Here is a list of all modules:

- License Terms and Copyright Information
- Abbreviations and Definitions
- Overview
- Architecture Description
- APP Configuration Parameters
- Enumerations
- Data structures
- Methods
- Usage
- Release History
License Terms and Copyright Information

Copyright (c) 2015, Infineon Technologies AG All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

To improve the quality of the software, users are encouraged to share modifications, enhancements or bug fixes with Infineon Technologies AG (dave@infineon.com).
## Abbreviations and Definitions

### Abbreviations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAVE™</td>
<td>Digital Application Virtual Engineer</td>
</tr>
<tr>
<td>APP</td>
<td>DAVE™ Application</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>LLD</td>
<td>Low Level Driver</td>
</tr>
<tr>
<td>IO</td>
<td>Input Output</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog to Digital Conversion</td>
</tr>
<tr>
<td>VADC</td>
<td>Versatile Analog to Digital Converter</td>
</tr>
</tbody>
</table>

### Definitions:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singleton</td>
<td>Only single instance of the APP is permitted</td>
</tr>
<tr>
<td>Sharable</td>
<td>Resource sharing with other APPs is permitted</td>
</tr>
<tr>
<td>initProvider</td>
<td>Provides the initialization routine</td>
</tr>
<tr>
<td>Physical connectivity</td>
<td>Hardware inter/intra peripheral (constant) signal connection</td>
</tr>
<tr>
<td>Conditional connectivity</td>
<td>Constrained hardware inter/intra peripheral signal connection</td>
</tr>
<tr>
<td>Aggregation</td>
<td>Indicates consumption of low level (dependent) DAVE APPs</td>
</tr>
</tbody>
</table>
ADC_MEASUREMENT_ADV

Overview

Overview

**Scope:** This APP measures multiple analog signals using the Versatile Analog to Digital Converter (VADC) peripheral. It provides advanced features such as sequencing, post processing, synchronized measurements and hardware boundary check.

Some configurations are set in a required APP (ADC_SCAN or ADC_QUEUE). These configurations are read only in this APP. The features in detailed are:

1. Up to 8 analog channels (more with sync. conversion)
2. Measures a linear sequence of analog inputs (Scan request source).
3. Measures a flexible sequence of analog inputs (Queue request source).
4. Supports internal and external consumption of the request sources.
5. Software and hardware controlled start of measurements.
6. Results post processing with FIFO and Filter.
7. Synchronously channels measurement.
8. Boundary configurations for channels
9. Supports events (Request source event, Result event, Channel event, boundary flag).

**Scope delimitation:**
To avoid cross configuration, the background request source is not in the scope of this APP. This feature can be added without cross configuration by adding the ADC_MEASUREMENT APP to the project.

**Details of provided functionalities:**
1. **Number of channels and conversion details:**

   The number of channels can be configured in the APP GUI "Overview". Also the type of conversion can be selected. The conversion details (Timing, Resolution...) are read from required APP.

2. **Conversion sequence:**

   The channel names configured in the APP GUI "Channel Configuration". Also for each channel the "Wait for read" can be enabled.

   - **Scan sequence:**

     The scan sequence is a linear sequence starting with the highest enabled channel (CH7 to CH0). The assignment of the "Channel name" to the physical channel CHx is done by assigning the channel resource or by assigning the input pin. This is explained with an example as follows:

     If 6 measurements are configured in the GUI named as CH_A, CH_B, CH_C, CH_D, CH_E, CH_F. The channels assigned to the measurements is as follows

     CH_A = CHANNEL-2
     CH_B = CHANNEL-5
     CH_C = CHANNEL-6
     CH_D = CHANNEL-0
     CH_E = CHANNEL-4
     CH_F = CHANNEL-1

     The order of conversion sequence for these measurements is as shown in the following figure.
**Queue sequence:**

The queue sequence is a flexible 8 stages sequence. Each channel can be placed in any position and also multiple entries of one channel are possible. The TAB "Sequence Plan" in the APP GUI allows configuring the sequence. The assignment of the "Channel name" to the physical channel CHx is done by assigning the channel resource or by assigning the input pin. This is explained with an example as follows:

If 6 measurements are configured in the GUI named as CH_A, CH_B, CH_C, CH_D, CH_E, CH_F. The channels assigned to the measurements is as follows:

- CH_A = CHANNEL-2
- CH_B = CHANNEL-5
- CH_C = CHANNEL-6
- CH_D = CHANNEL-0
- CH_E = CHANNEL-4
- CH_F = CHANNEL-1

And the Queue sequence is selected as follows:

- CH_B = CHANNEL-5
- CH_B = CHANNEL-5
- CH_B = CHANNEL-5

![Figure 1: Scan Conversion Sequence](image-url)
\[ \begin{align*}
\text{CH}_A & = \text{CHANNEL-2} \\
\text{CH}_C & = \text{CHANNEL-6} \\
\text{CH}_D & = \text{CHANNEL-0} \\
\text{CH}_F & = \text{CHANNEL-1} \\
\text{CH}_E & = \text{CHANNEL-4}
\end{align*} \]

The order of conversion sequence for these measurements is as shown in the following figure

---

3. **Expose APP:**

The APP supports the consumption of the request source inside or outside the APP. This feature shall by selecting the "Expose APP" option in the overview TAB of the APP GUI. If Expose APP is
enabled, the APP consumes the required request source APP either ADC_SCAN or ADC_QUEUE depending on the selection being made. If the expose APP option is disabled then the request source will be consumed inside the APP. Use the "Expose APP" option when multiple use case APPs like ADC_MEASUREMENT_ADV and BUCK need to share the request source.

4. **Synchronous conversion:**

Two or more measurements can be requested at the same time when they share the same trigger. If one of the ADC groups is already converting at this moment this measurement will be delayed compared to the other measurements. In some applications the measurement has to be done completely synchronous. For this case the APP supports the synchronous conversion mode. The synchronous conversion is implemented as master/slave approach. If one group is a master the following group(s) can be slaves.

To configure a group as sync master the number of slaves need to be selected in the TAB "Sync. Conversion". After this configuration each channel can be configured as master channel. A master channel will allocate the selected amount of slave channels. The master and slave group is different but the channel number is the same. Every time the master channel starts a conversion, the slave channels also start a conversion.

For example G0 is a master group with 1 slave. This makes G1 to a slave group. As soon as "Channel_A" (from G0) is configured as Sync Master and assigned to CH3, G1 CH3 becomes a slave channel. When the master G0 CH3 is requested to convert also the slave G1 CH3 will be converted.

Sync slave channels are named with the <master channel>_slave_<slave token>. Slave channels are exclusive consumed by the APP. The class selection, post processing and boundary configurations are copied from the master to all slaves.
If required, the class configuration can be copied via APP API. This API can overwrite a class configuration if it is configured by another APP.

Note: Ensure that the "Arbitration mode" for all the sync groups is set to "Run Permanently" in GLOBAL_ADC APP.

If 6 measurements are configured in the GUI named as CH_A, CH_B, CH_C, CH_D, CH_E, CH_F slaves groups needed as 3 (Group-0 is the master). CH_F(from Group-0) and CH_B(from Group-0) are configured as sync master channels. The channels in the master are using scan request source for conversion and the order of conversion is as show in the Figure 1. The sync conversion with scan request source is as shown in the following figure.

![Figure 3: Sync conversion with scan request source](image)

5. **Result processing:**

The APP supports result post processing in the hardware. The post processing options can be selected from the APP GUI tab "Post Processing". The options for post processing include the following options:

- FIFO buffer
This allows up to 16 stage FIFO buffer for the selected channel.

- **Result filtering**
  Allows either FIR filter or IIR filter for filtering of conversion results.

- **Result accumulation**
  Allows up to 4 accumulations of conversion results.

- **Result subtraction**
  Subtracts the conversion result by the value stored in the GxRES0 register. The result register is not allocated when this mode is selection.

### 6. Limit Checking (Boundary):

The APP supports result monitoring with limit checking. In normal mode the converted value is automatically checked with two boundary values. If required, a channel event is generated on every result or if the result is inside or outside the boundary. In fast compare mode the result is checked with a compare value. In this mode a channel event can be generated if the result is lower or higher the compare value or if the value is crossing the compare value. The fast compare mode can be selected in the required APP ADC_SCAN or ADC_QUEUE. Additional for both modes and up to 4 channels a boundary flag is available.

### 7. Interrupt Notifications:

- **Result event:**
  A notification or an event is generated when a new valid result is present in the result register

- **Channel event:**
  A notification or an event is generated when either a channel has finished its conversion or the channel's converted result is inside/outside the boundary values.

- **Request source interrupt:**
  A interrupt is generated when all the channels are converted
once. For a scan request source the interrupt is generated after the least channel number gets converted. For a queue request source the interrupt is generated after a specific entry finishes its conversion.

**APP Structure**

**Figure 4** shows how the APP is structured in DAVE. XMC controllers provides the VADC module for analog to digital conversion. The XMC Lib layer provides abstraction for these hardware modules. The **ADC_MEASUREMENT_ADV** APP uses VADC and SCU LLDs and other dependent APPS such as **ADC_QUEUE**, **ADC_SCAN**, **GLOBAL_ADC**, **ANALOG_IO** and **CLOCK_XMCx** for the functional execution.
**Figure 4**: Hardware and Software connectivity of 
**ADC_MEASUREMENT_ADV** APP

**Limitations:**

- Channel events are only supported for boundary channels, hence is limited to maximum 4.
- The sync. slave channel events are not available for signal connection.
- For XMC1x devices when alias feature is used, it causes the gain value to be configured for the source channel and not for the alias channel.

**Supported Devices**

*The APP supports below devices:*

1. XMC4800 / XMC4700 Series
2. XMC4500 Series
3. XMC4400 Series
4. XMC4300 Series
5. XMC4200 / XMC4100 Series
6. XMC1400 Series
7. XMC1300 Series
8. XMC1200 Series

**Reference**

1. XMC4800 / XMC4700 Reference Manual
2. XMC4500 Reference Manual
5. XMC4200 / XMC4100 Reference Manual
7. XMC1300 Reference Manual
ADC_MEASUREMENT_ADV

Architecture Description

The following diagrams represents the internal software architecture of the ADC_MEASUREMENT_ADV APP. The figure shows the consumed hardware resources, dependent APPs and various signals which are exported.

- Figure 1 represents the architecture of the ADC_MEASUREMENT_ADV APP with expose APP option enabled in the GUI.
- Figure 2 represents the architecture of the ADC_MEASUREMENT_ADV APP with expose APP option disabled in the GUI. This configuration consumes the request source inside ADC_MEASUREMENT_ADV itself.

A ADC_MEASUREMENT_ADV APP instance exists in a DAVE™ project with fixed attributes as shown and uses the VADC peripheral's scan or queue request source for converting a channel. This in addition requires the consumption of the GLOBAL_ADC and CLOCK APPS for functional configurations. The ADC_MEASUREMENT_ADV APP also provides output signals, these are described in Table-1.
Figure 1: Architecture of the ADC_MEASUREMENT_ADV APP with exposed request source APP.
Figure 2: Architecture of the ADC_MEASUREMENT_ADV APP with internally consumed request source.

Following are the features of the ADC_MEASUREMENT_ADV APP.

1. Pin Sharing
2. Result Registers
3. Interrupt/Event Generation
4. Queue Sequencer
5. Insert channels
6. Sample and Hold Gain Configuration
7. Limit checking (Boundary configuration)
Detailed descriptions are as follows.

1. **Pin Sharing:**

   ANALOG_IO APP is conditionally used by ADC_MEASUREMENT_ADV APP when an "Expose pin" is selected in the UI. This is applicable for all the channels. By using the ANALOG_IO, the ADC_MEASUREMENT_ADV can share the pin with other APPs such as DAC, ACMP_CONFIG etc. It is possible to connect the same ANALOG_IO APP to multiple channels. This involves the use of the ALIAS feature of the ADC channels. In this the same pin gets converted by multiple channels from the same group.

   For example: Two instances of the ADC_MEASUREMENT_ADV are present in the project. In ADC_MEASUREMENT_ADV_0, 3 channels are used from group-0 and in ADC_MEASUREMENT_ADV_1 another 3 channels are used from Group-1. The example is taken with reference to XMC4500. The pins and channels distribution are shown in the following figure.

   ![Figure 3: Pins and channel consumption without ANALOG_IO](image)

   The alias feature of the VADC allows the Channels 0 and 1 to convert any pin available in the same group. In XMC4500 the pin P14.7 is only connected to Channel-7 of Group-0. With the alias feature we can use the Group-0 Channel-0 to convert P14.7.
The alias feature of the VADC allows the conversion of any given pin of the group by the alias channels Channel-0 or Channel-1. This means that the Channel-0 and Channel-1 can convert not only the pins assigned to it but also the other pins available for the entire group. This feature can be used to convert a pin multiple times. In order to do this a pin must get shared with the other channels as well. This is only possible through the consumption of the ANALOG_IO APP. The following figure depicts how the channel numbers get changed when the same pin is being shared between channels.
Figure 5: Pins and channel consumption with ANALOG_IO with a group

In different groups of the VADC the channels can have the same pin number. For example in XMC4500 P14.3 is shared between Group-0 Channel-3 and Group-1 Channel-3. Thus two instances of the `ADC_MEASUREMENT_ADV` APP can convert the same pin. The following figure shows that it is possible to connect the Group-1 Channel-3 to the ANALOG_IO that is consuming P14.3. With the alias feature in the Group-1 is also possible to convert that pin using Group-1 Channel-0 and Group-1 Channel-1.

Figure 6: Pins and channel consumption with ANALOG_IO shared between groups

2. Result Registers:

Each channel is mapped to one result register(excluding FIFO related configuration). There are 3 different categories of result registers based on different functions it provides.

- **result_adv**: Provide boundary flag outputs.
- **result_filter**: Provide filtered output.
- **result**: Provide accumulation(1x, 2x, 3x, 4x)/subtraction mode.
In **ADC_MEASUREMENT_ADV** APP the `result_adv` is used when Boundary is used for Normal Comparison Mode. The `result_filter` is used when result post processing requires Result Filtering Mode. This is either FIR (Finite Impulse Response) or the IIR (Infinite Impulse Response) filters. The `result` is used when ever whenever Standard Data Reduction Mode or Subtraction Mode is selected.

3. **Interrupt/Event Generation:**

   - **Request source interrupt:**

     In the Interrupts TAB the **ADC_MEASUREMENT_ADV** doesn’t consume a NVIC node for interrupt. But it uses a callback registration mechanism that is called whenever the request source interrupt is raised. This callback is registered with the request source APP, ADC_QUEUE/ADC_SCAN. The callback is registered in the queue/scan entry by considering the following rules.

     - For queue request source the callback would be registered to the entry that has the "Source event" enabled in the Sequence Plan TAB.
     - For scan request source the callback is registered with the lowest numbered channel number. Since the lowest channel numbered entry is the last channel to get executed in the scan sequence.

   - **Result Event:**

     Receive an event every time a new result is available in the result register. This event has to be enabled in the "Channel Configuration" TAB. To generate an interrupt from this event, connect the result event signal from the **ADC_MEASUREMENT_ADV** APP to INTERRUPT APP using the H/W Signal connections. This setup would provide result interrupts for the channels.

   - **Channel Event:**
This event has to be enabled in the "Boundary Settings" TAB. To enable the channel events, select the channel event combo box for the required channel. To generate an interrupt from this event, connect the result event signal from the ADC_MEASUREMENT_ADV APP to INTERRUPT APP using the H/W Signal connections.

In Normal Conversion Mode it is possible to get a channel event only if the converted value is inside/outside the boundary values or on each new result. The boundary values must be configured using the API ADC_MEASUREMENT_ADV_SetBoundary().

In Fast Compare Mode it is possible to get a channel event only if the measured value is above or below a compare value, or if it crosses the compare value. The compare value is located in the result register and must be configured with the API ADC_MEASUREMENT_ADV_SetFastCompareValue().

4. **Queue Sequencer:**

   In the Queue sequence it is possible to have a flexible sequence of channels for conversion.

   The "Sequence Plan" allows a flexible configuration of the sequence. Each Channel can be placed on any position. Also multiple selections are allowed. Additional for each position the configurations "Wait for Trigger", "Refill" and "Source Event" are available. With the "Wait for Trigger" "configuration" the queue is waiting for a trigger before the selected channel is converted. The "Refill" option allows a one time measurement. This position is not refilled in the queue and will not execute in the next rounds. The "Source Event" configuration will execute a source event when the conversion is finished.

   The following examples explain the usage of the "Wait for Trigger"
and "Refill" configuration.

In the first example the sequence is starting with CH_C. This channel has the configuration "Wait for Trigger" therefore the sequence is waiting for a Trigger before it starts with the conversion of CH_C. No other channel is configured with "Wait for Trigger" therefore the sequence is immediately continuing with the conversion. When the sequence is reaching the end all channels with "Refill" are refilled and the sequence starts again as new. In this case it is waiting again for a trigger.

![Queue sequence: Wait for trigger enabled for 1 Queue Entry](image)

**Figure 7**: Queue sequence: Wait for trigger enabled for 1 Queue Entry

In the second example additional CH_A is configured with "Wait for Trigger". This results in a two steps sequence. First the sequence is waiting for a trigger, after this trigger CH_C, CH_B and CH_E are converted. Now the sequence is again waiting for a trigger to continue with CH_A. After a trigger CH_A, CH_F, CH_D and CH_B are converted. Hence all channels provide the configuration "Refill" the sequence is repeated.
Figure 8: Queue sequence: Wait for trigger enabled for 2 Queue Entries

In the third example again only CH_C provides a "Wait for Trigger" configuration but know don't has a "Refill" configuration. This means the sequence is waiting for a trigger. After the trigger it start with the conversion of CH_C and continuous with all other channels. At the end of the sequence all channels with "Refill" configuration are refilled. Hence CH_C don't has a "Refill" it will not copied in the sequence. This means after the second round there is no entry with "Wait for Read" and the sequence repeat endless.
5. **Insert channels**:

The insert operation in `ADC_MEASUREMENT_ADV` binds the channels to the selected request source in the hardware. Since the ADC_QUEUE is shared it is possible to assign the same queue position from other instances of `ADC_MEASUREMENT_ADV` (or from other APPs). In order to prevent such operations, a MACRO is constructed with the following format.

`VADC_QUEUE_GROUP_<group_number>_POSITION_<position>`.

This would prevent the APPs that are sharing the ADC_QUEUE APP from trying to push a queue entry into the same location. If the same position is used a error message is displayed. If such a error message is received then the queue sequence needs to be modified appropriately in the GUI.

- `ADC_QUEUE`
When a ADC_MEASUREMENT_ADV_InsertChannels() API is invoked or the GUI check box "Insert channels at initialization" is enabled with the "Wait for Trigger", the channels will wait until a trigger event occurs. A trigger event can be either a ADC_MEASUREMENT_ADV_StartConversion() or an external trigger edge. When a trigger event occurs the particular channel will go for conversion.

When a ADC_MEASUREMENT_ADV_InsertChannels() API is invoked or the GUI check box "Insert channels at initialization" is enabled without the "Wait for Trigger", The channel being inserted will get converted immediately. And if the refill is enabled for the channel then it will continuously keep converting.

○ ADC_SCAN

When a ADC_MEASUREMENT_ADV_InsertChannels() API is invoked or the GUI check box "Insert channels at initialization" is enabled, the channels will be inserted into the hardware. There these channels will continue to wait until a trigger event occurs. A trigger event can be either a ADC_MEASUREMENT_ADV_StartConversion() or an external trigger edge. The trigger event will make all the channels to go for conversion. When all the channels are converted it will go back to wait state when "continuous conversion" is disabled. If "Continuous conversion" is enabled the last channel to get converted will trigger all the channels to get converted again.

6. Sample and Hold Gain Configuration:

The gain related configuration is applicable for only the XMC1x series. This feature allows the input signal to be amplified by the required factor. The gain values can be selected in the GUI for the required channel or by invoking the API ADC_MEASUREMENT_ADV_SetChannelGain(). The following example explains how the gain is configured when alias feature is used. The following figure explains the gain configuration by taking
3 channels as an example. The configuration is as follows, CH_A is Group-0 Channel-3, CH_B and CH_C are the alias channels which consume group-0 channel-0 and group-0 channel-1. CH_A and CH_B need gain of 1:1 and CH_C needs 1:3 as the gain ratio. The source channel in this case is Group-0 Channel-3 and the source pin is P14.3 and the other 2 are aliased to this channel. When alias feature is used and gain configuration is needed for the alias channel it is necessary to configure the gain of the source channel rather than the gain of the alias channel. In this example in-order to configure the gain for the CH_C it is necessary to configure the gain in the source channel i.e. group-0 channel-3. Due to such a configuration of the SHS the CH_A and CH_B would also convert with a gain factor of 1:3.

Figure 10: Sample and Hold gain configuration example with alias enabled
7. **Limit checking (Boundary configuration):**

The Limit checking can automatically compare each conversion to an upper or lower bound values. Accordingly a channel event can be triggered if the result is inside/outside the user-defined band. This limit checking can only be done by specific registers with boundary being configured in GxBOUND or GLOBBOUND. In XMC45 device the limit checking can be done the first 4 channels in each group and in all the other devices the limit checking is done by the first 4 result registers in each group. Hence in the UI the boundary configuration is limited to only 4.

The Boundary values can only be configured by invoking the API 
\texttt{ADC\_MEASUREMENT\_ADV\_SetBoundaryUpper()} and \texttt{ADC\_MEASUREMENT\_ADV\_SetBoundaryLower()}. These API would configure the respective boundary registers mentioned in the "Boundary Settings" Tab in APP's GUI.

