hybrid interface to the Analogparadigm Model-1 analog computer.
   The same holds true for the two external halt conditions (overflow and the EXT-HALT-input).
   Be careful not to issue c/C (set OP/IC time) or d/D (switch digital output on or off) commands during a single or repetitive run as these commands
   use blocking IO to wait for their respective argument!

   04-AUG-2016  B. Ulmann   Begin implementation
   05-AUG-2016  B. Ulmann   Single/repetitive run implemented
   06-AUG-2016  B. Ulmann   Code-Cleanup, improved timing accuracy, added "t"-command
   07-AUG-2016  B. Ulmann   Changes to match the new Perl-module HyCon.pm
   01-AUG-2016  B. Ulmann   Added basic support for digital potentiometers
   03-SEP-2016  B. Ulmann   Digital potentiometers are now fully supported
   12-MAY-2017  B. Ulmann   Start of 2560-version for the new hybrid controller (lots of new features)
                            New: 'S' -> Enter PUTSET-mode
                                 'g<4 hex digits>' -> set address bus and return element ID and readout value
   13-MAY-2017  B. Ulmann   Changed baudrate to 250000, some communication protocol changes
   14-MAY-2017  B. Ulmann   Minor bug fixes
   16-MAY-2017  B. Ulmann   single_run with synchronous mode, PUTSET behaviour changed
   22-AUG-2018  B. Ulmann   Start of the port to the new HC module, got rid of the slow digitalWrite calls etc.
   23-AUG-2018  B. Ulmann   Implemented the DPT module routines, only the builtin DPTs are supported right now
   31-AUG-2018  B. Ulmann   Started ADC support
   21-FEB-2019  B. Ulmann   Changed baud rate to 115200 since at least some LINUX systems can't cope with 250000
   05-SEP-2019  B. Ulmann   Added group-commands G, f
   21-SEP-2019  B. Ulmann   Modified group-readout to minimize skew, added address search in setup, baudrate back to 250000
   02-DEC-2019  B. Ulmann   Started adding support for DPT24
   06-DEC-2019  B. Ulmann   Continued adding DPT24 support
   13-DEC-2019  B. Ulmann   Added support for the prototype XBAR-module
   17-DEC-2019  B. Ulmann   Added logging functionality
   18-DEC-2019  B. Ulmann   Single-run-OP-time is now interrupt controlled to minimize timing jitter...
                            Known bug: Data logging logs too few data points when the compute intervals get small.
                            At several seconds everything is as expected, at several ms, things get messy which is
                            not () causes by the MIN_SAMPLE_INTERVAL value (a value too small causes the interrupt
                            routine to not finish before it is called the next time).
   23-DEC-2019  B. Ulmann   The data logging got out of sequence after the first tuple, causing a glitch at the start of gnuplot output.
   01-MAR-2020  B. Ulmann   readout is now more precise.
   09-MAR-2020  B. Ulmann   Baud rate set to 250000.
   04-APR-2020  B. Ulmann   ADC read routine modified, it now performs two reads with arithmetic mean which smoothes the result considerably
   05-APR-2020  B. Ulmann   ADC read routine is now about 4 times fast! Who knew that shifts on 32 bit integers were so slow?
   11-APR-2020  B. Ulmann   Oh dear, ADC_READ_DELAY was too short which caused the ADC to lose the MSB, but this only happened
                            due to an erroneous configration procedure... Added locate command 'L' to simplify locating components in
                            larger setups.
   12-APR-2020  B. Ulmann   locate() again.
   26-APR-2020  B. Ulmann   xbar configuration now expects 40 hex nibbles for the production XBAR card with two AD8113.
   27-APR-2020  B. Ulmann   PS-card got an ECO changing its address from $000x to $00Fx to avoid collision with the card in slot 0.
   17-MAY-2020  B. Ulmann   USB interface set to 1 MBit/s - and back, as not being reliable on the Mac.
   08-DEC-2020  B. Ulmann   Rewrote read_adc(...) using Karl-Heinz SPI-implementation as the basis, added init_adc(...).
   09-DEC-2020  B. Ulmann   Some minor cleanups and added comments.
   15-DEC-2020  B. Ulmann   E and F with OVL-Halt enabled and a previous overload only entered OP every second call! Fixed...
*/

/* Port mapping:

     Description  Port   Direction Implemented Tested
     A0-A7        PA0-7  Output    *           *
     A8-A15       PH0-7  Output    *           *
     D0-D7        PK0-7  In/Out    *           *
     Digital IN   PL0-7  Input     *           *
     Digital OUT  PC0-7  Output    *           *
     ModeOP       PE3    Output    *           *
     ModeIC       PE4    Output    *           *
     EXTHALT      PE5    Input     *           *
     /Fotset      PE6    Output    *
     ERR          PE7    Output    *           *
     /CS_ADC      PG1    Output    *           *
     /Read        PG2    Output    *           *
     /Write       PG3    Output    *
     ADC_CNV      PG4    Output    *           *
     /Overload    PG5    Input     *           *
     PB1          SCLK   Output    *           *
```

```
    PB2             MOSI    Output   *          *
    PB3             MISO    Input    *          *
    SYNC            PB6     Output   *          *

  Spare ports:
    PF0-7, PJ0-7, PD2 / PD3 (second UART TX/RX), PB0/SS (Slave select, unused by SPI, should only be used with caution - may have side effects),
    PB4, PB5, PB7, PD4, PD5, PD6, PD7
*/

#undef DEBUG

#define VERSION "v2.1"
#define VERSION_DATE "20201215"

#include <SPI.h>
#include <TimerThree.h>
#include <TimerFive.h>

#ifndef FALSE
# define FALSE 0
# define TRUE !FALSE
#endif

#define PORTC_DEFAULT B00000000   // Default state of PORTC (digital outputs on front panel)
#define PORTE_DEFAULT B11011111   // Default state of PORTE (control pins etc.)
#define PORTG_DEFAULT B00001110   // /CS_ADC = 1, /READ = 1, /WRITE = 1, CNV = 0

