The ADuC812 MicroConverter® as an IEEE 1451.2 Compatible Smart Transducer Interface.

The application note outlines the implementation of an IEEE 1451.2 compatible interface on the Analog Devices ADuC812 MicroConverter.
It will specify how to create a true minimal IEEE 1451.2 standard implementation, it will logically introduce you to the development of modular, scaleable and readable code, and it will outline how this code may be modified to adjust itself to end user to requirements.

1. Introduction

1.1 What is the IEEE 1451.2 Standard?

In it's simplest terms, the 1451.2 smart sensor standard specifies a 'plug-and-play' capability in a transducer module, which is achieved through an 'electronic data sheet'. It specifies a digital interface to access this data sheet, read sensor data and set actuators. A set of read and write logic functions to access the 'transducer electronic data sheet' and transducers are defined.
The standard attempts to reduce the complexities designers have traditionally faced in establishing communications between various networks and transducers. The ultimate goal of the standard is to provide an industry standard interface to efficiently connect transducers to µCs and to connect µCs to networks.

Terminology that is used when talking about 1451 include:
XDCR – An abbreviation of 'transducer', which is a sensor or an actuator.
STIM – "Smart Transducer Interface Module" [figure 1]
A STIM can range in complexity from a single-channel sensor or actuator to many channels of transducers. A transducer channel is denoted 'smart' in this context because :
- It is described by a machine-readable "Transducer Electronic Data Sheet"
- The control and data associated with the channel are digital.
- Triggering, status and control are provided to support the proper functioning of the channel.

NCAP – "Network Capable Application Processor"
The NCAP mediates between the STIM and a digital network, and may provide local intelligence. The STIM communicates to the network transparently via the "Transducer Independent Interface" that links it to the NCAP.

TII – "Transducer Independent Interface"
The TII is simply a 10-wire serial I/O bus that also defines:
- A triggering function that triggers reading/writing from/to a transducer.
- A bit transfer methodology.
- A byte-write data-transport protocol (NCAP to STIM).
- A byte-read data-transport protocol (STIM to NCAP).
- Data transport frames.

TEDS – "Transducer Electronic Data Sheet"

® MicroConverter is a registered trademark of Analog Devices, Inc.

The TEDS is a data sheet written in electronic format that describes the STIM and the transducers associated with it. The TEDS must remain with the STIM for the duration of the STIM's lifetime.

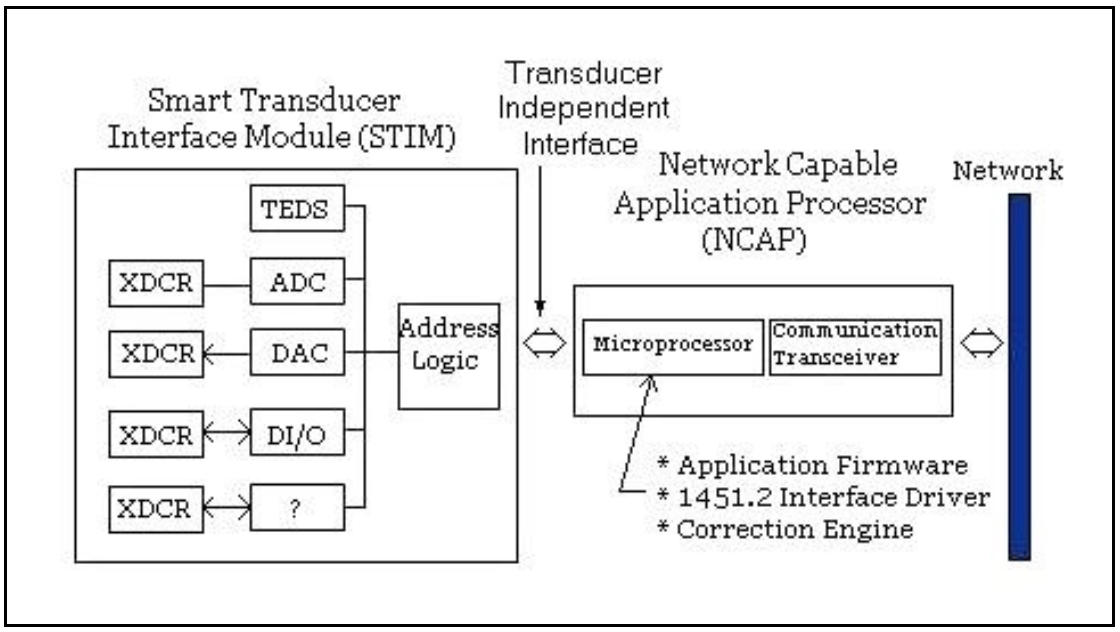

Figure 1. Overview of a STIM and how it associates through the TII to an NCAP and then onto a Network

1.2 The 1451.2 Software Issues

Overall, the STIM contains the TEDS, the control and status registers, the transducer channels, interrupt masks, address and function decoding logic, data transport handling functions, trigger and trigger acknowledge functions for the digital interface to the TII, a TII 'driver' and a transducer interface.

The TII contains data-transport, clock, triggering and acknowledge lines.