**Note:** The value to be configured in the boundary registers have to be left aligned compare values. For example, a compare value in 10 bit resolution is 500 in order to configure this in the boundary registers it has to be first left aligned(left alignment is done by left shift operation). After the left shift operation the resulting value is 2000 this value has to be passed as an argument into the \texttt{ADC\_MEASUREMENT\_ADV\_SetBoundaryUpper()} or \texttt{ADC\_MEASUREMENT\_ADV\_SetBoundaryLower()} API.

**Signals:**

The following table provides all APP signals for connection.

**Table 1:** APP IO signals

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Input/Output</th>
<th>Availability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result event</td>
</tr>
</tbody>
</table>
event_res_channel_x
Where channel_x represents the channels that are being used.

channel_x:
- It can be connected to INTERRUPT APP to generate an interrupt after the result is generated by a particular channel_x.
- The result event is available after the check box in the UI "Result event for the particular channel_x" is enabled.
- The channel_x can be configured in the "Channel Configuration" Tab of the APP GUI.

Channel event:
- It can be connected to INTERRUPT APP.
- The channel event is...
event_ch\_channel\_x
Where \textit{channel\_x} represents the channels that are being used.

Output Unconditional

Available after the combo box selection in the UI "Boundary Settings" for the particular channel is enabled.

- The label \textit{channel\_x} can be configured in the "Channel Configuration" Tab of the APP GUI.

bound_fl\_channel\_x
Where \textit{channel\_x} represents the channels that are being used.

Output Conditional

Boundary flag output for \textit{channel\_x}:

- It can be connected to other peripherals like CCU/POSIF/ERU etc.
- The boundary flag signal is available only when the combo box selection in the UI "Boundary Settings" for the particular channel is enabled.
channel is enabled.

- The label "channel_x" can be configured in the "Channel Configuration" Tab of the APP GUI.

\[ \text{channel}_x \text{input} \]

Where \( \text{channel}_x \) represents the channels that are being exposed.

Input connection from ANALOG_IO APP pin to group channel:

- The pin to channel connection is visible only when "Expose Pin" option is selected for a channel configured under "Channel Configuration TAB".

- The Expose option allows the pin to be shared between peripherals.

- The label "channel_x" can be
channel_{x}_slavey_input
Where channel_{x} represents the slave channels that are being exposed.

Input connection from ANALOG_IO APP pin to slave group channel:
- The pin to channel connection is visible when the "Exposure" option is selected for a synchronous channel under "Channel Configuration Tab".
- The label channel_{x} can be configured in the "Channel Configuration Tab".
- The Exposure option allows the pin to be shared between peripherals.
Tab of the APP GUI.

Input connection from request source APP: (ADC_SCAN / ADC_QUEUE) channel:

- It selects a particular group channel, establishes connection from group request source to selected channel.
- The label \( channel_x \) can be configured in the "Channel Configuration" Tab of the APP GUI.

\( channel_x \_sel \)

Where \( channel_x \) represents the channels that are being used.

**Timing Calculations:**

**ADC_MEASUREMENT_ADV** APP uses the following equations to calculate the sample time and Total conversion time. Refer the reference manual for the detailed information.

1. **Total Conversion time:** XMC4000 devices
Note: PC value is configured as 2 i.e with post calibration always enabled.
If post calibration is disabled (in GLOBAL_ADC APP), the total conversion time will be reduced by 2/fADC (GLOBAL_ADC APP).

1. Sample Time:

   Actual Sample Time = (2 + STC) * tADCI
   where,
   STC : Sample time control (Value - 0 to 256)
   fADCI : Analog clock frequency
   tADCI = 1/fADCI

2. Standard Conversion Mode:

   Total Conversion time = (2 + STC + N + DM + PC) * tADCI
   Where,
   N = 8, 10, 12 for n bit resolution.
   tADC = ADC module clock = system clock
   tADCI = Analog clock
   STC = Sample time control (Value - 0 to 256)
   DM = The selected duration of the MSB conversion
   PC = The post-calibration time PC, if selected

3. Fast Compare Mode:

   Total Conversion time = (2 + STC + 2) * tADCI + 2 * tAI
   Where,
   tADC = ADC module clock = system clock
   tADCI = Analog clock
   STC = Sample time control (Value - 0 to 256)

2. Total Conversion time: XMC1000 devices

   "Total conversion time" for XMC1400/XMC1300/XMC1200 devices are calculated with post calibration.

   1. Sample Time:
   Minimum sample time is calculated to in accordance with Errata ADC_AI.H007. To ensure proper operation of the internal control logic, tS must be at least four cycles of the prescaled converter clock fSH, i.e. tS >= 4 tCONV x
Actual Sample Time \( t_S \) = \((2 + \text{STC}) \times t_{\text{ADCI}}\)

where,

STC : Sample time control (Value - 0 to 256)
f\( _{\text{ADCI}} \) : Analog clock frequency
\( t_{\text{ADCI}} = 1/f_{\text{ADCI}} \)

2. **Standard Conversion Mode:**

Total Conversion time = \((2 + \text{STC}) \times t_{\text{ADCI}} + (4 \times t_{\text{SH}}) + (N + 8) \times t_{\text{SH}} + (5 \times t_{\text{ADC}}) + (12 \times t_{\text{SH}})\)

Where,

\( N \) = 8, 10, 12 for \( n \) bit resolution.
\( t_{\text{SH}} \) = Sample and Hold clock (Converter clock time period)
\( t_{\text{ADC}} \) = ADC module clock = system clock
\( t_{\text{ADCI}} \) = Analog clock
STC = Sample time control (Value - 0 to 256)

\[ t_{\text{SH}} = t_{\text{CONV}} \times (\text{DIVS} + 1) \]

Where,

\( t_{\text{SH}} \) = Sample and Hold clock (Converter clock time period)
\( t_{\text{CONV}} \) = Converter clock time period (SHS Clock)
\( \text{DIVS} \) = Divider Factor for the SHS Clock

3. **Fast Compare Mode:**

Total Conversion time = \((\text{FCRT} + 1) \times 2 \times t_{\text{ADCI}} + (2 + \text{STC}) \times t_{\text{ADCI}}\)

Where,

\( \text{FCRT} \) = Fast Compare Mode Response Time (Value - 0 to 15)
\( t_{\text{SH}} \) = Sample and Hold clock (Converter clock time period)
\( t_{\text{ADCI}} \) = Analog clock
STC = Sample time control (Value - 0 to 256)

Note: FCRT value is configured as 0.
Figure 1: Overview
## Figure 2: General Settings

<table>
<thead>
<tr>
<th>Request Source Settings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger edge selection</td>
<td>No External Trigger</td>
</tr>
<tr>
<td>Getting selection</td>
<td>All Conversion Requests are Issued</td>
</tr>
<tr>
<td>Priority of queue source</td>
<td>Priority-0 (Lowest Priority)</td>
</tr>
<tr>
<td>Conversion start mode</td>
<td>Wait For Start Mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Settings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conversion mode</td>
<td>10 Bit Fast Conversion</td>
</tr>
<tr>
<td>Desired sample time [nsec]</td>
<td>100</td>
</tr>
<tr>
<td>Actual sample time [nsec]</td>
<td>83.333</td>
</tr>
<tr>
<td>Total conversion time [nsec]</td>
<td>166.667</td>
</tr>
</tbody>
</table>
Figure 3: Channel Configuration

<table>
<thead>
<tr>
<th>Channel name</th>
<th>Expose Pin</th>
<th>Wait for read</th>
<th>Result Event</th>
<th>Reference GND</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel_A</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
<tr>
<td>Channel_B</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
<tr>
<td>Channel_C</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
<tr>
<td>Channel_D</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
<tr>
<td>Channel_E</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
<tr>
<td>Channel_F</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
<tr>
<td>Channel_G</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
<tr>
<td>Channel_H</td>
<td></td>
<td></td>
<td></td>
<td>Vss</td>
<td>1:1</td>
</tr>
</tbody>
</table>
Figure 4: Sequence Plan
Figure 5: Post Processing
Figure 6: Sync. Conversion

Note: Ensure that the "Arbitration mode" for all the sync groups are set to "Runs Permanently" in GLOBAL_ADC APP.
Figure 7: Boundary Settings

Note: Configure boundary levels with provided API.
Figure 8: Interrupts
## ADC_MEASUREMENT_ADV

### Enumerations

<table>
<thead>
<tr>
<th>Enum Name</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_MEASUREMENT_ADV_STATUS</td>
<td>ADC_MEASUREMENT_ADV_STATUS_SUCCESS, ADC_MEASUREMENT_ADV_STATUS_FAILURE, ADC_MEASUREMENT_ADV_STATUS_UNINITIALIZED</td>
<td>Return value of an API.</td>
</tr>
<tr>
<td>ADC_MEASUREMENT_ADV_REQUEST_SOURCE</td>
<td>ADC_MEASUREMENT_ADV_REQUEST_SOURCE_SCAN, ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN, ADC_MEASUREMENT_ADV_REQUEST_SOURCE_QUEUE, ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_QUEUE</td>
<td>The selected Request source.</td>
</tr>
<tr>
<td>ADC_MEASUREMENT_ADV_GAIN</td>
<td>ADC_MEASUREMENT_ADV_GAIN_1, ADC_MEASUREMENT_ADV_GAIN_3, ADC_MEASUREMENT_ADV_GAIN_6, ADC_MEASUREMENT_ADV_GAIN_12</td>
<td>The gain applied on the input signal.</td>
</tr>
<tr>
<td>ADC_MEASUREMENT_ADV_SUBTRATION</td>
<td>ADC_MEASUREMENT_ADV_SUBTRATION_12BIT_LEFT_ALIGN, ADC_MEASUREMENT_ADV_SUBTRATION_12BIT_RIGHT_ALIGN, ADC_MEASUREMENT_ADV_SUBTRATION_10BIT_LEFT_ALIGN, ADC_MEASUREMENT_ADV_SUBTRATION_10BIT_RIGHT_ALIGN, ADC_MEASUREMENT_ADV_SUBTRATION_8BIT_LEFT_ALIGN</td>
<td>} The selected Request source.</td>
</tr>
</tbody>
</table>

```c
enum ADC_MEASUREMENT_ADV_STATUS {
    ADC_MEASUREMENT_ADV_STATUS_SUCCESS,
    ADC_MEASUREMENT_ADV_STATUS_FAILURE,
    ADC_MEASUREMENT_ADV_STATUS_UNINITIALIZED
}
```

```c
enum ADC_MEASUREMENT_ADV_REQUEST_SOURCE {
    ADC_MEASUREMENT_ADV_REQUEST_SOURCE_SCAN,
    ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN,
    ADC_MEASUREMENT_ADV_REQUEST_SOURCE_QUEUE,
    ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_QUEUE
}
```

```c
enum ADC_MEASUREMENT_ADV_GAIN {
    ADC_MEASUREMENT_ADV_GAIN_1,
    ADC_MEASUREMENT_ADV_GAIN_3,
    ADC_MEASUREMENT_ADV_GAIN_6,
    ADC_MEASUREMENT_ADV_GAIN_12
}
```
<table>
<thead>
<tr>
<th>Type</th>
<th>Enum Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum</td>
<td>ADC_MEASUREMENT_ADV_SUBTRACTION</td>
<td>Alignment options for the subtraction value.</td>
</tr>
<tr>
<td>typedef enum</td>
<td>ADC_MEASUREMENT_ADV_STATUS</td>
<td>Return value of an API.</td>
</tr>
<tr>
<td>typedef enum</td>
<td>ADC_MEASUREMENT_ADV_REQUEST_SOURCE</td>
<td>The selected Request source.</td>
</tr>
<tr>
<td>typedef enum</td>
<td>ADC_MEASUREMENT_ADV_GAIN</td>
<td>The gain applied on the input signal.</td>
</tr>
<tr>
<td>typedef enum</td>
<td>ADC_MEASUREMENT_ADV_SUBTRACTION</td>
<td>Alignment options for the subtraction value.</td>
</tr>
<tr>
<td>typedef enum</td>
<td>ADC_MEASUREMENT_ADV_FAST_COMPARE</td>
<td>The result of the fast compare operation.</td>
</tr>
<tr>
<td>typedef enum</td>
<td>ADC_MEASUREMENT_ADV_SYNC_SEQ</td>
<td>The result of the fast compare operation.</td>
</tr>
</tbody>
</table>
# Enumeration Type Documentation

**enum** `ADC_MEASUREMENT_ADV_STATUS`  

Return value of an API.

**Enumerator:**

- `ADC_MEASUREMENT_ADV_STATUS_SUCCESS`  
The API call is success

- `ADC_MEASUREMENT_ADV_STATUS_FAILURE`  
The API call is failed

- `ADC_MEASUREMENT_ADV_STATUS_UNINITIALIZED`  
APP has not been Initialize

Definition at line 107 of file `ADC_MEASUREMENT_ADV.h`. 

### ADC_MEASUREMENT_ADV

#### Data structures

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ADC_MEASUREMENT_ADV_EVENT_CONFIG_t</code></td>
<td>typedef void(*ADC_MEASUREMENT_ADV_EVENT_CONFIG_t)(void)</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_NVIC_CONFIG_t</code></td>
<td>typedef struct ADC_MEASUREMENT_ADV_NVIC_CONFIG</td>
</tr>
<tr>
<td></td>
<td>NVIC Configuration structure for request source.</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_SCAN_t</code></td>
<td>typedef struct ADC_MEASUREMENT_ADV_SCAN</td>
</tr>
<tr>
<td></td>
<td>Configuration Data structure of scan source.</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_QUEUE_t</code></td>
<td>typedef struct ADC_MEASUREMENT_ADV_QUEUE</td>
</tr>
<tr>
<td></td>
<td>Configuration Data structure of queue source.</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_CHANNEL_t</code></td>
<td>typedef struct ADC_MEASUREMENT_ADV_CHANNEL</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_t</code></td>
<td>typedef struct ADC_MEASUREMENT_ADV</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Typedef Documentation**

**typedef struct** `ADC_MEASUREMENT_ADV_CHANNEL ADC_MEASUREMENT_ADV_CHANNEL_t`  
Structure to configure the channels in the `ADC_MEASUREMENT_ADV` APP.

**typedef void(** `ADC_MEASUREMENT_ADV_EVENT_CONFIG_t` **)**(void)  
Function pointer to the mux configuration  
Definition at line 190 of file `ADC_MEASUREMENT_ADV.h`.

**typedef struct** `ADC_MEASUREMENT_ADV ADC_MEASUREMENT_ADV_t`  
Structure to configure `ADC_MEASUREMENT_ADV` APP.
### ADC_MEASUREMENT_ADV

#### Methods

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DAVE_APP_VERSION_t</code></td>
<td><strong>ADC_MEASUREMENT_ADV_GetAppVersion</strong></td>
<td>Get ADC_MEASUREMENT_ADV version</td>
</tr>
<tr>
<td><code>void</code></td>
<td><strong>ADC_MEASUREMENT_ADV_StartADC</strong></td>
<td>Starts the VADC request source.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><strong>ADC_MEASUREMENT_ADV_SoftwareTrigger</strong></td>
<td>Starts the conversion of the required channels.</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_STATUS_t</code></td>
<td><strong>ADC_MEASUREMENT_ADV_Init</strong></td>
<td>Initializes the ADC_MEASUREMENT_ADV instance.</td>
</tr>
<tr>
<td><code>__STATIC_INLINE uint16_t</code></td>
<td><strong>ADC_MEASUREMENT_ADV_GetResult</strong> (handle_ptr)</td>
<td>Returns the conversion result.</td>
</tr>
<tr>
<td><code>__STATIC_INLINE XMC_VADC_DETAILED_RESULT_t</code></td>
<td><strong>ADC_MEASUREMENT_ADV_GetDetailedResult</strong> (const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr)</td>
<td>Returns the complete conversion result.</td>
</tr>
<tr>
<td><code>__STATIC_INLINE uint16_t</code></td>
<td><strong>ADC_MEASUREMENT_ADV_GetFifoResult</strong> (handle_ptr)</td>
<td>Returns the conversion result.</td>
</tr>
<tr>
<td><code>__STATIC_INLINE XMC_VADC_DETAILED_RESULT_t</code></td>
<td><strong>ADC_MEASUREMENT_ADV_GetFifoDetailedResult</strong> (const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr)</td>
<td>Returns the complete conversion result.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>__STATIC_INLINE ADC_MEASUREMENT_ADV_FAST_COMPARE_t</td>
<td>(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr) Returns the result of fast conversion.</td>
<td></td>
</tr>
<tr>
<td>ADC_MEASUREMENT_ADV_STATUS_t</td>
<td>(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, uint16_t compare_value) Sets the fast conversion value.</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, ADC_MEASUREMENT_ADV_SUBTRACTION_t subtraction_alignment, uint16_t subtraction_value) Set the subtraction value if a result difference mode is required.</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, const XMC_VADC_GROUP_CLASS_t *config) Configures the input class (Resolution and Sampling time).</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, const XMC_VADC_CHANNEL_REF_t reference_select) Selects alternate reference voltage for the channel.</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, const ADC_MEASUREMENT_ADV_GAIN_t gain_factor) Sets the channel gain.</td>
<td></td>
</tr>
<tr>
<td>void</td>
<td>(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, XMC_VADC_BOUNDARY_SELECT_t boundary, XMC_VADC_CHANNEL_BOUNDARY_t boundary_selection)</td>
<td></td>
</tr>
</tbody>
</table>

Note: The text seems to be a part of a C/C++ code snippet, detailing functions associated with advanced ADC measurement in an embedded system.
Select the boundary for the channel.

```c
void ADC_MEASUREMENT_ADV_SetBoundaryUpper(const ADC_MEASUREMENT_ADV_CHANNEL_t*const handle_ptr, uint32_t boundary_value)
```
Sets the upper boundary value for the channel.

```c
void ADC_MEASUREMENT_ADV_SetBoundaryLower(const ADC_MEASUREMENT_ADV_CHANNEL_t*const handle_ptr, uint32_t boundary_value)
```
Sets the lower boundary value for the channel.

```c
XMC_VADC_CHANNEL_ALIAS_t ADC_MEASUREMENT_ADV_GetAliasValue(ADC_MEASUREMENT_ADV_CHANNEL_t handle_ptr)
```
Returns the configured alias value.

```c
void ADC_MEASUREMENT_ADV_SetIclass(ADC_MEASUREMENT_ADV_t)
```
Enables uniform conversion configurations across slaves.

### Methods
Function Documentation

void ADC_MEASUREMENT_ADV_ConfigureChannelClass ( const A
const X
)

Configures the input class (Resolution and Sampling time).

Parameters:

iclass_selection  Select the input class to be configured
Pass
XMC_VADC_CHANNEL_CONV_GROUP_CLA
to configure group input class 0. Pass
XMC_VADC_CHANNEL_CONV_GROUP_CLA
to configure group input class 1.

config  Constant pointer to the input class configuration

return_num  The group number whose input class needs to be configured.

Returns:
None

Description:
Configures the input class for standard conversion (GxICLASS[0] OR GxICLASS[1]). A call to this API would configure the Resolution and sampling time for standard conversion. The group-specific input class registers define the sample time and data conversion mode for each channel of the respective group. And each channel can use these by selecting the input class in GxCHCTRy.ICLSEL.

Example Usage:

// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and Channel_B at position-1.
// Enable Wait for trigger for the Channel_A.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_QUEUE APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h>     //Declarations from DAVE Code Generation (includes SFR declarations)

uint16_t result[2];
void adc_measurement_adv_callback(void)
{
    // converted result will be of 8bit resolution.
    result[0] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[1] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
}

int main(void)
{
    DAVE_STATUS_t status;
    XMC_VADC_GROUP_CLASS_t res_8bit = {
        .conversion_mode_standard = XMC_VADC_CONVMODE_8BIT,
        .sample_time_std_conv     = 10U
    };
    status = DAVE_Init();           // Initialization of DAVE APPs
if(status == DAVE_STATUS_FAILURE) {
   // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
   XMC_DEBUG("DAVE APPs initialization failed\n");

   while(1U) {
   
   }
}

ADC_MEASUREMENT_ADV_ConfigureChannelClass(&ADC_MEASUREMENT_ADV_0_Channel_A,&res_8bit);

   // Start the selected request source
   ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

   // Start the queue conversion sequence
   ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);

   while(1U) {
   
   }
}
**XMC_VADC_CHANNEL_ALIAS_t ADC_MEASUREMENT_ADV_GetAlias**

Returns the configured alias value.

**Parameters:**

- `handle_ptr` constant pointer to the channel handle structure.  
  (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)

**Returns:**

- `XMC_VADC_CHANNEL_ALIAS_t` returns  
  `XMC_VADC_CHANNEL_ALIAS_DISABLED` if the alias is not applicable to the channel or if alias is not enabled  
  else it returns the alias value.

**Description:**

Return the alias value for the channel. If the alias feature is enabled then the channels CH-0 or CH-1 can convert any other channel's input signal. The API returns  
`XMC_VADC_CHANNEL_ALIAS_DISABLED` if the Channel is neither CH-0 nor CH-1. Also the value  
`XMC_VADC_CHANNEL_ALIAS_DISABLED` is returned when the CH-0 or CH-1 is not configured with alias. If either CH-0 or CH-1 is configured with alias then the appropriate aliased channel number is returned. A call to this API would access the register GxALIAS.

**Example Usage:**

```c
#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)

// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.  
// Select the request source APP as ADC_SCAN.  
// Generate the code and build.  
// Replace this in the main.c.
```
#define MAX_LOCAL_BUFFER (16U)
uint16_t result[16];
uint16_t i = 0U;

void channel_event_callback(void)
{
    result[i++] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    if(MAX_LOCAL_BUFFER == i)
    {
        i = 0U;
    }
}

int main(void)
{
    DAVE_STATUS_t status;
    XMC_VADC_CHANNEL_ALIAS_t alias_ch;

    status = DAVE_Init(); // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
        }
    }
}
// Check if the given channel is aliased.
alias_ch = ADC_MEASUREMENT_ADV_GetAliasValue(&ADC_MEASUREMENT_ADV_0_Channel_A);

if(XMC_VADC_CHANNEL_ALIAS_DISABLED == alias_ch)
{
    // do something
}

// Start the selected request source
ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

while(1U)
{
    // Continuously re-trigger the scan conversion sequence
    ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
}

Definition at line 1054 of file ADC_MEASUREMENT_ADV.c.