// Some useful global states of the analog computer:
#define PORTE_IC      B11001111   // MIC = 0, MOP = 1, PS = 1, ERR = 1
#define PORTE_OP      B11010111   // MIC = 1, MOP = 0, PS = 1, ERR = 1
#define PORTE_HALT    B11011111   // MIC = 1, MOP = 1, PS = 1, ERR = 1
#define PORTE_POTSET  B10011111   // MIC = 1, MOP = 1, PS = 0, ERR = 1

#define OVL_PORT      PING        // Port to which the overload bus line is mapped
#define OVL_BIT       B00100000   // Number of the bit, the overload bus line is mapped to

#define EXT_HALT_PORT PINE        // Port to which the external halt input is mapped
#define EXT_HALT_BIT  B00100000   // Respective bit on that particular port

#define RW_PORT       PORTG       // Caution: The R/W-bits of port G are addressed in a hardcoded fashion in assert/deassert_read/write()!
#define SYNC_PORT     PORTB       // Caution: The SYNC-bit handling is hardcoded in data2potentiometers()!

#define D_OUT   PORTC             // Port used for the front panel digital outputs
#define D_IN    PINL              // Port used for the front panel digital inputs
#define DBUS_IN PINK              // Port used to read from the data bus of the analog computer

#define RAW     0
#define COOKED  1

// Various default values:
#define BAUDRATE            250000          // Baud rate, can be as high as 2 MBit/s but some LINUX distributions don't support this
#define MAX_GROUP_SIZE      100             // Maximum number of addresses in a readout group
#define TIMEOUT             10000           // Timeout in ms for the serial line when reading an integer
#define TIME_DIGITS         6               // All times (IC/OP) will be specified with six digits (leading zeros)
#define PT_HC               8               // We have eight builtin digital potentiometers in the hybrid controller
#define PT_DPT24            24              // ...and 24 DPTs on each DPT24 module
#define PT_VAL_DIGITS       4               // Digits to specify a potentiometer's wiper setting
#define PT_VAL_MASK         0x03ff          // The AD5293 digital potentiometers require 10 bit for a wiper position
#define MAX_DPT_MODULES     16              // A system may accomodate (arbitrarily chosen) up to 16 modules containing digital potentiometers
#define INPUT_BUFF_LEN      MAX_GROUP_SIZE * 5 + 4  // Length of the input buffer for all numeric values
#define READ_DELAY          4               // Delay for readout of module ids (in microseconds)
#define PS_READ_DELAY       1000            // A long delay prior to reading the machine units to make sure everything is a stable as can be
#define MAX_SAMPLES         1152            // Number of 16 bit values usable for logging purposes
#define CONTINGENCY_SAMPLES 128             // This is subtracted from MAX_SAMPLES to avoid problems due to small timer inaccuracies
#define MIN_SAMPLE_INT      50              // Minimum duration of a sample interval in microseconds
#define XBAR_CONF_BYTES     20              // Number of bytes required to configure an XBAR-module - the prototype features one XBAR-chip, thus
                                            // 80 bits, the production module will have two of these chips with 160 bits of configuration data

#define MODULE_ID_HC    0x0008
#define MODULE_ID_DPT24 0x0009
```

```
#define M_UNIT_POS_ADDR 0x00F0                  // Address to read out the positive machine unit
#define M_UNIT_NEG_ADDR 0x00F1                  // Address to read out the negative machine unit
#define M_UNIT_POS_ADDR_OLD 0x0000              // Old addresses to remain compatible with older customer machines - for positive
#define M_UNIT_NEG_ADDR_OLD 0x0001              // and negative reference voltages

// Operating modes of the analog computer
#define MODE_IC     0
#define MODE_OP     1
#define MODE_HALT   2
#define MODE_POTSET 3

// Simple state machine for single/repetitive operation
#define STATE_NORMAL          0
#define STATE_SINGLE_RUN_IC 1
#define STATE_SINGLE_RUN_OP 2
#define STATE_REPETITIVE_IC 3
#define STATE_REPETITIVE_OP 4

// Modes for interrupt control
#define MODE_INACTIVE   -1
#define MODE_ARMED      -2

#define SINGLE_RUN     0
#define REPETITIVE_RUN  1

//*****************************************************************************************
// Global variables
//*****************************************************************************************
struct DPT_MODULE {  // Each module containing DPTs is associated with one of these structures.
  unsigned int number_of_potentiometers, address, module_id, values[PT_DPT24];
};

int mode,                               // Current mode of operation of the analog computer
    sync_mode,                          // If set, single_run (F) will generate a completion message
    module_id;                          // Remember the ID of the module last read by the ADC
unsigned long int op_start = - 1,       // Start of OP-mode in microseconds
                  op_end = -1;          // End of last OP-mode in microseconds
volatile unsigned int ro_group[MAX_GROUP_SIZE],  // Address list for readout group
         ro_group_size = 0;             // Number of entries in the readout group
int16_t machine_unit_pos, machine_unit_neg;  // Remember the machine units for later conversion
volatile int16_t no_of_samples = 0,     // Index of last sample, volatile is required due to the interrupt routine
               samples[MAX_SAMPLES],    // Array for data logging
               state = STATE_NORMAL;    // The state machine is setup for external control
struct DPT_MODULE dpt_modules[MAX_DPT_MODULES];  // Each module containins DPTs is represented by one entry of this array
SPISettings adc_spi(10000000, MSBFIRST, SPI_MODE3); // Set SPI-mode and speed for accessing the ADC