The TEDS is the electronic data sheet. Each STIM will have a TEDS, which may be broken down into 8 different sections, which are:

1. **Meta TEDS** *[mandatory]*
- Makes available at the interface all the information needed to gain access to any channel
- Contains information common to all channels
- Information is constant and read-only

2. **Channel TEDS** *[mandatory]*
- Makes available at the interface all the information concerning the channel being addressed to enable proper operation of that channel
- Information is constant and read-only

3. **Calibration TEDS** *[optional]*
- Makes available at the interface all of the information used by the correction engine in connection with the channel being addressed

- Information may be configured to be read and write capable, or it may be configured as read-only

4. **Meta-Identification TEDS** *[optional]*
- Makes available at the interface the information needed to identify the STIM
- Contains any identification information common to all channels
- Information is constant and read-only

5. **Channel-Identification TEDS** *[optional]*
- Makes available at the interface all of the information needed to identify the channel being addressed
- Information is constant and read-only

6. **Calibration-Identification TEDS** *[optional]*
- Makes available at the interface the information describing the calibration of the STIM
- Information may be configured to be read and write capable, or it may be configured as read-only (it must be the same as for the 'Calibration TEDS').

7. **End Users' Application-Specific TEDS** *[optional]*
- Contains end-users' writable application-specific data
- Information is non-volatile

8. **Industry Extensions TEDS** *[optional]*
- The function of the extension TEDS, the appropriate functional and channel address range where it may reside, and the meaning and type of the data fields will be defined by the creator of the extension.

The software implementation for the 1451.2 standard may be represented as in the diagram in figure 2. It breaks down into five main partitions:

1. the STIM Control and Channel Data block,
2. the STIM transducer interface,
3. the TII block,
4. the TEDS block, and
5. the Address and Function block

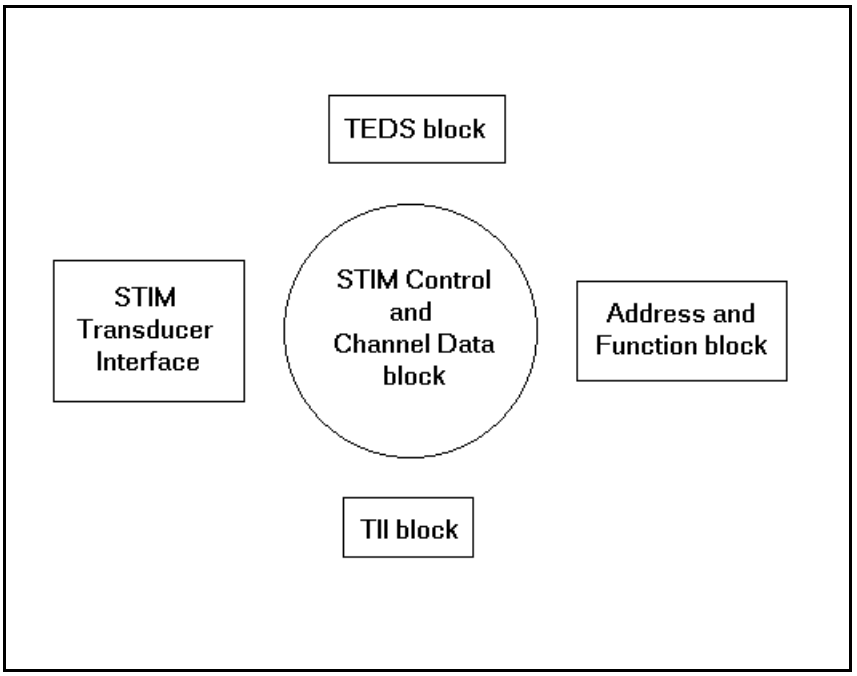

Figure 2. 1451.2 broken down into it's logical software blocks

1.3 The ADuC812

The ADuC812 contains an 8051 compatible MCU, 8kb of program flash/EE, 640 bytes of data flash/EE, 256 bytes of RAM, up to 32 programmable I/O lines, an SPI® serial I/O port, dual DACs and an 8-channel true 12-bit ADC.

Figure 3 shows a block diagram containing all of the above features of the ADuC812.

The SPI port is an industry standard four wire synchronous serial communications interface. It can be configured for master or slave operation, and is externally clocked when in slave mode.

The data flash/EE is a memory array which consists of 640 bytes, configured into 160 four-byte pages. The interface to this memory space is via a group of registers that is mapped in the SFR space.
The 8kb program flash/EE will ultimately store and run the end users' application code.

® SPI is a registered trademark of Motorola, Inc.

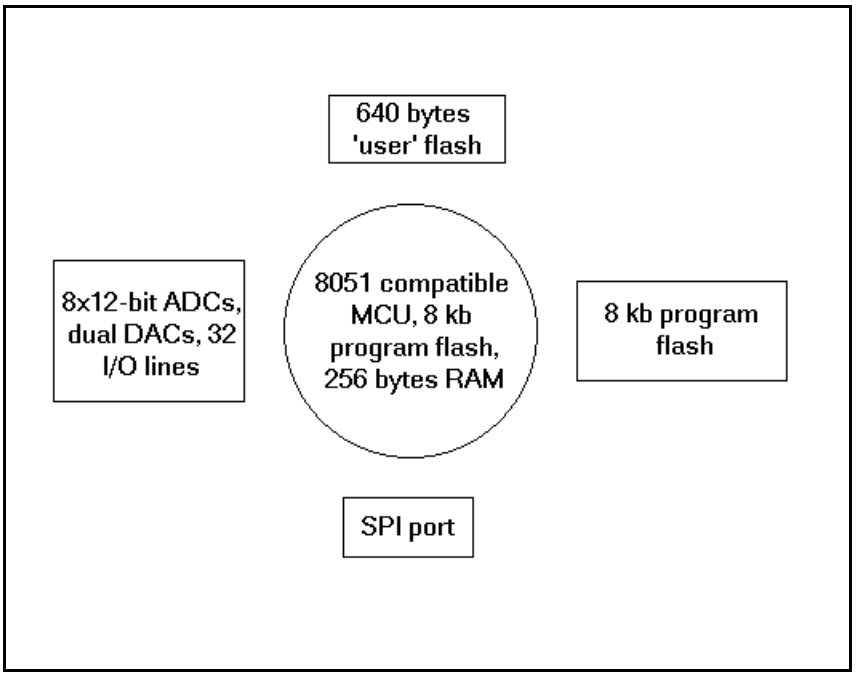

Figure 3. A block diagram of the principal features contained on the ADuC812

1451.2 and the ADuC812

A quick comparison of figures 2 and 3 will emphasise the suitability of the ADuC812 as an implementation platform for a STIM based on the IEEE 1451.2 standard. This is shown diagrammatically below in figure 4.

