Freescale Semiconductor
Application Note

Document Number: AN4529
Rev. 0, 5/2012

Using ADC Built In Self Test in the MPC5675K Microcontroller

by: Curtis Hiller, Motoki Sakurai, and Shanaka Yapa
Automotive and Industrial Solutions Group

1 Introduction

The MPC5675K microcontroller supports run-time available analog-to-digital converter (ADC) hardware built in self-test to verify the operation of ADC analog components. The ADC self-test feature supports the testing of capacitive, resistive, and power supply integrity. The ADC self-test includes:

- S Algorithm to verify in-range / out-of-range for ADC supply voltage (VDD_HV_ADV) and reference voltage (VDD_HV_ADR)
- C Algorithm to check for opens or shorts in the capacitive network used for the internal digital-to-analog converter (DAC)
- RC Algorithm to check for opens or shorts in the resistive network used for the internal DAC

This application note details supplemental information required to operate the ADC self-test feature. The application note shows the structure and detail of thresholds programmed into test flash. Two use case samples are shown to help users understand how to program the ADC self-test feature.

Contents

1 Introduction....................1
2 ADC self-test feature description....................2
3 ADC self-test parameters....................3
4 Considerations for software-based comparison of converted data to high / low thresholds....................4
4.1 S1 Algorithm: Logical checking of integer data and fractional data method....................4
4.2 S1 Algorithm: Calculated voltage comparison method....................5
5 Sample code A: ADC self-test using Scan mode....................6
6 Sample code B: Single-Shot ADC self-test....................9

2 ADC self-test feature description

For safety devices used in very critical applications, it is important to check at regular intervals that the ADC is functioning correctly. For this purpose, the self-testing feature has been incorporated inside the ADC. The self-tests use analog watchdogs to verify the result of self-test conversions. The threshold of these watchdogs is saved in the test flash. Before running the self-test, the user must copy these values from the test flash to the STAWxR registers.

Three types of self-testing algorithms have been implemented inside the ADC:

- **Supply self-test—Algorithm S:** It includes the conversion of the ADC internal bandgap voltage, ADC supply voltage, and ADC reference voltage. It includes a sequence of three test conversions (steps). The supply test conversions must be an atomic operation with no functional conversions interleaved.
- **Resistive-capacitive self-test—Algorithm RC:** It includes a sequence of 19 test conversions (steps) by setting the ADC internal resistive digital-to-analog converter (DAC).
- **Capacitive self-test: Algorithm C:** It includes a sequence of 17 test conversions (steps) by setting the capacitive elements comprising the sampling capacitor/ capacitive DAC.

The ADC performs the following functions:

- Implements an additional test channel dedicated for self-testing
- Provides signals to schedule self-testing algorithms using configuration registers
- Monitors the converted data using analog watchdog registers, and flags the error to the Fault Collection and Control Unit (FCCU) in case any of the algorithms fails.

