0. Overview

author: Lutz Rossa
version: 0.2 of 2024-08-17

EPICS support for Measurement Computing Corporation daq hats for Raspberry Pi

vendor of hats:

- Measurement Computing Corporation <https://www.mccdaq.com/>
- now Digilent (part of Emerson) <https://digilent.com/>

possible types:

- MCC118: 8-ch 12-bit 100kS/s analog input, single ended -10V...+10V, common clock input/output, common trigger input
- MCC128: 8-ch 16-bit 100kS/s analog input, single ended or 4-ch 16-bit 100kS/s analog input, differential, -10V...+10V, -5V...+5V, -2V...+2V, -1V...+1V, common clock input/output, common trigger input
- MCC134: 4-ch 24-bit thermocouple input
- MCC152: 2-ch 12-bit analog output 0V...5V, 8-ch bit-configureable digital I/O
- MCC172: 2-ch 24-bit 51.2kS/s analog input, differential -5V...+5V, common trigger input

1. Requirements

1.1. Raspberry PI

This should be self-explaining for intended use of the hardware. If possible, install an operating system, which is supported by EPICS (see next chapter). Raspberry-Pi-OS (Linux) is recommended. Additionally these packets should be installed on this Linux system:

- `git`
- g++ (GNU C++ compiler)
- `make` (GNU make)
- optional `libreadline-dev` (GNU readline and history)

This documentation often says to use `make`, which means to build the sources in the directory with a terminal and using a single command line

```
make
```

. You could build the sources in different ways, if you know how. The output should not show any errors while compiling or linking output files.

1.2. required MCC library

Please clone this MCC library with `git` into a separate directory and build it with `make`. The library comes with a configuration tool `daqhats_read_eeproms`, which should be used to read the installed modules and store some required information.

1.3. required EPICS

EPICS documentation could be found here . You should clone the repositories of base and asyn with `git` into separate directories, called *<base>* and *<asyn>* here. To configure EPICS base, optionally modify file

<base>/configure/CONFIG_SITE

or put a local file

<base>/configure/CONFIG_SITE.local

with a single line

BUILD_IOCS=YES

and build it with `make`.

To configure the asyn package, modify the file

<asyn>/configure/RELEASE

or put a local file

<base>/configure/RELEASE.local

and point to EPICS base with a single line (absolute directory path)

EPICS_BASE=<base>

and build it with `make`.

2. MCC DAQ hats support module

2.1. Cloning and configuring

Please clone the support module with `git` into a separate directory called *<mccdaqhats>* here. To configure the support module, modify the file

<mccdaqhats>/configure/RELEASE

or put a local file

<mccdaqhats>/configure/RELEASE.local

and point to EPICS base with these lines (absolute directory paths)

EPICS_BASE=<base>

ASYN=<asyn>

.

Please open the file `<mccdaqhats>/iocBoot/iocmccdaqhats/st.cmd` in your favorite text editor and go to the line starting with

```
dbLoadRecords("generated.db","P=pi,
```

and modify substition `P=pi` (which is used as prefix) into another EPICS prefix in your EPICS set up, e.g. `PI=myprefix` or something better.

2.2. Compiling

Execute `make` in the directory *<mccdaqhats>*, which should not produce an error message. The build process should generated these directories/files:

- file `<mccdaqhats>/bin/linux-arm/mccdaqhats` : generated IOC
- directory `<mccdaqhats>/dbd/` with EPICS description(s) of possibile fields
- file `<mccdaqhats>/iocBoot/iocmccdaqhats/envPaths` : absolute path information
- possibly more directories and files, which are not important here.

Use the command `make distclean` to remove most of generated files.

2.3. Using

The file `<mccdaqhats>/iocBoot/iocmccdaqhats/st.cmd` should contain an example, how to automatically find and use all installed MCC hats.