References ADC_MEASUREMENT_ADV_CHANNEL::ch_num, and ADC_MEASUREMENT_ADV_CHANNEL::group_index.

DAVE_APP_VERSION_t ADC_MEASUREMENT_ADV_GetAppVersion

Get ADC_MEASUREMENT_ADV APP version.

Returns:
    DAVE_APP_VERSION_t APP version information (major, minor and patch number)

Description:
The function can be used to check application software compatibility with a specific version of the APP.

Example Usage:

```c
#include <DAVE.h>

int main(void) {
    DAVE_STATUS_t init_status;
    DAVE_APP_VERSION_t version;

    // Initialize ADC_MEASUREMENT_ADV APP:
    // ADC_MEASUREMENT_ADV_Init() is called from within DAVE_Init().
    init_status = DAVE_Init();

    version = ADC_MEASUREMENT_ADV_GetAppVersion();
    if (version.major != 1U) {
        // Probably, not the right version.
    }

    // More code here
    while(1) {

    }
    return (0);
}
```

Definition at line 682 of file ADC_MEASUREMENT_ADV.c.

__STATIC_INLINE XMC_VADC_DETAILED_RESULT_t ADC_MEASUREMENT_ADV_GetDetailedResult

Returns the complete conversion result.

**Parameters:**

- **handle_ptr** constant pointer to the channel handle structure.
(Use the channel handle related macros which are defined in adc_measure_adv_conf.h)

**Returns:**

XMC_VADC_DETAILED_RESULT_t returns the complete result register

**Description:**

Return the completely 32 bit result register (GxRESy). In the APP each channel is configured to a particular group result register (excluding FIFO). The result of conversion as well as other information is returned from this API. The detailed result contains result of the most recent conversion, the channel number requested the conversion, valid flag, converted request source, fast compare result, the result data reduction counter and the EMUX channel number (if GxRES[0] only). In polling mechanism the converted result can be read out after checking the valid flag bit. The result register is defined in the channel handle structure ADC_MEASUREMENT_ADV_CHANNEL_t. Hence this API shall call be called with a pointer to the channel handle of type ADC_MEASUREMENT_ADV_CHANNEL_t (Directly use the channel handle related macros which are defined in adc_measure_adv_conf.h).

**Note:**

This API is not Applicable for reading the result from the result FIFO registers Use ADC_MEASUREMENT_ADV_GetFifoDetailedResult in order to read the FIFO result.

**Example Usage:**

```c
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and Channel_B at position-1.
```
// Enable Wait for trigger for the Channel_A.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_QUEUE APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)
#define QUEUE_SRC (0U)

XMC_VADC_DETAILED_RESULT_t result[2];
uint32_t queue_flag = 0U;

void adc_measurement_adv_callback(void)
{
    // Use the channel handle parameter in this format "<APP Name>_<CHANNEL Name>"
    result[0] = ADC_MEASUREMENT_ADV_GetDetailedResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[1] = ADC_MEASUREMENT_ADV_GetDetailedResult(&ADC_MEASUREMENT_ADV_0_Channel_B);

    if((result[0].converted_request_source == QUEUE_SRC) && (QUEUE_SRC == result[1].converted_request_source))
    {
        queue_flag++;
    }
}

int main(void)
{
    DAVE_STATUS_t status;
}
status = DAVE_Init(); // Initialization of DAVE APPs

if (status == DAVE_STATUS_FAILURE)
{
    // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
    XMC_DEBUG("DAVE APPs initialization failed\n");

    while (1U)
    {
    
    }
}

ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

    // Start the queue conversion sequence
ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);

    while(queue_flag != 0U);
    // do something

while(1U)
{
}

Definition at line 731 of file ADC_MEASUREMENT_ADV.h.

References ADC_MEASUREMENT_ADV_CHANNEL::ch_handle, and ADC_MEASUREMENT_ADV_CHANNEL::group_index.
Parameters:

handle_ptr constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)

Returns:

XMC_VADC_FAST_COMPARE_t fast conversion result.
Returns
::ADC_MEASUREMENT_ADV_FAST_COMPARE_HIGH if the sampled signal is greater than the compare value. Returns
::ADC_MEASUREMENT_ADV_FAST_COMPARE_LOW if the sampled signal is lower than the compare value. Returns
::ADC_MEASUREMENT_ADV_FAST_COMPARE_INVALID if there is no valid result available.

Description:

Returns the fast conversion result stored in the result register [GxRESy.FCR]. In the APP each channel is configured to a particular group result register (excluding FIFO). The result register is defined in the channel handle structure

ADC_MEASUREMENT_ADV_CHANNEL_t. Hence this API shall call be called with a pointer to the channel handle of type

ADC_MEASUREMENT_ADV_CHANNEL_t (Directly use the channel handle related macros which are defined in adc_measure_adv_conf.h).

Note:

This API is only applicable to read fast compare result of the particular channel.

Example Usage:

// Initialize the ADC_MEASUREMENT_ADV APP. Set th
e number of required channels to 1.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and enable the refill.
// Enable Wait for trigger for the Channel_A.
// Goto interrupts tab enable result event for Channel_A.
// Instantiate the interrupt APP.
// In the UI of the interrupt APP change the interrupt handler to "channel_event_callback"
// goto HW signal connectivity and connect event_result_Channel_A to interrupt APP.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h>       //Declarations from DAVE Code Generation (includes SFR declaration)
#define MAX_LOCAL_BUFFER (16U)
ADC_MEASUREMENT_ADV_FAST_COMPARE_t result[16];
uint16_t i = 0U;

void channel_event_callback(void)
{
    result[i++] = ADC_MEASUREMENT_ADV_GetFastCompareResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    if(MAX_LOCAL_BUFFER == i)
    {
        i = 0U;
    }
}

int main(void)
{
    DAVE_STATUS_t status;
status = DAVE_Init();  // Initialization of DAVE APPs

if(status == DAVE_STATUS_FAILURE)
{
    // Placeholder for error handler code. The while loop below can be replaced
    // with an user error handler.
    XMC_DEBUG("DAVE APPs initialization failed\n");

    while(1U)
    {
    }
}

    // Set the threshold value as Vdd/2
    ADC_MEASUREMENT_ADV_SetFastCompareValue(&ADC_MEASUREMENT_ADV_0_Channel_A, 512U);

    // Start the selected request source
    ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

    while(1U)
    {
        // Continuously trigger the queue conversion sequence
        ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
    }
}

Definition at line 1001 of file ADC_MEASUREMENT_ADV.h.

References ADC_MEASUREMENT_ADV_CHANNEL::ch_handle,
and `ADC_MEASUREMENT_ADV_CHANNEL::group_index`.

```c
__STATIC_INLINE XMC_VADC_DETAILED_RESULT_t ADC_MEASUREMENT_ADV_GetFifoDetailedResult
```

Returns the complete conversion result.

**Parameters:**

- `handle_ptr` constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in `adc_measure_adv_conf.h`)

**Returns:**

- `XMC_VADC_DETAILED_RESULT_t` returns the complete result register

**Description:**

Return the completely 32 bit FIFO result register (GxRESy). If result FIFO is configured then the results are available in the FIFO tail register. The result of conversion as well as other information is returned from this API. The detailed result contains result of the most recent conversion, the channel number requested the conversion, valid flag, converted request source, fast compare result, the result data reduction counter and the EMUX channel number (if GxRES[0] only). In polling mechanism the converted result can be read out after checking the valid flag bit. The result register is defined in the channel handle structure `ADC_MEASUREMENT_ADV_CHANNEL_t`. Hence this API shall call be called with a pointer to the channel handle of type `ADC_MEASUREMENT_ADV_CHANNEL_t` (Directly use the channel handle related macros which are defined in `adc_measure_adv_conf.h`).

**Note:**

This API is not Applicable for reading the result from channels which done use FIFO. Use `ADC_MEASUREMENT_ADV_GetDetailedResult` in order to
read from a single result register.

Example Usage:

```c
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and Channel_B at position-1.
// Enable Wait for trigger for the Channel_A.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_QUEUE APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)
#define MAX_FIFO (16U)
XMC_VADC_DETAILED_RESULT_t result[16];
uint16_t i = 0U;
void adc_measurement_adv_callback(void)
{
    // Use the channel handle parameter in this format "<APP Name>_<CHANNEL Name>"
    result[i++] = ADC_MEASUREMENT_ADV_GetFifoDetailedResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[i++] = ADC_MEASUREMENT_ADV_GetFifoDetailedResult(&ADC_MEASUREMENT_ADV_0_Channel_B);

    if(MAX_FIFO == i)
    {
        i = (uint32_t)0;
    }
```
int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init();    // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
            
        }
    }

    // Start the selected request source
    ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

    // Start the queue conversion sequence
    ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);

    while(1U)
    {
        
    }
}

Definition at line 908 of file ADC_MEASUREMENT_ADV.h.
Returns the conversion result.

**Parameters:**
- `handle_ptr` constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)

**Returns:**
- `uint16_t` conversion result.
- Range: [0x0 to 0xFFF] without any filters/accumulation/subtraction enabled.

**Description:**
Returns the converted result stored in the result FIFO register [GxRESy.RESULT]. If result FIFO is configured then the results are available in the FIFO tail register. The result register can only be read at the tail of the FIFO. This result register number is defined in the channel handle structure `ADC_MEASUREMENT_ADV_CHANNEL_t`. Hence this API shall call be called with a pointer to the channel handle of type `ADC_MEASUREMENT_ADV_CHANNEL_t` (Directly use the channel handle related macros which are defined in adc_measure_adv_conf.h).

**Note:**
This API is not Applicable for reading the result from channels which done use FIFO. Use `ADC_MEASUREMENT_ADV_GetResult` in order to read from a single result register.

**Example Usage:**
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and Channel_B at position-1.
// Enable Wait for trigger for the Channel_A.
// Goto postprocessing Tab and select 8 stages FIFO for both Channel_A and Channel_B
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_QUEUE APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.

#include <DAVE.h>

//Declarations from DAVE Code Generation (includes SFR declaration)
#define MAX_FIFO (16U)
uint16_t result[16];
uint16_t i;
void adc_measurement_adv_callback(void)
{
    // Use the channel handle parameter in this format "<APP Name>_<CHANNEL Name>"
    result[i++] = ADC_MEASUREMENT_ADV_GetFifoResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[i++] = ADC_MEASUREMENT_ADV_GetFifoResult(&ADC_MEASUREMENT_ADV_0_Channel_B);

    if(MAX_FIFO == i)
    {
        i = (uint32_t)0;
    }
}
int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init();       // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
        }
    }

    // Start the selected request source
    ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

    // Start the queue conversion sequence
    ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);

    while(1U)
    {
    }
}
ADC_MEASUREMENT_ADV_CHANNEL::group_index, and ADC_MEASUREMENT_ADV_CHANNEL::result_fifo_tail_number.

__STATIC_INLINE uint16_t ADC_MEASUREMENT_ADV_GetResult (handle_ptr)

Returns the conversion result.

**Parameters:**
- **handle_ptr** constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)

**Returns:**
- uint16_t conversion result.
- Range: [0x0 to 0xFFF] without any filters/ accumulation/subtraction enabled.

**Description:**
- Return the converted result stored in the result register [GxRESy.RESULT]. In the APP each channel is configured to a particular group result register (excluding FIFO). The result register is defined in the channel handle structure ADC_MEASUREMENT_ADV_CHANNEL_t. Hence this API shall call be called with a pointer to the channel handle of type ADC_MEASUREMENT_ADV_CHANNEL_t (Directly use the channel handle related macros which are defined in adc_measure_adv_conf.h).

**Note:**
- This API is not Applicable for reading the result from the result FIFO registers Use ADC_MEASUREMENT_ADV_GetFifoResult in order to read the FIFO result.

**Example Usage:**
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and Channel_B at position-1.
// Enable Wait for trigger for the Channel_A.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_QUEUE APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)

uint16_t result[2];
void adc_measurement_adv_callback(void) {
    // Use the channel handle parameter in this format "<APP Name>_<CHANNEL Name>"
    result[0] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[1] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
}

int main(void) {
    DAVE_STATUS_t status;
    
    status = DAVE_Init();  // Initialization of DAVE APPs
    
    if(status == DAVE_STATUS_FAILURE) {
        
    
}
// Placeholder for error handler code. The while loop below can be replaced with an user error handler.

XMC_DEBUG("DAVE APPs initialization failed\n");

while (1U)
{

}

// Start the selected request source
ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

// Start the queue conversion sequence
ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);

while (1U)
{

}

Definition at line 638 of file ADC_MEASUREMENT_ADV.h.

References ADC_MEASUREMENT_ADV_CHANNEL::ch_handle, and ADC_MEASUREMENT_ADV_CHANNEL::group_index.

**ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_Init**

Initializes the **ADC_MEASUREMENT_ADV** APP Instance.

**Parameters:**
- **handle_ptr** constant pointer to the APP handle structure
Returns:
None

Description:
Initialize the ADC and all the required configurations. A call to this API would initialize the queue request source or the scan request source depending on the GUI selection. The initialization is taken up by calling ADC_QUEUE_Init() or ADC_SCAN_Init(). Following this the synchronous conversion related initializations are taken up. In the sync initialization the slave groups and the master group are powered down and either GxSYNCTR.STSEL (if Slave) or GxSYNCTR.EVALRy (if master) are configured. After the sync related configurations are completed the master group alone is powered on. Following this the result event or channel event related service request node configurations are done (if required). Then the GxCHCTR configurations are completed. After the channel initialization the result handling initializations are done. This entails configuring the GxRCR registers for result filtering, accumulation, subtraction and FIFO. After all these initialization are completed the channels configured in the GUI is inserted into the appropriate ADC_QUEUE or ADC_SCAN APP buffer. If the GUI check box "Insert channels at initialization" is enable then these entries is pushed to the Hardware.

Example Usage:

```c
#include <DAVE.h>
int main(void)
{
    DAVE_Init(); //ADC_MEASUREMENT_ADV_Init is called within DAVE_Init
    return 0;
}
```

Definition at line 796 of file ADC_MEASUREMENT_ADV.c.
void ADC_MEASUREMENT_ADV_SelectBoundary ( const ADC_MEASUREMENT_ADV_CHANNEL_t XMC_VADC_BOUNDARY_SELECT_t XMC_VADC_CHANNEL_BOUNDARY_t )

Select the boundary for the channel.

**Parameters:**

- **handle_ptr** constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)
- **boundary** The lower boundary or upper boundary of the channel to be configured.
- **boundary_selection** Boundary register selection for the particular channel.

**Returns:**

None
Description:
Select the boundary for the channel. This API will select either lower boundary or the upper boundary depending on boundary and configures the boundary_selection value into the GxCHCTRy.BNDSELL or GxCHCTRy.BNDSELU accordingly.

Example Usage:

```c
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)

#define MAX_LOCAL_BUFFER (16U)
uint16_t result[16];
uint16_t i = 0U;

void channel_event_callback(void)
```
{  
    result[i++] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    if (MAX_LOCAL_BUFFER == i)  
    {  
        i = 0U;
    }
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init();  // Initialization of DAVE APPs

    if (status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while (1U)
        {
        }
    }

    // Set the boundary selection for Channel_A lower Bound as Group Bound-1
    ADC_MEASUREMENT_ADV_SelectBoundary(&ADC_MEASUREMENT_ADV_0_Channel_A, XMC_VADC_BOUNDARY_SELECT_LOWER_BOUND,
                                        XMC_VADC_CHANNEL_BOUNDARY_GROUP_BOUND1);
// Set the boundary selection for Channel_A upper Bound as Group Bound-0 and also the value as 2048U
ADC_MEASUREMENT_ADV_SetBoundaryUpper(&ADC_MEASUREMENT_ADV_0_Channel_A, 2048U);

// Start the selected request source
ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

while(1U)
{
  // Continuously trigger the queue conversion sequence
  ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
}

Definition at line 1014 of file ADC_MEASUREMENT_ADV.c.

References ADC_MEASUREMENT_ADV_CHANNEL::ch_num, and ADC_MEASUREMENT_ADV_CHANNEL::group_index.

void ADC_MEASUREMENT_ADV_SetAlternateReference (const ADC_MEASUREMENT_ADV_CHANNEL_t * handle_ptr)

Selects alternate reference voltage for the channel.

Parameters:

- **handle_ptr** constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)
**reference_select** Voltage reference for the channel.

**Returns:**
None

**Description:**
Select the reference voltage for conversion. For XMC4000 series, an internal voltage reference (VARef) or an external voltage reference fed to Ch-0 can serve as a voltage reference for conversions. For XMC1000 series, an internal ground reference (Vss) or an external reference ground from CH-0 can serve as an alternate reference. A call to this API would configure the register bit field GxCHCTR.REFSEL.

**Example Usage:**

```c
// Initialize the ADC_MEASUREMENT_ADV ADV APP. Set the number of required channels to 2.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_SCAN APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)

uint16_t result[2];
void adc_measurement_adv_callback(void)
{
    result[0] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[1] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
}
```
int main(void) {
    DAVE_STATUS_t status;

    status = DAVE_Init();    // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE) {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U) {

        }
    }

    // Start the selected request source
    ADC_MEASUREMENTADV_StartADC(&ADC_MEASUREMENTADV_0);

    // connect the alternate reference to the channel-0 pin of Channel_A's group.
    ADC_MEASUREMENTADV_SetAlternateReference(&ADC_MEASUREMENTADV_0_Channel_A, XMC_VADC_CHANNEL_REF_ALT_CH0);

    while(1U) {
        // Continuously re-trigger the scan conversion sequence
        ADC_MEASUREMENTADV_SoftwareTrigger(&ADC_MEASUREMENTADV_0);
void ADC_MEASUREMENT_ADV_SetBoundaryLower ( const ADC_MEASUREMENT_ADV_CHANNEL_t uint32_t )

Sets the lower boundary value for the channel.

**Parameters:**
- **handle_ptr** constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)
- **boundary_value** The boundary value that needs to be configured in the lower boundary register.

**Returns:**
None

**Description:**
Sets the lower boundary value for the channel. This API will set the lower boundary value depending on boundary selected for the channel in the GUI of the APP. Thus also configured in the API is either the register GLOBBOUND or GxBOUND.

**Note:**
This API will configure the lower boundary for the channel according to the selection done in the GUI. Runtime change of the lower boundary selection will not be handled by this API.
Example Usage:

// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 1.
// Select the request source APP as ADC_SCAN.
// Goto the Boundary Settings tab and select the channel name as Channel_A.
// Select the Upper boundary as Group boundary 1.
// Change the channel event from No to "If Result Inside Band".
// Change the "Generate boundary flag" to Yes/Non-Inverted.
// Instantiate the interrupt APP.
// In the UI of the interrupt APP change the interrupt handler to "channel_event_callback"
// goto HW signal connectivity and connect event_channel_Channel_A to interrupt APP.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)

#define MAX_LOCAL_BUFFER (16U)
uint16_t result[16];
uint16_t i = 0U;

void channel_event_callback(void)
{
    result[i++] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    if(MAX_LOCAL_BUFFER == i)
    {
        i = 0U;
    }
}
int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init(); // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
            // Set the boundary selection for Channel_A lower Bound as Group Bound-1
            ADC_MEASUREMENT_ADV_SetBoundaryLower(&ADC_MEASUREMENT_ADV_0_Channel_A,1024);

            // Set the boundary selection for Channel_A upper Bound as Group Bound-0 and also the value as 2048U
            ADC_MEASUREMENT_ADV_SetBoundaryUpper(&ADC_MEASUREMENT_ADV_0_Channel_A,2048);

            // Start the selected request source
            ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

            while(1U)


Continuously re-trigger the scan conversion sequence:
```c
    ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
```

Definition at line 1040 of file `ADC_MEASUREMENT_ADV.c`.
References `ADC_MEASUREMENT_ADV_CHANNEL::ch_handle`.

```c
void ADC_MEASUREMENT_ADV_SetBoundaryUpper ( const ADC_uint32_t 
)
```

Sets the upper boundary value for the channel.

**Parameters:**
- **handle_ptr** constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in `adc_measure_adv_conf.h`)
- **boundary_value** The boundary value that needs to be configured in the upper boundary register.

**Returns:**
None

**Description:**
Sets the upper boundary value for the channel. This API will set the upper boundary value depending on boundary selected for the channel in the GUI of the APP. Thus also configured in the API is either the register GLOBBOUND or GxBOUND.

**Note:**
This API will configure the upper boundary for the channel according to the selection done in the GUI. Runtime change of the Upper boundary selection will not be handled by this API.

Example Usage:

```c
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 1.
// Select the request source APP as ADC_SCAN.
// Goto the Boundary Settings tab and select the channel name as Channel_A.
// Select the Upper boundary as Group boundary 1.
// Change the channel event from No to "If Result Inside Band".
// Change the "Generate boundary flag" to Yes/Non-Inverted.
// Instantiate the interrupt APP.
// In the UI of the interrupt APP change the interrupt handler to "channel_event_callback"
// goto HW signal connectivity and connect event_channel_Channel_A to interrupt APP.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)

#define MAX_LOCAL_BUFFER (16U)
uint16_t result[16];
uint16_t i = 0U;

void channel_event_callback(void)
{
    result[i++] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    if(MAX_LOCAL_BUFFER == i)
    {
    
```
int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init(); // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
        }
    }

    // Set the boundary selection for Channel_A lower Bound as Group Bound-1
    ADC_MEASUREMENT_ADV_SetBoundaryLower(&ADC_MEASUREMENT_ADV_0_Channel_A,1024);

    // Set the boundary selection for Channel_A upper Bound as Group Bound-0 and also the value as 2048U
    ADC_MEASUREMENT_ADV_SetBoundaryUpper(&ADC_MEASUREMENT_ADV_0_Channel_A,2048);

    // Start the selected request source


```c
ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

while(1U)
{
    // Continuously re-trigger the scan conversion sequence
    ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
}
```

Definition at line 1027 of file `ADC_MEASUREMENT_ADV.c`.

