Simple ADC example `adc_basic`

1 Overview

The simplest implementation of analog to digital conversion is measuring the voltage of a single channel. The measurement is initiated by the software, and the ADC peripheral is continually polled by the software until the conversion finishes. It is a simple method and easy to understand, but it has a few drawbacks:

1. The timing cannot be made precise because the main loop is generally not timing sensitive.
2. During the conversion, the main loop waits until the conversion is completed. It wastes processor time.

However, it is a good preliminary example. Once the code is running, open a serial port terminal to see its output. When you press a key on your PC, Alakart starts conversion of the voltage applied to ADC0 Channel 1 which is labeled as P06. When it completes, the result value is written back on the serial terminal. Note that the serial terminal communication speed must be set to 115200 baud. You can apply a voltage to `PIO0_6` pin by uing your potentiometer.

- Connect the viper (middle pin) of the potentiometer to Alakart `P06` pin
- Connect one of the outermost pins of the potentiometer to Alakart `GND`
- Ant the other outermost pin to Alakart `3V3`
- **CAUTION!**: Never use the Alakart `5V` pin as **it will damage the processor!**

2 The Setructure of the Program

The program will be detailed below. The general flow is as follows:

1. In `main()`, the processor pins, clocks and peripherals such as UART (for serial terminal) are initialized.

```
BOARD_InitPins();
BOARD_InitBootClocks();
CLOCK_EnableClock(kCLOCK_Uart0);     // Enable UART0 clock
CLOCK_SetClkDivider(kCLOCK_DivUsartClk, 1U);// Set UART0 clock divider
BOARD_InitDebugConsole();
```

2. ADC0 clock is enabled and initialized.

```
CLOCK_EnableClock(kCLOCK_Adc);       // Enable ADC clock
POWER_DisablePD(kPDRUNCFG_PD_ADC0); // Power on ADC0
```

3. ADC0 self calibration is performed. This is completely done by the hardware and it must be performed after each device reset. See Sec. 21.3.4 “Hardware self-calibration” for details.

```
    if (true == ADC_DoSelfCalibration(ADC0, frequency)) {
    ...
```

4. ADC0 configuration function is called.

```
ADC_Configuration(&ADCResultStruct);
```

5. Finally, in the main loop, the program waits for a character to arrive from the terminal:

```
GETCHAR();
```

   After which, it starts an ADC conversion:

```
ADC_DoSoftwareTriggerConvSeqA(ADC0);
```

   And polls the appropriate register and **waits until the conversion is complete**:

```
while (!ADC_GetChannelConversionResult(ADC0, ...
```

   When the conversion is complete, it prints out the result on the serial terminal.

The rest of the program is the initialization and configuration of the ADC.

3 ADC0 Configuration

This is performed in the function:

```
void ADC_Configuration(adc_result_info_t * ADCResultStruct)
```

It follows the guideline set out in Sec. 21.3.1 “Perform a single ADC conversion using a software trigger” of the Reference Manual. Please read it.

The configuration is done using the processor support libraries (PSL), and a struct is provided to store its various parameters. The configuration parameters that are shown below are bit fields of **A/D Control Register (CTRL)** in Sec. 21.6.1 “ADC Control Register”. Please read it while examining the part below.

The clock frequency of ADC0 is the system clock (initialized in this example to 24MHz; see `BOARD_InitBootClocks()` function). However, it can be set to a slower value using the clock divider. In this example the clock divider is set to 1. Low power mode is disabled.

```
adcConfigStruct.clockDividerNumber = ADC_CLOCK_DIVIDER; // Defined above.
adcConfigStruct.enableLowPowerMode = false;
```

To obtain more correct measurement results, the operating voltage range of the processor is written in an ADC register(See Sec. 21.6.11 A/D trim register (voltage mode)):

```
adcConfigStruct.voltageRange = kADC_HighVoltageRange;
```

The second part of the configuration is about a conversion sequence. In the LPC824, either a single conversion or a preset sequence of conversions can be performed in one go. Two such sequences can be used **Sequence A** and **Sequence B**. Here we define Sequence A but register only 1 channel. See Sec. 21.6.2 "A/D Conversion Sequence A Control Register". In the first line, we register the only channel we will be using into the channel sequence.

```
 adcConvSeqConfigStruct.channelMask = (1U << ADC_CHANNEL);
```

Then the trigger source is selected as software (See Table 277. "ADC hardware trigger inputs"). The rest can be checked by comparing with the information at Sec. 21.6.2.

```
 adcConvSeqConfigStruct.triggerMask      = 0U;
 adcConvSeqConfigStruct.triggerPolarity  = kADC_TriggerPolarityPositiveEdge
  ;
 adcConvSeqConfigStruct.enableSingleStep = false;
 adcConvSeqConfigStruct.enableSyncBypass = false;
 adcConvSeqConfigStruct.interruptMode    = kADC_InterruptForEachSequence;

```

Finally conversion sequence A is configured and enabled:

```
ADC_SetConvSeqAConfig(ADC0, &adcConvSeqConfigStruct);
ADC_EnableConvSeqA(ADC0, true);
```

One conversion is performed to prime ADC0 before the function returns.

4 Other Code Components of the Project

The project consists of many other files. You can see some of the files in the current directory of the project, and others are in the processor support libraries directory, `Xpresso_SDK/devices/LPC824/drivers/`. The files required for the project can be seen in the `Makefile` following the line `# C sources`. Notably these files are used:

- `Makefile`: This is the orchestra chief. It provides the following information:
  - The files that will be used in the compilation of the project are listed.
  - The directories that the files can be searched and found.
  - The rules by which a file type to be compiled using given options so that the resulting object file can be added into the final executable binary.
  - How the partially compiled files (`.obj`) are linked together.
  - Other actions such as programming the processor can also be described here (see the part `21flash`)
- `LPC824_flash.ld`: This describes such things as:
  - The microcontroller structure such as sizes of RAM and ROM memory,
  - Where they are located in the address map,

  - Where in the memory space should the text of the program or the variables or the vectors should be written,
  - Which location in the memory is the first line of code that must be executed after power on,
  - etc.

- `adc_basic.c`: Has been explained here.
- `pin_mux.c`: Assignments of peripheral input/output signals to physical package pins. This was discussed under the SWM peripheral subject.
- `board.c` and `board.h`: Sets the properties of the board, most notably configures the serial port and its baud rate (speed).
- `clock_config.c`: Configures the system clock to 24MHz.
- `startup_LPC824.S`: Initializes the processor, the interrupt vector table and variables in the program, then calls `main()`
- `system_LPC824.c`: Initializes the processor clock and provides some low level functions.
- All source files starting with `fsl_`: They provide the processor support library functions for each peripheral. If there is a PSL function that you do not understand, you can check its source code in these files.

As can be seen, the project requires quite a few files and has a complex structure. However, once the orchestra chief `Makefile` is written correctly, all that is required to compile it is to issue `make` in the shell.

Ahmet Onat 2023