//*****************************************************************************************
//  setup() is called once during startup of the HC module and initializes the various
// components of the system:
//
// - Set the baudrate of the USB interface.
// - Scan the bus for modules such as the HC and DPT24 containing digital potentiometers.
//   Each such interface is associated with a global data structure holding the current
//   configuration bits for the digital potentiometers of this module.
// - Send the Soft Span configuration word to the ADC on the HC board which is done by
//   calling init_adc().
// - Scan the bus for the PS module (this is searched for at the old standard addresses
//   0x0000/0x0001 as at the new address 0x00f0/0x00f1 - the new address is used
//   in all systems with a SYSBUS v. 1.4 while previous systems use the old addresses).
//   The two machine units are then read out and the associated 16 bit values stored in
//   two global variables machine_unit_pos and machine_unit_neg. These values are used
//   in the routine convert_adc2float(...).
// - Initialize Timer3 and Timer5 which are used to control OP time and sampling
//   intervals.
//*****************************************************************************************
void setup() {  // Setup interfaces
  int i;
  char buffer[10];          // General purpose buffer, required to read adc to get module_ids
```

```
Serial.begin(BAUDRATE);     // Initialize serial line interfacen
Serial.setTimeout(TIMEOUT); // The c- and C-commands require an integer as parameter, thus an explicit timeout

DDRA = B11111111;           // Port A is an output port connected to A0-A7 on the bus
DDRB = B01000110;           // PB6 = SYNC (Output), PB3 = MISO (Input), PB2 = MOSI (Output), PB1 = SCLK (Output)
DDRC = B11111111;           // Port C controls the eight digital outputs on the front panel
DDRE = B11011111;           // Port E controls the various bus control lines and LEDs on the front panel
DDRG = B00011110;
DDRH = B11111111;           // Port H is connected to A8-A15 on the bus
DDRL = B00000000;           // Port L is mapped to the eight digital inputs on the front panel

D_OUT = 0;                  // Set all digital outputs to 0
PORTE = PORTE_DEFAULT;
PORTG = PORTG_DEFAULT;
SYNC_PORT |= B01000000;     // Deactivate SYNC for the digital potentiometers
RW_PORT = B00001110;        // N/A, SYNC = 0, N/A, CNV = 0, /WRITE = 1, /READ = 1, /CSADC = 1, N/A

halt();                     // Switch on the HALT-LED to show that the bus scan is running...

// Scan the bus to determine the address of the hybrid controller and the addresses of all optional
// DPT24 modules. These addresses are required to initialize the digital potentiometers of these
// modules and to build the associated data structures.
for (i = 0; i < MAX_DPT_MODULES; i++)
  dpt_modules[i].address = dpt_modules[i].number_of_potentiometers = 0;
i = 0;
for (unsigned int rack = 0; rack < 0x10; rack++) {
  for (unsigned chassis = 0; chassis < 0x10; chassis++) {
    for (unsigned module = 0; module < 0x10; module++) {
      unsigned int address = (rack << 12) | (chassis << 8) | (module << 4);
      write_address(address);               // Address the requested computing element
      PORTG &= B11111011;                   // Set the busline /READ = 0
      delayMicroseconds(READ_DELAY);        // This is necessary to allow the buslines to settle - otherwise spurious module ids will be read
      module_id = DBUS_IN;                  // Remember the ID of the module just read out - this is ugly since module_id is global... *sigh*

      if (module_id == MODULE_ID_HC)
        dpt_modules[i].number_of_potentiometers = PT_HC;  // This is the HC's slot
      else if (module_id == MODULE_ID_DPT24)              // A HC has PT_HC digital potentiometers builtin
        dpt_modules[i].number_of_potentiometers = PT_DPT24;  // We found a DPT24 module and have to remember its address for initialization
      else
        continue;

      dpt_modules[i].address = address;
      dpt_modules[i].module_id = module_id;
#ifdef DEBUG
      Serial.print(String(i) + ": "); Serial.print(dpt_modules[i].address, HEX); Serial.print(": " + String(dpt_modules[i].module_id) + "\n");
#endif

      initialize_dpts(&dpt_modules[i]);  // Initalize all DPTs on this particular board

      i++;
      if (i >= MAX_DPT_MODULES) {
        Serial.print("ERROR: Too many DPT24 modules found!\nMaximum is " + String(MAX_DPT_MODULES) + "\n");
        set_err_led();
        i--;
      }
    }
  }
}

ic();                               // Bus scan complete, switch to initial condition
clear_err_led();

init_adc();                         // Send Soft Span configuration word, this is absolutely necessary for the ADC to work correctly!

machine_unit_pos = read_adc(M_UNIT_POS_ADDR_OLD, PS_READ_DELAY);
if (module_id == 0) {               // module_id is globale, 0 is the id of the PS card
#ifdef DEBUG
  Serial.print("PS card found at old address\n");
```

```
#endif
   machine_unit_neg = read_adc(M_UNIT_NEG_ADDR_OLD, PS_READ_DELAY);
  } else {                          // No PS card at the old address, it must be a new machine
  #ifdef DEBUG
    Serial.print("No PS card found at old address - module_id = " + String(module_id) + "\n");
  #endif
    machine_unit_pos = read_adc(M_UNIT_POS_ADDR, PS_READ_DELAY);
    if (module_id != 0)
      Serial.print("PANIC: PS card neither found at old nor at new address - module_id = " + String(module_id) + "\n");
    machine_unit_neg = read_adc(M_UNIT_NEG_ADDR, PS_READ_DELAY);
  }
  #ifdef DEBUG
  Serial.print("+ = " + String(machine_unit_pos) + ", - = " + String(machine_unit_neg) + "\n");
  #endif
  no_of_samples = 0;                // Clear all data samples gathered previously

  Timer3.stop();
  Timer5.stop();
  Timer5.attachInterrupt(sample_elements);
  Timer3.attachInterrupt(stop_single_run);
}

//*****************************************************************************************
//  The following function must be called once for each DPT-module in order to
//  initialize it and its corresponding datastructure properly.
//*****************************************************************************************
void initialize_dpts(struct DPT_MODULE *data) {
  int i;

  for (i = 0; i < data->number_of_potentiometers; data->values[i++] = 0x1802);          // Set the builtin digital potentiometers to write-enabled
  data2potentiometers(RAW, data->address, data->values, data->number_of_potentiometers);   // Transmit raw data to potentiometers

  for (i = 0; i < data->number_of_potentiometers; data->values[i++] = 0);               // Set all wiper positions to zero
  data2potentiometers(COOKED, data->address, data->values, data->number_of_potentiometers); // Transmit wiper position data to potentiometers
}