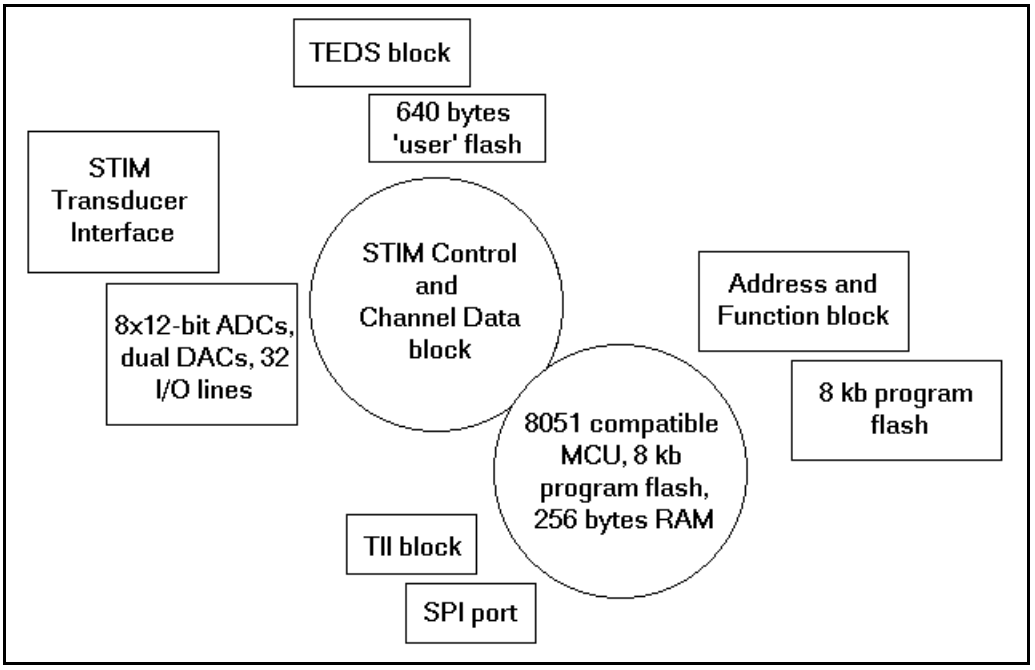

Figure 4. How the software blocks overlay onto the ADuC812 feature blocks

1. the STIM will be controlled from the program flash/EE, and each channel's transducer data, status and control registers will be held in RAM for the duration of the STIM lifetime,
2. the transducer interface will be mapped onto the ADCs, DACs and I/O lines,
3. the TII will be a super-set of the SPI port (plus some I/O lines),
4. the TEDS map into the 640 bytes of data flash/EE, and finally
5. the Address and Function block will be stored into the program flash/EE,

2. Implementation

2.1 Software Overview

The 1451.2 software is implemented in four modules, which reflect the diagram shown in figure 2. The two 'STIM' blocks have been put together into one software module. The modules are implemented as described in the following paragraphs.

2.2 Software Modules

2.2.1 STIM block / STIM transducer interface

- *software module name "stim.c / stim.h"*
- *will require modifications for end user requirements*

This module contains the definitions for the channels associated with the STIM, the data associated with each channel and the main control flow of the program. This particular implementation defines two channels: one sensor (an AD590 temperature sensor) and one actuator (a simple digital-output controlled fan, that is either 'on' or 'off'). As with the TII module, this module contains certain definitions that are necessarily hardware-coupled (i.e. the physical lines that the sensor and actuator are realised on).

During code-customisation, the hardware-specifics will need to be re-defined. The software provides one function for setting up and initialising each channel, and so a function for each channel will have to be written, or the existing one modified. The means by which data is written to or read from each channel will also need to be considered, although this should be a more straight-forward addition / modification to the existing functions. Finally, the channel type determines how each channel reacts to a trigger. The triggering operation will have to be addressed – the examples provided in this application will serve as a good starting point.

2.2.2 TII block

- *software module name "tii.c / tii.h"*
- *will not require modification for end user requirements*

The TII module defines the physical interface on the NCAP side. As has been pointed out previously, the TII is a superset of the SPI port that exists on the ADuC812. It is

written on the hardware level, as it has to interact with the physical layer. Therefore, it is coupled tightly with the hardware.

On the other hand, looking at the TII functions from the API (i.e. end user's) perspective, the hardware coupling is transparent.

This module will only require changes if the end user needs to re-define some of the TII port-pins that are used. If the user is happy with the definitions as they exist, no changes will be necessary.

2.2.3 TEDS block

- *software module name "teds.c / teds.h"*
- *will require modification for end user requirements*

The TEDS module defines the TEDS that are in use for the current implementation of the 1451.2. It defines where the TEDS are mapped to, how they are written, how they are retrieved and what they contain.

