AD7147 Slider Firmware

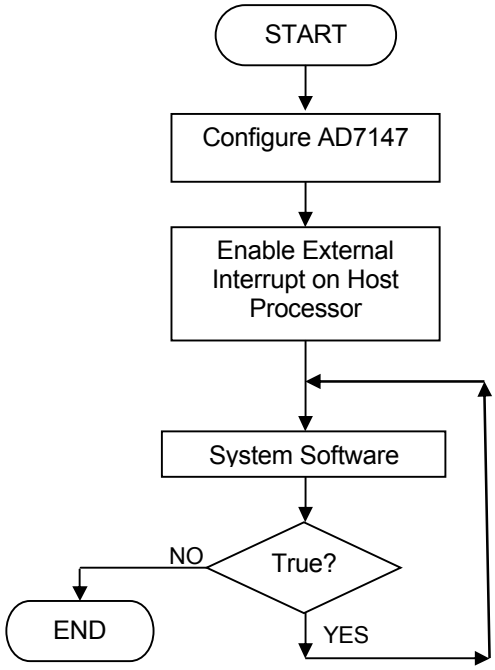
Integration Guide

Introduction

The following guide attempts to describe the software integration process required to implement an 8-Segment slider sensor application.
The 8-Segment slider firmware configures the device and then reads the Interrupt Status Registers, the ADC results, the Max average values and the Ambient values for each stage of the sensor from the AD7147 when a sensor is activated.

AD7147 Configuration

On power-up the AD7147 registers must be configured to allow the part to function correctly in the application. This configuration step is done in the main system software; once the AD7147 register map is initialised an external host interrupt must then be configured. Touching any area of the 8-Segment Slider will then cause an interrupt to fire which in turn causes software to jump to an Interrupt Service Routine (ISR) where registers will be read from the AD7147 and the absolute position will be computed.
The following flowchart describes the AD7147 and ISR configuration process.



The AD7147 register configuration is done in two stages; firstly all 12 stages must be initialised with the sensor configuration including CIN connection, initial offset and threshold sensitivity information, there are 8 registers for each of the 12 stages. The second stage of the register configuration is to program the Bank 1 registers of the AD7147; these registers contain the power modes, environmental calibration and interrupt configuration settings.

Once these registers have been initialised a single read of the Interrupt Status registers is performed to clear the INT pin in case an interrupt was detected during the register configuration process, the part is then ready to respond to sensor activations immediately.
The AD7147 register configuration information is usually contained in an “AD7147 Config.c” file provided by ADI. The complete register configuration takes approximately 6.5mS with a 400kHz I2C interface as shown in Fig.1.

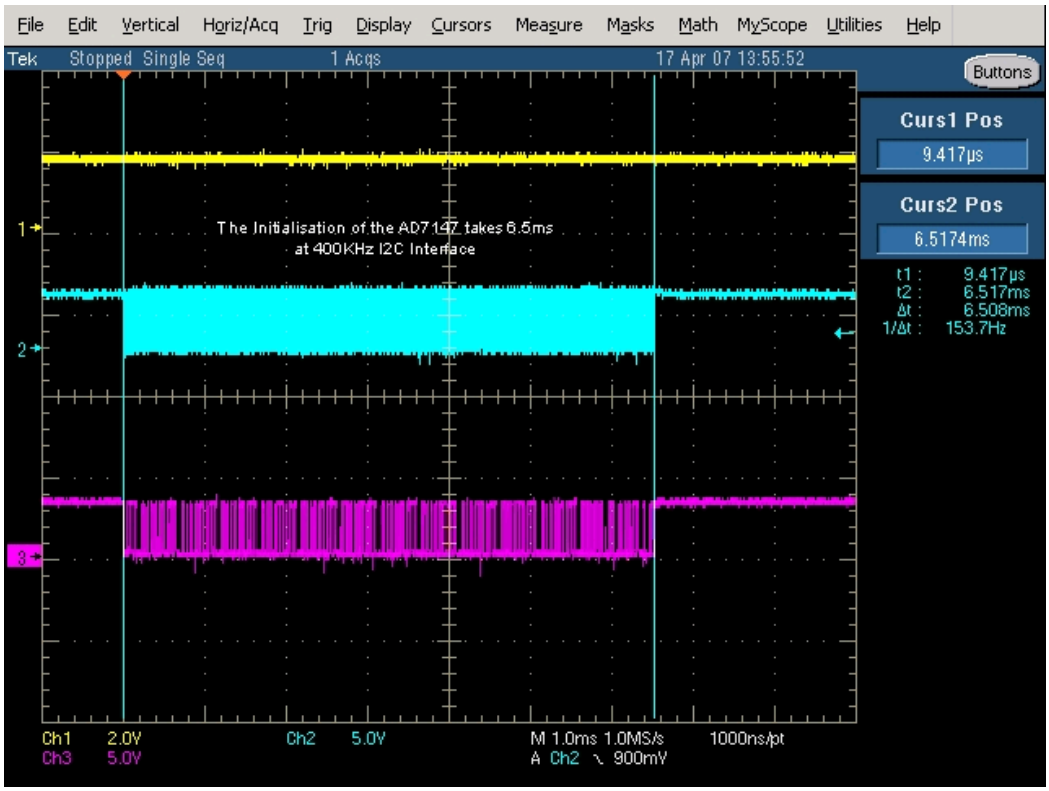


Fig.1 AD7147 Register Configuration

8-Segment Slider Configuration

In the following “AD7147 - Slider Firmware Config.c” file, an 8-Segment Slider sensor made out of 8 sensors is connected to CIN5, CIN6… CIN12. The inputs are configured to connect to the positive input of Stages 0 to 7 respectively, all other CIN inputs are not connected and bank 1 and bank 2 registers are initialised as follows:

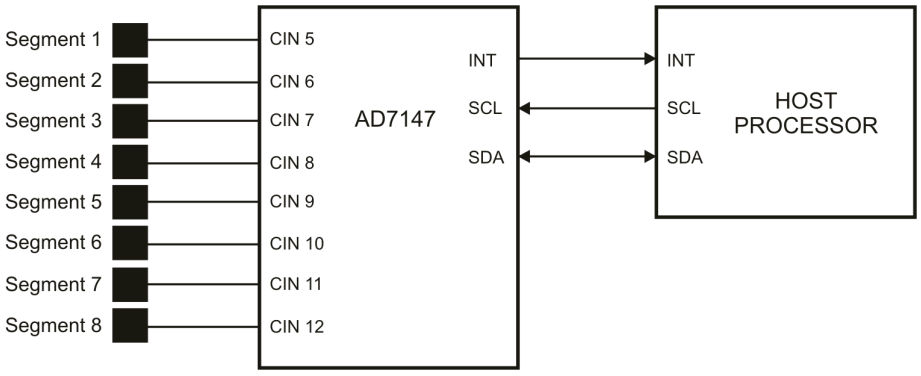


Fig.2 AD7147 – 8 Segment Slider Connection Diagram

```
#include "Include\General Definitions.h"
#include "Include\AD7147RegMap.h"
#include "Include\AD7147 - Slider Definitions.h"


//---------------------------------
//Function prototypes
//---------------------------------
//External functions
extern void ReadFromAD7147(const WORD RegisterStartAddress, const
BYTE NumberOfRegisters, WORD *DataBuffer, const WORD OffsetInBuffer);
extern void WriteToAD7147(const WORD RegisterAddress, const BYTE
NumberOfRegisters, WORD *DataBuffer, const BYTE OffsetInBuffer);

//Local functions
void ConfigAD7147(void);


//---------------------------------
//Global variables
//---------------------------------

//External global variables
//---------------------------------
extern WORD xdata AD7147Registers[NUMBER_OF_AD7147_REGISTERS];
extern WORD xdata UpperClampValue[NB_OF_SENSORS_FOR_SLIDER];


//----------------------
//Function declarations
//----------------------
void ConfigAD7147(void);

//---------------------
```

```
//Function definitions
//---------------------
void ConfigAD7147(void)
{
   WORD xdata StageBuffer[8];
   //=========================================
   //============ Bank 2 Registers ============
   //=========================================

   //=========================================
   //= Stage 0 - CIN5(+) - Slider Segment S1 =
   //=========================================
   StageBuffer[0]=0xFBFF; //Register  0x88
   StageBuffer[1]=0x1FFF; //Register  0x89
   StageBuffer[2]=0x0100; //Register  0x8A
   StageBuffer[3]=0x2626; //Register  0x8B
   StageBuffer[4]=2000;   //Register  0x8C
   StageBuffer[5]=2000;    //Register  0x8D
   StageBuffer[6]=2000;    //Register  0x8E
   StageBuffer[7]=2000;    //Register  0x8F
   WriteToAD7147(STAGE0_CONNECTION, 8, StageBuffer, 0);

   //=========================================
   //= Stage 1 - CIN6(+) - Slider Segment S2 =
   //=========================================
   StageBuffer[0]=0xEFFF; //Register  0x90
   StageBuffer[1]=0x1FFF; //Register  0x91
   StageBuffer[2]=0x0100; //Register  0x92
   StageBuffer[3]=0x2626; //Register  0x93
   StageBuffer[4]=2000;   //Register  0x94
   StageBuffer[5]=2000;   //Register  0x95
   StageBuffer[6]=2000;   //Register  0x96
   StageBuffer[7]=2000;   //Register  0x97
   WriteToAD7147(STAGE1_CONNECTION, 8, StageBuffer, 0);

   //=========================================
   //= Stage 2 - CIN7(+) - Slider Segment S3 =
   //=========================================
   StageBuffer[0]=0xFFFF; //Register  0x98
   StageBuffer[1]=0x1FFE; //Register  0x99
   StageBuffer[2]=0x0100; //Register  0x9A
   StageBuffer[3]=0x2626; //Register  0x9B
   StageBuffer[4]=2000;   //Register  0x9C
   StageBuffer[5]=2000;   //Register  0x9D
   StageBuffer[6]=2000;   //Register  0x9E
   StageBuffer[7]=2000;   //Register  0x9F
   WriteToAD7147(STAGE2_CONNECTION, 8, StageBuffer, 0);

   //=========================================
   //= Stage 3 - CIN8(+) - Slider Segment S4 =
   //=========================================
   StageBuffer[0]=0xFFFF; //Register  0xA0
   StageBuffer[1]=0x1FFB; //Register  0xA1
   StageBuffer[2]=0x0100; //Register  0xA2
   StageBuffer[3]=0x2626; //Register  0xA3
   StageBuffer[4]=2000;   //Register  0xA4
   StageBuffer[5]=2000;   //Register  0xA5
   StageBuffer[6]=2000;   //Register  0xA6
   StageBuffer[7]=2000;   //Register  0xA7
   WriteToAD7147(STAGE3_CONNECTION, 8, StageBuffer, 0);
```

```
//=======================================
//= Stage 4 - CIN9(+) - Slider Segment S5 =
//=======================================
StageBuffer[0]=0xFFFF; //Register  0xA8
StageBuffer[1]=0x1FEF; //Register  0xA9
StageBuffer[2]=0x0100; //Register  0xAA
StageBuffer[3]=0x2626; //Register  0xAB
StageBuffer[4]=2000;   //Register  0xAC
StageBuffer[5]=2000;   //Register  0xAD
StageBuffer[6]=2000;   //Register  0xAE
StageBuffer[7]=2000;   //Register  0xAF
WriteToAD7147(STAGE4_CONNECTION, 8, StageBuffer, 0);

//=======================================
//= Stage 5 - CIN10(+) - Slider Segment S6 =
//=======================================
StageBuffer[0]=0xFFFF; //Register  0xB0
StageBuffer[1]=0x1FBF; //Register  0xB1
StageBuffer[2]=0x0100; //Register  0xB2
StageBuffer[3]=0x2626; //Register  0xB3
StageBuffer[4]=2000;   //Register  0xB4
StageBuffer[5]=2000;   //Register  0xB5
StageBuffer[6]=2000;   //Register  0xB6
StageBuffer[7]=2000;   //Register  0xB7
WriteToAD7147(STAGE5_CONNECTION, 8, StageBuffer, 0);

//=======================================
//= Stage 6 - CIN11(+) - Slider Segment S7 =
//=======================================
StageBuffer[0]=0xFFFF; //Register  0xB8
StageBuffer[1]=0x1EFF; //Register  0xB9
StageBuffer[2]=0x0100; //Register  0xBA
StageBuffer[3]=0x2626; //Register  0xBB
StageBuffer[4]=2000;   //Register  0xBC
StageBuffer[5]=2000;   //Register  0xBD
StageBuffer[6]=2000;   //Register  0xBE
StageBuffer[7]=2000;   //Register  0xBF
WriteToAD7147(STAGE6_CONNECTION, 8, StageBuffer, 0);

//=======================================
//= Stage 7 - CIN12(+) - Slider Segment S8 =
//=======================================
StageBuffer[0]=0xFFFF; //Register  0xC0
StageBuffer[1]=0x1BFF; //Register  0xC1
StageBuffer[2]=0x0100; //Register  0xC2
StageBuffer[3]=0x2626; //Register  0xC3
StageBuffer[4]=2000;   //Register  0xC4
StageBuffer[5]=2000;   //Register  0xC5
StageBuffer[6]=2000;   //Register  0xC6
StageBuffer[7]=2000;   //Register  0xC7
WriteToAD7147(STAGE7_CONNECTION, 8, StageBuffer, 0);

//===========================
//= Stage 8 - Not connected =
//===========================
StageBuffer[0]=0xFFFF; //Register  0xC8
StageBuffer[1]=0x3FFF; //Register  0xC9
StageBuffer[2]=0x0000; //Register  0xCA
StageBuffer[3]=0x2626; //Register  0xCB
StageBuffer[4]=5000;   //Register  0xCC
StageBuffer[5]=5000;   //Register  0xCD
```

```
StageBuffer[6]=5000;   //Register  0xCE
StageBuffer[7]=5000;   //Register  0xCF
WriteToAD7147(STAGE8_CONNECTION, 8, StageBuffer, 0);

//============================
//= Stage 9 - Not connected =
//============================
StageBuffer[0]=0xFFFF; //Register  0xD0
StageBuffer[1]=0x3FFF; //Register  0xD1
StageBuffer[2]=0x0000; //Register  0xD2
StageBuffer[3]=0x2626; //Register  0xD3
StageBuffer[4]=5000;   //Register  0xD4
StageBuffer[5]=5000;   //Register  0xD5
StageBuffer[6]=5000;   //Register  0xD6
StageBuffer[7]=5000;   //Register  0xD7
WriteToAD7147(STAGE9_CONNECTION, 8, StageBuffer, 0);

//============================
//= Stage 10 - Not connected =
//============================
StageBuffer[0]=0xFFFF; //Register  0xD8
StageBuffer[1]=0x3FFF; //Register  0xD9
StageBuffer[2]=0x0000; //Register  0xDA
StageBuffer[3]=0x2626; //Register  0xDB
StageBuffer[4]=5000;   //Register  0xDC
StageBuffer[5]=5000;   //Register  0xDD
StageBuffer[6]=5000;   //Register  0xDE
StageBuffer[7]=5000;   //Register  0xDF
WriteToAD7147(STAGE10_CONNECTION, 8, StageBuffer, 0);

//============================
//= Stage 11 - Not connected =
//============================
StageBuffer[0]=0xFFFF; //Register  0x80
StageBuffer[1]=0x3FFF; //Register  0x81
StageBuffer[2]=0x0000; //Register  0x82
StageBuffer[3]=0x2626; //Register  0x83
StageBuffer[4]=5000;   //Register  0x84
StageBuffer[5]=5000;   //Register  0x85
StageBuffer[6]=5000;   //Register  0x86
StageBuffer[7]=5000;   //Register  0x87
WriteToAD7147(STAGE11_CONNECTION, 8, StageBuffer, 0);

//========================================
//============ Bank 1 Registers ============
//========================================
//Initialisation of Bank 1 Registers
AD7147Registers[PWR_CONTROL]=0x02B2;      //Register 0x00
WriteToAD7147(PWR_CONTROL, 1, AD7147Registers, 0);

//Read High and Low Limit Status registers to clear INT pin
ReadFromAD7147(STAGE_LOW_LIMIT_INT, 3, AD7147Registers,
              STAGE_LOW_LIMIT_INT);//Registers 0x08, 0x09 and 0x0A

AD7147Registers[AMB_COMP_CTRL0]=0x3233;       //Register 0x02
AD7147Registers[AMB_COMP_CTRL1]=0x0A19;       //Register 0x03
AD7147Registers[AMB_COMP_CTRL2]=0x0832;       //Register 0x04
AD7147Registers[STAGE_LOW_INT_EN]=0x0000;     //Register 0x05
AD7147Registers[STAGE_HIGH_INT_EN]=0x0000;    //Register 0x06
AD7147Registers[STAGE_COMPLETE_INT_EN]=0x0001;//Register 0x07
```

```
   WriteToAD7147(AMB_COMP_CTRL0, 6, AD7147Registers, AMB_COMP_CTRL0);

   //Enable calibration for all sequences in use
   AD7147Registers[STAGE_CAL_EN]=0x00FF;
   WriteToAD7147(STAGE_CAL_EN, 1, AD7147Registers,
                 STAGE_CAL_EN);//Register 0x01

   //Read High and Low Limit Status registers to clear INT pin
   ReadFromAD7147(STAGE_LOW_LIMIT_INT, 3, AD7147Registers,
                  STAGE_LOW_LIMIT_INT);//Registers 0x08, 0x09 and 0x0A
}
```