//*****************************************************************************************
//  init_adc() initializes the LTC2357 analog digital converter by sending an apropriate
//  Soft Span configuration word with explicit bit banging. This routine must be called
//  once before using the ADC. It is normally called from within setup().
//*****************************************************************************************
int16_t init_adc() {                // Initiate a conversion and readout the result from the ADC (LTC2357)
  uint32_t result;                  // This will hold one result value with the lower 24 bits containing the actual bits read
  int16_t value,                    // Actual value read from the 16 bit ADC
          adccfg;                   // ADC configuration bit mask

  RW_PORT |= B00010000;             // Toggle CNV: First, set CNV = 1, this is independent from /CS
  RW_PORT &= B11101111;             // Reset CNV - these two successive port manipulations yield a high-time of 120 ns on the Mega2650-CORE
  PORTB   |= B00000010;             // SCLK = 1
  RW_PORT &= B11111101;             // Set /CS to low, thus enabling the ADC's serial line interface
  adccfg = 0x70;                    // ADCCONFIG word sets number of channels and SOFTSPAN word
  PORTB &= B11111011;               // Clear MOSI first

  for (int i = 24; i > 0; i--) {    // Read and write 24 bits all in all

    if (adccfg & 0x8000)
      PORTB |= B00000100;           // Set MOSI according to the ADCCONFIG word
    else
      PORTB &= B11111011;

    adccfg <<= 1;

    PORTB &= B11111101;             // Toggle SCLK: SCLK = 0
    PORTB |= B00000010;             // ...and SCLK = 1
  }

  read_adc(M_UNIT_POS_ADDR, READ_DELAY);    // Perform one dummy read - this is necessary only for some (?!) of the LTC2357 chips...
  read_adc(M_UNIT_POS_ADDR, READ_DELAY);    // Perform one dummy read - this is necessary only for some (?!) of the LTC2357 chips...
}
```

```
//*****************************************************************************************
// read_adc(...) reads the output voltage of a computing element at a given address.
// Before using this routine, init_adc() must have been called one (which is done in
// setup()).
//  This routine returns a 16 bit integer value which can be converted to a single
// precision floating point number by calling convert_adc2float(...) subsequently.
//  This routine also sets the global variable module_id to the ID value returned by
// the element addressed on the data bus. Only the lower 7 bits of the module ID read
// are returned.
//*****************************************************************************************
int16_t read_adc(unsigned int address, unsigned int delay) {
  uint8_t rx0, rx1, rx2;

  write_address(address);                 // Address the requested computing element
  assert_read();                          // Activate the /READ bus line
  delayMicroseconds(delay);               // Wait a moment for the signal on readout to settle (long bus line)
  module_id = DBUS_IN & 0x7f;             // Remember the ID of the module just read out - this is ugly since module_id is global... *sigh*
  RW_PORT |= B00010000;                   // Toggle CNV: First, set CNV = 1, this is independent from /CS
  RW_PORT &= B11101111;                   // Reset CNV - these two successive port manipulations yield a high-time of 120 ns on the Mega2650-CORE
  RW_PORT &= B11111101;                   // Set /CS to low, thus enabling the ADC's serial line interface
  delayMicroseconds(2);                   // TODO: This is required only on one of the HCs known today. Does this HC has a too slowly falling /CS edge?

  SPI.beginTransaction(adc_spi);
  rx0 = SPI.transfer(0);                  // Read three single bytes (only the first two contain the actual data,
  rx1 = SPI.transfer(0x70);               // the third one contains the Soft Span word which is checked
  rx2 = SPI.transfer(0);                  // just to be sure everything worked OK).
  SPI.end();
#ifdef DEBUG
  Serial.print("Result = "); Serial.print(rx0, HEX); Serial.print(" "); Serial.print(rx1, HEX); Serial.print(" "); Serial.print(rx2, HEX); Serial.print("\n");
#endif
  RW_PORT |= B00000010;                   // Set /CS to high
  deassert_rw();                          // Deactivate /READ, thus releasing the bus again

  if (rx2 != 0x7) {                       // The third word must contain the Soft Span configuration word which is 0x7 in our case, just be paranoid
    Serial.print("ERROR: Illegal value read from ADC! ");
    Serial.print("Result = "); Serial.print(rx0, HEX); Serial.print(" "); Serial.print(rx1, HEX); Serial.print(" "); Serial.print(rx2, HEX); Serial.print("\n");
    Serial.print("\n");
  }

  return (rx0 << 8) | rx1;                // Combine high and low byte of the result
}

//*****************************************************************************************
//  convert_adc2float(...) converts a 16 bit value read from the ADC to a single precision
// floating point number. Therefore the basis values of the positive and negative machine
// units are required which were already read out during setup().
//*****************************************************************************************
float convert_adc2float(int16_t value) {
  float result;

  if (value >= 0)
    result = (float) value / (float) machine_unit_pos;
  else
    result = -((float) value / (float) machine_unit_neg);

  return result;
}

//*****************************************************************************************
//  write_address(...) writes a 16 bit address to the address bus thus selecting one
// computing element for a further operation.
//*****************************************************************************************
void write_address(unsigned int address) {  // Set the 16 address lines to the address of the computing element to be selected
  PORTA = address & 0xff;                   // Output the eight low bits of the address
  PORTH = (address >> 8) & 0xff;            // Output the upper eight bits
}

//*****************************************************************************************
```

```
// data2potentiometers(...) sends data (16 bit per potentiometer) to daisy-chained
// potentiometers like those contained on the HC and DPT24 modules.
//
// Parameters:
//   mode:       RAW    = transmit data as is, used for mode setup etc.
//               COOKED = transmit wiper data
//   address:    Address of the DPT module on the bus. The local DPTs must be
//               addressed using the HC's own address
//   data[]:     Array holding the data for all daisy-chained potentiometers of
//               board being addressed
//   number_of_pt: Number of entries in the aforementioned array
//*****************************************************************************
void data2potentiometers(int mode, unsigned int address, unsigned int data[], unsigned int number_of_pt) {
  int i;
  unsigned char high, low;