Naturally, as all implementations will have a different set of TEDS associated with them, this module will need changes during code customisation. Specifically, one function and a new TEDS start address (within the memory map) must be added for each TEDS. See the example given in section XX.

2.2.4 Address and Function Block

- *software module name "function.c / function.h"*
- *will require only minimal modification for end user requirements*

This module implements all of the main functionality that is defined by the standard. It takes care of the 'Data Transport', 'Control', 'Interrupt', 'Status' and 'Trigger' functions. Each of the different function types is preceded by a three letter abbreviation that represents the grouping to which it belongs, e.g. the function '`DAT_ReadMetaTEDS()`' belongs to the 'Data Transport' function group.

This module is transparent to the user, and will not require any major changes during end-user code-customisation.

2.3 Hardware Overview

Analog Devices' ADuC812 evaluation board, "Eval-ADuC812QS, Rev: B01", was used for this 1451.2 realisation.

The following modifications need to be made to this board:
- Connect the link, LK5.
- Lift pin 6 of U5 [1].
- Wire in the Sensor and Actuator set, as defined in figure 5.

The NCAP used was the HP Bfoot 66501. The HP Bfoot 66501 is a complete thin web-server solution for manufacturers of smart sensors and actuators, and it is also an IEEE 1451.2 NCAP. It provides two IEEE 1451.2 2x5 connectors on-board.

[1] Before lifting, U5 pin 6 was driving Port 3.2 [INT0, or NTRIG in this case] high, which would invalidate the signal coming in on NTRIG from the NCAP.

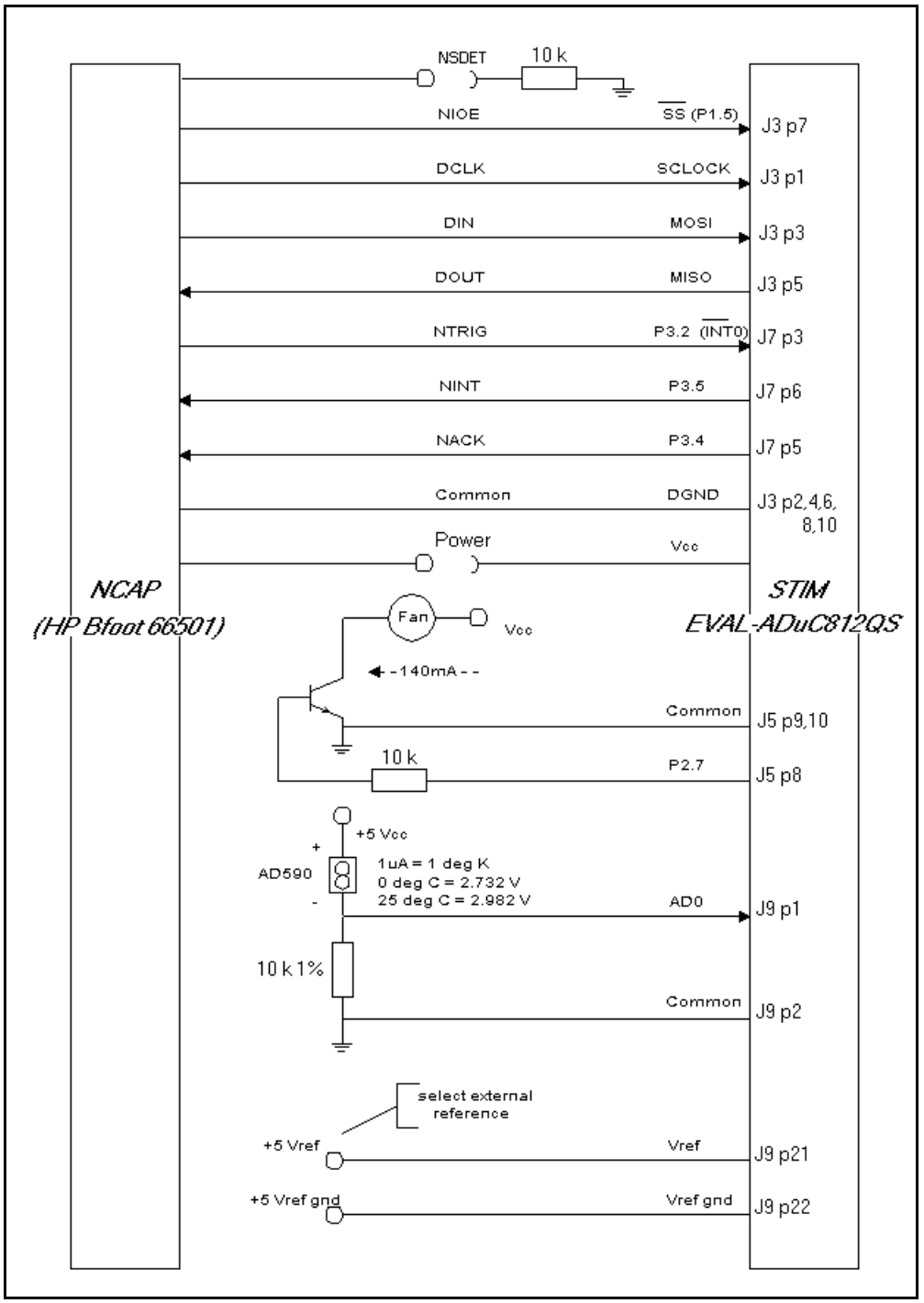

Figure 5. The hardware connections to be made on the 'Eval-ADuC812QS' board

2.4 The 1451.2 General Program Control Flow

The program control flow is shown in figure 6. This represents the '`main()`' function contained within the STIM code module.
When power is supplied to the STIM board, the STIM will go through an initialisation routine.
During this routine:-

- all memory and flash/EE areas will be cleared in preparation for loading
- the TEDS will be loaded into the data flash/EE area
- three channels[2], together with their associated data buffers, register sets and interrupt masks will be set up in memory
- the TII will be initialised

On the first iteration of the main loop there can be no 'reset request', as this request can only occur as a result of a 'control' function written to the STIM via the data transport protocol.