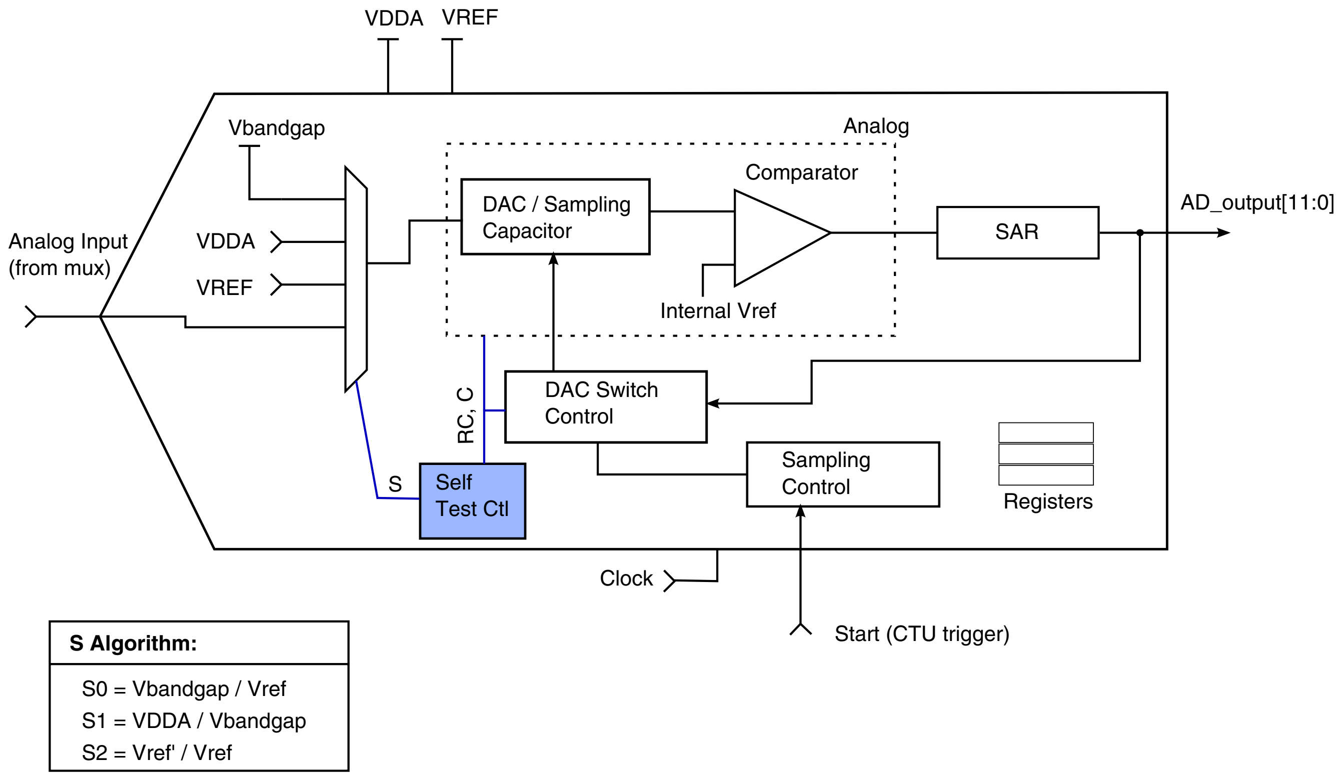

Figure 1. ADC block diagram with self-test feature

3 ADC self-test parameters

The MPC5675K uses two types of parameter settings to define the ADC self-test operation:

- Sample phase duration settings programmed into the INPSAMP_S, INPSAMP_RC, and INPSAMP_C fields of the Self-Test Configuration Register (STCR1).
- Threshold values for analog watchdog algorithms

Sample phase duration settings define the amount of time required during the ADC sample phase. Each algorithm (C, RC, S) has a dedicated register field to define the sample phase duration. Recommended settings are shown in the table below:

Table 1. Sample phase (INPSAMP_x) settings

Register field	Recommended setting
INPSAMP_C	18h
INPSAMP_RC	60h
INPSAMP_S	FFh[1]

1. Recommended setting is the maximum value of this register field due to slow sample capacitor settling time at low temperature for S0 algorithm.

Assuming that ADC_ckfreq = 60 MHz, the sample time can be calculated using the following formula:

$$\text{Sample time} = (\text{INPSAMP\_}n-1)/\text{ADC\_ckfreq}$$

$$\text{Sample time} = (0\text{xFF}-1)/60\ \text{MHz}$$

$$\text{Sample time} = 4.233\ \mu\text{s}$$

Threshold values for analog watchdog algorithms are stored in test flash and retrieved by the user application at configuration time and loaded into ADC Self Test registers. The following table defines the test flash mapping to the ADC Self-Test registers:

Table 2. Sample test flash values for ADC self-test thresholds

Word	Flash location offset	Value in flash	Loads to STAWxR	Step	THRH	THRL	THRH	THRL
W1	10h	FE20F1E0h	STAW3R	RC	E20h	1E0h	3616d	480d
W2	14h	F856F732h	STAW4R	C0	856h	732h	2134d	1842d
W3	18h	F873F732h	STAW5R	C1-C17	873h	732h	2163d	1842d
W4	1Ch	F75AF4DFh	STAW0R	S0_3.3V	75Ah	4DFh	1882d	1247d
W5	20h	F4D0F2DBh	STAW0R	S0_5.0V	4D0h	2DBh	1232d	731d
W6	24h	F003F002h	STAW1AR	S1(INT)	3h	2h	3d	2d
W7	28h	F3D9F1E3h	STAW1BR	S1(FRAC)	3D9h	1E3h	985d	483d
W8	2Ch	FFFFFFF9h	STAW2R	S2	FFFh	FF9h	4095d	4089d

4 Considerations for software-based comparison of converted data to high / low thresholds

It is possible to implement software methods comparing the ADC Self Test converted results in STDR1.TCDATA, STDR2.IDATA, and STDR2.FDATA to the high threshold (THRH) and low threshold (THRL) values stored in Test Flash. Software-based comparison methods allow the users to average two or more ADC converted data results; which can reduce the effects of system, ground, and/or power supply noise.

- Supply algorithm step 0 (S0), Supply algorithm step 2 (S2), all Resistive-Capacitive algorithm steps (RCn), and all Capacitive algorithm steps (Cn) can be simply averaged using the results present in STDR1.TCDATA after every conversion.
- For Supply algorithm step 1 (S1), since both the integer data (STDR2.IDATA) and fractional data (STDR2.FDATA) are available for each conversion, the user software must account for the integer and fractional part of the conversions.

There are two possible ways to account for S1 integer and fractional parts:

- A logical comparison of IDATA and FDATA results against THRH and THRL
- Calculated voltage comparison

These two methods are detailed in the following sections.

4.1 S1 Algorithm: Logical checking of integer data and fractional data method

The user software can implement a logical, step-by-step checking of IDATA and FDATA results against THRH and THRL. Since the S1 results are represented by both an integer part (IDATA) and a fractional part (FDATA), it is necessary to consider a flow chart based approach to compare the S1 results against THRH and THRL.

Figure 2 illustrates the logical requirements for comparing IDATA and FDATA results to THRH / THRL.