Hardware Interrupt Configuration

In the 8-Segment Slider application firmware, ADI developed a routine that configures the AD7147 either in End of Conversion interrupt mode or Threshold interrupt mode. On power up the AD7147 is configured in End of Conversion Interrupt mode. This mode is enabled by configuring the 3 interrupt enable registers as follow:

```
AD7147Registers[STAGE_LOW_INT_EN] = 0x0000;   //Register 0x05
AD7147Registers[STAGE_HIGH_INT_EN] = 0x0000;  //Register 0x06
AD7147Registers[STAGE_COMPLETE_INT_EN]=0x0001;//Register 0x07
```

`AD7147Registers[STAGE_COMPLETE_INT_EN] = 0x0001;`
This configures the Stage Complete Interrupt Enable Register 0x07 Bit <0> so that the INT pin will be asserted on completion of each stage 0 ADC conversion, all the stages will also be converted on each interrupt cycle. The STAGE_0_COMPLETE_STATUS_INT bit in the STAGE_COMPLETE_LIMIT_INT register (address 0x0A) is set automatically by the AD7147 when it has completed the conversion of all stages. When this bit is set, the interrupt pin is asserted which causes software to jump to an Interrupt Service Routine where the 3 interrupt status registers at address 0x08, 0x09 and 0x0A are read to clear the hardware interrupt.