References `ADC_MEASUREMENT_ADV_CHANNEL::ch_handle`.

```c
void ADC_MEASUREMENT_ADV_SetChannelGain (const ADC_MEASUREMENT_ADV_CHANNEL_t handle_ptr,
                                            const ADC_MEASUREMENT_ADV_GAIN_t gain_factor)
```

Sets the channel gain.

**Parameters:**
- `handle_ptr` constant pointer to the channel handle structure.
  
  (Use the channel handle related macros which are defined in `adc_measure_adv_conf.h`)

- `gain_factor` The gain factor value.

**Returns:**
None

**Description:**
Set the gain value for the particular channel. This API would set the SHS gain factor for the channel. The input voltage will get a gain proportional to the selected `gain_factor` then will be converted by the ADC.
Example Usage:

```c
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_SCAN APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)

uint16_t result[2];
void adc_measurement_adv_callback(void)
{
    result[0] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[1] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init(); // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");
    }
```
while(1U) {
}

// Start the selected request source
ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

// select the gain value of 12
ADC_MEASUREMENT_ADV_SetChannelGain(&ADC_MEASUREMENT_ADV_0_Channel_A, ADC_MEASUREMENT_ADV_GAIN_12);

while(1U) {
    // Continuously re-trigger the scan conversion sequence
    ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
}

Definition at line 992 of file ADC_MEASUREMENT_ADV.c.

References ADC_MEASUREMENT_ADV_CHANNEL::ch_handle, ADC_MEASUREMENT_ADV_CHANNEL::ch_num, and ADC_MEASUREMENT_ADV_CHANNEL::group_index.

ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV\n
Sets the fast conversion value.
Parameters:

- **handle_ptr** constant pointer to the channel handle structure. (Use the channel handle related macros which are defined in adc_measure_adv_conf.h)
- **compare_value** constant pointer to the channel handle structure.

Returns:

None

Description:

Returns the converted result stored in the result register [GxRESy.RESULT]. In the APP each channel is configured to a particular group result register. If FIFO is enabled for the particular channel this reads the fast compare result from the FIFO tail register. The result register is defined in the channel handle structure **ADC_MEASUREMENT_ADV_CHANNEL_t**. Hence this API shall be called with a pointer to the channel handle of type **ADC_MEASUREMENT_ADV_CHANNEL_t** (Directly use the channel handle related macros which are defined in adc_measure_adv_conf.h).

Example Usage:

```c
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 1.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and enable the refill.
// Enable Wait for trigger for the Channel_A.
// Goto interrupts tab enable result event for Channel_A.
// Instantiate the interrupt APP.
// In the UI of the interrupt APP change the interrupt handler to "result_event_callback"
```
// goto HW signal connectivity and connect event_result_Channel_A to interrupt APP.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)

#define MAX_LOCAL_BUFFER (16U)
ADC_MEASUREMENT_ADV_FAST_COMPARE_t result[16];
uint16_t i = 0U;

void result_event_callback(void)
{
    result[i++] = ADC_MEASUREMENT_ADV_GetFastCompareResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    if(MAX_LOCAL_BUFFER == i)
    {
        i = 0U;
    }
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init(); // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");
    }
while(1U)
{
}
}

// Set the threshold value as Vdd/2
ADC_MEASUREMENT_ADV_SetFastCompareValue(&ADC_MEASUREMENT_ADV_0_Channel_A,512U);

// Start the selected request source
ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

while(1U)
{
  // Continuously trigger the queue conversion sequence
  ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
}

Definition at line 921 of file ADC_MEASUREMENT_ADV.c.

References ADC_MEASUREMENT_ADV_STATUS_FAILURE, ADC_MEASUREMENT_ADV_STATUS_SUCCESS, ADC_MEASUREMENT_ADV_CHANNEL::ch_handle, and ADC_MEASUREMENT_ADV_CHANNEL::group_index.

**void** ADC_MEASUREMENT_ADV_SetIclass (const **ADC_MEASUREMENT_ADV**

Enables uniform conversion configurations across slaves.

**Parameters:**

  * **handle_ptr** constant pointer to the channel handle structure.
Returns:
None

Description:
Enables uniform conversion configurations across slaves. The ADC_QUEUE configures the input class settings for the master group. When slaves need to convert the input signals at the same configuration as the master group then a call to this API is needed. A call to this API will ensure that the master and the slave channels are converting the input signals at the same resolution and sampling time.

Note:
- ADC ADVANCE_MEASURE APP will configure the input class used by ADC_QUEUE or ADC_SCAN in the channel configuration. Thus the input class is either GxICLASS[0] or GxICLASS[1]. The slaves channels are also configured by the same input class number in the GxCHCTR.ICLSEL. A call to this API will copy the configurations from the master groups input class(GxICLASS[z]) to the slaves groups input class (GyICLASS[z], where in x is the master group, y is the slave group and z is the input class number which is common across master and slave).
- The channel iclass can be changed to global iclass at any time by calling runtime APIs. This API cannot support such a situation.

Example Usage:

// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 1.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at
position-0 and enable the refill.
// Enable Wait for trigger for the Channel_A.
// Goto the Sync. Conversion tab and select the synchronized groups as 1 slave.
// Goto the Sync Master check box for the Channel_A and enable it.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_QUEUE APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.

#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)

#define MAX_LOCAL_BUFFER (16U)
uint16_t result[16], sync_result[16];
uint16_t i = 0U;

void adc_measurement_adv_callback(void)
{
    result[i++] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    sync_result[i++] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A_SLAVE_A);
    if(MAX_LOCAL_BUFFER == i)
    {
        i = 0U;
    }
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init();  // Initializati
on of DAVE APPs

    if (status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while (1U)
        {
            
        }
    }

    // Set the same conversion characteristics for the slave groups channels as well
    ADC_MEASUREMENT_ADV_SetIclass(&ADC_MEASUREMENT_ADV_0);

    // Start the selected request source
    ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

    while (1U)
    {
        
    }
}

Definition at line 1090 of file ADC_MEASUREMENT_ADV.c.

References ADC_MEASUREMENT_ADV::group_index.

void ADC_MEASUREMENT_ADV_SetSubtractionValue (const ADC
Set the subtraction value if a result difference mode is required.

**Parameters:**

- **handle_ptr**
  - constant pointer to the APP handle structure

- **subtraction_alignment**
  - The result alignment in the result register.

- **subtraction_value**
  - constant value that is used for subtraction.

**Returns:**

None

**Description:**

Set the subtraction value in the result register 0 [GxRES[0].RESULT]. In the subtraction mode the result register 0 is used as the subtrahend. Any channel in the GUI which has selected the subtraction mode will have its converted value subtracted from the value stored in the result register-0. This can be used as an offset for the converted values.

**Example Usage:**

```c
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Select the request source APP from ADC_SCAN to ADC_QUEUE.
// Goto the sequence plan and select Channel_A at position-0 and Channel_B at position-1.
// Enable Wait for trigger for the Channel_A.
// Goto interrupts tab enable request source interrupt.
```
Goto the ADC_QUEUE APP and enable the request source interrupt.
Generate the code and build.
Replace this in the main.c.

```c
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)

#define VOLTAGE_OFFSET (50U)
uint16_t result[2];
void adc_measurement_adv_callback(void)
{
    result[0] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[1] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init(); // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
        }
    }
```
void ADC_MEASUREMENT_ADV_SoftwareTrigger (const ADC_MEASUREMENT_ADV_t handle_ptr) {

Starts the conversion of the required channels.

Parameters:
handle_ptr constant pointer to the APP handle structure

Returns:
None

Description:
Trigger a load event for the required channels thus starting the conversion of the ADC channels. If scan request source is selected then this API would write to GxASMR.LDEV bit, causing the conversion to start. If queue request source is selected then this API would write to GxQMR0.TREV bit. At the time of writing to TREV bit for the queue. If there was queue entry waiting in the queue buffer for a hardware trigger, writing to TREV bit triggers the conversion for that entry.

Example Usage:

```c
// Initialize the ADC_MEASUREMENT_ADV APP. Set the number of required channels to 2.
// Goto interrupts tab enable request source interrupt.
// Goto the ADC_SCAN/ADC_QUEUE APP and enable the request source interrupt.
// Generate the code and build.
// Replace this in the main.c.
#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)

uint16_t result[2];
void adc_measurement_adv_callback(void)
{
    result[0] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result[1] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
}

int main(void)
{
    DAVE_STATUS_t status;
    status = DAVE_Init();  // Initialization
```
on of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");
        
        while(1U)
        {

        }
    }

    // Start the selected request source
    ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);
    
    while(1U)
    {
        // Continuously re-trigger the scan conversion sequence
        ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
    }

Definition at line 718 of file ADC_MEASUREMENT_ADV.c.

References ADC_MEASUREMENT_ADV::group_index, and ADC_MEASUREMENT_ADV::req_src.

void ADC_MEASUREMENT_ADV_StartADC (const ADC_MEASUREME
Starts the VADC request source.

**Parameters:**
- `handle_ptr` constant pointer to the APP handle structure

**Returns:**
- None

**Description:**
This API would start the request source of the VADC. After this API is invoked the ADC is ready for accepting conversion requests. If scan request source is selected then this API would write to GxARBPR.ASEN1 bit. If queue request source is selected then this API would write to GxARBPR.ASEN0 bit. At the time of writing to ASEN0 bit for the queue, If there was queue entry waiting in the queue buffer without waiting for a hardware trigger, this API would start the conversion of such a queue.

**Example Usage:**

```c
#include <DAVE.h>

#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)

uint16_t result[2];
void adc_measurement_adv_callback(void)
{
```
result[0] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
result[1] = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init(); // Initialization of DAVE APPs

    if(status == DAVE_STATUS_FAILURE)
    {
        // Placeholder for error handler code. The while loop below can be replaced with an user error handler.
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
            
        }
    }

    // Start the selected request source
    ADC_MEASUREMENT_ADV_StartADC(&ADC_MEASUREMENT_ADV_0);

    while(1U)
    {
        // Continuously re-trigger the scan conversion sequence
        ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
    }
Definition at line 695 of file ADC_MEASUREMENT_ADV.c.

References ADC_MEASUREMENT_ADV::group_index, and ADC_MEASUREMENT_ADV::req_src.

Referenced by ADC_MEASUREMENT_ADV_Init().
## Usage

The following examples demonstrate some of the use cases of the ADC_MEASUREMENT_ADV APP.

1. **Example to illustrate the autoscan mode (Software start continuous mode).**
2. **Example to illustrate the software triggered conversions (Software start single shot mode).**
3. **Example to illustrate the hardware triggered conversions (Hardware trigger single shot mode).**
4. **Example to illustrate synchronous conversions.**

---

**Use case 1:** The use case illustrates the use of the "autoscan mode" (continuous conversion). An API call starts the conversion round. After one conversion round (2 Channels) the round is automatically repeated. Hence no additional start of the conversion round is necessary. The result is cautiously read in the main loop.

In this example the "ADC_SCAN_APP" is used. See following configuration.

**Note:** The conversion result is read continuously and it is independent of ADC result being ready.

The use case can also be handled with "ADC_QUEUE_APP" by disabling all "Wait for trigger" in the "Sequence Plan" tab.

**Note:** This configuration will immediately start the conversion after the ADC start (GUI checkbox "Start after initialization" or API for start ADC). This use case is not shown in this example.

---

**Instantiate the required APPs**
Drag an instance of **ADC_MEASUREMENT_ADV** APP. Update the fields in the GUI of the APP with the following configuration.

**Configure the APP:**

**ADC_MEASUREMENT_ADV_0 APP:**

1. Set the number of channels to 2.

2. Goto General settings Tab.
3. Check the "Enable continuous Conversion" check box.
Manual pin allocation

4. Select the potentiometer Pin present in the boot kit and also another pin in available for the select board. **Note:** The pin number is specific to the development board chosen to run this example. The pin shown in the image above may not be available on every XMC boot kit. Ensure that a proper pin is selected according to the board.

Generate code
Files are generated here: '<project_name>/Dave/Generated/'
('project_name' is the name chosen by the user during project creation).
APP instance definitions and APIs are generated only after code generation.

- **Note:** Code must be explicitly generated for every change in the GUI configuration.
  **Important:** Any manual modification to the APP specific generated files will be overwritten by a subsequent code generation operation.

**Sample Application (main.c)**

```c
#include <DAVE.h>  //Declarations from DAVE Code Generation (includes SFR declaration)

int main(void)
{
    DAVE_STATUS_t status;
    uint16_t resultA, resultB;

    status = DAVE_Init();  /* Initialization of DAVE APPs */

    if(status == DAVE_STATUS_FAILURE)
    {
        /* Placeholder for error handler code. The while loop below can be replaced with an user error handler. */
        XMC_DEBUG("DAVE APPs initialization failed\n");

        while(1U)
        {
```
```c
```
One Software trigger to start the conversion of the 2 channels.
// After each conversion round (i.e. after 2 channels) the conversion will be automatically repeated.

ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
while(1U)
{
    //continuously read the result value.
    //NOTE: The result read doesn't wait for the new result to be ready.
    resultA = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    resultB = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
    XMC_UNUSED_ARG(resultA);
    XMC_UNUSED_ARG(resultB);
}

Build and Run the Project

Observation
The measurement is started with the
ADC_MEASUREMENT_ADV_SoftwareTrigger() API call. The conversion results are read continuously and stored in "resultA" and "resultB" variable.

Use case 2:
The use case illustrates the software triggered conversions. In this use case a sequence of 2 ADC Channels will be converted. The conversion round is started with an API. After one conversion round (2 Channels)
the conversion is stopped and waits for the next software start. After each conversion round a result event is generated and the ADC results are read in the ISR. After the results have been read, a new conversion is started with an API call.
In this example the "ADC_QUEUE_APP" is used. See following configuration.

The use case can also be handled with "ADC_SCAN_APP". This use case is not shown in this example.

**Note:** This use case uses an INTERRUPT APP for the result events. For those configurations refer to INTERRUPT APP help.
**Note:** In "ADC_SCAN_APP" the conversion sequence depends on the channels consumed. If Channel_A consumes a lower channel number than Channel_B then Channel_B would be converted first. Hence care needs to be taken when a similar setup is done while using ADC_SCAN_APP.

**Instantiate the required APPs**
Drag an instance of ADC_MEASUREMENT_ADV APP and one instance of INTERRUPT APP. Update the fields in the GUI of the APP with the following configuration.

**Configure the APP:**
ADC_MEASUREMENT_ADV_0 APP:

1. Set the number of channels to 2.
2. Select the request source as ADC_QUEUE_APP.
3. Goto Channel configuration Tab.
4. Enable the result event for Channel_B.
5. Goto Sequence Plan Tab.
6. Select the Queue Position-0 as Channel-A, Position-1 as Channel_B.
7. Select "Wait for Trigger" for Channel-A.
Manual pin allocation

8. Select the potentiometer Pin present in the boot kit and also another pin in available for the select board.

Note: The pin number is specific to the development board chosen to run this example. The pin shown in the image above may not be available on every XMC boot kit. Ensure that a proper pin is selected according to the board.
HW Signal Connections

9. Connect the ADC_MEASUREMENT_ADV_0 event_res_Channel_B signal to INTERRUPT_0 sr_irq

Generate code
Files are generated here: '<project_name>/Dave/Generated/
('project_name' is the name chosen by the user during project creation). APP instance definitions and APIs are generated only after code generation.

- **Note:** Code must be explicitly generated for every change in the GUI configuration.
- **Important:** Any manual modification to the APP specific generated files will be overwritten by a subsequent code generation operation.
Sample Application (main.c)

```c
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)

uint16_t resultA,resultB;

// Result event for Channel_B
void UserIRQHandler(void)
{
    // read the results of the conversion
    resultA = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    resultB = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);

    //Start the next round of conversion
    ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init(); /* Initialization of DAVE APPs */

    if(status == DAVE_STATUS_FAILURE)
    {
        /* Placeholder for error handler code. The while loop below can be replaced with an user error handler. */
        XMC_DEBUG("DAVE APPs initialization failed\n")
    }
}
```
// One Software trigger to start the conversion of the 2 channels.
ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0);
while(1U)
{

}

Build and Run the Project

Observation
The measurement is started with the ADC_MEASUREMENT_ADV_SoftwareTrigger() API call. After Channel_B is converted an event is triggered and executes the ISR UserIRQHandler(). In this ISR the results of Channel_A and Channel_B are stored in "resultA" and "resultB" variables. The API call ADC_MEASUREMENT_ADV_SoftwareTrigger(&ADC_MEASUREMENT_ADV_0) in the ISR starts a new conversion round.

Use case 3:
The use case illustrates hardware triggered conversions. A hardware trigger starts the conversion round of 3 Channels. When the conversion round is finished a source interrupt is executed. In the ISR the results are read. The ADC waits now for a new hardware trigger.

In this example the "ADC_QUEUE_APP" is used, this allows a user defined sequence. See following configuration.
The use case can also be handled with "ADC_SCAN_APP". In that case it is mandatory to disable the "autoscan mode". Also the "result event" can be used to read the ADC results. Both the use cases are not handled in this example.

**Note:** The 10kHz hardware trigger is generated with a PWM APP. For PWM configuration refer to the PWM APP help.

**Instantiate the required APPs**
Drag an instance of `ADC_MEASUREMENT_ADV` APP and one instance of PWM APP. Update the fields in the GUI of the APP with the following configuration.

**Configure the APP:**
`ADC_MEASUREMENT_ADV_0` APP:

1. Set the number of channels to 3.
2. Set the request source to `ADC_QUEUE_APP`.

4. Select the Trigger edge as Rising Edge.

5. Goto Sequence Plan Tab.
6. Select the Queue Position-0 as Channel-C, Position-1 as Channel-A, Position-2 as Channel_B.
7. Select "Wait for Trigger" for Channel-C.
8. Select "Source Event" for Channel-B.
9. Goto Interrupts Tab.
10. Check the “Enable callback” check box.
11. Change the Callback function to queue_src_event.

**Manual pin allocation**

12. Select the potentiometer Pin present in the boot kit and also the
other pins which are available for the select board.  
**Note:** The pin number is specific to the development board chosen to run this example. The pin shown in the image above may not be available on every XMC boot kit. Ensure that a proper pin is selected according to the board.

**HW Signal Connections**

5. Connect the PWM_0 event_period_match signal to ADC_MEASUREMENT_ADV_0 trigger_input.  
**Note:** Ensure that the period match event in the PWM_0 is enabled.

**Generate code**
Files are generated here: `<project_name>/Dave/Generated/`  
(‘project_name’ is the name chosen by the user during project creation).
APP instance definitions and APIs are generated only after code generation.

- **Note:** Code must be explicitly generated for every change in the GUI configuration.
  **Important:** Any manual modification to the APP specific generated files will be overwritten by a subsequent code generation operation.

**Sample Application (main.c)**

```c
#include <DAVE.h>                         //Declarations from DAVE Code Generation (includes SFR declaration)

uint16_t resultA,resultB,resultC;

// Request source event for Channel_B
void queue_src_event(void)
{
    // read the results of the conversion
    resultA = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    resultB = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_B);
    resultC = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_C);
}

int main(void)
{
    DAVE_STATUS_t status;

    status = DAVE_Init();                 /* Initialization of DAVE APPs */
```
if(status == DAVE_STATUS_FAILURE)
{
    /* Placeholder for error handler code. The while loop below can be replaced with an user error handler. */
    XMC_DEBUG("DAVE APPs initialization failed\n");

    while(1U)
    {
    
    }
}

// Start the PWM trigger.
PWM_Start(&PWM_0);
while(1U)
{
}

Build and Run the Project

Observation
The measurement is started after initialization and waits for a hardware trigger. With a hardware trigger Channel_C, Channel_A and Channel_B are converted in this sequence. The source event for Channel_B is enabled and execute the ISR queue_src_event(void). In this ISR the results of Channel_A, Channel_B and Channel_C are stored in "resultA", "resultB" and "resultC" variables. A new sequence is started with the next hardware trigger

Use case 4:
The use case illustrates synchronous conversions. The synchronous conversion is used when one or more slaves are
busy while the master starts a request. In this case the master group cancel the slave group conversion, wait until all slaves are ready and starts a conversion. This provides an exact synchronous conversion.

In this use case one ADC_MEASUREMENT_ADV APPs are used. The APP is used to configure the master which is controlled by a hardware trigger. The result event of the master is generating an interrupt where both results are read.

The ADC_MEASUREMENT_ADV APP for master is using the "ADC_QUEUE_APP" with a hardware trigger input. A Sync. Conversion with one slave is configured, see following configuration. The ADC_MEASUREMENT_ADV APP for master is triggered by a hardware trigger. See Use Case 3 for configuration.

**Instantiate the required APPs**
Drag an instance of ADC_MEASUREMENT_ADV APP, one instance of PWM APP and INTERRUPT APP. Update the fields in the GUI of the APP with the following configuration.