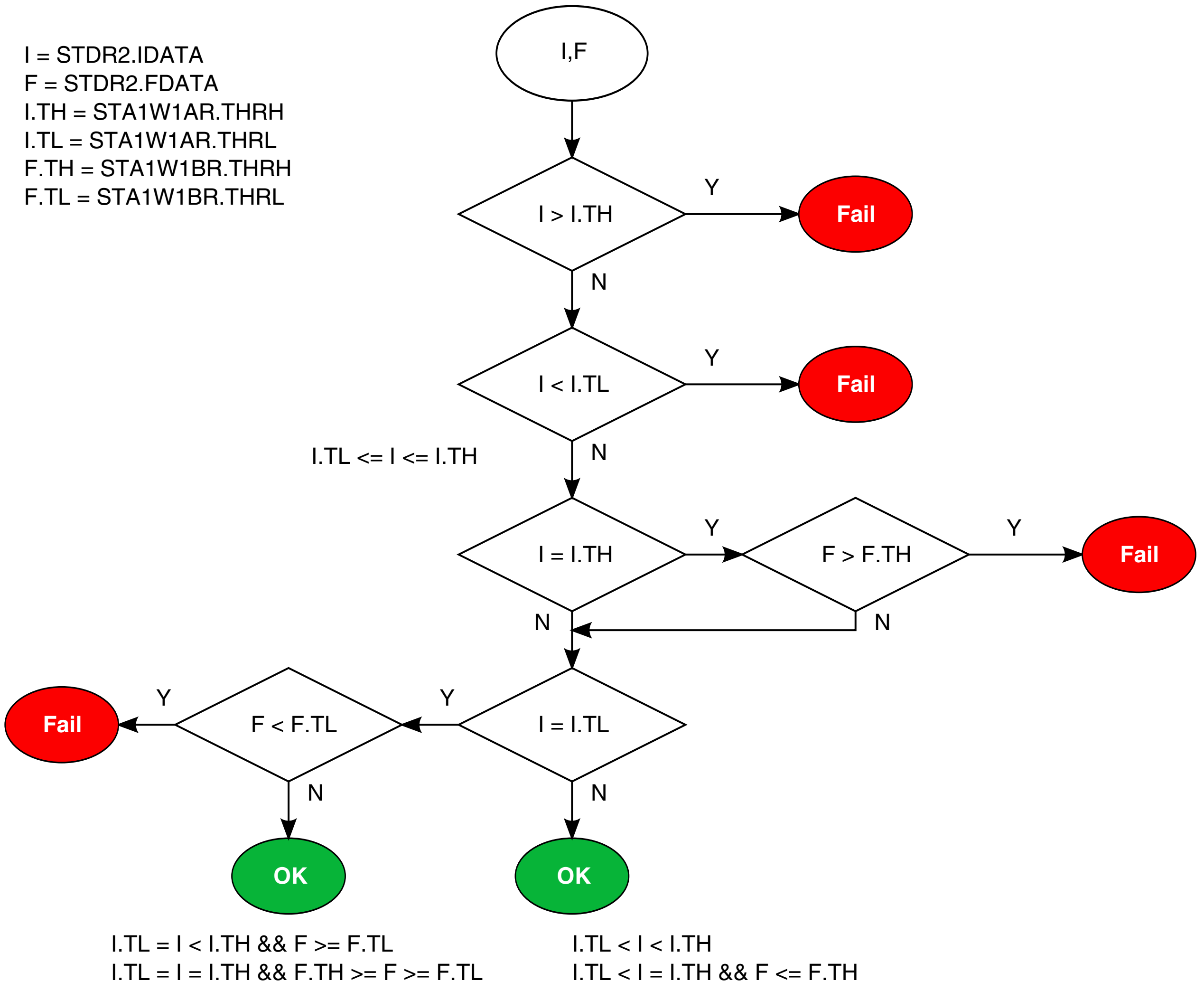

Figure 2. Flow chart showing IDATA and FDATA comparison checking against THRH and THRL values

4.2 S1 Algorithm: Calculated voltage comparison method

The user software can implement a method of calculating S1 algorithm IDATA, FDATA, the THRH values, and the THRL values to voltages, and then perform a comparison of the calculated voltage to the calculated THRH voltage and calculated THRL voltage.

The steps involved in this method are as follows:

1. Calculate the high threshold voltage:

$$\left(\mathrm{STA1}W\mathrm{1AR.THRH} + \left(\mathrm{STA1}W\mathrm{1BR.THRH}/_{4096}\right)\right) * V_{\mathrm{bandgap}}$$

2. Calculate the low threshold voltage:

$$\left(\mathrm{STA1}W\mathrm{1AR.THRL} + \left(\mathrm{STA1}W\mathrm{1BR.THRL}/_{4096}\right)\right) * V_{\mathrm{bandgap}}$$

3. Calculate the converted results to voltage:

$$\left(\text{STDR2.IDATA} + \left(\text{STDR2.FDATA}/4096\right)\right) * V_{\text{bandgap}}$$

4. Compare result from step 3 to THRH and THRL. S1 algorithm passes if low threshold voltage ≤ converted voltage ≤ high threshold voltage.

5 Sample code A: ADC self-test using Scan mode

Sample code A shows a typical usage for setting both the analog watchdogs which monitor various capacitors (open / short), resistors (open / short), and power supplies (in range / out of range) and the ADC watchdog timers (ADC conversions finished within watchdog time).