In the End of Conversion interrupt mode the frequency of the interrupt depends on bit 8 and bit 9 of the Power Control Register (referenced as `AD7147Registers[PWR_CONTROL]` in the code at address 0x00). These bits control the decimation rate.

Bit 8	Bit 9	Decimation rate	Frequency of the Interrupt for 12 conversion stages
0	0	256	36ms
0	1	128	18ms
1	0	64	9ms
1	1	64	9ms

Table.1 Interrupt Frequency

In order to minimise the interrupt frequency to the host, ADI implemented a software routine in the ISR which changes the AD7147 interrupt mode depending on whether the user is contacting the sensor or not. When the user touches the sensor, the firmware configures the AD7147 in End of Conversion interrupt mode. When the user lifts off the sensor, the firmware configures the AD7147 in Threshold mode shortly after lifting off.

This is the code that implements this functionality:

```
/************************/
/* Change interrupt mode */
/************************/
if (((AD7147Registers[STAGE_HIGH_LIMIT_INT] &
      POWER_UP_INTERRUPT)!=0x0000) ||
     ((AD7147Registers[STAGE_LOW_LIMIT_INT] &
      POWER_UP_INTERRUPT)!=0x0000))
{
    //Configure the AD7147 in End of Conversion Interrupt mode
    if (AD7147Registers[STAGE_COMPLETE_INT_EN]==0x0000)
    {
        AD7147Registers[STAGE_LOW_INT_EN] &= 0xF000;
        AD7147Registers[STAGE_HIGH_INT_EN] &= 0xF000;
        AD7147Registers[STAGE_COMPLETE_INT_EN]=0x0001;
        WriteToAD7147(STAGE_LOW_INT_EN, 3, AD7147Registers,
                          STAGE_LOW_INT_EN);
    }
    InterruptCounterForBtnIntMode=NUMBER_OF_INTS_BEFORE_BTN_INT_MODE;
}
else
{
    //Configure the AD7147 in threshold interrupt mode
    if (InterruptCounterForBtnIntMode>0)
        InterruptCounterForBtnIntMode--;
    if (AD7147Registers[STAGE_HIGH_INT_EN]==0x0000 &&
        InterruptCounterForBtnIntMode==0)
    {
        AD7147Registers[STAGE_LOW_INT_EN] |= POWER_UP_INTERRUPT;
        AD7147Registers[STAGE_HIGH_INT_EN] |= 0x00FF;
        AD7147Registers[STAGE_COMPLETE_INT_EN]=0x0000;
        WriteToAD7147(STAGE_LOW_INT_EN, 3, AD7147Registers,
                      STAGE_LOW_INT_EN);
    }
}
```

When the sensor is not being touched, the AD7147 is configured in threshold mode so that no hardware interrupt are generated by the AD7147. When the sensor is touched a bit will be set in the Stage High Limit Interrupt Status register (address 0x09) and a hardware interrupt will be generated. As soon as a bit is set in this register, we clear bit 0 to 11 in the High and Low Limit Interrupt Enable registers and set bit 0 in the Stage Complete Interrupt Enable Register (address 0x07).

```
AD7147Registers[STAGE_LOW_INT_EN] &= 0xF000;
AD7147Registers[STAGE_HIGH_INT_EN] &= 0xF000;
AD7147Registers[STAGE_COMPLETE_INT_EN]=0x0001;
WriteToAD7147(STAGE_LOW_INT_EN,3,AD7147Registers,STAGE_LOW_INT_EN);
```

When the user lifts off the sensor, the firmware waits for 8 End of Conversion interrupts and changes the AD7147 back into threshold interrupt mode by writing to the following registers:

```
AD7147Registers[STAGE_LOW_INT_EN] |= POWER_UP_INTERRUPT;
AD7147Registers[STAGE_HIGH_INT_EN] |= 0x00FF;
AD7147Registers[STAGE_COMPLETE_INT_EN]=0x0000;
WriteToAD7147(STAGE_LOW_INT_EN,3,AD7147Registers,STAGE_LOW_INT_EN);
```

POWER_UP_INTERRUPT has the value 0x00FF.

`AD7147Registers[STAGE_HIGH_INT_EN] |= 0x00FF;`
This configures the High Limit Interrupt Enable Register 0x06 Bits 0-7; if any of these bits are set, then a hardware interrupt is generated on the INT pin when a sensor is activated and a high limit threshold is exceeded on the AD7147.

`AD7147Registers[STAGE_LOW_INT_EN] |= POWER_UP_INTERRUPT;`
This configures the Low Limit Interrupt Enable Register 0x05 Bits 0-7; this will only generate a hardware interrupt if an error needs to be corrected by a forced recalibration on the AD7147.

There are two sources of error that may need to be corrected:

1. If the sensor is touched when the part is powered up, the initial sensor thresholds calculated by the AD7147 will be incorrect, when the user lifts off the sensor a low limit threshold will be asserted and software must then recalibrate the part in the interrupt service routine
2. The second error that may occur is when the sensor value drifts below the low limit threshold due to excessive temperature or humidity errors. In this case the recalibration function in the interrupt service routine also needs to be called.

`AD7147Registers[STAGE_COMPLETE_INT_EN] = 0x0000;`
If this Register were set to 0x0001 then hardware interrupts would be asserted after each Stage0 conversion regardless of whether we are touching the sensor or not. CDC results would then be available for all 12 stages.

`AD7147Registers[STAGE_CAL_EN] = 0x00FF;`
In this configuration file, the STAGE_CAL_EN bits are set to 0x00FF, this enables the environmental calibration and adaptive threshold logic from stage 0 to stage 7 only as these are the only stages used in this application. Any unused or un-configured stages should not be enabled in the STAGE_CAL_EN register.

Interrupt Service Routine (ISR)

The following diagram shows the interrupt sequence used in the 8-Segment Slider application.