The data transport is tested for activity (`DAT_TransportActive()`), and if there is transport activity pending, the 'data transport' processing block is entered. It is in this block that read and write functions are deciphered and the correct response action taken.

On completion of data transport, or if there is no transport active, the triggering function is polled for activity (`TRIG_PollTrigger()`). The trigger is the means by which a sensor sample is 'triggered' or an actuator data-set is written. In order that a trigger can validly occur, the 'triggered channel address' must first be written during a data transport function.
If the actuator is triggered, the actuator data set should already have been written into the channel data buffer, and the trigger simply causes this data to be sent to the actuator from that buffer. Conversly, if the sensor channel has been triggered, the sample is acquired and stored into the channel data buffer at trigger time.
The sensor data will only be returned to the NCAP if it is subsequently requested during a data transport frame.

If (during any of the activity described above) there is an error detected on any of the channels, the STIM must set an appropriate flag in the channel status registers. Any of the status flags[3] may cause the STIM to assert the 'interrupt request' line, to which the NCAP will be obliged to respond, via the data transport.

[2] CHANNEL_ZERO (global channel), channel 1 (sensor channel) and channel 2 (actuator channel).
[3] If the status flag is enabled in the channel's interrupt or auxiliary mask registers.

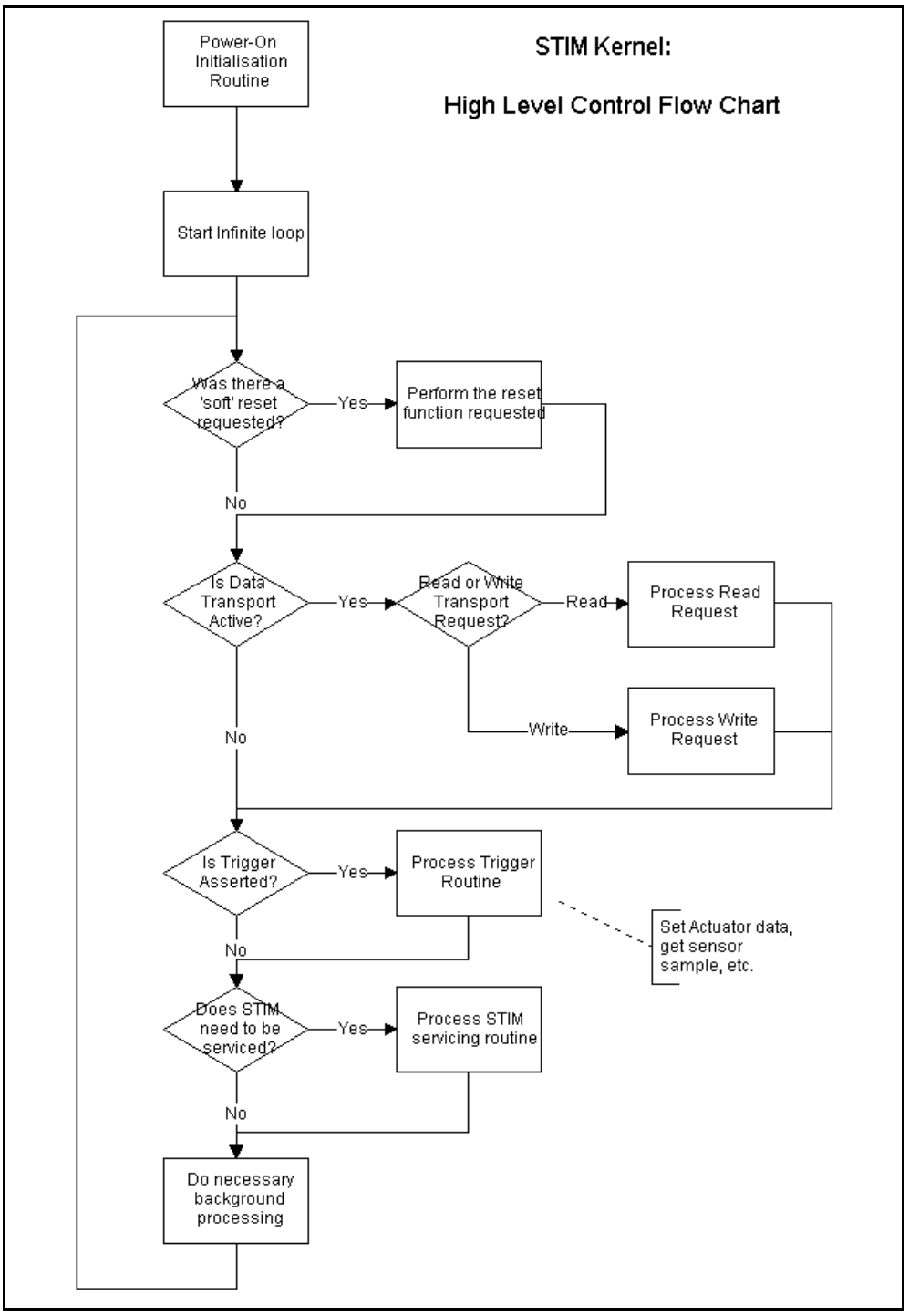

Figure 6. 1451.2 'main' program control flow chart

2.5 Building the Code

The code may be built using the Keil development environment.