  #ifdef DEBUG
  Serial.print("data2potentiometers: number_of_pt = " + String(number_of_pt) + "\n");
  #endif

  write_address(address & 0xfff0);      // Address the potentiometer group - the four lowest address bits must be 0!
  assert_write();                       // and make sure /WRITE is active so that MOSI/SCKL etc. are gated, this also ensures that /CS = 1

  SPI.begin();
  SPI.setBitOrder(MSBFIRST);            // This is pretty specific to the digital potentiometers used and should be
  SPI.setClockDivider(SPI_CLOCK_DIV2);  // done before any access to make sure that we have the right settings
  SPI.setDataMode(SPI_MODE1);

  SYNC_PORT &= B10111111;               // Initiate transfer and transfer data to the daisy-chained potentiometers, data for the first one must come last!
  for (i = number_of_pt - 1; i >= 0; i--) {
    high = (data[i] >> 8) & 0xff;
    low  = data[i] & 0xff;

    if (mode != RAW) {                  // Transfer wiper data and update wiper position
      high &= 0x3;
      low  &= 0xff;
      high |= 0x4;
    }

    SPI.transfer(high);
    SPI.transfer(low);
  }

  SYNC_PORT |= B01000000;               // Let the potentiometers know that the data upload is finished
  SPI.end();
  deassert_rw();                        // Deassert /WRITE
}

//*****************************************************************************
// data2xbar(...) sends a configuration bit stream to the XBAR module at the address
// specified in the first parameter.
//*****************************************************************************
void data2xbar(unsigned int address, unsigned char *data) {
  int i;

  write_address(address & 0xfff0);      // Address the potentiometer group - the four lowest address bits don't matter
  assert_write();                       // and make sure /WRITE is active so that MOSI/SCKL etc. are gated, this also ensures that /CS = 1

  SPI.begin();
  SPI.setBitOrder(MSBFIRST);            // This is pretty specific to the digital potentiometers used and should be
  SPI.setClockDivider(SPI_CLOCK_DIV2);  // done before any access to make sure that we have the right settings
  SPI.setDataMode(SPI_MODE2);           // The AD5113 runs in SPI-mode 2

  SYNC_PORT |= B01000000;               // Load the shift register

  for (i = 0; i < XBAR_CONF_BYTES; i++) {
    SPI.transfer(data[i]);
  }
```

```
  SYNC_PORT &= B10111111;             // Activate the new configuration loaded into the shift register
  SPI.end();
  deassert_rw();                      // Deassert /WRITE
}

//*****************************************************************************************
// sample_elements() samples data from all computing elements defined in the current
// readout-group.
// It is called by a Timer1-interrupt at regular intervals. The data can then be sent
// to the attached digital computer by the 'l' command.
//*****************************************************************************************
void sample_elements() {
  int i;
  uint32_t value;

  if (mode != MODE_OP)  // Reading micros() at the beginning of op() takes too much time
    return;             // Enabling the timer in op() would be to slow, too, so let's make
                        // sure here that we have reached OP_MODE already.

  for (int i = 0; i < ro_group_size; i++) { // Read all elements defined in the readout-group
    samples[no_of_samples++] = read_adc(ro_group[i], READ_DELAY);
#ifdef DEBUG
    Serial.print(String(no_of_samples) + ": " + String(samples[no_of_samples - 1]) + " ");
#endif
    if (no_of_samples == MAX_SAMPLES) {
      Serial.print("Sampling buffer overflow!\n");
      no_of_samples = 0;
    }
  }
#ifdef DEBUG
  Serial.print("\n");
#endif
}

//*****************************************************************************************
// ic() switches the analog computer into initial condition mode.
//*****************************************************************************************
void ic() {
  op_end = micros();
  PORTE = PORTE_IC;
  mode = MODE_IC;
}

//*****************************************************************************************
// op() switches the analog computer into operate mode.
//*****************************************************************************************
void op() {
  op_start = micros();  // Remember start time to compute delta-t later
  PORTE = PORTE_OP;
  mode = MODE_OP;
}

//*****************************************************************************************
// halt() switches the analog computer into halt mode.
//*****************************************************************************************
void halt() {
  op_end = micros();
  PORTE = PORTE_HALT;
  mode = MODE_HALT;
}

//*****************************************************************************************
// ps() switches the analog computer into potentiometer set mode.
//*****************************************************************************************
void ps() {
  PORTE = PORTE_POTSET;
  mode = MODE_POTSET;
}
```

```
//*************************************************************************************
// set_err_led() switches the error LED on.
//*************************************************************************************
void set_err_led() {
  PORTE &= B01111111;
}

//*************************************************************************************
// clear_err_led() switches the error LED off.
//*************************************************************************************
void clear_err_led() {
  PORTE |= B10000000;
}

//*************************************************************************************
// clear_digital_output() clears a specific digital output line (0 to 7).
//*************************************************************************************
void clear_digital_output(unsigned int value) { // Clear a digital output line
  if (value > 7) {
    set_err_led();
    Serial.print("ERR\n");
  } else
    D_OUT &= ~(1 << value) & 0xff;
}

//*************************************************************************************
// set_digital_output() sets a specific digital output line (0 to 7).
//*************************************************************************************
void set_digital_output(unsigned int value) {   // Set a digital output line
  if (value > 7) {
    set_err_led();
    Serial.print("ERR\n");
  } else
    D_OUT |= 1 << value;
}

//*************************************************************************************
// assert_read() activates the /READ control line on the system bus.
//*************************************************************************************
void assert_read() {    // Activate /READ only
  deassert_rw();
  RW_PORT &= B11111011;
}

//*************************************************************************************
// assert_write() activates the /WRITE control line on the system bus.
//*************************************************************************************
void assert_write() {   // Activate /WRITE only
  deassert_rw();
  RW_PORT &= B11110111;
}