**Configure the APP:**
ADC_MEASUREMENT_ADV_0 APP:

1. Set the request source to ADC_QUEUE_APP.
2. Goto Channel configuration Tab.
3. Enable the result event for the Channel_A.
9. Goto Sync. conversion Tab.
10. Select the synchronized groups to "1 Slaves".
11. Enable the Configure global ICLASS-1.
12. Enable the sync master checkbox for Channel_A.
Manual pin allocation

12. Select the potentiometer Pin present in the boot kit and also the other pins which are available for the select board.

Note: The pin number is specific to the development board chosen to run this example. The pin shown in the image above may not be available on every XMC boot kit. Ensure that a proper pin is selected according to the board.

HW Signal Connections
5. Connect the PWM_0 event_period_match signal to ADC_MEASUREMENT_ADV_0 trigger_input. **Note:** Ensure that the event_period_match in the PWM_0 is enabled.

![HW Signal Connections](image1)

5. Connect the ADC_MEASUREMENT_ADV_0 event_res_Channel_A signal to INTERRUPT_0 sr_irq.

![HW Signal Connections](image2)

**Generate code**
Files are generated here: `<project_name>/Dave/Generated/` ('project_name' is the name chosen by the user during project creation). APP instance definitions and APIs are generated only after code generation.

- **Note:** Code must be explicitly generated for every change in the GUI configuration.
- **Important:** Any manual modification to the APP specific generated files will be overwritten by a subsequent code generation operation.
Sample Application (main.c)

```
#include <DAVE.h>    //Declarations from DAVE Code Generation (includes SFR declaration)

  uint16_t resultA,result_slaveA,result_slaveB,result_slaveC;

  // Result event for Channel_A
  void UserIRQHandler(void)
  {
    // read the results of the conversion
    resultA = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_A);
    result_slaveA = ADC_MEASUREMENT_ADV_GetResult(&ADC_MEASUREMENT_ADV_0_Channel_Async_slaveA);
  }

  int main(void)
  {
    DAVE_STATUS_t status;

    status = DAVE_Init();   /* Initialization of DAVE APPs */

    if(status == DAVE_STATUS_FAILURE)
    {
      /* Placeholder for error handler code. The while loop below can be replaced with an user error handler. */
      XMC_DEBUG("DAVE APPs initialization failed\n");

      while(1U)
  ```
{ }

// start the trigger signal for the ADC_MEASUREMENT_ADV_0
  PWM_Start(&PWM_0);

  while(1U)
  {
  
  
  }

Build and Run the Project

Observation
The voltage at sync pins are converted for each rising edge of the PWM signal. The conversion results are stored in "resultA" and "result_slaveA" variables.
## ADC_MEASUREMENT_ADV

<table>
<thead>
<tr>
<th>Home</th>
</tr>
</thead>
</table>

### Release History

Release History

---

---
Here are the data structures with brief descriptions:

<p>| ADC_MEASUREMENT_ADV_ADV     |
|----------------------------|--------------------------------------------------|
| ADC_MEASUREMENT_ADV_ADV_CHANNEL | NVIC Configuration structure for request source interrupt |
| ADC_MEASUREMENT_ADV_ADV_NVIC_CONFIG |
| ADC_MEASUREMENT_ADV_ADV_QUEUE | Configuration Data structure of queue request source |
| ADC_MEASUREMENT_ADV_ADV_SCAN | Configuration Data structure of scan request source |</p>
<table>
<thead>
<tr>
<th>Home</th>
<th>Data Structures</th>
<th>Data Structure Index</th>
<th>Data Fields</th>
</tr>
</thead>
</table>

**ADC_MEASUREMENT_ADV Struct Reference**
Detailed Description

Structure to configure ADC_MEASUREMENT_ADV APP.

Definition at line 286 of file ADC_MEASUREMENT_ADV.h.

#include <ADC_MEASUREMENT_ADV.h>
## Data Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const ADC_MEASUREMENT_ADV_CHANNEL_t**</code></td>
<td>channel_array</td>
</tr>
<tr>
<td><code>const ADC_MEASUREMENT_ADV_EVENT_CONFIG_t</code></td>
<td>event_config</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_STATUS_t*</code></td>
<td>init_state</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_REQUEST_SOURCE_t</code></td>
<td>req_src</td>
</tr>
<tr>
<td><code>const uint8_t</code></td>
<td>group_index</td>
</tr>
<tr>
<td><code>const uint8_t</code></td>
<td>total_number_of_entries</td>
</tr>
<tr>
<td><code>const uint8_t</code></td>
<td>total_number_of_channels</td>
</tr>
<tr>
<td><code>const bool</code></td>
<td>start_at_initialization</td>
</tr>
<tr>
<td><code>const bool</code></td>
<td>configure_glob_class1</td>
</tr>
<tr>
<td><code>const ADC_SCAN_ENTRY_t**</code></td>
<td>scan_entries</td>
</tr>
<tr>
<td><code>const ADC_QUEUE_ENTRY_t**</code></td>
<td>queue_entries</td>
</tr>
<tr>
<td><code>const XMC_VADC_QUEUE_ENTRY_t**</code></td>
<td>local_queue_entries</td>
</tr>
<tr>
<td><code>ADC_SCAN_t*</code></td>
<td>scan_handle</td>
</tr>
<tr>
<td><code>ADC_QUEUE_t*</code></td>
<td>queue_handle</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_SCAN_t*</code></td>
<td>local_scan_handle</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_QUEUE_t*</code></td>
<td>local_queue_handle</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>sync_slave_g0</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>sync_slave_g1</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>sync_slave_g2</td>
</tr>
<tr>
<td><code>uint8_t</code></td>
<td>sync_slave_g3</td>
</tr>
</tbody>
</table>
Field Documentation

**const ADC_MEASUREMENT_ADV_CHANNEL_t**

This holds an array of channels configured by the current instance of the `ADC_MEASUREMENT_ADV` APP

Definition at line 288 of file `ADC_MEASUREMENT_ADV.h`.

Referenced by `ADC_MEASUREMENT_ADV_Init()`.

**const bool ADC_MEASUREMENT_ADV::configure_grobiclass1**

Copy the master channels conversion parameters to the global iclass 1. Hence the slave channels are using the same features as that of the master.

Definition at line 356 of file `ADC_MEASUREMENT_ADV.h`.

**ADC_MEASUREMENT_ADV_EVENT_CONFIG_t**

This hold the pointer to the function that does mux configuration. Which entails channel node and result node configuration

Definition at line 306 of file `ADC_MEASUREMENT_ADV.h`.

Referenced by `ADC_MEASUREMENT_ADV_Init()`.

**const uint8_t**

The group index number for the APP

Definition at line 345 of file `ADC_MEASUREMENT_ADV.h`.
Referenced by `ADC_MEASUREMENT_ADV_SetIclass()`, `ADC_MEASUREMENT_ADV_SetSubtractionValue()`, `ADC_MEASUREMENT_ADV_SoftwareTrigger()`, and `ADC_MEASUREMENT_ADV_StartADC()`.

<table>
<thead>
<tr>
<th>ADC_MEASUREMENT_ADV_STATUS_t*</th>
<th>ADC_MEASUREMENT_ADV_STATUS_t* ADC_MEASUREMENT_ADV::init_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>This enumeration gives information about the status of the APP</td>
<td>Definition at line 325 of file <code>ADC_MEASUREMENT_ADV.h</code>.</td>
</tr>
<tr>
<td>Referenced by <code>ADC_MEASUREMENT_ADV_Init()</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>const XMC_VADC_QUEUE_ENTRY_t**</th>
<th>const XMC_VADC_QUEUE_ENTRY_t** const ADC_MEASUREMENT_ADV::local_queue_entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holds the pointer to the queue entries.</td>
<td>Definition at line 301 of file <code>ADC_MEASUREMENT_ADV.h</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADC_MEASUREMENT_ADV_QUEUE_t*</th>
<th>const ADC_MEASUREMENT_ADV_QUEUE_t* const ADC_MEASUREMENT_ADV::local_queue_handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to the queue handle</td>
<td>Definition at line 321 of file <code>ADC_MEASUREMENT_ADV.h</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADC_MEASUREMENT_ADV_SCAN_t*</th>
<th>const ADC_MEASUREMENT_ADV_SCAN_t* const ADC_MEASUREMENT_ADV::local_scan_handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to the scan handle</td>
<td>Definition at line 318 of file <code>ADC_MEASUREMENT_ADV.h</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>const ADC_QUEUE_ENTRY_t**</th>
<th>const ADC_QUEUE_ENTRY_t** const ADC_MEASUREMENT_ADV::queue_entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>ADC_QUEUE_t* const ADC_MEASUREMENT_ADV::queue_handle</code></td>
<td>Pointer to the ADC_QUEUE APP handle</td>
</tr>
<tr>
<td><code>ADC_MEASUREMENT_ADV_REQUEST_SOURCE_t ADC_MEASUREMENT_ADV::req_src</code></td>
<td>The request source used by this instance of the ADC_MEASUREMENT_ADV APP</td>
</tr>
<tr>
<td></td>
<td>Referenced by <code>ADC_MEASUREMENT_ADV_Init()</code>, <code>ADC_MEASUREMENT_ADV_SoftwareTrigger()</code>, and <code>ADC_MEASUREMENT_ADV_StartADC()</code></td>
</tr>
<tr>
<td><code>const ADC_SCAN_ENTRY_t** const ADC_MEASUREMENT_ADV::scan_entries</code></td>
<td>Holds the pointer to the scan entries.</td>
</tr>
<tr>
<td><code>ADC_SCAN_t* const ADC_MEASUREMENT_ADV::scan_handle</code></td>
<td>Pointer to the ADC_SCAN APP handle</td>
</tr>
</tbody>
</table>
**const bool ADC_MEASUREMENT_ADV::start_at_initialization**

This determines if the insertion of the queue or scan entries should happen after initialization of the APP.

Definition at line 353 of file ADC_MEASUREMENT_ADV.h.

Referenced by ADC_MEASUREMENT_ADV_Init().

**uint8_t ADC_MEASUREMENT_ADV::sync_slave_g0**

If set the group-0 will be configured as the slave group.

Definition at line 335 of file ADC_MEASUREMENT_ADV.h.

**uint8_t ADC_MEASUREMENT_ADV::sync_slave_g1**

If set the group-1 will be configured as the slave group.

Definition at line 336 of file ADC_MEASUREMENT_ADV.h.

**uint8_t ADC_MEASUREMENT_ADV::sync_slave_g2**

If set the group-2 will be configured as the slave group.

Definition at line 337 of file ADC_MEASUREMENT_ADV.h.

**uint8_t ADC_MEASUREMENT_ADV::sync_slave_g3**

If set the group-3 will be configured as the slave group.

Definition at line 338 of file ADC_MEASUREMENT_ADV.h.
const uint8_t ADC_MEASUREMENT_ADV::total_number_of_channels

Indicates the total number of channels configured in the current APP instance

Definition at line 350 of file ADC_MEASUREMENT_ADV.h.

Referenced by ADC_MEASUREMENT_ADV_Init().

const uint8_t ADC_MEASUREMENT_ADV::total_number_of_entries

Indicates the total number of entries configured in the current APP instance

Definition at line 347 of file ADC_MEASUREMENT_ADV.h.

Referenced by ADC_MEASUREMENT_ADV_Init().

The documentation for this struct was generated from the following file:

- ADC_MEASUREMENT_ADV.h
ADC_MEASUREMENT_ADV

<table>
<thead>
<tr>
<th>Home</th>
<th>Data Structures</th>
<th>Data Structure Index</th>
<th>Data Fields</th>
</tr>
</thead>
</table>

ADC_MEASUREMENT_ADV_CHANNEL Struct Reference
Detailed Description

Structure to configure the channels in the
ADC_MEASUREMENT_ADV APP.

Definition at line 257 of file ADC_MEASUREMENT_ADV.h.

#include <ADC_MEASUREMENT_ADV.h>
## Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMC_VADC_CHANNEL_CONFIG_t *</td>
<td>ch_handle</td>
</tr>
<tr>
<td>XMC_VADC_RESULT_CONFIG_t *</td>
<td>res_handle</td>
</tr>
<tr>
<td>ANALOG_IO_t *</td>
<td>analog_io_config</td>
</tr>
<tr>
<td>uint8_t</td>
<td>max_fifo_required</td>
</tr>
<tr>
<td>uint8_t</td>
<td>result_fifo_tail_number</td>
</tr>
<tr>
<td>uint8_t</td>
<td>group_index</td>
</tr>
<tr>
<td>uint8_t</td>
<td>ch_num</td>
</tr>
<tr>
<td>ADC_MEASUREMENT_ADV_GAIN_t</td>
<td>shs_gain_factor</td>
</tr>
</tbody>
</table>
Field Documentation

**ANALOG_IO_t**  
\texttt{ADC_MEASUREMENT_ADV_CHANNEL::analog_io}

This holds the address of the ANALOG_IO configuration structure

Definition at line \texttt{265} of file \texttt{ADC_MEASUREMENT_ADV.h}.

Referenced by \texttt{ADC_MEASUREMENT_ADV_Init()}. 

**XMC_VADC_CHANNEL_CONFIG_t**  
\texttt{ADC_MEASUREMENT_ADV_CHANNEL::ch_handle}

This holds the VADC Channel LLD structures

Definition at line \texttt{259} of file \texttt{ADC_MEASUREMENT_ADV.h}.

Referenced by
- \texttt{ADC_MEASUREMENT_ADV_ConfigureChannelClass()},
- \texttt{ADC_MEASUREMENT_ADV_GetDetailedResult()},
- \texttt{ADC_MEASUREMENT_ADV_GetFastCompareResult()},
- \texttt{ADC_MEASUREMENT_ADV_GetResult()},
- \texttt{ADC_MEASUREMENT_ADV_Init()},
- \texttt{ADC_MEASUREMENT_ADV_SetBoundaryLower()},
- \texttt{ADC_MEASUREMENT_ADV_SetBoundaryUpper()},
- \texttt{ADC_MEASUREMENT_ADV_SetChannelGain()}, and
- \texttt{ADC_MEASUREMENT_ADV_SetFastCompareValue()}. 

**uint8_t**  
\texttt{ADC_MEASUREMENT_ADV_CHANNEL::ch_num}

This holds the Channel Number

Definition at line \texttt{275} of file \texttt{ADC_MEASUREMENT_ADV.h}.

Referenced by \texttt{ADC_MEASUREMENT_ADV_GetAliasValue()},
uint8_t ADC_MEASUREMENT_ADV_CHANNEL::group_index

This holds the group index

Definition at line 273 of file ADC_MEASUREMENT_ADV.h.

Referenced by
ADC_MEASUREMENT_ADV_ConfigureChannelClass(),
ADC_MEASUREMENT_ADV_GetAliasValue(),
ADC_MEASUREMENT_ADV_GetDetailedResult(),
ADC_MEASUREMENT_ADV_GetFastCompareResult(),
ADC_MEASUREMENT_ADV_GetFifoDetailedResult(),
ADC_MEASUREMENT_ADV_GetFifoResult(),
ADC_MEASUREMENT_ADV_GetResult(),
ADC_MEASUREMENT_ADV_Init(),
ADC_MEASUREMENT_ADV_SelectBoundary(),
ADC_MEASUREMENT_ADV_SetAlternateReference(),
ADC_MEASUREMENT_ADV_SetChannelGain(), and
ADC_MEASUREMENT_ADV_SetFastCompareValue().

uint8_t ADC_MEASUREMENT_ADV_CHANNEL::max_fifo_required

The required number of FIFO elements

Definition at line 269 of file ADC_MEASUREMENT_ADV.h.

XMC_VADC_RESULT_CONFIG_t* ADC_MEASUREMENT_ADV_CHANNEL::res_handle

This hold the VADC LLD Result configuration structures
Definition at line 261 of file ADC_MEASUREMENT_ADV.h.

`uint8_t ADC_MEASUREMENT_ADV_CHANNEL::result_fifo_tail_number` The tail result register number if FIFO is selected.

Definition at line 271 of file ADC_MEASUREMENT_ADV.h.

Referenced by 
ADC_MEASUREMENT_ADV_GetFifoDetailedResult(), and
ADC_MEASUREMENT_ADV_GetFifoResult().

ADC_MEASUREMENT_ADV_GAIN_t ADC_MEASUREMENT_ADV_CHANNEL::shs_gain_factor

The required gain factor for the channel.

Definition at line 278 of file ADC_MEASUREMENT_ADV.h.

Referenced by ADC_MEASUREMENT_ADV_Init().

The documentation for this struct was generated from the following file:

- ADC_MEASUREMENT_ADV.h
<table>
<thead>
<tr>
<th>ADC_MEASUREMENT_ADV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
</tr>
<tr>
<td>Data Structures</td>
</tr>
<tr>
<td>Data Fields</td>
</tr>
</tbody>
</table>

Struct Reference
Detailed Description

NVIC Configuration structure for request source interrupt.

Definition at line 195 of file ADC_MEASUREMENTADV.h.

#include <ADC_MEASUREMENTADV.h>
## Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32_t</td>
<td>node_id</td>
</tr>
<tr>
<td>uint32_t</td>
<td>priority</td>
</tr>
<tr>
<td>uint32_t</td>
<td>sub_priority</td>
</tr>
<tr>
<td>bool</td>
<td>interrupt_enable</td>
</tr>
<tr>
<td>uint8_t</td>
<td>irqctrl</td>
</tr>
</tbody>
</table>
Field Documentation

bool ADC_MEASUREMENT_ADV_NVIC_CONFIG::interrupt_enable

This flag indicates if a Interrupt has been requested.

Definition at line 203 of file ADC_MEASUREMENT_ADV.h.

uint8_t ADC_MEASUREMENT_ADV_NVIC_CONFIG::irqctrl

This indicates the service request source selected for the consumed NVIC node.

Definition at line 205 of file ADC_MEASUREMENT_ADV.h.

uint32_t ADC_MEASUREMENT_ADV_NVIC_CONFIG::node_id

This indicates the NVIC Node number.

Definition at line 197 of file ADC_MEASUREMENT_ADV.h.

uint32_t ADC_MEASUREMENT_ADV_NVIC_CONFIG::priority

This indicates the NVIC priority.

Definition at line 199 of file ADC_MEASUREMENT_ADV.h.

uint32_t ADC_MEASUREMENT_ADV_NVIC_CONFIG::sub_priority

This indicates the NVIC sub priority in XMC4x Devices.

Definition at line 201 of file ADC_MEASUREMENT_ADV.h.
The documentation for this struct was generated from the following file:

- ADC_MEASUREMENT_ADV.h
<table>
<thead>
<tr>
<th>Home</th>
<th>Data Structures</th>
<th>Data Structure Index</th>
<th>Data Fields</th>
<th>Data Fields</th>
</tr>
</thead>
</table>

**ADC_MEASUREMENT_ADV**

ADC_MEASUREMENT_ADV_QUEUE Struct Reference
Detailed Description

Configuration Data structure of queue request source.

Definition at line 236 of file ADC_MEASUREMENT_ADV.h.

#include <ADC_MEASUREMENT_ADV.h>
### Data Fields

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const ADC_MEASUREMENT_ADV_NVIC_CONFIG_t</code></td>
<td><code>rs_intr_handle</code></td>
</tr>
<tr>
<td><code>const XMC_VADC_GROUP_CLASS_t</code></td>
<td><code>iclass_config_handle</code></td>
</tr>
<tr>
<td><code>const XMC_VADC_QUEUE_CONFIG_t</code> *const</td>
<td><code>queue_config_handle</code></td>
</tr>
<tr>
<td><code>const XMC_VADC_GATEMODE_t</code></td>
<td><code>gating_mode</code></td>
</tr>
<tr>
<td><code>const XMC_VADC_SR_t</code></td>
<td><code>srv_req_node</code></td>
</tr>
<tr>
<td><code>const uint8_t</code></td>
<td><code>iclass_num</code></td>
</tr>
</tbody>
</table>
Field Documentation

const XMC_VADC_GATEMODE_t ADC_MEASUREMENT_ADV_QUEUE::gating_mode

Gating mode configuration needed for Scan request source
Definition at line 244 of file ADC_MEASUREMENT_ADV.h.

const XMC_VADC_GROUP_CLASS_t ADC_MEASUREMENT_ADV_QUEUE::iclass_config_handle

Holds the ICLASS Configurations
Definition at line 240 of file ADC_MEASUREMENT_ADV.h.

const uint8_t ADC_MEASUREMENT_ADV_QUEUE::iclass_num

Holds the ICLASS ID either ICLASS-0 or ICLASS-1
Definition at line 248 of file ADC_MEASUREMENT_ADV.h.

const XMC_VADC_QUEUE_CONFIG_t* const ADC_MEASUREMENT_ADV_QUEUE::queue_config_handle

Holds the LLD QUEUE Structure
Definition at line 242 of file ADC_MEASUREMENT_ADV.h.

const ADC_MEASUREMENT_ADV_NVIC_CONFIG_t ADC_MEASUREMENT_ADV_NVIC_CONFIG

Holds the ISR Handle
Definition at line 238 of file ADC_MEASUREMENT_ADV.h.
const XMC_VADC_SR_t ADC_MEASUREMENT_ADV_QUEUE::srv_

Source event interrupt node pointer

Definition at line 246 of file ADC_MEASUREMENT_ADV.h.

The documentation for this struct was generated from the following file:

- ADC_MEASUREMENT_ADV.h
ADC_MEASUREMENT_ADV

<table>
<thead>
<tr>
<th>Home</th>
<th>Data Structures</th>
<th>Data Structure Index</th>
<th>Data Fields</th>
</tr>
</thead>
</table>

ADC_MEASUREMENT_ADV_SCAN Struct Reference
Detailed Description

Configuration Data structure of scan request source.

Definition at line 213 of file ADC_MEASUREMENT_ADV.h.