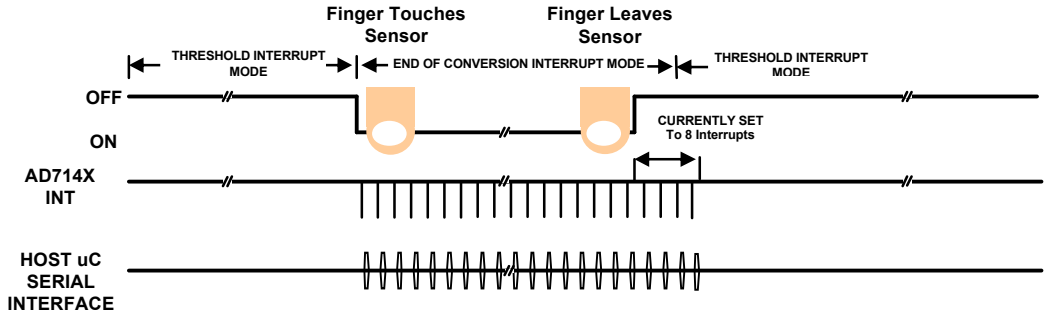


Fig.3 Interrupt Sequence

The interrupt service routine is executed every time the AD7147 generates a hardware interrupt on the INT pin. This interrupt is generated in the following conditions:

- When the user touches the sensor
- When a recalibration event is required
- Every 9ms, 18ms or 36ms when the AD7147 is operating in End of Conversion interrupt mode

In End of Conversion interrupt mode, the AD7147 continuously generates hardware interrupts while the sensor is being touched. As shown in Fig.3 the ISR is called every 9ms, 18ms or 36ms depending on the decimation rate.

The INT pin on the AD7147 is cleared by a read of the Interrupt status registers, 0x08, 0x09 and 0x0A. However, on the very first interrupt, the firmware will write to registers 0x05, 0x06 and 0x07 to change the AD7147 from Threshold interrupt mode to End of Conversion interrupt mode.

Once the user lifts off the sensor software will write again to registers 0x05, 0x06 and 0x07 to change the AD7147 back to Threshold Interrupt mode.

Recalibration

The code checks if a recalibration of the sensors is necessary in case an error has occurred as described below. If a low limit status bit is set which signifies an error then the ReCalibrate() function is called which forces a recalibration of the sensors by writing to the Forced_CAL bit<14> in the Ambient Compensation Control register (at address 0x02). A simple way to test the functionality of this code is to physically touch the sensor while power is applied to the part and the sensors are configured by the ConfigAD7147()

function, when the use lifts off the sensor this software routine should recalibrate the sensors to the untouched value allowing the sensor to function correctly afterwards as shown in Fig.4

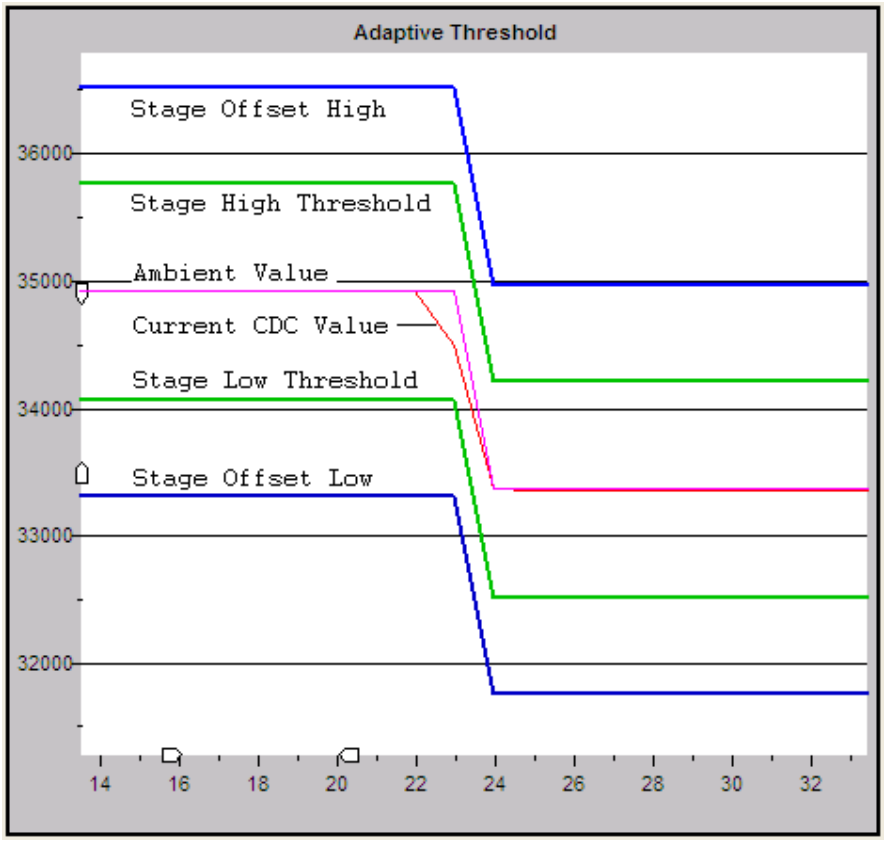


Fig.4 Recalibration

It is absolutely necessary to have the recalibration routine in your code, this software does two things:

- Firstly it recalibrates the sensor response if the user has been touching the sensor while the part is being configured after power up. If the user has been touching the sensor while the part is initialised the upper and lower sensor thresholds will be set at the incorrect levels around the touched sensor value, therefore without the recalibration code the high sensor threshold would never get set as it would always be higher than the touched sensor value. The recalibration code determines if a low threshold limit is exceeded as the user lifts off the sensor after part initialisation and then forces a recalibration of the sensor thresholds on chip so that they now centre around the untouched sensor value which is correct. When the user then touches the sensor again the high threshold will be exceeded as normal.

- Secondly the recalibration code corrects for various errors that may occur causing the sensor value to drift lower very quickly due to

excessive temperature drifts, if the sensor value drifts below the low limit threshold the recalibration code will then re-centre the high and low thresholds around the current sensor value.