The following sample software is divided into three major sections:

- Setup ADC_0: It programs the ADC clock prescaler, self-test algorithm (S + RC + C), sample settings, self-test enable, and self-test mode (Single-Shot or Scan mode). Additionally, the code in this section reads the self-test threshold values from test flash and copies those values to the self-test threshold registers
- Start ADC_0 self-test: It enables analog watchdogs and watchdog timers and starts the conversion process by setting MCR[NSTART] = 1.
- Stop ADC_0 self-test: It disables analog watchdogs, and watchdog timers, and stops conversions by clearing MCR[NSTART].

```
/************************************************************************/
/* Sample project for MPC5675K (Komodo)                                 */
/* 2012 March 12                                                        */
/************************************************************************/

/*  **********************************************************************************/
/*  *  THIS FILE IS ONLY INTENDED AS AN EXAMPLE CODE FOR THE                        */
/*  *  DEVICES AND HAS ONLY BEEN GIVEN A MINIMUM LEVEL OF TEST.                     */
/*  *  IT IS PROVIDED 'AS IS' WITH NO GUARANTEES AND NO PROMISE                     */
/*  *  OF SUPPORT.  USE AT YOUR OWN RISK !!                                         */
/*
**********************************************************************************/
#include "mpc5675k-2.02.h"
#include "typedefs.h"

/**********************************************************************************/
/*                              Function Definitions                              */
/**********************************************************************************/
void latest_ADC0_self_test(void);
/**********************************************************************************/
/*                                   Constants                                    */
/**********************************************************************************/

/**********************************************************************************/
/*                                Global Variables                                */
/**********************************************************************************/
  uint32_t adc_status1[4];
  uint32_t adc_status2[4];
  uint32_t adc_status3[4];

/**********************************************************************************/
/* Function : Main                                                                */
/**********************************************************************************/
int main0(void) {
  volatile int i = 0;
  volatile int k = 0;
  volatile int j = 0;

  RGM.FES.R = 0xffff;   // clear FCCU interrupts

  init_me();            /* mode entry */
  setup_fmpll0();       /* configure clock */
  setup_fmpll1();       /* configure clock */
  setup_peri_clk();     /* configure peripherals */
  init_run_mode();      /* enter run0 mode */

  ADC_0.MCR.B. PWDN = 0;// exit from power down state (offset cancellation is performed)
enable in early stage
  SIU.PCR[148].R = 0x0220;                          // Enable LED1
  SIU.PCR[150].R = 0x0220;                          // Enable LED2
  SIU.PCR[0].R = 0x0200;                            // J67 1
  SIU.PCR[1].R = 0x0200;                            // J67 5

  SIU.GPDO[148].R = 1;
  for (i=0 ; i< 500 ; i++)
  SIU.GPDO[150].R ^= 1;
  SIU.GPDO[0].R = 0;
  SIU.GPDO[1].R = 0;


  latest_ADC0_self_test(); // Final fixed test tested on 2012/02/17

  for (;;)
  {
//    if (i >= 6000) //~100 usec
    if (i >= 12000) //~200 usec
    {
      SIU.GPDO[148].R  ^= 1; // toggle led1
      i = 0;
    }
    else
    {
       i++;
    }
  }

}


//==========================================================================//
//    ADC0-self test with workaround for Self-test SCAN mode                //
//    User must power up the ADC before using this function                 //
//==========================================================================//
void latest_ADC0_self_test(void)
{
// start test
  int i;
  //--------------------------------------------------------------------
  //----------- ADC self test for one-shot S algo ----------------------
  //----------- Workaround for ERR003866 -------------------------------
  //--------------------------------------------------------------------
  ADC_0.MCR.B.MODE      = 0;      // One shot
  ADC_0.STSR1.B.ST_EOC = 0x1;     // clear end of conversion flag
  ADC_0.STCR1.R       = 0x1860FF07;   // Self test Sampling settings for C_RC_S
  ADC_0.STCR3.R       = 0x00000000;   // Self test S algo for Single shot and MSTEP=0
  ADC_0.STCR2.R       = 0x00000080;   // enable Selftest
  ADC_0.STBRR.R       = 0x00000000;   // BR=0  WDT=0.1ms

  ADC_0.STAW0R.R      = 0x0fff0fff & (*(uint32_t *)0x00400020);    // S step0
  ADC_0.STAW1AR.R     = 0x0fff0fff & (*(uint32_t *)0x00400024);    // S step1
  ADC_0.STAW1BR.R     = 0x0fff0fff & (*(uint32_t *)0x00400028);    // S step1
  ADC_0.STAW2R.R      = 0x00000fff & (*(uint32_t *)0x0040002C);    // S step2

  ADC_0.STAW3R.R      = 0x0fff0fff & (*(uint32_t *)0x00400010);    // RC
  ADC_0.STAW4R.R      = 0x0fff0fff & (*(uint32_t *)0x00400014);    // C
  ADC_0.STAW5R.R      = 0x0fff0fff & (*(uint32_t *)0x00400018);    // C step 1 to CS-1

  ADC_0.MCR.B.NSTART    = 1;            // Start the ADC trigger

  while(ADC_0.STSR1.B.ST_EOC == 0){;}   // wait for end of conversion flag
  ADC_0.STSR1.B.ST_EOC = 0x1;           // clear end of conversion flag
  ADC_0.STCR2.B.EN      = 0;            // disable Selftest
  //----------------------------------------------------------------------
  //--------- ADC self test for SCAN mode-------------------------------
  //----------------------------------------------------------------------
  ADC_0.MCR.B.MODE      = 1;            // SCAN mode
  ADC_0.STCR1.R         = 0x1860FF07;   // Self test Sampling for C_RC_S
  ADC_0.STCR3.R         = 0x00000300;   // Self test Set S+RC+C for SCAN and MSTEP=0
  ADC_0.STCR2.R         = 0x00000080;   // enable Selftest
  ADC_0.STBRR.R         = 0x00000000;   // BR=0  WDT=0.1ms


  ADC_0.STAW0R.B.AWDE   = 1;    // S Analog WDT enable
  ADC_0.STAW1AR.B.AWDE  = 1;    // S Analog WDT enable
  ADC_0.STAW2R.B.AWDE   = 1;    // S Analog WDT enable
  ADC_0.STAW3R.B.AWDE   = 1;    // RC Analog WDT enable
  ADC_0.STAW4R.B.AWDE   = 1;    // C Analog WDT enable

  // disable interrupts before x.WDTE = 1
  ADC_0.STAW0R.B.WDTE   = 1;    // S WDT enable
  ADC_0.STAW3R.B.WDTE   = 1;    // RC WDT enable
  ADC_0.STAW4R.B.WDTE   = 1;    // C WDT enable

  // WARNING : watchdog timer has already started at WDTE = 1
  // (ADC self test execution time is also included into the time out)

  ADC_0.MCR.B.NSTART    = 1;            // Start the ADC0 self test trigger
  // enable interrupts after NSTART=1

  // wait loop for desired period
  for(i=0 ; i <180000 ;i++);

  ADC_0.STAW0R.B.AWDE   = 0;    // S Analog WDT disable
  ADC_0.STAW1AR.B.AWDE  = 0;    // S Analog WDT disable
  ADC_0.STAW2R.B.AWDE   = 0;    // S Analog WDT disable
  ADC_0.STAW3R.B.AWDE   = 0;    // RC Analog WDT disable
  ADC_0.STAW4R.B.AWDE   = 0;    // C Analog WDT disable

  ADC_0.STAW0R.B.WDTE   = 0;    // S WDT disable
  ADC_0.STAW3R.B.WDTE   = 0;    // RC WDT disable
  ADC_0.STAW4R.B.WDTE   = 0;    // C WDT disable
  ADC_0.MCR.B.NSTART    = 0;    // Stop the ADC trigger
  ADC_0.STCR2.B.EN      = 0;    // disable Selftest

//----------------------------------------------------------------------
//------------- end of test ---------------------------------------
//----------------------------------------------------------------------
}
```