There are two ways to start the IOC:

1. Make the file `<mccdaqhats>/iocBoot/iocmccdaqhats/st.cmd` as executable

   ```
   chmod +x <mccdaqhats>/iocBoot/iocmccdaqhats/st.cmd
   ```

   go into the same directory

   ```
   cd <mccdaqhats>/iocBoot/iocmccdaqhats/
   ```

   and start it with

   ```
   ./st.cmd
   ```

2. Go into the same directory

   ```
   cd <mccdaqhats>/iocBoot/iocmccdaqhats/
   ```

   and start the IOC with the command line

   ```
   <mccdaqhats>/bin/linux-arm/mccdaqhats st.cmd
   ```

The IOC is started and reads the file `st.cmd` as input, which triggers these important functions:

1. The line with `envPaths` reads the absolute paths, which are used here.
2. The function `dbLoadDatabase` reads and prepares the database definitions.
3. The function `mccdaqhats_registerRecordDeviceDriver` initializes the support, that next commands are useable.
4. The function `mccdaqhatsInitialize` initializes the required MCC library and searches for / reads all installed modules and generates an internal parameter list.
5. The function `mccdaqhatsWriteDB` writes the internal parameter list into a file suitable as EPICS database.
6. The function `dbLoadRecords` reads this freshly generated EPICS database and generates EPICS PVs for them, here the prefix is important for using unique EPICS PV names.
7. The function `iocInit` starts the IOC.

No error message should appear and a prompt as last line. A command `dbl` could show existing EPICS PVs. See EPICS documentation for more help.

3. Automatically generated parameters

3.1. Naming scheme

The support is using asyn module. Every parameter has its unique name starting with *MCC_* followed by a module address *A<n>_* and a suffix described in the next chapters. The module address depends on your hardware setup, the first has the address 0 and is directly attached on the Raspberry Pi. A maximum of 8 hats are supported, so the address could be 0 to 7.

An example for a parameter could be *MCC_A0_C1*, which means the 2nd channel of the first module. The function `mccdaqhatsWriteDB` writes a substitution prefix `$(P):` to every parameter, so the EPICS PV name (after loading with `dbLoadRecords` and `P=example` for the example parameter above) would result in

```
example:MCC_A0_C1
```

3.2. MCC118 analog input

name	dir	type	description
C0 ... C7	R	float32[]	channel values as *aai* array
SLOPE0 ... SLOPE7	R	float32	used EEPROM correction factor
OFFSET0 ... OFFSET7	R	float32	used EEPROM correction offset
START	RW	enum	acquisition state: 0=stop, 1=start
MASK	RW	uint8	channel selection bit mask: bit0=C0 .. bit7=C7
TRIG	RW	enum	trigger mode: 0=none, 1=rising, 2=falling, 3=high, 4=low
RATE	RW	float	ADC clock rate selection

In stopped state, you have to set up first *MASK*, *TRIG* and *RATE*. Set bits in *MASK* enable the specified channel, cleared bits disable the specified channel. The *TRIG* trigger mode tells the hardware to optionally wait for the specified signal on the *TRG* input. The *RATE* as positive value selects to use the internal clock, 0 or negative value means an external clock on *CLK* input and a negative value gives a hint, what clock frequency could be expected.

Setting *START* to 1/*start* starts the data acquisition until *START* is set to 0/*stop*. The support will set an alarm, if some settings is wrong, e.g. a wrong sample rate (max. 100kHz on one channel, max. 12.5kHz with 8 channels).

While in acquisition mode, only *START* could be set to 0/*stop* to stop the data acquisition.

3.3. MCC128 analog input

name	dir	type	description
C0 ... C7	R	float32[]	channel values as *aai* array
SLOPE0 ... SLOPE3	R	float32	used EEPROM correction factor for every analog input range
OFFSET0...OFFSET3	R	float32	used EEPROM correction offset for every analog input range
START	RW	enum	acquisition state: 0=stop, 1=start
MASK	RW	uint8	channel selection bit mask: bit0=C0 ... bit3=C3/bit7=C7
TRIG	RW	enum	trigger mode: 0=none, 1=rising, 2=falling, 3=high, 4=low
MODE	RW	enum	analog input mode: 0=single-ended, 1=differencial
RANGE	RW	enum	analog input range selection: 0=10V, 1=5V, 2=2V, 3=1V
RATE	RW	float	ADC clock rate selection