#include <ADC_MEASUREMENT_ADV.h>
## Data Fields

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const ADC_measurement_adv_nvic_config_t</td>
<td>rs_intr_handle</td>
</tr>
<tr>
<td>const XMC_VADC_GROUP_CLASS_t</td>
<td>iclass_config_handle</td>
</tr>
<tr>
<td>const XMC_VADC_SCAN_CONFIG_t *const</td>
<td>scan_config_handle</td>
</tr>
<tr>
<td>const XMC_VADC_GATEMODE_t</td>
<td>gating_mode</td>
</tr>
<tr>
<td>const XMC_VADC_SR_t</td>
<td>srv_req_node</td>
</tr>
<tr>
<td>const uint32_t</td>
<td>insert_mask</td>
</tr>
<tr>
<td>const uint8_t</td>
<td>iclass_num</td>
</tr>
</tbody>
</table>
Field Documentation

const XMC_VADC_GATEMODE_t ADC_MEASUREMENT_ADV_SCAN::gating_mode

Gating mode configuration needed for Scan request source
Definition at line 221 of file ADC_MEASUREMENT_ADV.h.

const XMC_VADC_GROUP_CLASS_t ADC_MEASUREMENT_ADV_SCAN::iclass_config_handle

Holds the ICLASS Configurations
Definition at line 217 of file ADC_MEASUREMENT_ADV.h.

const uint8_t ADC_MEASUREMENT_ADV_SCAN::iclass_num

Holds the ICLASS ID either ICLASS-0 or ICLASS-1
Definition at line 227 of file ADC_MEASUREMENT_ADV.h.

const uint32_t ADC_MEASUREMENT_ADV_SCAN::insert_mask

Insert Mask for the scan request source
Definition at line 225 of file ADC_MEASUREMENT_ADV.h.

const ADC_MEASUREMENT_ADV_NVIC_CONFIG_t ADC_MEASUREMENT_ADV_SCAN::rs_intr_handle

Holds the ISR Handle
Definition at line 215 of file ADC_MEASUREMENT_ADV.h.
const XMC_VADC.Scan_Config_t* const ADC_MEASUREMENT_ADV_SCAN::scan_config_handle

Holds the LLD SCAN Structure
Definition at line 219 of file ADC_MEASUREMENT_ADV.h.

const XMC_VADC.SR_t ADC_MEASUREMENT_ADV_SCAN::srv_req_node

Source event interrupt node pointer
Definition at line 223 of file ADC_MEASUREMENT_ADV.h.

The documentation for this struct was generated from the following file:

- ADC_MEASUREMENT_ADV.h
# ADC_MEASUREMENT_ADV

## Data Structure Index

| ADC_MEASUREMENT_ADV_ADV_CHANNEL |
| ADC_MEASUREMENT_ADV_NVIC_CONFIG |
| ADC_MEASUREMENT_ADV |
ADC_MEASUREMENT_ADV

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

- **a** -
  - analog_io_config: ADC_MEASUREMENT_ADV_CHANNEL

- **c** -
  - ch_handle: ADC_MEASUREMENT_ADV_CHANNEL
  - ch_num: ADC_MEASUREMENT_ADV_CHANNEL
  - channel_array: ADC_MEASUREMENT_ADV
  - configure_globiclass1: ADC_MEASUREMENT_ADV

- **e** -
  - event_config: ADC_MEASUREMENT_ADV

- **g** -
  - gating_mode: ADC_MEASUREMENT_ADV_SCAN, ADC_MEASUREMENT_ADV_QUEUE
  - group_index: ADC_MEASUREMENT_ADV_CHANNEL, ADC_MEASUREMENT_ADV

- **i** -
  - iclass_config_handle: ADC_MEASUREMENT_ADV_SCAN, ADC_MEASUREMENT_ADV_QUEUE
  - iclass_num: ADC_MEASUREMENT_ADV_QUEUE,
• init_state: ADC_MEASUREMENT_ADV
• insert_mask: ADC_MEASUREMENT_ADV_SCAN
• interrupt_enable: ADC_MEASUREMENT_ADV_NVIC_CONFIG
• irqctrl: ADC_MEASUREMENT_ADV_NVIC_CONFIG

- l -

• local_queue_entries: ADC_MEASUREMENT_ADV
• local_queue_handle: ADC_MEASUREMENT_ADV
• local_scan_handle: ADC_MEASUREMENT_ADV

- m -

• max_fifo_required: ADC_MEASUREMENT_ADV_CHANNEL

- n -

• node_id: ADC_MEASUREMENT_ADV_NVIC_CONFIG

- p -

• priority: ADC_MEASUREMENT_ADV_NVIC_CONFIG

- q -

• queue_config_handle: ADC_MEASUREMENT_ADV_QUEUE
• queue_entries: ADC_MEASUREMENT_ADV
• queue_handle: ADC_MEASUREMENT_ADV

- r -

• req_src: ADC_MEASUREMENT_ADV
• res_handle: ADC_MEASUREMENT_ADV_CHANNEL
• result_fifo_tail_number: ADC_MEASUREMENT_ADV_CHANNEL
• rs_intr_handle: ADC_MEASUREMENT_ADV_SCAN, ADC_MEASUREMENT_ADV_QUEUE

- s -
- t -

- **total_number_of_channels**: `ADC_MEASUREMENT_ADV`
- **total_number_of_entries**: `ADC_MEASUREMENT_ADV`
ADC_MEASUREMENT_ADV

- a -
  - analog_io_config : ADC_MEASUREMENT_ADV_CHANNEL

- c -
  - ch_handle : ADC_MEASUREMENT_ADV_CHANNEL
  - ch_num : ADC_MEASUREMENT_ADV_CHANNEL
  - channel_array : ADC_MEASUREMENT_ADV
  - configure_globiclass1 : ADC_MEASUREMENT_ADV

- e -
  - event_config : ADC_MEASUREMENT_ADV

- g -
  - gating_mode : ADC_MEASUREMENT_ADV_SCAN, ADC_MEASUREMENT_ADV_QUEUE
  - group_index : ADC_MEASUREMENT_ADV_CHANNEL, ADC_MEASUREMENT_ADV

- i -
  - iclass_config_handle : ADC_MEASUREMENT_ADV_SCAN, ADC_MEASUREMENT_ADV_QUEUE
  - iclass_num : ADC_MEASUREMENT_ADV_QUEUE, ADC_MEASUREMENT_ADV_SCAN
- l -
  - local_queue_entries : ADC_MEASUREMENT_ADV
  - local_queue_handle : ADC_MEASUREMENT_ADV
  - local_scan_handle : ADC_MEASUREMENT_ADV

- m -
  - max_fifo_required : ADC_MEASUREMENT_ADV_CHANNEL

- n -
  - node_id : ADC_MEASUREMENT_ADV_NVIC_CONFIG

- p -
  - priority : ADC_MEASUREMENT_ADV_NVIC_CONFIG

- q -
  - queue_config_handle : ADC_MEASUREMENT_ADV_QUEUE
  - queue_entries : ADC_MEASUREMENT_ADV
  - queue_handle : ADC_MEASUREMENT_ADV

- r -
  - req_src : ADC_MEASUREMENT_ADV
  - res_handle : ADC_MEASUREMENT_ADV_CHANNEL
  - resultfifo_tail_number : ADC_MEASUREMENT_ADV_CHANNEL
  - rs_intr_handle : ADC_MEASUREMENT_ADV_SCAN, ADC_MEASUREMENT_ADV_QUEUE

- s -
- t -

- total_number_of_channels : ADC_MEASUREMENT_ADV
- total_number_of_entries : ADC_MEASUREMENT_ADV
Here is a list of all documented files with brief descriptions:

- ADC_MEASUREMENT_ADV.c [code]
- ADC_MEASUREMENT_ADV.h [code]
/*CODE_BLOCK_BEGIN*/
#ifndef ADC_MEASUREMENT_ADV_H
#define ADC_MEASUREMENT_ADV_H

/*******************************************
**************************************************
**************************
* HEADER FILES
**************************************************
**************************
#include "GLOBAL_ADC/global_adc.h"
#include "adc_measurement_adv_conf.h"

/******************************************
**************************************************
**************************
* MACROS
**************************************************
**************************
#if (!((XMC_LIB_MAJOR_VERSION == 2U) &&
(XMC_LIB_MINOR_VERSION >= 1U) &&
(XMC_LIB_PATCH_VERSION >= 8U)))
#error "ADC_MEASUREMENT_ADV requires XMC Per
typedef enum ADC_MEASUREMENT_ADV_STATUS {
    ADC_MEASUREMENT_ADV_STATUS_SUCCESS = 0,
    ADC_MEASUREMENT_ADV_STATUS_FAILURE,
    ADC_MEASUREMENT_ADV_STATUS_UNINITIALIZED
} ADC_MEASUREMENT_ADV_STATUS_t;

typedef enum ADC_MEASUREMENT_ADV_REQUEST_SOURCE {
    ADC_MEASUREMENT_ADV_REQUEST_SOURCE_SCAN = 0,
    ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN,
    ADC_MEASUREMENT_ADV_REQUEST_SOURCE_QUEUE,
} ADC_MEASUREMENT_ADV_REQUEST_SOURCE_t;

typedef enum ADC_MEASUREMENT_ADV_GAIN {
    ADC_MEASUREMENT_ADV_GAIN_1 = 0U,
    ADC_MEASUREMENT_ADV_GAIN_3 = 1U,
    ADC_MEASUREMENT_ADV_GAIN_6 = 2U,
    ADC_MEASUREMENT_ADV_GAIN_12 = 3U
} ADC_MEASUREMENT_ADV_GAIN_t;
typedef enum ADC_MEASUREMENT_ADV_SUBTRACTION {
    ADC_MEASUREMENT_ADV_SUBTRACTION_12BIT_LEFT_ALIGN = 0U,
    ADC_MEASUREMENT_ADV_SUBTRACTION_12BIT_RIGHT_ALIGN = 0U,
    ADC_MEASUREMENT_ADV_SUBTRACTION_10BIT_LEFT_ALIGN = 2U,
    ADC_MEASUREMENT_ADV_SUBTRACTION_10BIT_RIGHT_ALIGN = 0U,
    ADC_MEASUREMENT_ADV_SUBTRACTION_8BIT_LEFT_ALIGN = 4U,
    ADC_MEASUREMENT_ADV_SUBTRACTION_8BIT_RIGHT_ALIGN = 0U
} ADC_MEASUREMENT_ADV_SUBTRACTION_t;

typedef enum ADC_MEASUREMENT_ADV_FAST_COMPARE {
    ADC_MEASUREMENT_ADV_FAST_COMPARE_LOW = 0U,
    ADC_MEASUREMENT_ADV_FAST_COMPARE_HIGH = 1U,
    ADC_MEASUREMENT_ADV_FAST_COMPARE_INVALID = 2U
} ADC_MEASUREMENT_ADV_FAST_COMPARE_t;

typedef enum ADC_MEASUREMENT_ADV_SYNC_SEQ {
    ADC_MEASUREMENT_ADV_SYNC_SEQ_POWER_DOWN = 0U,
    ADC_MEASUREMENT_ADV_SYNC_SEQ_STSEL_CONFIG,
    ADC_MEASUREMENT_ADV_SYNC_SEQ_EVAL_CONFIG
} ADC_MEASUREMENT_ADV_SYNC_SEQ_t;
/*Anonymous structure/union guard start*/
#endif

typedef void (*ADC_MEASUREMENT_ADV_EVENT_CONFIG_t)(void);

typedef struct ADC_MEASUREMENT_ADV_NVIC_CONFIG
{
    uint32_t node_id;
    uint32_t priority;
    #if(UC_FAMILY == XMC4)
        uint32_t sub_priority;
    #endif
    bool interrupt_enable;
    #ifdef ADC_MEASUREMENT_ADV_NON_DEFAULT_IRQ_SOURCE_SELECTED
        uint8_t irqctrl;
    #endif
} ADC_MEASUREMENT_ADV_NVIC_CONFIG_t;

#ifdef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
    typedef struct ADC_MEASUREMENT_ADV_SCAN
    {
        uint32_t node_id;
        uint32_t priority;
        #if(UC_FAMILY == XMC4)
            uint32_t sub_priority;
        #endif
        bool interrupt_enable;
        #ifdef ADC_MEASUREMENT_ADV_NON_DEFAULT_IRQ_SOURCE_SELECTED
            uint8_t irqctrl;
        #endif
    } ADC_MEASUREMENT_ADV_NVIC_CONFIG_t;
#endif
const ADC_MEASUREMENT_ADV_NVIC_CONFIG_t rs_intr_handle;
const XMC_VADC_GROUP_CLASS_t iclass_config_handle;
const XMC_VADC_SCAN_CONFIG_t *const scan_config_handle;
const XMC_VADC_GATEMODE_t gating_mode;
const XMC_VADC_SR_t srv_req_node;
const uint32_t insert_mask;
const uint8_t iclass_num;

} ADC_MEASUREMENT_ADV_SCAN_t;
#endif
#endif
#ifdef ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED
typedef struct ADC_MEASUREMENT_ADV_QUEUE {
const ADC_MEASUREMENT_ADV_NVIC_CONFIG_t rs_intr_handle;
const XMC_VADC_GROUP_CLASS_t iclass_config_handle;
const XMC_VADC_QUEUE_CONFIG_t *const queue_config_handle;
const XMC_VADC_GATEMODE_t gating_mode;
const XMC_VADC_SR_t srv_req_node;
const uint8_t iclass_num;
} ADC_MEASUREMENT_ADV_QUEUE_t;
#endif
typedef struct ADC_MEASUREMENT_ADV_CHANNEL {
  XMC_VADC_CHANNEL_CONFIG_t *ch_handle;
  XMC_VADC_RESULT_CONFIG_t *res_handle[ADC_MEASUREMENT_ADV_RESULT_REG];
  #ifdef ADC_MEASUREMENT_ADV_ANALOG_IO_USED
    ANALOG_IO_t *analog_io_config;
  #endif
  #ifdef ADC_MEASUREMENT_ADV_FIFO_USED
    uint8_t max_fifo_required;
    uint8_t result_fifo_tail_number;
  #endif
  #if(UC_FAMILY == XMC1)
    ADC_MEASUREMENT_ADV_GAIN_t shs_gain_factor;
  #endif
} ADC_MEASUREMENT_ADV_CHANNEL_t;

typedef struct ADC_MEASUREMENT_ADV {
  const ADC_MEASUREMENT_ADV_CHANNEL_t **const channel_array;
  #if defined(ADC_MEASUREMENT_ADV_ADC_SCAN_USED) || defined(ADC_MEASUREMENT_ADV_ADC_QUEUE_USED) || defined(ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED)
    union {
      #ifdef ADC_MEASUREMENT_ADV_ADC_SCAN_USED
        const ADC_SCAN_ENTRY_t **const scan_entries;
      #endif
      #endif
  #endif
} ADC_MEASUREMENT_ADV_ADV;
ifdef ADC_MEASUREMENT_ADV_ADC_QUEUE_USED
const ADC_QUEUE_ENTRY_t **const queue_entries;
#endif

ifdef ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED
const XMC_VADC_QUEUE_ENTRY_t **const local_queue_entries;
#endif

};
#endif

ADC_MEASUREMENT_ADV_EVENT_CONFIG_t event_config;
union {
ifdef ADC_MEASUREMENT_ADV_ADC_SCAN_USED
ADC_SCAN_t *const scan_handle;
#endif
ifdef ADC_MEASUREMENT_ADV_ADC_QUEUE_USED
ADC_QUEUE_t *const queue_handle;
#endif
ifdef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
ADC_MEASUREMENT_ADV_SCAN_t *const local_scan_handle;
#endif
ifdef ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED
ADC_MEASUREMENT_ADV_QUEUE_t *const local_queue_handle;
#endif
};

ADC_MEASUREMENT_ADV_STATUS_t *init_state;

ADC_MEASUREMENT_ADV_REQUEST_SOURCE_t req_src;
#ifdef ADC_MEASUREMENT_ADV_SYNC_USED
union
struct {
    uint8_t sync_slave_g0 :1;
    uint8_t sync_slave_g1 :1;
    uint8_t sync_slave_g2 :1;
    uint8_t sync_slave_g3 :1;
    uint8_t :4;
};

uint8_t sync_slaves;

const uint8_t group_index;
const uint8_t total_number_of_entries;
const uint8_t total_number_of_channels;
const bool start_at_initialization;

#ifdef ADC_MEASUREMENT_ADV_SYNC_USED
    const bool configure_globiclass1;
#endif

} ADC_MEASUREMENT_ADV_t;

/*Anonymous structure/union guard end*/

#if defined(__CC_ARM)
    #pragma pop
#elif defined(__TASKING__)
    #pragma warning restore
#endif

#ifdef __cplusplus
    extern "C" {
#endif

/* API Prototypes */
DAVE_APP_VERSION_t ADC_MEASUREMENT_ADV_GetAppVersion(void);

void ADC_MEASUREMENT_ADV_StartADC(const ADC_MEASUREMENT_ADV_t *const handle_ptr);

void ADC_MEASUREMENT_ADV_SoftwareTrigger(const ADC_MEASUREMENT_ADV_t *const handle_ptr);

ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_Init(const ADC_MEASUREMENT_ADV_t *const handle_ptr);

__STATIC_INLINE uint16_t ADC_MEASUREMENT_ADV_GetResult(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr)
{
    uint16_t result;
    extern XMC_VADC_GROUP_t *const group_ptrs[XMC_VADC_MAXIMUM_NUM_GROUPS];
    XMC_ASSERT("ADC_MEASUREMENT_ADV_GetResult: Invalid handle_ptr", (handle_ptr != NULL))
    result = XMC_VADC_GROUP_GetResult(group_ptrs[handle_ptr->group_index],
        (uint32_t)handle_ptr->ch_handle->result_reg_number);
    return(result);
}

__STATIC_INLINE XMC_VADC_DETAILED_RESULT_t ADC_MEASUREMENT_ADV_GetDetailedResult(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr)
NNEL_t *const handle_ptr)  
00733  
00734   XMC_VADC_DETAILED_RESULT_t result_register;
00735  
00736   extern XMC_VADC_GROUP_t *const group_ptrs[XMC_VADC_MAXIMUM_NUM_GROUPS];
00737  
00738   XMC_ASSERT("ADC_MEASUREMENT_ADV_GetDetailedResult:Invalid handle_ptr", (handle_ptr != NULL))
00739  
00740   result_register.res = XMC_VADC_GROUP_GetDetailedResult(group_ptrs[handle_ptr->group_index],
00741  
00742   (uint32_t) handle_ptr->ch_handle->result_reg_number);
00743  
00744   return(result_register);
00745 
00746 #ifdef ADC_MEASUREMENT_ADV_FIFO_USED
00747 
00821  
00822   __STATIC_INLINE uint16_t ADC_MEASUREMENT_ADV_GetFifoResult(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr) 
00823  
00824   
00825   uint16_t result;
00826   
00827   extern XMC_VADC_GROUP_t *const group_ptrs[XMC_VADC_MAXIMUM_NUM_GROUPS];
00828   
00829   XMC_ASSERT("ADC_MEASUREMENT_ADV_GetFifoResult:Invalid handle_ptr", (handle_ptr != NULL))
00830 
00831   result = XMC_VADC_GROUP_GetResult(group_ptrs[handle_ptr->group_index], handle_ptr->result_fifo_tail_number);
00832  
00833   return(result);
00834 
00835 }  
00908  
00909 __STATIC_INLINE XMC_VADC_DETAILED_RESULT_t ADC_MEASUREMENT_ADV_GetFifoDetailedResult(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr) 
00909
NNEL_t *const handle_ptr) {
    XMC_VADC_DETAILED_RESULT_t result_register;
    extern XMC_VADC_GROUP_t *const group_ptrs[XMC_VADC_MAXIMUM_NUM_GROUPS];
    XMC_ASSERT("ADC_MEASUREMENT_ADV_GetFifoDet
tailedResult:Invalid handle_ptr", (handle_ptr != NULL))
    result_register.res = XMC_VADCGROUP_GetDe
tailedResult(group_ptrs[handle_ptr->group_index],
    handle_ptr->result_fifo_tail_number);
    return(result_register);
}

__STATIC_INLINE ADC_MEASUREMENT_ADV_FAST_COMPARE_t ADC_MEASUREMENT_ADV_GetFastCompareResult(const
NNEL_t *const handle_ptr) {
    ADC_MEASUREMENT_ADV_FAST_COMPARE_t fast_compare_result;
    extern XMC_VADC_GROUP_t *const group_ptrs[XMC_VADC_MAXIMUM_NUM_GROUPS];
    XMC_ASSERT("ADC_MEASUREMENT_ADV_GetFastCom
pareResult:Invalid handle_ptr", (handle_ptr != NULL))
    fast_compare_result = (ADC_MEASUREMENT_ADV
_FAST_COMPARE_t)XMC_VADCGROUP_GetFastCompareResul
group_ptrs[handle_ptr->group_index], (uint32_t) handle_ptr->ch_handle->result_reg_number);
01010
01011    return(fast_compare_result);
01012  }
01013
01089 ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_SetFastCompareValue(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
01090
    uint16_t compare_value);
01091
01159 void ADC_MEASUREMENT_ADV_SetSubtractionValue(const ADC_MEASUREMENT_ADV_t *const handle_ptr,
01160
    ADC_MEASUREMENT_ADV_SUBTRACTION_t subtraction_alignment,
01161
    uint16_t subtraction_value);
01162
01236 void ADC_MEASUREMENT_ADV_ConfigureChannelClass(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
01237
    const XMC_VADC_GROUP_CLASS_t *config);
01238
01301 void ADC_MEASUREMENT_ADV_SetAlternateReference(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
01302
    const XMC_VADC_CHANNEL_REF_t reference_select);
01303
01304 #if (XMC_VADC_SHS_AVAILABLE == 1U)
01305
01365 void ADC_MEASUREMENT_ADV_SetChannelGain(const
ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
const ADC_MEASUREMENT_ADV_GAIN_t gain_factor);
#endif

void ADC_MEASUREMENT_ADV_SelectBoundary(const
ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
ADC_MEASURE)))),
XMC
_VADC_BOUNCARY_SELECT_t boundary,
XMC
_VADC_CHANNEL_BOUNDARY_t boundary_selection);

void ADC_MEASUREMENT_ADV_SetBoundaryUpper(const
ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
uint32_t boundary_value);

void ADC_MEASUREMENT_ADV_SetBoundaryLower(const
ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
uint32_t boundary_value);