The users might find it beneficial to add software for storing ADC self-test conversions to SRAM for troubleshooting purposes. The following C code stores S algorithm results (Step 0, Step 1, and Step 2) to SRAM.

```
//----------------------------------------------------------------------
//------ Read results into an array starting at 'start' --------------
//----------------------------------------------------------------------
  start = 0x40001000;
  pt = (int *)(start);  //set pointer to start.

  for(k = 1; k < 16; k++)
  {
    while(ADC_0.STSR1.B.ST_EOC == 0){;}  // wait for end of conversion flag
    *pt++ = ADC_0.STDR1.B.TCDATA;        // record conversion result S0
    ADC_0.STSR1.B.ST_EOC = 0x1;          // clear end of conversion flag

    while(ADC_0.STSR1.B.ST_EOC == 0){;} // wait for end of conversion flag
    *pt = ADC_0.STDR1.B.TCDATA;     // record conversion result, allow for overwrite
    *pt++ = ADC_0.STDR2.B.IDATA;    // record integer data S1
    *pt++ = ADC_0.STDR2.B.FDATA;    // record fractional data S1
    ADC_0.STSR1.B.ST_EOC = 0x1;     // clear end of conversion flag
    while(ADC_0.STSR1.B.ST_EOC == 0){;} // wait for end of conversion flag
    ADC_0.STSR1.B.ST_EOC = 0x1;         // clear end of conversion flag

    while(ADC_0.STSR1.B.ST_EOC == 0){;} // wait for end of conversion flag
    *pt++ = ADC_0.STDR1.B.TCDATA;       // record conversion result  S2
    ADC_0.STSR1.B.ST_EOC = 0x1;         // clear end of conversion flag

    // read RC and C algorithm results, write to SRAM but don't incr ptr
    for(j = 1; j < 36; j++)
    {
      while(ADC_0.STSR1.B.ST_EOC == 0){;} // wait for end of conversion flag
      *pt = ADC_0.STDR1.B.TCDATA;         // record result, allow for overwrite data
      ADC_0.STSR1.B.ST_EOC = 0x1;         // clear end of conversion flag
    }
  }
```