1. Run the Keil 'µVision-51' application.
2. Open a 'New Project'. Call it '1451.prj'.
3. Go to the directory that you have down-loaded the source code into, and select the following files to be added to your project:
   - function.c,
   - teds.c,
   - tii.c, and
   - stim.c
4. Click 'Open All' and then the 'Save' button.
5. Go to 'Options' – 'BL51 Code Banking Linker', then select the 'Size/Location' tab. Change 'Ram Size' to 256 bytes. Click 'OK'.
6. Hit 'Alt-F8' to build the hex file.
7. There should now be a hex file called '1451.hex' in the same directory as the source code.

2.6 Programming the evaluation board

1. Connect the RS-232 cable between COM1 on a PC and the EVAL-ADuC812QS evaluation board.
2. Connect link LK3.
3. Apply power to the board. Press 'reset'.
4. Run the Analog Devices serial down-loader program (found in the ADuC812 directory).
5. Type in the entire path and filename of the '1451.hex' file.
6. When the serial down-loader has finished (it should take a few seconds), pull out the jumper on LK3, then press 'reset' again. The fan should visibly (though briefly) rotate on initialisation.
7. Disconnect the power.
8. You are now ready to connect the evaluation board to an NCAP over the TII.

2.7 Testing the application

We used the Hewlett Packard Bfoot 66501 industrial ethernet board to run tests and to communicate with this STIM implementation.

The HP Bfoot 66501 is a complete thin web-server solution for manufacturers of smart sensors and actuators, and it is also an IEEE 1451.2 NCAP. It provides 2 IEEE 1451.2 2x5 connectors on-board. It provides the ability to execute 1451.2 data-transport transactions, and to display the results on any web-browser.

It has an on-board 'trend-chart' applet that automatically triggers any attached sensors, and trends the results graphically in real-time.

3. Customising this code to End-User Requirements

The source code as described in this application note may be down-loaded from our website at http://www.analog.com/microconverter.

3.1 The source files available

The source files that are available from the web-site are described in table 1.

Filename	*Description*	*Customisation Changes Required?*
Stim.c	Contains the definitions for the location of the sensor and actuator, and their access methods. It also declares an instance of a 'CHANNEL' type for each implemented channel. It provides functions that allow access to each channel and it's associated data. Channel initialisation routines, STIM version information and STIM reset routines are defined. The 'main' program (i.e. the STIM Kernel from figure 6) is defined.	Changes will be required to reflect your custom version of STIM. This module, more than any of the others, will need to be thought through carefully. The 'main' structure itself will not change, but the definition of channels, their initialisation and how their data should be handled will.
Tii.c	Contains the physical definition of the 'transducer independent interface'. Maps the interface to the hardware SPI and general port I/O pins. Provides hardware independent function definitions for TII integration. Defines the interrupt routine for handling data via the SPI .	No changes will be required in this module, if the user is content to use the physical mapping provided. If not, only the physical I/O definitions will need to be changed. Logically, the TII functions should not be affected by changes to the physical mapping.
Teds.c	Contains the TEDS memory-map definition, the method of setting up the TEDS and of writing them into a non-volatile medium.	Almost all TEDS for all implementations will be different. The actual contents of the TEDS, the number of TEDS defined, the TEDS memory map and the medium to which the TEDS are written may all need to be changed.
Function.c	Contains most of the IEEE 1451.2 standard functions. These include the 'data transport', 'triggering', 'status', 'interrupt' and 'control' functions.	Very little in customisation will be needed here, if any at all. This module is best left as is. [4]
Stim.h	Contains the definition for the 'transducer data' generic structure, XDCR_DATA (required in CHANNEL_ZERO). Contains the definition for the 'CHANNEL' type, of which there will be an instance for each channel (see 'stim.c'). Defines all of the STIM function prototypes.	Changes should reflect any changes made to 'stim.c'. There may also be changes required to the structure of the 'XDCR_DATA' and 'CHANNEL' data-types.
Tii.h	Contains the function prototypes for 'tii.c'.	Changes should only reflect changes made to 'tii.c'.
Teds.h	Defines the number of channels in use,	The number of TEDS, the

[4] In this code version:
- the data-transport functions that write 'calibration' and 'calibration id' TEDS will not work
- the data-transport functions that read and write transducer data only support two channels

	CHANNEL_ZERO as a constant and the 'data model length' for each channel. Defines the structures for the Meta- and Channel- TEDS (there are no other TEDS types defined in this version of the module). The TEDS function prototypes are included.	channel 'data model lengths' and the TEDS structures must reflect the actual TEDS that are realised in your customised STIM. The function prototypes should reflect any changes made to 'teds.c'.
Function.h	Contains all of the definitions, typedefs, etc. relevant to 'function.c'. Contains the function prototypes, and descriptions of each of the function-groupings, as derived from the IEEE 1451.2 standard (section 4).	Changes should only reflect changes made to 'function.c'.
Datatype.h	Defines the data-types as described in the IEEE 1451.2 standard section 3.3. Also contains more generic data-types (e.g. NULL, boolean, NaN32) that are useful in the context of a 1451.2 code implementation.	Will not require changes.
Aduc812.h	Contains the SFR and register definitions specific to the ADuC812.	Will not require changes.