//*************************************************************************************
// deassert_rw() deactivates the /READ and /WRITE control lines on the system bus.
//*************************************************************************************
void deassert_rw() {
  RW_PORT |= B00001100; // Set /WRITE and /READ to 1 - the interlock logic then suppresses the signals altogether
}

//*************************************************************************************
// stop_single_run() is controlled by TimerThree and stops the OP-portion of a
// single-run when its run time is over.
//*************************************************************************************
void stop_single_run() {
  static int first_call = TRUE;

  if (state != STATE_SINGLE_RUN_OP) // If this is true, the routine has been called spuriously,
    return;                         // which should be ignored.
```

```
#ifdef DEBUG
  Serial.print("F=" + String(first_call) + "\nState = " + String(state) + "\n");
#endif
  if (first_call) // The routine is called twice - once at the beginning of a timer period and once at the end, only the last one is relevant
    first_call = FALSE;
  else {
    Timer3.stop();          // Stop the OP-interval timer
    Timer5.stop();          // Stop data gathering
    state = STATE_NORMAL;
    first_call = TRUE;      // Prepare for next call to this routine

    halt();
    if (sync_mode) {
      Serial.print("EOSR\n"); // End of single-run reached
      sync_mode = FALSE;
    }
  }
}

//*****************************************************************************************
// loop() contains the main control logic of the HC module firmware. This is the place
// where commands are read and interpreted and results are sent back to the attached
// digital computer.
//*****************************************************************************************
void loop() {
  static int enable_ovl_halt = FALSE,       // Halt on overload is disabled by default
             enable_ext_halt = FALSE;       // as is halt on external input
  static unsigned long op_time = 0,         // OP-time in microseconds
                       ic_time = 0,         // IC-time in microseconds
                       now,                 // Current time in microseconds, used for single/repetitive run
                       sample_interval;     // Length of sample interval in microseconds
  unsigned int value, i, j,
               total_samples;               // Number of samples which can be logged during a single-run
  int16_t address;
  int index, potentiometer_address;
  float result,
        ro_group_values[MAX_GROUP_SIZE];
  char input[INPUT_BUFF_LEN], cmd, buffer[10],  // Character buffer for hexadecimal addresses
       *p;                                      // Scratch pointer
  unsigned char xbar_buffer[XBAR_CONF_BYTES];   // Buffer holding the configuration bit stream for an XBAR module

  for (;;) { // This is a tad faster than relying on loop being called repeatedly
    // Take care of halt conditions - this has to be done before any single/repetitive run logic!
    if (mode == MODE_OP) { // Either halt condition only has an effect if the current analog computer mode is OP
      if (!(OVL_PORT & OVL_BIT) && enable_ovl_halt) {
        if (state == STATE_SINGLE_RUN_OP)
          stop_single_run();
        else
          halt();
        state = STATE_NORMAL;
        Serial.print("\tOverload halt!\n");
      }
      else if ((EXT_HALT_PORT & EXT_HALT_BIT) && enable_ext_halt) {
        halt();
        if (state == STATE_SINGLE_RUN_OP && sync_mode)
          Serial.print("EOSRHLT\n");
        state = STATE_NORMAL;
      }
    }

    // Take care of single and repetitive runs
    if (state == STATE_SINGLE_RUN_IC) {
      if (micros() - now >= ic_time) { // End of IC-period reached
        state = STATE_SINGLE_RUN_OP;

        if (ro_group_size) { // Activate logging if a readout-group has been defined
          no_of_samples = 0;                    // Clear all data samples gathered previously
          total_samples = (MAX_SAMPLES - CONTINGENCY_SAMPLES) / ro_group_size;
          sample_interval = op_time / total_samples; // Delta-t for the interrupt routine gathering data
```

```
    if (sample_interval < MIN_SAMPLE_INT)      // Make sure we do not sample way too often thus avoiding
      sample_interval = MIN_SAMPLE_INT;        // potential problems with updating micros() etc.
  #ifdef DEBUG
      Serial.print("Total samples = " + String(total_samples) + ", Delta-t = " + String(sample_interval) + "\n");
  #endif
      Timer5.initialize(sample_interval);
      Timer5.restart();
    }

    now = micros(); // Remember start of OP-period
    op(); // This takes a tick :-) for reading out micros() to determine run time, so let's start the interrupt after calling it
    Timer3.initialize(op_time); // Prepare time to end the OP-interval
    Timer3.restart();
  }
 }
 else if (state == STATE_SINGLE_RUN_OP) {  // This became obsolete due to the interrupt controlled routine stop_single_run()
 }
 else if (state == STATE_REPETITIVE_IC) {
  if (micros() - now >= ic_time) {
    state = STATE_REPETITIVE_OP;
    now = micros();
    op();
  }
 }
 else if (state == STATE_REPETITIVE_OP) {
  if (micros() - now >= op_time) {
    state = STATE_REPETITIVE_IC;
    now = micros();
    ic();
  }
 }

 if (Serial.available() > 0) { // If there is anything on the serial line, read and process it
   clear_err_led();
   switch (cmd = Serial.read()) {
     case 'a': // Disable halt on overload
       enable_ovl_halt = FALSE;
       Serial.print("OVLH=DISABLED\n");
       break;
     case 'A': // Enable halt on overload
       enable_ovl_halt = TRUE;
       Serial.print("OVLH=ENABLED\n");
       break;
     case 'b': // Disable external halt
       enable_ext_halt = FALSE;
       Serial.print("EXTH=DISABLED\n");
       break;
     case 'B': // Enable external halt
       enable_ext_halt = TRUE;
       Serial.print("EXTH=ENABLED\n");
       break;
     case 'c': // Set OP-time, format: c\d{6}, time is in milliseconds
       input[Serial.readBytesUntil('\n', input, TIME_DIGITS)] = 0;
       op_time = strtol(input, 0, 10) * 1000;
       Serial.print("T_OP=" + String(op_time / 1000) + "\n");
       break;
     case 'C': // Set IC-time, format: C\d{6}, time is in milliseconds
       input[Serial.readBytesUntil('\n', input, TIME_DIGITS)] = 0;
       ic_time = strtol(input, 0, 10) * 1000;
       Serial.print("T_IC=" + String(ic_time / 1000) + "\n");
       break;
     case 'd': // Clear digital output, format: d[0-7]
       while (Serial.available() <= 0); // Blocking IO
       clear_digital_output(Serial.read() - '0');
       break;
     case 'D': // Set digital output, format: D[0-7]
       while (Serial.available() <= 0); // Blocking IO
       set_digital_output(Serial.read() - '0');
       break;
```