8-Segment Slider Algorithm

As soon as the 8-Segment Slider is touched the INT pin on the AD7147 is asserted which causes a hardware interrupt on the host processor to call the Interrupt Service Routine (ISR). In the AD7147 ISR, new position is available after calling the function "`GetNewSliderUpdate()`".

```
BEGIN
    Read Interrupt Status Registers at 0x008, 0x009, 0x00A

    Increment Interrupt Counter

    IF Interrupt counter = 2 THEN
        //Initialisation
        Initialise slider algorithm
    ELSE IF Interrupt counter > 2 THEN
        IF Sensor Error is detected THEN
            //Recalibration
            Force Calibration
            Reset Interrupt Counter to reload the initialisation
        ELSE
            Get new Slider position
        END IF

        //Change of Interrupt mode
        IF slider is touched AND AD7147 is in Threshold Interrupt Mode THEN
            Change AD7147 from Threshold Interrupt Mode to End of Conversion
            Interrupt mode.
            Reload Interrupt Counter for Changing interrupt mode
        ELSE IF slider is not touched AND AD7147 is in End of Conversion Interrupt Mode
                THEN
            Decrement Interrupt Counter for Changing interrupt mode
            IF Counter for Changing interrupt mode = 0 THEN
                Change AD7147 from End of Conversion Interrupt mode to Threshold
                Interrupt Mode.
            END IF
        END IF
    END IF
END
```

Once the ISR is called the first thing that is done is to read the status registers from the AD7147.
Next the firmware keeps track of the first few interrupts after powering up the AD7147. On the second interrupt after power up, initialisation of some variables needs to be done. On the 3rd interrupt after power up, the 8-Segment Slider algorithm can be processed.

First the code checks if a recalibration of the sensors is necessary in case an error has occurred as described above. If a low limit status bit is set which signifies an error then the "`ForceCalibration()`" function is called which

forces a recalibration of the sensors by writing to the Forced_CAL bit<14> in the Ambient Compensation Control register, 0x02.
A simple way to test the functionality of this code is to physically touch the sensor while power is applied to the part and the sensors are configured by the ConfigAD7147() function. When the use lifts off the sensor this software routine should recalibrate the 8-Segment Slider to the untouched value allowing the sensor to function correctly afterwards.
If an error is not detected then we process the change of interrupt mode as described previously and the 8-Segment Slider algorithm. The slider algorithm is executed calling the function "`GetNewSliderUpdate()`".

The following pseudo code describes the functionality behind the "`GetNewSliderUpdate()`"function.

```
BEGIN
    IF any stages used in the slider is activated THEN
        //Read from AD7147
        Read ADC Values
        Read Ambient Values.
    END IF
    Calculate Max Average values.
    Detect touch errors on 8-Segment Slider

    FOREACH Stage
        //Calculate a sensor response representing the distance between the
        //current value and the ambient value
        Sensor value = Absolute (Current ADC Value - Ambient Value)
    NEXT


    //Determine activation
    IF Any bit of the 8 stages is set in Stage High Limit Threshold register THEN
        Slider activated = TRUE
        Increment interrupt counter for the TAP
        IF interrupt counter > (T_MIN+T_MAX) THEN
            Clear No Touch Interrupt Counter
        END IF
    ELSE
        Slider activated = FALSE
        Reset activation variables
        Reset list box variables
        //Determine fast scroll
        IF Slider Activation Counter < T_MAX_TOUCHING THEN
            NumberOfUpdates = abs(PositionOnLiftOff - PositionOnActivation) /
                                Menu Item Resolution;
            IF (PositionOnLiftOff < PositionOnActivation)
                ScrollingDirection = UP;
            ELSE IF (PositionOnLiftOff > PositionOnActivation)
                ScrollingDirection = DOWN;
            END IF

            IF abs(PositionOnLiftOff - PositionOnActivation) > 1/4 of screen length THEN
                NumberOfUpdates = NumberOfUpdates + 5
            END IF
        ELSE
            NumberOfUpdates = 0
        END IF
```

```
        //Work out the tap
        Increment No Touch Interrupt Counter
        IF T_MIN > interrupt counter for the TAP > T_MAX AND
            No Touch Interrupt Counter > interrupt counter for the TAP THEN
                Set tapping bit for a few interrupts.
            END IF
        END IF
    END IF

    IF Slider activated = TRUE THEN
        //Calculate absolute position
        IF No touch errors were detected THEN
            Find sensor with highest response

            SELECT CASE (sensor with highest response)
                CASE 0
                    A parameter = Sensor value 1
                    B parameter = Sensor value 0 + Sensor value 1

                    Slider position = (PIXEL_RESOLUTION * A_parameter) /
                                      B_parameter
                CASE Last sensor
                    A parameter = (Last sensor value * Index of last sensor) +
                                  (Second last sensor value * Index of second last sensor)
                    B parameter = Last sensor value + Second last sensor value

                    Slider position = (PIXEL_RESOLUTION * A_parameter) /
                                      B_parameter
                CASE ELSE
                    A parameter = (Sensor with highest response – 1) *
                                  (Index of sensor with highest response-1) +
                                  (Sensor with highest response) *
                                  (Index of sensor with highest response) +
                                  (Sensor with highest response + 1) *
                                  (Index of sensor with highest response + 1)

                    B parameter = (Sensor with highest response – 1) +
                                  (Sensor with highest response) +
                                  (Sensor with highest response + 1)

                    Slider position = (PIXEL_RESOLUTION * A_parameter) /
                                      B_parameter
            END SELECT

            //Apply IIR filter to smooth slider response
            IF first interrupt where the activation is registered THEN
                Initialise IIR filter with slider position
            ELSE
                Update IIR filter with new slider position
            END IF

            //Calculate relative position
            IF it is the first interrupt since activation THEN
                Position on first touch = Scaled Down Displacement
            ELSE
                //Auto scroll
                IF Position on first touch < Item resolution AND
                   we've moved since the activation of the slider THEN
                    Auto scroll upward
                ELSE IF Position on first touch < (Number of Wanted Positions –
```

```
                        Item resolution) AND we've moved since the activation THEN
                        Auto scroll downward
                    ELSE
                        IF (abs(Position on first touch - Scaled Down Displacement)
                            > Item resolution) THEN
                            Only scroll by 1 item
                            Moved since the activation = TRUE
                        END IF
                    END IF
                END IF


                //Clear tap if we're scrolling
                IF Movement has been detected AND
                   Slider Activation Counter < T_MAX_TOUCHING THEN
                    Cancel Tap
                END IF

                //Format position data
                Slider status = Touch error
                IF Slider status = TRUE THEN
                    Slider status = Slider status OR Activation bit
                    IF Lift off detected = FALSE AND 2 finger flag=FALSE THEN
                        Slider status = Slider status OR new position
                    END IF
                ELSE
                    Slider status = Slider status with activation bit cleared
                    IF Tap detected =TRUE THEN
                        Set tap bit
                    ELSE
                        Clear tap bit
                    END IF
                    //Send UP and DOWN commands  if a fast scroll was detected
                    IF NumberOfUpdates > 0 THEN
                        IF Fast scroll update counter = 0 THEN
                            Reload fast scroll update counter
                            IF Scrolling Direction = UP THEN
                                Send UP command
                            ELSE
                                Send DOWN command
                            END IF
                            Fast scroll update counter = Fast scroll update counter - 1
                        END IF
                    END IF
                END IF
            END IF
        END IF
END
```