XMC_VADC_CHANNEL_ALIAS_t ADC_MEASUREMENT_ADV
_GetAliasValue(const ADC_MEASUREMENT_ADV_CHANNEL_t
*const handle_ptr);

#endif

#include "adc_measurement_adv_extern.h"
#ifdef __cplusplus
}
#endif
#endif /* _ADC_MEASUREMENT_ADV_H_ */
# include "adc_measurement_adv.h"

/*
 * Pointer to the VADC GLOBAL*/

#define ADC_MEASUREMENT_ADV_GLOBAL_PTR ((XMC_VADC_GLOBAL_t*) (void*) VADC)

/* Max value possible with 10 bit resolution is 1023*/

#define ADC_MEASUREMENT_ADV_10_BIT_MAX_VALUE ((uint32_t)1023)

#if (XMC_VADC_SHS_AVAILABLE == 1U)

/* Pointer to the SHS unit */
```c
#define ADC_MEASUREMENT_ADV_SHS_PTR ((XMC_VADC_GLOBAL_SHS_t *) (void *) SHS0)
#endif

#define ADC_MEASUREMENT_ADV_RESERVED_REGISTERS ((uint32_t)2)

#define ADC_MEASUREMENT_ADV_HEAD_RESULT_REG_CONFIG ((uint32_t)0)

#define ADC_MEASUREMENT_ADV_TAIL_RESULT_REG_CONFIG ((uint32_t)1)

/* Configure the slave input class as global input class 1*/
#define ADC_MEASUREMENT_ADV_GLOBICLASS1 ((uint32_t)1)

/* Since the SCU is different for various devices a macro is defined here to enable check of clock-ungating*/
#if UC_FAMILY == XMC1
#define ADC_MEASUREMENT_ADV_CHECK_CLOCK_GATING ((uint32_t)1)
#endif

/*******************************************
**************************************************
**************************
* LOCAL DATA
*******************************************
**************************************************
***************************/

/* Array of Group pointers*/
XMC_VADC_GROUP_t *const group_ptrs[XMC_VADC_MAXIMUM_NUM_GROUPS] =
```
(VADC_G_TypeDef*)(void*) VADC_G0,
        (VADC_G_TypeDef*)(void*) VADC_G1
#if (XMC_VADC_MAXIMUM_NUM_GROUPS > 2U)
        ,(VADC_G_TypeDef*)(void*) VADC_G2,
        (VADC_G_TypeDef*)(void*) VADC_G3
#endif
};
#endif
static const XMC_VADC_RESULT_CONFIG_t ADC_MEASUREMENT_ADV_fifo_intermediate_stage = {
    .data_reduction_control = 0,
    .post_processing_mode = 0,
    .wait_for_read_mode = 0,
    .part_of_fifo = (bool)true,
    .event_gen_enable = 0
};
#endif
/*Anonymous structure/union guard start*/
#if defined(__CC_ARM)
#pragma push
#pragma anon_unions
#elif defined(__TASKING__)
#pragma warning 586
#endif
/* Private structure to determine the ALIAS*/
typedef struct ADC_MEASUREMENT_ADV_ALIAS { union
{ struct
typedef struct {
  uint32_t alias0 : 5; /* ALIAS for Channel 0*/
  uint32_t : 3;
  uint32_t alias1 : 5; /* ALIAS for Channel 1*/
  uint32_t : 19;
} ADC_MEASUREMENT_ADV_ALIAS_t;

/*Anonymous structure/union guard end*/
#if defined(__CC_ARM)
#pragma pop
#elif defined(__TASKING__)
#pragma warning restore
#endif

/*LOCAL ROUTINES*/
#if defined(ADC_MEASUREMENT_ADV_ADC_QUEUE_USED) || defined(ADC_MEASUREMENT_ADV_ADC_SCAN_USED)
/* Local function to insert an entry into the H/W*/
__STATIC_INLINE void ADC_MEASUREMENT_ADV_InsertEntry(const ADC_MEASUREMENT_ADV_t *const handle_ptr, uint8_t_t ch_num) {
  #if defined( ADC_MEASUREMENT_ADV_SCAN_USED) && defined(ADC_MEASUREMENT_ADV_ADC_SCAN_USED)
  #if defined(ADC_MEASUREMENT_ADV_QUEUE_USED) && defined(ADC_MEASUREMENT_ADV_ADC_QUEUE_USED)
if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_SCAN == handle_ptr->req_src)

ADC_SCAN_InsertScanEntry(handle_ptr->scan_handle, handle_ptr->scan_entries[ch_num]);

#else
ADC_QUEUE_InsertQueueEntry(handle_ptr->queue_handle, handle_ptr->queue_entries[ch_num]);

#endif

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

XMC_VADC_CHANNEL_CONV_t ADC_MEASUREMENT_ADV_lGetIclass(const ADC_MEASUREMENT_ADV_t *const handle_ptr)

{ 
XMC_VADC_CHANNEL_CONV_t req_iclass;

#ifdef ADC_MEASUREMENT_ADV_SCAN_USED
#ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN >= handle_ptr->req_src)
#endif
#endif

{
#ifndef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
#define ADC_MEASUREMENT_ADV_ADC_SCAN_USED

if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_SCAN != handle_ptr->req_src)
#endif
#endif

#else
/* Call the function to initialise Clock and ADC global functional units*/
req_iclass = (XMC_VADC_CHANNEL_CONV_t)handle_ptr->local_scan_handle->iclass_num;
#endif
#endif
#else
#ifndef ADC_MEASUREMENT_ADV_ADC_SCAN_USED
#endif
#else
else
#endif
#endif
#endif
#endif
#endif
#else
#endif
#else
#endif
#endif

if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_QUEUE != handle_ptr->req_src)

{ req_iclass = (XMC_VADC_CHANNEL_CONV_t)handle_ptr->local_queue->iclass_num;
#endif
#endif
#endif
#endif
}
else
#endif
{
req_iclass = (XMC_VADC_CHANNEL_CONV_t)handle_ptr->queue_handle->iclass_num;
}
#endif
}
#endif
return (req_iclass);
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
#ifdef ADC_MEASUREMENT_ADV_SYNC_USED

/*	Helper	function	to	configure	the	eval	bits	in	the	slave*/
void ADC_MEASUREMENT_ADV_lSyncEvalConfig(uint32_t master_group, uint32_t slave_selected,
uint32_t sync_group)
{
    int8_t group_index;
    sync_group |= (1U << master_group);
    sync_group &= ~(1U << slave_selected);
    for( group_index = XMC_VADC_MAXIMUM_NUM_GROUPS - (int32_t)1; group_index >= (int32_t)0 ; group_index--)
    {
    }
}
if ((bool)false != (bool)((sync_group > group_index) & 0x1))
{
    XMC_VADC_GROUP_SetSyncSlaveReadySignal(group_ptrs[slave_selected], slave_selected, group_index);
}

/* Helper function to execute the sync init sequence*/

void ADC_MEASUREMENT_ADV_lSyncSequencer(const ADC_MEASUREMENT_ADV_t *const handle_ptr,
uint32_t sync_group,
ADC_MEASUREMENT_ADV_SYNC_SEQ_t sequence)
{
    int8_t group_index;
    for( group_index = XMC_VADC_MAXIMUM_NUM_GROUPS - (int32_t)1; group_index >= (int32_t)0 ; group_index--)
    {
        if ((bool)false != (bool)((sync_group > group_index) & 0x1))
        {
            switch( sequence)
            {
                case ADC_MEASUREMENT_ADV_SYNC_SEQ_POWER_DOWN:
                    XMC_VADC_GROUP_SetPowerMode(group_ptrs[group_index],XMC_VADC_GROUP_POWERMODE_OFF)
break;

case ADC_MEASUREMENT_ADV_SYNC_SEQ_STSEL_CONFIG:
    XMC_VADC_GROUP_SetSyncSlave(group_ptrs[group_index], handle_ptr->group_index, group_index);
    XMC_VADC_GROUP_CheckSlaveReadiness(group_ptrs[handle_ptr->group_index], group_index);
    break;

case ADC_MEASUREMENT_ADV_SYNC_SEQ_EVAL_CONFIG:
    ADC_MEASUREMENT_ADV_lSyncEvalConfig(handle_ptr->group_index, group_index, sync_group);
    default:
    break;

} 
}
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

/* Initialization of the all the sync related functions */
__STATIC_INLINE void ADC_MEASUREMENT_ADV_lSyncInit(const ADC_MEASUREMENT_ADV_t *const handle_ptr)
{
    uint8_t sync_group;
    
    /* shift to get the 4 bit position needed to or it with the slave groups */
    sync_group = handle_ptr->sync_slaves | ( 1
<< handle_ptr->group_index);  
00325  ADC_MEASUREMENT_ADV_lSyncSequencer(handle_ptr, sync_group, ADC_MEASUREMENT_ADV_SYNC_SEQ_POWER_DOWN);
00326  
00327  sync_group = handle_ptr->sync_slaves;
00328  ADC_MEASUREMENT_ADV_lSyncSequencer(handle_ptr, sync_group, ADC_MEASUREMENT_ADV_SYNC_SEQ_STSEL_CONFIG);
00329  ADC_MEASUREMENT_ADV_lSyncSequencer(handle_ptr, sync_group, ADC_MEASUREMENT_ADV_SYNC_SEQ_EVAL_CONFIG);
00330  
00331  /* Configure the iclass settings needed for the sync slaves*/
00332  if( (bool) false != handle_ptr->configure_globiclass1)  
00333  {
00334    ADC_MEASUREMENT_ADV_SetIclass(handle_ptr);
00335  }
00336  
00337  XMC_VADC_GROUP_SetSyncMaster(group_ptrs[handle_ptr->group_index]);
00338  
00339  XMC_VADC_GROUP_SetPowerMode(group_ptrs[handle_ptr->group_index],XMC_VADC_GROUP_POWERMODE_NORMAL);
00340  }
00341  #endif
00342  
00343  /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
00344  #ifdef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
00345  __STATIC_INLINE ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_lScanInit(ADC_MEASUREMENT_ADV
```c
_SCAN_t *const handle_ptr,
 00346      uint8_t group_index)
 00347 {
 00348       ADC_MEASUREMENT_ADV_STATUS_t status;
 00349
 00350    /* Initialization of APP 'ADCGroup' */
 00351    status = (ADC_MEASUREMENT_ADV_STATUS_t) GLOBAL_ADC_Init(ADC_MEASUREMENT_ADV_GLOBAL_HANDLE);
 00352
 00353    XMC_VADC_GROUP_InputClassInit(group_ptrs[group_index], handle_ptr->iclass_config_handle,
 00354       XMC_VADC_GROUP_CONV_STD, (uint32_t)handle_ptr->iclass_num);
 00355
 00356    /* Initialization of scan request source */
 00357    XMC_VADC_GROUP_ScanInit(group_ptrs[group_index], handle_ptr->scan_config_handle);
 00358
 00359    /* Configure the gating mode for Scan */
 00360    XMC_VADC_GROUP_ScanSetGatingMode(group_ptrs[group_index], handle_ptr->gating_mode);
 00361
 00362    /* Interrupt Configuration */
 00363    if ((bool)true == handle_ptr->rs_intr_handle.interrupt_enable)
 00364      {
 00365         #if (UC_FAMILY == XMC1)
 00366            NVIC_SetPriority((IRQn_Type)handle_ptr->rs_intr_handle.node_id, handle_ptr->rs_intr_handle.priority);
 00367         #else
 00368             NVIC_SetPriority((IRQn_Type)handle_ptr->rs_intr_handle.node_id,
 00369                NVIC_EncodePriority(NVIC_GetPriorityGrouping(),
 00370                }
handle_ptr->rs_intr_handle.priority, handle_ptr->rs_intr_handle.sub_priority));
#endif
#endif
XMC_SCU_SetInterruptControl(handle_ptr->rs_intr_handle.node_id,
((handle_ptr->rs_intr_handle.node_id << 8) | handle_ptr->rs_intr_handle irqctrl));
#endif

/* Connect RS Events to NVIC nodes */
XMC_VADC_GROUP_ScanSetReqSrcEventInterruptNode(group_ptrs[group_index], handle_ptr->srv_req_node);
}
return (status);
}
#endif

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

__STATIC_INLINE ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_lQueueInit(ADC_MEASUREMENT_ADV_QUEUE_t *const handle_ptr,
uint8_t group_index)
{
ADC_MEASUREMENT_ADV_STATUS_t status;

/* Initialization of APP 'GLOBAL_ADC'*/
status = (ADC_MEASUREMENT_ADV_STATUS_t) GLOBAL_ADC_Init(ADC_MEASUREMENT_ADV_GLOBAL_HANDLE);
/* Class Configuration */
XMC_VADC_GROUP_InputClassInit(group_ptrs[group_index], handle_ptr->iclass_config_handle, XMC_VADC_GROUP_CONV_STD, handle_ptr->iclass_num);

/* Initialize the Queue hardware */
XMC_VADC_GROUP_QueueInit(group_ptrs[group_index], handle_ptr->queue_config_handle);

/* Configure the gating mode for queue */
XMC_VADC_GROUP_QueueSetGatingMode(group_ptrs[group_index], handle_ptr->gating_mode);

/* Interrupt Configuration */
if ((bool)true == handle_ptr->rs_intr_handle.interrupt_enable) {
    #if (UC_FAMILY == XMC1)
        NVIC_SetPriority((IRQn_Type)handle_ptr->rs_intr_handle.node_id, handle_ptr->rs_intr_handle.priority);
    #else
        NVIC_SetPriority((IRQn_Type)handle_ptr->rs_intr_handle.node_id, NVIC_EncodePriority(NVIC_GetPriorityGrouping(), handle_ptr->rs_intr_handle.priority, handle_ptr->rs_intr_handle.sub_priority));
    #endif
    #ifdef ADC_MEASUREMENT_ADV_NON_DEFAULT_IRQ_SOURCE_SELECTED
        XMC_SCU_SetInterruptControl(handle_ptr->rs_intr_handle.node_id, ((handle_ptr...
- `rs_intr_handle.node_id << 8 | handle_ptr->rs_intr_handle.irqctrl));
  #endif
  */ Connect RS Events to NVIC nodes */
  XMC_VADC_GROUP_QueueSetReqSrcEventInterruptNode(group_ptrs[group_index], (XMC_VADC_SR_t)handle_ptr->srv_req_node);
  } }
  return (status);
  #endif
  */~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
  /* Local function to do the request source initialization.*/
  __STATIC_INLINE ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_lRequestSrcInit(const ADC_MEASUREMENT_ADV_t *const handle_ptr) {
    ADC_MEASUREMENT_ADV_STATUS_t status;
    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
      #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
        if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN >= handle_ptr->req_src)
          #endif
      #endif
    #endif
  { #ifdef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
    if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN != handle_ptr->req_src)
00442    #endif
00443    {
00444    status = ADC_MEASUREMENT_ADV_lScanInit(handle_ptr->local_scan_handle, handle_ptr->group_index);
00445    }
00446    #endif
00447
00448    #ifdef  ADC_MEASUREMENT_ADV_ADC_SCAN_USED
00449    #ifdef  ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
00450    else
00451    #endif
00452    {
00453    /* Call the function to initialise C lock and ADC global functional units*/
00454    status = (ADC_MEASUREMENT_ADV_STATUS_t) ADC_SCAN_Init(handle_ptr->scan_handle);
00455    }
00456    #endif
00457    }
00458    #endif
00459
00460    #ifdef  ADC_MEASUREMENT_ADV_QUEUE_USED
00461    #ifdef  ADC_MEASUREMENT_ADV_SCAN_USED
00462    else
00463    #endif
00464    {
00465    #ifdef  ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED
00466    #ifdef  ADC_MEASUREMENT_ADV_ADC_QUEUE_USED
00467    if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_QUEUE != handle_ptr->req_src)
00468    #endif
00469    {
00470    status = ADC_MEASUREMENT_ADV_lQueueInit(handle_ptr->local_queue_handle, handle_ptr->group_index);
00471    }
#else
    #ifdef ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED

    else
        #endif
    
    { /* Call the function to initialise Clock and ADC global functional units*/

        status = (ADC_MEASUREMENT_ADV_STATUS_t) ADC_QUEUE_Init(handle_ptr->queue_handle);
    }
#endif

return (status);

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

#ifdef ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED

/* Local function to insert the queue entries into the hardware.*/
__STATIC_INLINE void ADC_MEASUREMENT_ADV_LocalQueueInsertEntries(const ADC_MEASUREMENT_ADV_t *const handle_ptr)
{
    uint32_t entry_index;

    XMC_VADC_GROUP_t *queue_group_ptr = group_ptrs[handle_ptr->group_index];

    const XMC_VADCQUEUE_ENTRY_t **const entries_array = handle_ptr->local_queue_entries;

    for(entry_index = 0; entry_index < handle_
ptr->total_number_of_entries; entry_index++)
    00499    {
    00500        XMC_VADC_GROUP_QueueInsertChannel(queue_group_ptr, *entries_array[entry_index]);
    00501    }
    00502 }
    00503 #endif
    00504
    00505 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
    00506 __STATIC_INLINE bool ADC_MEASUREMENT_ADV_ArbitrationStatus(const ADC_MEASUREMENT_ADV_t *const handle_ptr)
    00507 {
    00508    bool clock_reset_check;
    00509    bool arbitration_status;
    00510
    00511 #if !defined(CLOCK_GATING_SUPPORTED) || !defined(ADC_MEASUREMENT_ADV_CHECK_CLOCK_GATING)
    00512        clock_reset_check = (bool)false;
    00513    #endif
    00514    arbitration_status = (bool)false;
    00515
    00516    /* To check if the arbiter is already enabled. Before checking this ensure that clock and reset states are correct */
    00517 #if defined(CLOCK_GATING_SUPPORTED) && defined(ADC_MEASUREMENT_ADV_CHECK_CLOCK_GATING)
    00518        clock_reset_check = !XMC_SCU_CLOCK_IsPeripheralClockGated(XMC_SCU_PERIPHERAL_CLOCK_VADC);
    00519    #endif
    00520 #ifdef PERIPHERAL_RESET_SUPPORTED
    00521        clock_reset_check |= !XMC_SCU_RESET_IsPeripheralResetAsserted(XMC_SCU_PERIPHERAL_RESET_VADC);
    00522    #endif
if(clock_reset_check != (bool)false) {
    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
        if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN >= handle_ptr->req_src)
    #endif
    #endif

    arbitration_status = XMC_VADC_GROUP_ScanIsArbitrationSlotEnabled(group_ptrs[handle_ptr->group_index]);
#endif

    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
        else
    #endif
    #endif

    arbitration_status = XMC_VADC_GROUP_QueueIsArbitrationSlotEnabled(group_ptrs[handle_ptr->group_index]);
#endif

    #endif
}

return (arbitration_status);

__STATIC_INLINE void ADC_MEASUREMENT_ADV_lDisableArbitration(const ADC_MEASUREMENT_ADV_t *const handle_ptr,
bool arbitration_status) {

if (arbitration_status == (bool)false)
{
    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
        if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN >= handle_ptr->req_src)
    #endif
    
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
        else
    #endif
    
    XMC_VADC_GROUP_ScanDisableArbitrationSlot(group_ptrs[handle_ptr->group_index]);
    
    #endif
} /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

/* Insert channels into the hardware*/

void ADC_MEASUREMENT_ADV_lInsertChannels(const ADC_MEASUREMENT_ADV_t *const handle_ptr)
{
    XMC_ASSERT("ADC_MEASUREMENT_ADV_InsertChannels:Invalid handle_ptr", (handle_ptr != NULL))

    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN >= handle_ptr->req_src)
    #endif
{
    #ifdef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
    #ifdef ADC_MEASUREMENT_ADV_ADC_SCAN_USED
    if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_SCAN != handle_ptr->req_src)
    #endif
    {
        XMC_VADC_GROUP_ScanAddMultipleChannels(group_ptrs[handle_ptr->group_index], handle_ptr->local_scan_handle->insert_mask);
    }
    #endif
}
#ifdef ADC_MEASUREMENT_ADV_ADC_SCAN_USED
#ifdef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
else
else
    ADC_SCAN_AllEntriesInserted(handle_ptr->scan_handle);
#endif

#ifdef ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED
#ifdef ADC_MEASUREMENT_ADV_ADC_QUEUE_USED
if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_QUEUE != handle_ptr->req_src)
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

/* Select the boundary for a channel and configure its value as well.*/
void ADC_MEASUREMENT_ADV_lSetBoundary(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, 
XMC_VADC_CHANNEL_BOUNDARY_t boundary_select, 
uint32_t boundary_value)
{
    XMC_ASSERT("ADC_MEASUREMENT_ADV_SetBoundary:Invalid handle_ptr", (handle_ptr != NULL))

    switch(boundary_select)
{ 
    case XMC_VADC_CHANNEL_BOUNDARY_GROUP_BOUND0: 
    case XMC_VADC_CHANNEL_BOUNDARY_GROUP_BOUND1:
        XMC_VADC_GROUP_SetIndividualBoundary(
            group_ptrs[handle_ptr->group_index], boundary_select,
            (uint16_t)boundary_value);
        break;
    case XMC_VADC_CHANNEL_BOUNDARY_GLOBAL_BOUND0: 
    case XMC_VADC_CHANNEL_BOUNDARY_GLOBAL_BOUND1:
        XMC_VADC_GLOBAL_SetIndividualBoundary(ADC_MEASUREMENT_ADV_GLOBAL_PTR, 
            boundary_select,
            (uint16_t)boundary_value);
        break;
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
#ifndef ADC_MEASUREMENT_ADV_SYNC_USED
    /*Address the errata for the incorrect conversion.*/
    void ADC_MEASUREMENT_ADV_lSyncADCClocks(void)
    {
        int32_t group_index;
        for (group_index = (int32_t)XMC_VADC_MAXIM
UM_NUM_GROUPS - (int32_t)1; group_index >= (int32_t)0 ; group_index--)
00661    {
00662        XMC_VADC_GROUP_SetPowerMode(group_ptrs[group_index],XMC_VADC_GROUP_POWERMODE_OFF);
00663    }
00664
00665    for (group_index = (int32_t)XMC_VADC_MAXIM
UM_NUM_GROUPS - (int32_t)1; group_index > (int32_t)0 ; group_index--)
00666    {
00667        XMC_VADC_GROUP_SetSyncSlave(group_ptrs[group_index], (uint32_t)0, (uint32_t)group_index);
00668        XMC_VADC_GROUP_CheckSlaveReadiness(group_ptrs[0U], (uint32_t)group_index);
00669    }
00670    XMC_VADC_GROUP_SetSyncMaster(group_ptrs[0U]);
00671
00672    XMC_VADC_GROUP_SetPowerMode(group_ptrs[0U],XMC_VADC_GROUP_POWERMODE_NORMAL);
00673 00674    }
00675 #endif
00676
00677  /***************************************************************************************
00678  ***************************************************************************************
00679  ***************************************************************************************
00680  ***************************************************************************************
00681  */ API IMPLEMENTATION
00682  ***************************************************************************************
00683  ***************************************************************************************
00684  ***************************************************************************************
00685  /*This function returns the version of the ADC_MEASUREMENT App*/
00686  DAVE_APP_VERSION_t ADC_MEASUREMENT_ADV_GetApp
00687  (void)
{   DAVE_APP_VERSION_t version;

    version.major = (uint8_t) ADC_MEASUREMENT_ADV_MAJOR_VERSION;
    version.minor = (uint8_t) ADC_MEASUREMENT_ADV_MINOR_VERSION;
    version.patch = (uint8_t) ADC_MEASUREMENT_ADV_PATCH_VERSION;

    return version;
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

/* Enables the arbiter of the selected request source*/
void ADC_MEASUREMENT_ADV_StartADC(const ADC_MEASUREMENT_ADV_t *const handle_ptr) {
#if defined ADC_MEASUREMENT_ADV_SCAN_USED
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
        if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN >= handle_ptr->req_src)
        {
            XMC_VADC_GROUP_ScanEnableArbitrationSlot(group_ptrs[handle_ptr->group_index]);
        }
    #endif
#else
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
        else
    #endif
    #endif
    #endif
}
XMC_VADC_GROUP_QueueEnableArbitrationSlot(group_ptrs[handle_ptr->group_index]);
#endif
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
/*	Starts	the	ADC	conversions	by	causing	a	software	start	of	conversion*/
void ADC_MEASUREMENT_ADV_SoftwareTrigger(const ADC_MEASUREMENT_ADV_t *const handle_ptr)
{
    XMC_ASSERT("ADC_MEASUREMENT_ADV_StartConversion:Invalid handle_ptr", (handle_ptr != NULL))
    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
    if (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN >= handle_ptr->req_src)
    #endif
    {
        XMC_VADC_GROUP_ScanTriggerConversion(group_ptrs[handle_ptr->group_index]);
    }
    #endif
    #ifdef ADC_MEASUREMENT_ADV_QUEUE_USED
    #ifdef ADC_MEASUREMENT_ADV_SCAN_USED
    else
    #endif
    {
        XMC_VADC_GROUP_QueueTriggerConversion(group_ptrs[handle_ptr->group_index]);
    }
    #endif
}
/* Enables the NVIC (if needed) when scan/queue request source is consumed internally in the AP P. */

void ADC_MEASUREMENT_ADC_lNvicEnable(const ADC_MEASUREMENT_ADV_t *const handle_ptr)
{
    #ifdef ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED
        if (((bool)true == handle_ptr->local_scan_handle->rs_intr_handle.interrupt_enable) &&
            (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_SCAN == handle_ptr->req_src))
            NVIC_EnableIRQ((IRQn_Type)handle_ptr->local_scan_handle->rs_intr_handle.node_id);
    #endif

    #ifdef ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED
        if (((bool)true == handle_ptr->local_queue_handle->rs_intr_handle.interrupt_enable) &&
            (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_LOCAL_QUEUE == handle_ptr->req_src))
            NVIC_EnableIRQ((IRQn_Type)handle_ptr->local_queue_handle->rs_intr_handle.node_id);
    #endif

    /* Configures the result results. */
__STATIC_INLINE void ADC_MEASUREMENT_ADC_lResultInit(const ADC_MEASUREMENT_ADV_CHANNEL_t *indexed) {
  #ifdef ADC_MEASUREMENT_ADV_FIFO_USED
    uint8_t fifo_num_of_intermediate_stages;
    uint8_t fifo_index;
    uint8_t fifo_head;
  #endif
  #ifdef ADC_MEASUREMENT_ADV_FIFO_USED
    /* If FIFO is selected for the particular channel then do the FIFO initializations*/
    if ((uint32_t)0 != indexed->max_fifo_required) {
      /*Excluding the head and tail from the total number of FIFO elements needed*/
      fifo_num_of_intermediate_stages = indexed->max_fifo_required - ADC_MEASUREMENT_ADV_RESERVED_REGISTERS;
      fifo_head = (uint8_t)indexed->ch_handle->result_reg_number;
      for (fifo_index = 1; fifo_index <= fifo_num_of_intermediate_stages; fifo_index++) {
        XMC_VADC_GROUP_ResultInit(group_ptrs[indexed->group_index], (uint32_t)fifo_head - fifo_index, &ADC_MEASUREMENT_ADV_fifo_intermediate_stage);
      }
    }
    /* For the FIFO Tail configuration*/
    XMC_VADC_GROUP_ResultInit(group_ptrs[indexed->group_index], (uint32_t)indexed->result_fifo_tail_number,

/* Initialize for configured result registers For FIFO Head configuration*/
XMC_VADC_GROUP_ResultInit(group_ptrs[indexed->group_index], (uint32_t)indexed->ch_handle->result_reg_number,
indexed->res_handle[ADC_MEASUREMENT_ADV_HEAD_RESULT_REG_CONFIG]);
}
#endif
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
/* Initialization routine to call ADC LLD API's */
ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_Init(const ADC_MEASUREMENT_ADV_t *const handle_ptr)
{
const ADC_MEASUREMENT_ADV_CHANNEL_t *indexed;
uint8_t ch_num;
uint8_t total_number_of_channels;
ADC_MEASUREMENT_ADV_STATUS_t status;

#define ADC_MEASUREMENT_ADV_SHS_GAIN_NON_DEFAULT
uint8_t channel_number;
#endif

bool arbitration_status = (bool)false;
if (ADC_MEASUREMENT_ADV_STATUS_UNINITIALIZED == *handle_ptr->init_state)
{
    arbitration_status = ADC_MEASUREMENT_ADV_lArbitrationStatus(handle_ptr);
    /* Initialize the scan/queue request source.*/
    status = ADC_MEASUREMENT_ADV_lRequestSrcInit(handle_ptr);
    /* Disable the Arbitration if no other instance has enabled it*/
    ADC_MEASUREMENT_ADV_lDisableArbitration(handle_ptr, arbitration_status);
    if (ADC_MEASUREMENT_ADV_SYNC_USED) { /* Configure the Sync conversion operation */
        ADC_MEASUREMENT_ADV_lSyncInit(handle_ptr);
    } else
        ADC_MEASUREMENT_ADV_lSyncADCClocks();
    /* Initialize the SR lines for the Channel event and the Result event, if required*/
    if (ADC_MEASUREMENT_ADV_MUX_USED)
ifdef ADC_MEASUREMENT_ADV_MUX_NOT_ALL_USED
if (handle_ptr->event_config != NULL) endif
{
  (handle_ptr->event_config)();
}
endif

total_number_of_channels = (uint8_t)hand
le_ptr->total_number_of_channels;
for (ch_num = (uint8_t)0; ch_num < (uint8_t)total_number_of_channels; ch_num++)
{
  indexed = handle_ptr->channel_array[ch _num];
  /* Initialize for configured channels*/
  XMC_VADC_GROUP_ChannelInit(group_ptrs[ indexed->group_index],(uint32_t)indexed->ch_num, i
dexed->ch_handle);
#endif
#endif

#if (XMC_VADC_SHS_AVAILABLE == 1U)
#ifdef ADC_MEASUREMENT_ADV_SHS_GAIN_NON_DE
fault
channel_number = indexed->ch_num;
#endif
#endif
  if (indexed->ch_handle->alias_channel != XMC_VADC_CHANNEL_ALIAS_DISABLED)
  { channel_number = indexed->ch_handle ->alias_channel;
  }
  endif
  XMC_VADC_GLOBAL_SHS_SetGainFactor(ADC_
MEASUREMENT_ADV_SHS_PTR,
00860 t8_t)indexed->shs_gain_factor,
00861 (XMC _VADC_GROUP_INDEX_t)indexed->group_index,
00862 chan nel_number);
00863 #endif
00864 #endif
00865
00866 /* Result Init both with and without F IFO */
00867 ADC_MEASUREMENT_ADC_lResultInit(indexed);
00868
00869 #ifdef ADC_MEASUREMENT_ADV_ANALOG_IO_USED
00870 /* ANALOG_IO initialization for the channel*/
00871 #ifndef ADC_MEASUREMENT_ADV_ANALOG_IO_ALL_CHANNELS_USED
00872 if(indexed->analog_io_config != NULL)
00873 #endif
00874 {
00875 status |= (ADC_MEASUREMENT_ADV_STATU S_t) ANALOG_IO_Init(indexed->analog_io_config);
00876 }
00877 #endif
00878 }
00879
00880 #if defined(ADC_MEASUREMENT_ADV_ADC_SCAN_USED) || defined(ADC_MEASUREMENT_ADV_ADC_QUEUE_USED)
00881 /* Load the queue/scan entries into ADC_ QUEUE/ADC_SCAN.
00882 * This would load the scan/ queue entri es into the software buffers in the ADC_SCAN/ADC_Q UEUE APPs.
00883 * A call to this API would only configu
re the ADC_SCAN/ADC_QUEUE software buffers and will not be
  programmed into the Hardware. The programming into the hardware is taken care by another API.
/*
  if defined(ADC_MEASUREMENT_ADV_LOCAL_SCAN_USED) || defined(ADC_MEASUREMENT_ADV_LOCAL_QUEUE_USED)
  if ( (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_SCAN == handle_ptr->req_src) ||
       (ADC_MEASUREMENT_ADV_REQUEST_SOURCE_QUEUE == handle_ptr->req_src))
#endif
{
  for (ch_num = (uint8_t)0; ch_num < (uint8_t)handle_ptr->total_number_of_entries; ch_num++)
  {
    ADC_MEASUREMENT_ADV_lInsertEntry(handle_ptr,ch_num);
  }
#endif
/* Enables the NVIC node if NVIC node is consumed inside the APP*/
ADC_MEASUREMENT_ADC_lNvicEnable(handle_ptr);
/* Load the queue/scan entries into the hardware */
ADC_MEASUREMENT_ADV_lInsertChannels(handle_ptr);
/*Start the arbiter of the ADC request source after the initialization. */
```c
#ifdef ADC_MEASUREMENT_ADV_START_ADC
#ifdef ADC_MEASUREMENT_ADV_NOT_ALL_REQ_STA