```
case 'e': // Start repetitive operation
  Serial.print("REP-MODE\n");
  state = STATE_REPETITIVE_IC;
  now = micros();
  ic();
  break;
case 'E': // Start single IC/OP-cycle
  Serial.print("SINGLE-RUN\n");
  state = STATE_SINGLE_RUN_IC;
  now = micros();
  ic();
  break;
case 'F': // Start single IC/OP-cycle with completion message to synchronize with the Perl library
  Serial.print("SINGLE-RUN\n");
  state = STATE_SINGLE_RUN_IC;
  now = micros();
  ic();
  sync_mode = TRUE;
  break;
case 'f': // Fetch values for all elements defined in a readout group
  noInterrupts();
  for (int i = 0; i < ro_group_size; i++) // 1st, we read all elements to minimize jitter
    ro_group_values[i] = convert_adc2float(read_adc(ro_group[i], READ_DELAY));

  for (int i = 0; i < ro_group_size; i++) { // 2nd, we output the values
    dtostrf(ro_group_values[i], 4, 4, buffer);
    Serial.print(buffer);
    if (i < ro_group_size - 1)
      Serial.print(";");
  }
  Serial.print("\n");
  interrupts();
  break;
case 'G': // Define a readout group consisting of hex addresses delimited by semicola
  ro_group_size = 0;
  input[Serial.readBytesUntil(',', input, MAX_GROUP_SIZE * 5)] = 0;
  p = strtok(input, ";");
  while (p) {
    ro_group[ro_group_size++] = strtol(p, 0, 16);
    p = strtok(NULL, ";");
  }

  no_of_samples = 0;  // After defining a new readout-group there is no way to use old sampled data
  break;
case 'g': // Set address of a computing element and return its ID and associated value, format: g\x{4}
  input[Serial.readBytesUntil('\n', input, 4)] = 0;
  address = strtol(input, 0, 16);
  result = convert_adc2float(read_adc(address, READ_DELAY));
  dtostrf(result, 4, 4, buffer);
  Serial.print(buffer);
  Serial.print(" " + String(module_id) + "\n");
  break;
case 'h': // Set analog computer to halt
  Serial.print("HALT\n");
  state = STATE_NORMAL;  // Make sure to kill any active single/repetitive run
  halt();
  break;
case 'i': // Set analog computer to initial condition
  Serial.print("IC\n");
  state = STATE_NORMAL;  // Make sure to kill any active single/repetitive run
  ic();
  break;
case 'L': // Locate a computing element
  input[Serial.readBytesUntil('\n', input, 4)] = 0;          // Now read the element's address
  address = strtol(input, 0, 16);
  write_address(address);                                    // Address the requested computing element
  if (address == 0xffff)
    PORTG |= B00000100;                                      // Deassert /READ
  else
```

```
  PORTG &= B11111011;                     // Set the busline /READ = 0, thus turning the read LED on
  break;
case 'l': // Dump samples from last single-run
  if (!ro_group_size || !no_of_samples)
    Serial.print("No data!\n");
  else {
    for (i = 0; i < no_of_samples; i++) {
      dtostrf(convert_adc2float(samples[i]), 4, 4, buffer);
      Serial.print(buffer); Serial.print(" ");
      if (ro_group_size == 1 || (i + 1) % ro_group_size)))
        Serial.print("\n");
    }
    Serial.print("EOD\n");
  }
  break;
case 'o': // Set analog computer to operate mode
  Serial.print("OP\n");
  state = STATE_NORMAL;  // Make sure to kill any active single/repetitive run
  op();
  break;
case 'P': // Set a digital potentiometer to a value, format: P\x{4}\x{2}\d{4}
  input[Serial.readBytesUntil('\n', input, 4)] = 0;          // Read four hex digits for the potentiometer card's address
  address = strtol(input, 0, 16);
  input[Serial.readBytesUntil('\n', input, 2)] = 0;          // Now read the number of the potentiometer on the card to be set. Since some
  potentiometer_address = strtol(input, 0, 16);              // cards can have > 16 potentiometers, this cannot be part of the card's address!
  input[Serial.readBytesUntil('\n', input, PT_VAL_DIGITS)] = 0; // Read the value the potentiometer should be set to - this is a decimal
  value = strtol(input, 0, 10);                              // value - maybe I should rewrite everything and use hexadecimal consistently.

  index = -1;
  for (i = 0; i < MAX_DPT_MODULES; i++) {                    // Determine the index to the dpt_modules structure array
    if (dpt_modules[i].address == address) {
      index = i;
      break;
    }
  }