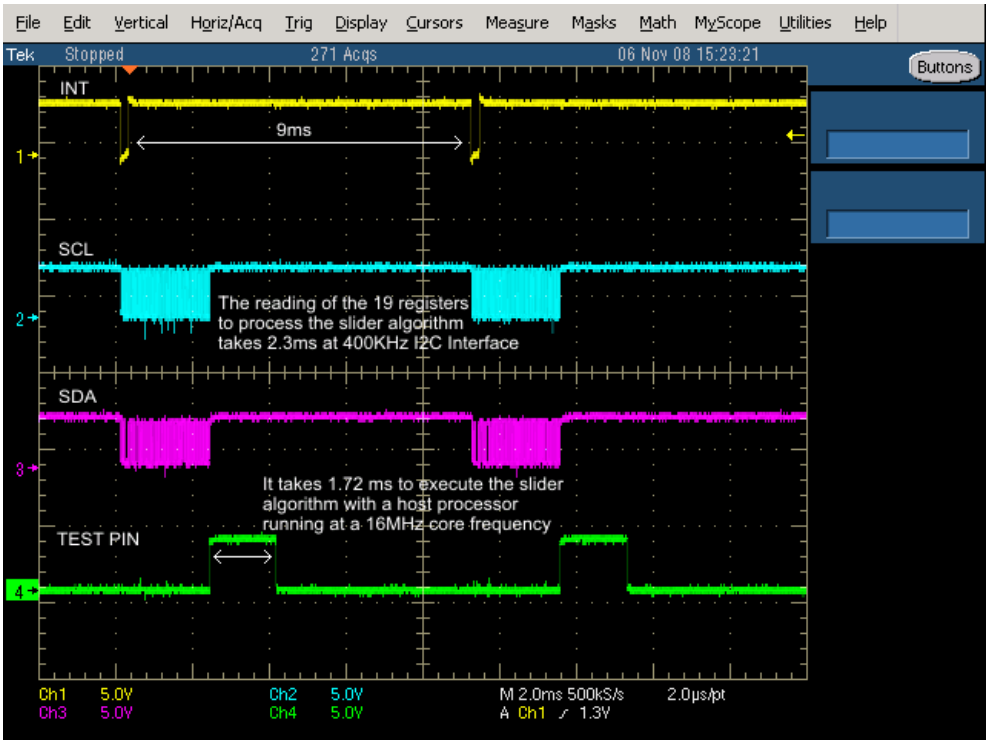


Fig.5 Communication and execution timing between the AD7147 and host processor

Explanation of the slider pseudo code

Activation, Tap and Fast Swipe

When calling the function "`GetNewSliderUpdate()`", the entire slider algorithm is executed. This function extracts on it own absolute positions data based on the CDC response of the 8 sensors making the slider.
This function returns formatted position data as well as commands to update a menu list control.

When entering this function, the firmware checks if any of the bits in the High Limit status register is set. If any is set the following registers are read from the AD7147:

- ADC values for the first 8 stages
- Ambient values for the first 8 stages

All registers are read back within 2.3ms using a 400KHz I2C interface, the algorithm to process position data takes a further 1.72ms with a host processor running at 16MHz core clock frequency.

After reading from the AD7147, the firmware computes the sensor responses.

The response of a sensor is defined by the absolute number of codes between the current CDC value and the ambient value.

$$Sensor\_value = abs(Current\_value - Ambient\_value)$$

Next, we look for touch errors. A touch error is registered when 2 fingers are detected or when a wide surface area of the sensor is in contact with user's hand. To detect 2 fingers, we look at the High Limit Interrupt Status register (at address 0x09) and search for the interrupt bits that are not contiguously set. An error registered due to a contact of a wide sensor area occurs when too many bits are set in the High Limit Interrupt Status register.

The next step in the algorithm is the registration of the activation. The activation is determined checking if any of the 8 lower bits in High Limit Interrupt Status register (0x09) is set. We consider the Slider activated if any of the 8 lower bits is set.

When the slider activation changes of state, we keep track for certain duration the number of interrupts elapsed since the change of state. We use for this two counters.
One counter keeps track of the interrupts when touching and the second keeps track of the interrupts when the sensor is untouched.

The counters used in these 2 operations are useful to determine a valid Tap event on the Slider.
When lifting off, we check if the number of interrupt when the user was touching is within certain bounds. If it is and if the number of interrupt registered when lifting off is greater that the number of interrupts when touching, then we register a valid tap.

$$Tap = (T\_MIN < TouchDownCounter < T\_MAX) AND (NoTouchCounter > TouchDownCounter)$$

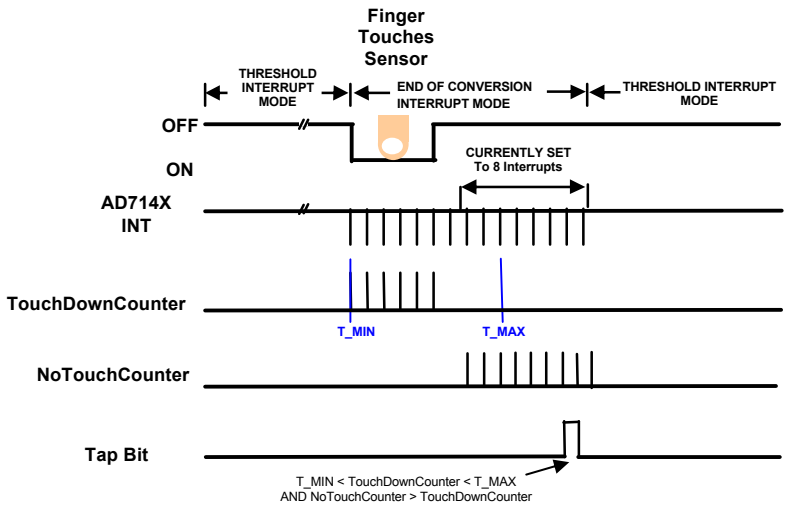


Fig.6 Tap timing

When the user lifts off the sensor, the next event the firmware tries to catch is a fast swipe. A fast swipe is detected when the user scrolls from one point to another within a certain time. If a fast swipe is detected, the firmware issues "UP" and "DOWN" commands that can be used by a higher level application to control a menu list.