if ((bool)false != handle_ptr->start_at_initialization)
{
    ADC_MEASUREMENT_ADV_StartADC(handle_ptr);
}
#endif

*handle_ptr->init_state = status;
}

return (*handle_ptr->init_state);

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

/*
Set the Fast compare value*/

ADC_MEASUREMENT_ADV_STATUS_t ADC_MEASUREMENT_ADV_SetFastCompareValue(const ADC_MEASUREMENT_ADV_CHANNEL_t

*const handle_ptr, uint16_t compare_value)
{
    ADC_MEASUREMENT_ADV_STATUS_t status;
    XMC_ASSERT("ADC_MEASUREMENT_ADV_SetFastCompareValue:Invalid handle_ptr", (handle_ptr != NULL ))

    status = ADC_MEASUREMENT_ADV_STATUS_FAILURE;

    if ( (uint32_t)compare_value <= ADC_MEASUREMENT_ADV_STATUS_FAILURE)
```
EMENT_ADV_10_BIT_MAX_VALUE)
00930  {
00931         XMC_VADC_GROUP_SetResultFastCompareValue(group_ptrs[handle_ptr->group_index],
00932         (uint32_t) handle_ptr->ch_handle->result_reg_number,
00933         (XMC_VADC_RESULT_SIZE_t)compare_value);
00934         status = ADC_MEASUREMENT_ADV_STATUS_SUCCESS;
00935     }
00936     return (status);
00937 }
00938
00939 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
00940 /* Set the Subtraction value */
00941 void ADC_MEASUREMENT_ADV_SetSubtractionValue(const ADC_MEASUREMENT_ADV_t *const handle_ptr,
00942 ADC_MEASUREMENT_ADV_SUBTRACTION_t subtraction_alignment,
00943     uint16_t subtraction_value)
00944 {
00945     uint32_t groups;
00946     uint8_t i;
00947     XMC_ASSERT("ADC_MEASUREMENT_ADV_SetSubtractionValue:Invalid handle_ptr", (handle_ptr != NULL ))
00948     #ifdef ADC_MEASUREMENT_ADV_SYNC_USED
00949     groups = handle_ptr->sync_slaves;
00950     #else
00951     groups = (uint32_t)0;
00952     #endif
00953     for (i = 0; i < groups; i++)
00954         for (j = 0; j < subtraction_alignment; j++)
00955             subtraction_value
00956             = subtraction_value + subtraction_alignment;
00957 #endif
00958 }
```c
#define groups |= (uint32_t)((uint32_t)1 << (uint32_t)handle_ptr->group_index);

for ( i = (uint8_t)0; i < (uint8_t)XMC_VADC_MAXIMUM_NUM_GROUPS ; i++)
{
    if ( (bool)false != (bool)((groups >> i) & (uint32_t)0x1 ))
    {
        XMC_VADC_GROUP_SetResultSubtractionValue(group_ptrs[i], (uint16_t)(subtraction_value << (uint32_t)subtraction_alignment));
    }
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

/* Configure the resolution and sampling time in an iclass */
void ADC_MEASUREMENT_ADV_ConfigureChannelClass(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
                                             const XMC_VADC_GROUP_CLASS_t *config)
{
    XMC_ASSERT("ADC_MEASUREMENT_ADV_ConfigureChannelClass:Invalid class configuration", (config != NULL))
    XMC_VADC_GROUP_InputClassInit(group_ptrs[handle_ptr->group_index], *config, XMC_VADC_GROUP_CONV_STD,
```
(uint32_t)handle_ptr->ch_handle->input_class);
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

/*
Sets the alternate reference for a particular channel*/
void ADC_MEASUREMENT_ADV_SetAlternateReference(
    const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
    const XMC_VADC_CHANNEL_REF_t reference_select)
{
    XMC_ASSERT("ADC_MEASUREMENT_ADV_SetAlternateReference:Invalid handle_ptr", (handle_ptr != NULL));
    XMC_VADC_GROUP_ChannelSetInputReference(group_ptrs[handle_ptr->group_index], (uint32_t)handle_ptr->ch_num,
    reference_select);
}

#if (XMC_VADC_SHS_AVAILABLE == 1U)
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/

/*
Sets the gain ratio for a particular channel*/
void ADC_MEASUREMENT_ADV_SetChannelGain(
    const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
    const ADC_MEASUREMENT_ADV_GAIN_t gain_factor)
```c
00994  {
00995      uint8_t channel_number;
00996      XMC_ASSERT("ADC_MEASUREMENT_ADV_SetChannel
00997          Gain:Invalid handle_ptr", (handle_ptr != NULL))
00998    channel_number = handle_ptr->ch_num;
00999  #ifdef ADC_MEASUREMENT_ADV_SHS_GAIN_ALIAS
01000      if (handle_ptr->ch_handle->alias_channel != XMC_VADC_CHANNEL_ALIAS_DISABLED)
01001          { channel_number = (uint8_t) handle_ptr->c
01002            h_handle->alias_channel;
01003      }
01004  #endif
01005  XMC_VADC_GLOBAL_SHS_SetGainFactor(ADC_MEAS
01006          UREMENT_ADV_SHS_PTR,
01007                              (uint8_t)gain_factor,
01008                              (XMC_VADC_GROUP_INDEX_t)handle_ptr->group_index,
01009    (uint32_t)channel_number);
01010  #endif
01011
01012  /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
01013      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
01014      ~~~~~~~~~~~~~~~~~~~~~~~~*/
01015
01016  /* Select the boundary for the channel*/
01017  void ADC_MEASUREMENT_ADV_SelectBoundary(const
01018      ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
01019      XMC_VADC_BOUNDARY_SELECT_t boundary,
01020      XMC_VADC_CHANNEL_BOUNDARY_t boundary_selection)
01021    {
01022      XMC_ASSERT("ADC_MEASUREMENT_ADV_SelectBoun
01023```
dary:Invalid handle_ptr", (handle_ptr != NULL))
01019
01020 XMC_VADC_GROUP_ChannelSetBoundarySelection(group_ptrs[handle_ptr->group_index], (uint32_t)handle_ptr->ch_num,
01021 boundary, boundary_selection);
01022
01023 }
01024
01025 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~*~/
01026 /* configure the upper boundary for a channel.*/
01027 void ADC_MEASUREMENT_ADV_SetBoundaryUpper(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr,
01028 uint32_t boundary_value)
01029 {
01030 uint32_t boundary_select;
01031 XMC_ASSERT("ADC_MEASUREMENT_ADV_SetBoundaryUpper:Invalid handle_ptr", (handle_ptr != NULL))
01032
01033 boundary_select = handle_ptr->ch_handle->upper_boundary_select;
01034
01035 ADC_MEASUREMENT_ADV_1SetBoundary(handle_ptr, (XMC_VADC_CHANNEL_BOUNDARY_t)boundary_select, boundary_value);
01036 }
01037
01038 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~*~/
01039 /* configure the lower boundary for a channel.
void ADC_MEASUREMENT_ADV_SetBoundaryLower(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr, uint32_t boundary_value) {
    uint32_t boundary_select;
    XMC_ASSERT("ADC_MEASUREMENT_ADV_SetBoundaryLower:Invalid handle_ptr", (handle_ptr != NULL))
    boundary_select = handle_ptr->ch_handle->lower_boundary_select;
    ADC_MEASUREMENT_ADV_lSetBoundary(handle_ptr, (XMC_VADC_CHANNEL_BOUNDARY_t)boundary_select, boundary_value);
}

/* Aliased channel number is returned if the channel has alias enabled */
XMC_VADC_CHANNEL_ALIAS_t ADC_MEASUREMENT_ADV_GetAliasValue(const ADC_MEASUREMENT_ADV_CHANNEL_t *const handle_ptr) {
    XMC_VADC_CHANNEL_ALIAS_t return_value;
    ADC_MEASUREMENT_ADV_ALIAS_t alias_value;
    XMC_ASSERT("ADC_MEASUREMENT_ADV_GetAliasValue:Invalid handle_ptr", (handle_ptr != NULL))
    alias_value.alias = XMC_VADC_GROUP_GetAlias(group_ptrs[handle_ptr->group_index]);
if ((uint8_t)0 == handle_ptr->ch_num )
{
    return_value = (XMC_VADC_CHANNEL_ALIAS_t)alias_value.alias0;
    if ((uint32_t)0 == alias_value.alias0)
    {
        return_value = XMC_VADC_CHANNEL_ALIAS_DISABLED;
    }
}
else if ((uint8_t)1 == handle_ptr->ch_num )
{
    return_value = (XMC_VADC_CHANNEL_ALIAS_t)alias_value.alias1;
    if ((uint32_t)1 == alias_value.alias1)
    {
        return_value = XMC_VADC_CHANNEL_ALIAS_DISABLED;
    }
}
else
{
    return_value = XMC_VADC_CHANNEL_ALIAS_DISABLED;
}
return(return_value);

#ifndef ADC_MEASUREMENT_ADV_SYNC_USED
/* Enables uniform conversion configurations across slaves*/
void ADC_MEASUREMENT_ADV_SetIclass(const ADC_MEASUREMENT_ADV_t *const handle_ptr) {
    XMC_VADC_CHANNEL_CONV_t req_iclass;
    XMC_VADC_GROUP_CLASS_t conv_class;
    XMC_VADC_GLOBAL_CLASS_t conv_class_global;
    
    XMC_ASSERT("ADC_MEASUREMENT_ADV_SetIclass: Invalid handle_ptr", (handle_ptr != NULL))
    
    req_iclass = ADC_MEASUREMENT_ADV_lGetIclasses(handle_ptr);
    conv_class = XMC_VADC_GROUP_GetInputClass(group_ptrs[handle_ptr->group_index], req_iclass);
    conv_class_global.globiclass = conv_class.g_iclass0;
    XMC_VADC_GLOBAL_InputClassInit(ADC_MEASUREMENT_ADV_GLOBAL_PTR, conv_class_global, XMC_VADC_GROUP_CONV_STD, (uint32_t)ADC_MEASUREMENT_ADV_GLOBAL1);
}