The following table shows results stored in MPC5675K SRAM for S0, S1, and S2.

Table 3. Results of ADC self-test S algorithm

SRAM address	S0 result	S1 IDATA result	S1 FDATA result	S2 result
SD : 40001000h	000003E0h	00000002h	00000B00h	00000FFFh
SD : 40001010h	000003DDh	00000002h	00000B1Dh	00000FFFh
SD : 40001020h	000003DAh	00000002h	00000B47h	00000FFFh
SD : 40001030h	000003DCh	00000002h	00000B28h	00000FFFh
...	...	...	...	...
SD : 400010D0h	000003DEh	00000002h	00000B1Ah	00000FFFh
SD : 400010E0h	000003DDh	00000002h	00000B21h	00000FFFh

Assuming that VDD_HV_ADV = 3.3 V, and VDD_HV_ADR = 5.0 V, it is possible to convert the results into voltages using the following equations.

For S0 results, the following equation applies:

$$V_{\text{bandgap}} = \left({}^{S0\_\text{result}}\!/_{\text{ADC\_full\_scale}}\right) * \text{ADR}$$

$$V_{\text{bandgap}} = \left({}^{3E0h}\!/_{\text{FFFh}}\right) * 5.0V$$

V_{bandgap} = 1.217 V. This result is expected, nominal V_{bandgap} = 1.2 V)

For S1 results, the following equation applies:

$$\text{ADV} = \left(\text{IDATA} + \left({}^{\text{FDATA}}\!/_{\text{ADC\_full\_scale}}\right)\right) * V_{\text{bandgap}}$$

$$\text{ADV} = \left(2 + \left({}^{B00h}\!/_{\text{FFFh}}\right)\right) * 1.217\ V$$

ADV = 3.27 V. This result is expected since ADV applied to the MPC5675K is 3.3 V.

6 Sample code B: Single-Shot ADC self-test

Sample code B can be used to run multiple iterations of the ADC self-test algorithms and store the results of each algorithm's conversions into an SRAM array. The sample code is designed for the single-shot operation.