Table 1. Existing Code Table

3.2 Example of Code Customisation

This is an outline of how the existing down-loadable code should be changed to reflect a STIM on which there are two sensors (on ADC channels 0 and 1) and two digital I/O actuators (on ports pins 2.6 and 2.7).

We locate the first sensor on the STIM's logical channel 1, and the second on logical channel 3. The actuators will be located on logical channels 2 and 4.

3.2.1 'stim.c' and 'stim.h'

'stim.h'

- 'Ch3' and 'Ch4' must be added into the XDCR_DATA union. As they are a sensor and actuator, their data sets will be 'unsigned int' and 'boolean' respectively. The data type for 'ChData' will not change in this example. If a larger type than any of the existing ones was introduced, 'ChData' would assume this type.
- add function prototypes for : STIM_InitCh3, STIM_InitCh4, STIM_GetCh3Sample, and STIM_SetCh4State

'stim.c'

- define a second actuator, on pin 2.6.

```
sbit FAN_CH2  = P2 ^ 7;   // on channel 2
sbit RELAY_CH4 = P2 ^ 6;  // on channel 4
```

- in 'STIM_GetXdcrData()', add two more 'channel' cases:

```
case 3:*(unsigned int*)pBuf = maChannelData[ucChNum].Data.Ch3;
break;
case 4:               *pBuf = maChannelData[ucChNum].Data.Ch4;
break;
```

- in 'STIM_SetXdcrData()' add two more channel cases:

```
case 3:  maChannelData[ucChNum].Data.Ch3 = *pBuf;
         maChannelData[ucChNum].Data.Ch3 <<= 8;
         maChannelData[ucChNum].Data.Ch3 |= *(pBuf+1);
break;
case 4:  maChannelData[ucChNum].Data.Ch4 = *pBuf;
break;
```

- add two 'channel initialisation' functions called STIM_InitCh3 and STIM_InitCh4. Use the existing functions STIM_InitCh1 and STIM_InitCh2 as example templates.

- add a function called STIM_GetCh3Sample, and one called STIM_SetCh4State. Again, the existing functions STIM_GetCh1Sample and STIM_SetCh2State can be used as example templates.

- in the 'main' routine:
  ('Initialisation' block – reference figure 6)
  - add function calls to STIM_InitCh3 and STIM_InitCh4
  - add function calls to TEDS_SetupCh3Teds and TEDS_SetupCh4Teds
  - add calls to INT_SetInterruptMask for both channels 3 and 4, and for both auxiliary and standard interrupt masks

  ('Trigger' block)
  - add cases that describe the action to be taken when a trigger occurs on either channel 3 or 4. Use the existing cases as examples.

3.2.2 'teds.c' and 'teds.h'

'teds.h'

- change the definition for '`NUM_CHANNELS`' to 4
- add two definitions for `CH3_DATA_MODEL_LENGTH` and `CH4_DATA_MODEL_LENGTH`, and make them '2' and '1' respectively
- add function prototypes for `TEDS_SetupCh3TEDS` and `TEDS_SetupCh4TEDS`

'teds.c'

- add the two new channel TEDS to the memory map, and assign them addresses:

```
#define  CH3_TEDS_ADDRESS     0x43
#define  CH4_TEDS_ADDRESS     0x5B
```

- add two new 'setup TEDS' functions, one for each new channel TEDS required (i.e. `TEDS_SetupCh3Teds` and `TEDS_SetupCh4Teds`). These functions can be based entirely on the existing functions, with only the actual content of the TEDS data structure changing. Any internal references to a specific channel number and / or the channel TEDS address must also be changed.
- In the '`TEDS_GetTEDSHandle`' function, add two more 'channel TEDS' cases, one each for channels 3 and 4.

3.2.3 'function.c' and function.h'

'function.h'

- there are no changes to be made here

'function.c'

- go to `DAT_ReadXdcrData` and add two more cases, i.e.

```
…
else if(ucChan==3)   DataLength = CH3_DATA_MODEL_LENGTH;
else if(ucChan==4)   DataLength = CH4_DATA_MODEL_LENGTH;
…
```

- add the same two lines to the 'switch…case' in `DAT_WriteXdcrData`.

3.2.4 All remaining modules

'tii.c', 'tii.h', 'datatype.h' and 'aduc812.h' will not require any modifications.

You can now build, link and down-load this code as described in sections 2.4 – 2.6.

4. Conclusions

The ultimate aims of this IEEE 1451.2 application were to:
- create a true minimal standard implementation
- create modular, scaleable and readable code
- make it easy for an end user to customise the code

The examples given in section 3 (and also in appendix 2) show how the existing code modules may be customised and expanded to fit user requirements.

Some of the issues to be aware of during code-modification are:

RAM:
There are 256 bytes of RAM available on-board. Of this,
- 100 bytes are currently reserved for loading the TEDS.
- 10 bytes are reserved for each implemented channel.
- the run-time data space requires over 30 bytes.
- the run-time stack space requires between 20 and 30 bytes.

Data Flash/EE:
There are 640 bytes of data flash/EE available on-board.
- 76 bytes are required for the smallest version of 'Meta TEDS'.
- 96 bytes are required for each of the smallest versions of the 'Channel TEDS'.
- the other TEDS types are undefined in length, but can be quite large.
- up to 5 channel TEDS (plus the Meta-TEDS) can be stored in the data flash/EE area.

Physical implementation:
The TII connector (e.g. a 10 wire ribbon cable) interface should be kept as short as possible. For reliable operation we recommend a cable connection of up to a few inches.