  if (index == -1) {  // No module containing digital potentiometers at this address found!
    Serial.print("P"); Serial.print(address, HEX); Serial.print(","); Serial.print(potentiometer_address, HEX); Serial.print("=ERROR!\n");
  } else {
    dpt_modules[index].values[potentiometer_address] = value & PT_VAL_MASK;
    data2potentiometers(CODED, dpt_modules[i].address, dpt_modules[index].values, dpt_modules[index].number_of_potentiometers); // Send wiper position data, not raw data!
    Serial.print("P"); Serial.print(address, HEX); Serial.print(","); Serial.print(potentiometer_address, HEX); Serial.print("=");
    Serial.print(String(dpt_modules[i].values[potentiometer_address]) + "\n");
  }
  break;
case 'q': // Dump digital potentiometer settings
  for (i = 0; i < MAX_DPT_MODULES; i++) {
    if (!dpt_modules[i].address) break; // The first address of 0 terminates the list of DPT-modules
    if (i > 0) Serial.print(";");
    Serial.print(dpt_modules[i].address, HEX);
    Serial.print(":");
    for (int j = 0; j < dpt_modules[i].number_of_potentiometers; j++) {
      if (j > 0) Serial.print(",");
      Serial.print(dpt_modules[i].values[j]);
    }
  }
  Serial.print("\n");
  break;
case 'R': // Read digital inputs and return their associated values
  value = D_IN;
  for (i = 0; i < 8; i++) Serial.print(String((value >> i) & 1) + " ");
  Serial.print("\n");
  break;
case 's': // Print system status
  Serial.print("STATE=");
  switch (state) {
    case STATE_NORMAL:
      Serial.print("NORM");
      break;
```

```
    case STATE_SINGLE_RUN_IC:
      Serial.print("SR-IC");
      break;
    case STATE_SINGLE_RUN_OP:
      Serial.print("SR-OP");
      break;
    case STATE_REPETITIVE_IC:
      Serial.print("REP-IC");
      break;
    case STATE_REPETITIVE_OP:
      Serial.print("REP-OP");
  }
  Serial.print(",+i=" + String(convert_adc2float(machine_unit_pos)) + ",-i=" + String(convert_adc2float(machine_unit_neg)));
  Serial.print(",\
MODE=" + (mode == MODE_IC ? String("IC") : (mode == MODE_OP ? String("OP") : String("HALT"))) + ",\
EXTH=" + (enable_ext_halt ? String("ENA") : String("DIS")) + ",\
OVLH=" + (enable_ovl_halt ? String("ENA") : String("DIS")) + ",\
IC-time=" + String(ic_time / 1000) + ",\
OP-time=" + String(op_time / 1000) + ",RO-GROUP=");
  for (int i = 0; i < ro_group_size; i++) {
    Serial.print(ro_group[i], HEX);
    if (i < ro_group_size - 1)
      Serial.print(";");
  }
  Serial.print(",DPTADDR=");
  for (int i = 0; i < MAX_DPT_MODULES; i++) {
    if (!dpt_modules[i].address) break; // The first address of 0 terminates the list of DPT-modules
    if (i > 0) Serial.print(";"); // Column separator if this is not the first column
    Serial.print(dpt_modules[i].address, HEX);
    Serial.print("/" + String(dpt_modules[i].module_id));
  }
  Serial.print("\n");
  break;
case 'S': // Switch the analog computer to POTSET-mode
  Serial.print("PS\n");
  state = STATE_NORMAL;  // Make sure to kill any active single/repetitive run
  ps();
  break;
case 't': // Print current time spent in OP-mode
  Serial.print("t_OP=");
  if (op_start == -1)     // We were never in OP-mode, so there is nothing to display
    Serial.print("N/A");
  else if (mode == MODE_OP)  // The analog computer is currently in OP-mode, so time is still running
    Serial.print(String((micros() - op_start) / 1000));
  else                       // We are no longer in OP-mode, use the stored end-time instead of current time
    Serial.print(String((op_end - op_start) / 1000));
  Serial.print("\n");
  break;
case 'x': // Reset hybrid controller
  setup();
  ro_group_size = 0;
  Serial.print("RESET\n");
  break;
case 'X': // Configure an XBAR-module
  input[Serial.readBytesUntil('\n', input, 4)] = 0; // Read four hex digits for the XBAR-module's address
  address = strtol(input, 0, 16);

  for (i = 0; i < XBAR_CONF_BYTES; i++) {          // Read configuration data in pairs of hex nibbles and store it
    input[Serial.readBytesUntil('\n', input, 2)] = 0;
    xbar_buffer[i] = strtol(input, 0, 16);
  }

  data2xbar(address, xbar_buffer);
  Serial.print("XBAR READY\n");
  break;
case '?': // Print help
  Serial.print("HC firmware " + String(VERSION) + " " + String(VERSION_DATE));
  Serial.print("\n\nCommands:\n\
    Disable Halt-on-overflow\n\
a
```

```
A       Enable halt-on-overflow\n\
b       Disable external halt\n\
B       Enable external halt\n\
c\\d{6}   Set OP time for repetitive/single operation\n\
C\\d{6}   Set IC time for repetitive/single operation\n\
d[0-7]  Clear digital output\n\
D[0-7]  Set digital output\n\
e       Start repetitive operation\n\
E       Start single IC/OP-cycle\n\
f       Get (fetch) data for a readout group defined by an address list via G\n\
F       Start single IC/OP-cycle with completion message (for sync operation)\n\
g\\x{4}   Set address of computing element and return its ID and value\n\
G\\w      Set addresses \\x;\\x. for group readout, '.' terminates the input\n\
h       Halt\n\
i       Initial condition\n\
l       Get data from last logging operation\n\
L\\x{4}   Locate a computing element by its 4 digit address by turning on the read LED\n\
          An address of ffff will deassert read.\n\
o       Operate\n\
P\\x{4}\\x{2}\\d{4} Set the digital potentiometer number \\x{2}on the card with\n\
          address \\x{4} to value \\d{4}\n\
q       Dump digital potentiometer settings\n\
R       Read digital inputs\n\
s       Print status\n\
S       Switch to PotSet-mode\n\
t       Print elapsed OP-time\n\
x       Reset\n\
X\\x{4}\\x{20} Send a configuration bitstream (40 hex nibbles) to the XBAR-module at address \\x{4}\n\
?       Print Help\n\n");
        break;
      default:  // Illegal command
        set_err_led(1);
        Serial.print("Illegal command: ");
        Serial.print(cmd, HEX);
        Serial.print("\n");
      }
    }
  }
}
```

Bibliography

[Ulmann(2020)] Bernd Ulmann, *Analog and hybrid computer programming*, DeGruyter, 2020