Sample code B: Single-Shot ADC self-test

The sample code depends on the following pre-conditions:

Preconditions: The sample code requires that ADC self-test parameters have been read from the test flash and written to the threshold registers (see Sample code A: ADC self-test using Scan mode)

This sample code is composed of the following major items:

- Disable normal conversions.
- Set ADC mode to One Shot by setting MCR[MODE] = 0.
- Set sample time by programming STCR1[INPSAMP_n] fields (n = S, RC, C).
- Enable desired algorithm (S, RC, or C).
- Cycle through the steps for the desired algorithm
    a. Set conversion step.
    b. Execute conversion by setting MCR[NSTART] = 1.
    c. Wait for the End Of Conversion flag (EOC).
    d. Read result and store to SRAM.

```
 /*********************************************************************************
FUNCTION          : AdcSelfTestSequence1( ) - sets ADC for Self Test
INPUTS NOTES      : start - address to store results
                    total_conv_per_step - number of iterations to perform
                    adc - number of the ADC (0..3)
                    alg - algorithm (0=S, 1=RC, 2 = C)
                    total_steps - total steps for the algorithm
RETURNS NOTES     :
GENERAL NOTES     : Run a number of ADC Self Test algorithms and store results.
                  : Before running this test, the user must run one step of
                  : the S algorithm with AWD disabled (AWDE = 0).  This is to account
                  : for ERR003866, it allows better settling time for S0.  The dummy
                  : S0 step must be executed immediately before the normal S0 step.

*********************************************************************************/

void AdcSelfTestSequence1 (UINT32 start, UINT32 total_conv_per_step, UINT8 adc, UINT8 alg,
UINT8 total_steps)
{
  UINT32 i, dly, lp;
  UINT16 *pt;
  volatile struct ADC_struct_tag *adcp;
  UINT32 data=0;
  static UINT16 v=1;
  UINT8 current_step;

  if (adc==0) adcp = &ADC_0;
  else if (adc==1) adcp = &ADC_1;
  else if (adc==2) adcp = &ADC_2;
  else adcp = &ADC_3;

// disable all normal conversions. ALG S requires back2back conversions of steps
  if (alg==ALG_S) adcp->NCMR[0].R = 0x0;

// enable normal conversion of ANA1. ALG S requires back2back conversions of steps
  else adcp->NCMR[0].R = 0x2;


  adcp->MCR.B.CTUEN = 0; // disable CTU
  adcp->MCR.B.MODE = 0; // to select one shot mode
  adcp->MCR.B.OWREN=1; // over write enable

  adcp->STCR1.B.INPSAMP_RC=inpsamp_rc; //sampling time for the RC algorithm
  adcp->STCR1.B.INPSAMP_C=inpsamp_c; //sampling time for the C algorithm
  adcp->STCR1.B.INPSAMP_S=inpsamp_s; //sampling time for the S algorithm

  adcp->STCR3.B.ALG = alg; // enable desired algorithm

  pt = (UINT16 *)(start);
  for(i=0;i<total_conv_per_step;i++)
  {
    for(current_step=0; current_step<total_steps; current_step++)
    {
      adcp->STCR3.B.MSTEP= current_step;
      adcp->STCR2.B.EN=1;//Self testing channel enable.

      adcp->ISR.R = 0x3; // clear EOC and ECH
      adcp->STSR1.B.ST_EOC = 0x1; //clear end of self test channel conversion flag
      adcp->MCR.B.NSTART = 1;// start conversions
      while(adcp->ISR.B.ECH == 0){;} // wait for end of chain
      while(adcp->STSR1.B.ST_EOC == 0){;} // wait for end of self test conv flag
      *pt++ = adcp->STDR1.B.TCDATA; // record conversion result
      if (alg!=ALG_S)
      {
        *pt++ = adcp->CDR1.B.CDATA; // record conversion result from 1
      }
      if ((alg==ALG_S)&&(current_step==1))
      {
        *pt++ = adcp->STDR2.B.IDATA; // record integer data
        *pt++ = adcp->STDR2.B.FDATA; // record fractional data
      }
    }

  }

}   /* end of AdcSelfTestSequence1 */
```

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4529
Rev. 0, 5/2012