As soon as the user touches, the firmware keeps a copy of the first position registered on activation. When lifting off, we check if the fast swipe took place within a 450ms window. If it did we calculate the distance swiped taking the absolute difference between the last position and the first position registered on activation. This distance is then divided by the resolution of 1 item of the menu list.
The resolution of 1 item corresponds to the number of positions required to swipe to select the next item in the menu list.
This method gives to the high level application the number of update to perform in the menu list depending on the distance swiped.
If the distance is too small, a minimal number of updates is set to 9. This setting can be changed altering the definition named "`MIN_NUMBER_OF_UPDATES`" in the file "`AD7147 - Slider Definitions.h`".
If the user swiped over a quarter of the length of the slider, then the number of updates is incremented by 5.

To know which way execute the updates, we must determine the direction of the swipe. If the last position is greater than the position on activation then we can be sure the user scrolled downwards and vice versa.

The updates of the menu list are taking place when the user lifts off the sensor. The updates are done inserting UP and Go DOWN commands in the slider status value returned from the function "`GetNewSliderUpdate()`".

As mentioned before, when the user lifts off the sensor, the firmware configures the AD7147 back to threshold interrupt mode. To make sure that this does not happen before all the updates have taken place, we alter the value of the interrupt counter dedicated to this task.

This is the code implementing this functionality:

```
//Check if there will be enough interrupts after lifting off before to switch to
//threshold mode.
MinimalNumberOfInterruptsAfterLiftingOff = NumberOfUpdates * LISTBOX_QUICK_UPDATE;
if (MinimalNumberOfInterruptsAfterLiftingOff > NUMBER_OF_INTS_BEFORE_THRES_INT_MODE)
    InterruptCounterForThresIntMode = MinimalNumberOfInterruptsAfterLiftingOff;
```

All the above calculation used to determine a fast swipe are only execute once, on the first interrupt after the user lifts off.

Absolute Position Calculation

The next big step of the algorithm is to calculate the absolute position based on the sensor responses.

The first step is to find out which sensor has the highest response. Then the following formulae can be applied

$$A = \sum_{i=(Index\_of\_peak\_Sensor-1)}^{i=(Index\_of\_peak\_Sensor+1)} Sensor\_response(i) \times i$$

$$B = \sum_{i=(Index\_of\_peak\_Sensor-1)}^{i=(Index\_of\_peak\_Sensor+1)} Sensor\_response(i)$$

$$Absolute\_Position = \frac{Touch\_resolution}{Number\_of\_Sensors - 1} \times \frac{A}{B}$$

The "A" and "B" formulae use the sensor with the highest response and the two adjacent sensor responses in the calculation. However, when the sensor with the highest response is the first one or the last one, only 1 adjacent sensor is used in the calculations.

In the integration code, we used a constant called "*PIXEL_RESOLUTION*" is defined by:

$$PIXEL\_RESOLUTION = \frac{Touch\_resolution}{Number\_of\_sensors - 1}$$

E.g: We want to achieve 128 positions with an 8 segment slider sensor. Therefore:

$$PIXEL\_RESOLUTION = \frac{Touch\_resolution}{Number\_of\_sensors - 1} = \frac{128}{8-1} = \frac{128}{7} = 18$$

Next, we must smooth the response of the slider applying an IIR filter to the position data calculated with the above formulae.

This is the formula of the IIR filter:

$$Average\_position = \frac{Average\_position \times 6 + New\_slider\_position \times 4}{10}$$

The average position is then inserted in the returned variable of the function "`GetNewSliderUpdate()`".

Relative position calculation

In order to create commands to update a list box in user's application, we must calculate the relative position. The calculation of the relative position is done by recording the absolute position on the first touch and then subtracting it from the latest absolute position.

When the relative position is greater than the item resolution in the list box (defined as `DISPLAY_ITEMS_CONSTANT` in the code), the position on the first touched is re-initialised to the current position and at the same time an update in the list box is done. If the relative position is negative then the firmware will generate a GO UP command. If the relative position is positive, the firmware will generate a GO DOWN command.
As soon as 1 update is made to the list box, we set a flag that indicates that user's finger moved since he/she touched the sensor.

To provide extra functionality ADI added auto scroll functionality.
This auto scroll enables the user to go through different items in a list box without having to scroll.
The auto scroll is enabled if the user has moved by at least 1 item in the list box and his/her finger is at the top or at the bottom of the slider.
If the user touches any of the extremities of the slider on activation and remains static, then the auto-scroll will enable after 1 second. This time is defined by the constant "`LISTBOX_SLOW_UPDATE`" in the file "`AD7147 - Slider Definitions.h`". It is set to 1000 by default.

Configuration of the firmware

All the constants used to tune the firmware have been described in the present document. All of them are located in the file "`AD7147 - Slider Definitions.h`".

The table below shows the data format returned by the "SliderStatus" variable.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Slider Status	Activation	Tap	Go UP	Go DOWN	Touch errors		Absolute positions ranging from 0 to127									

Table 2 – 8-Segment Slider Status

Code and data memory requirements

The 8-Segment Slider firmware requires 5.49Kb of program memory and 195 bytes of RAM. These are the requirements to implement the AD7147 register configuration, recalibration and 8-Segment Slider algorithm processing on a host controller. These code sizes do not include any software I2C or SPI

driver; it assumes the host processor will have a dedicated hardware driver. As an example to add a software I2C driver requires an additional 726 bytes of code memory and 16 bytes of data memory.