In stopped state, you have to set up first *MODE*, *RANGE*, *MASK*, *TRIG* and *RATE*. *MODE* selects, how many channels are available: in single-ended mode, all 8 channels could be used. In differential mode, two inputs are a pair of differential inputs, so 4 channels are available. *RANGE* sets the allowed input voltage range (lower voltages have higher resolution). Set bits in *MASK* enable the specified channel, cleared bits disable the specified channel. The *TRIG* trigger mode tells the hardware to optionally wait for the specified signal on the *TRG* input. The *RATE* as positive value selects to use the internal clock, 0 or negative value means an external clock on *CLK* input and a negative value gives a hint, what clock frequency could be expected.

Setting *START* to 1/*start* starts the data acquisition until *START* is set to 0/*stop*. The support will set an alarm, if some settings is wrong, e.g. a wrong sample rate (max. 100kHz on one channel, max. 12.5kHz with 8 channels).

While in acquisition mode, only *START* could be set to 0/*stop* to stop the data acquisition.

3.4. MCC134 thermocouple input

name	dir	type	description
C0 ... C3	R	float32	channel value
CJC0 ... CJC3	R	float32	cold junction compensation value
TCTYPE0...TCTYPE3	RW	enum	thermocouple type: 0=disabled, 1=J, 2=K, 3=T, 4=E, 5=R, 6=S, 7=B, 8=N
SLOPE	R	float32	used EEPROM correction factor
OFFSET	R	float32	used EEPROM correction offset
RATE	RW	uint8	update interval in seconds

The thermocouple will update every input with the specified *RATE*, if the *TCTYPE* of the channel is not 0/*disabled*.

3.5. MCC152 analog output, digital I/O

name	dir	type	description
C0 ... C1	R	float32	analog output channel values
DI	R	uint8	digital input state bit mask
DO	RW	uint8	digital output state bit mask, bits configured as input will be ignored
DIR	RW	uint8	digital I/O direction bit mask: cleared bit means output, set bit means input
IN_PULL_EN	RW	uint8	digital input pull resistor enable bit mask
IN_PULL_CFG	RW	uint8	digital input pull direction bit mask: 0=pull-down, 1=pull-up
IN_INV	RW	uint8	digital input inversion bit mask
IN_LATCH	RW	uint8	digital input latch enable bit mask
OUT_TYPE	RW	uint8	digital output configuration bit mask: 0=push/pull, 1=open-drain

Any write to analog or digital outputs will be directly written to hardware. The digital inputs are configured as interrupt, so any change should be detected in short time and distributed to PVs.

3.6. MCC172 analog input

name	**dir**	**type**	**description**
C0 ... C1	R	float32[]	channel values as *aai* array
SLOPE0 ... SLOPE1	R	float32	used EEPROM correction factor
OFFSET0...OFFSET1	R	float32	used EEPROM correction offset
IEPE0 ... IEPE1	RW	enum	IEPE sensor power: 0=off, 1=on
START	RW	enum	acquisition state: 0=stop, 1=start
MASK	RW	uint8	channel selection bit mask: bit0=C0, bit1=C1
TRIG	RW	enum	trigger mode: 0=none, 1=rising, 2=falling, 3=high, 4=low
CLKSRC	RW	enum	clock source: 0=local, 1=master, 1=slave
RATE	RW	float	ADC clock rate selection

In stopped state, you have to set up first *IEPE*, *MASK*, *TRIG*, *CLKSRC* and *RATE*. *IEPE* enables or disables sensor power on this channel. Set bits in *MASK* enable the specified channel, cleared bits disable the specified channel. The *TRIG* trigger mode tells the hardware to optionally wait for the specified signal on the *TRG* input. The *CLKSRC* selects and configures the *CLK* pin : the *local* mode ignores it, *master* configures it as output for other modules, which have to be configured as *slave*. The *RATE* is used on local and master mode and selects the acquisition frequency.

Setting *START* to 1/*start* starts the data acquisition until *START* is set to 0/*stop*. The support will set an alarm, if some settings is wrong.

While in acquisition mode, only *START* could be set to 0/*stop* to stop the data acquisition. *IEPE* is also writeable.