Resource Usage:

Original Code: incl. 2 Channel TEDS, 1 Meta TEDS		
Resource:	**Used:**	**Available:**
256 byte RAM	178 bytes	78 bytes
640 byte data flash/EE	268 bytes	372 bytes
8 kbyte program flash/EE	5,534 bytes	2,657 bytes
Example Code: incl. 4 Channel TEDS, 1 Meta TEDS, 1 Meta-Id TEDS		
Resource:	**Used:**	**Available:**
256 byte RAM	197 bytes	59 bytes
640 byte data flash/EE	552 bytes	88 bytes
8 kbyte program flash/EE	6,553 bytes	1,638 bytes

Table 2. Resources used and available.

If more TEDS are required, they may be located within the program flash/EE. Note that as more TEDS are required, the code space itself will grow too, so there is also a limit to the number of TEDS that may be fitted into this space.
Alternatively, an external EEPROM may be used to store the TEDS, if required. In this case the TEDS reading and writing functions would need to be re-written, and the memory map changed.

In table 2, there is no consideration given to the run-time stack requirements within the RAM space. The stack will use the excess RAM at the top of the RAM space. The amount of RAM required differs, and it is estimated to be somewhere between 20 and 30 bytes. Therefore the amount of RAM 'available' as shown in table 2 is actually not entirely correct.

5. Further Application Support on the IEEE1451.2 Implementation

The implementation described within this application note is based on the full release version of the IEEE1451.2 standard, and was developed by PEI Technologies, University of Limerick, Limerick, Ireland.

PEI Technologies specialises in research, design and development of hardware and software for distributed control systems, based on industry standard serial communications protocols.

PEI is available to support any IEEE1451.2 implementation, or other parts of the IEEE1451 series of standards, (e.g. the 1451.1 prov. standard (NCAP) for protocols such as Ethernet, CAN and LonWorks). Please visit http://www.ul.ie/~pei for further information.

6. References

"Sensors Smarten Up" EDN Cover Story, Bill Travis, March 1999
"Smart sensor standard looks set for takeoff" EE Times, Terry Costlow, Oct 1998
"IEEE Std 1451.2-1997" IEEE, Sept 1998

Appendix 1

Design Overview

Memory Map

Figure 7 shows the programming model of the ADuC812.
The ADuC812 has separate address spaces for Program and Data memory. The additional 640 bytes of Flash/EE that is available to the user may be accessed indirectly via a group of control registers in the SFR area of the Data Memory Space.
The lower 128 bytes of RAM are directly addressable, while the upper 128 bytes may only be addressed indirectly.
The SFR area is accessed by direct addressing only and provides an interface between the CPU and all on-chip peripherals.

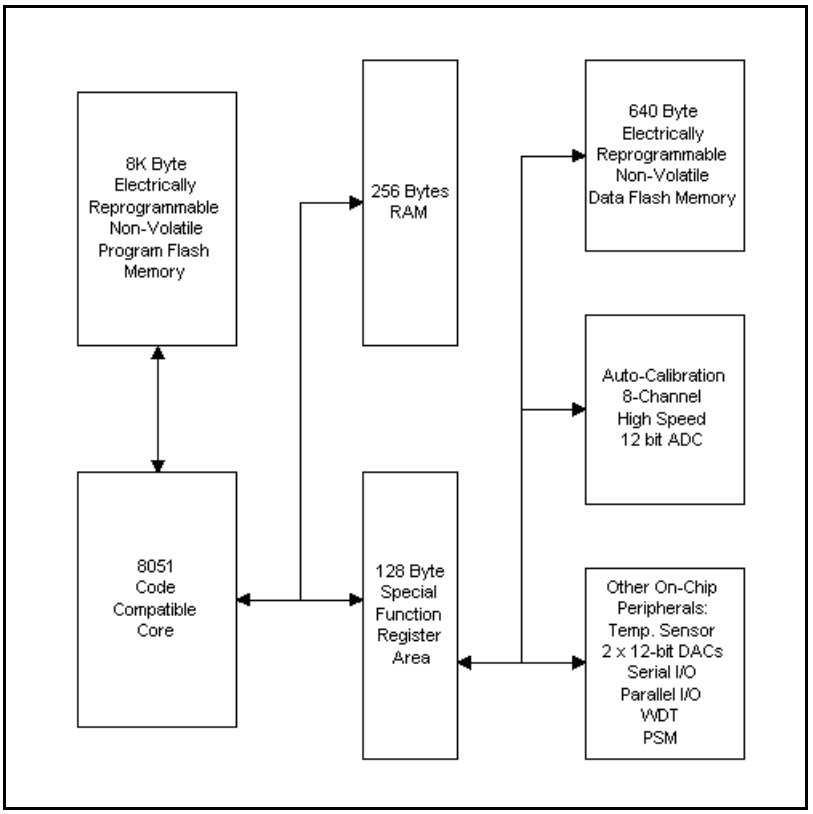

Figure 7. The ADuC812 Programming Model

Figure 8 shows how the 1451 implementation makes use of the ADuC812 Programming Model. The TEDS are located in the 640 byte data flash/EE, the TII and actuator are directly tied into the 'Peripherals' block and the sensor is hanging off of the ADC block. All of these features are accessed and controlled via the SFR area.

The standard data RAM is used for storing the STIM channel transducer data and registers. It must reserve a buffer from which the TEDS may be individually loaded, and into which the TEDS may be read back from the data flash/EE. Note also the the RAM must contain all 'local' and 'system' variables that are required by the code, and it must also allow for the run-time stack. All of these requirements place limitations on the size of the TEDS buffer, which the end user should be aware of.

Naturally, all of the programming functions are stored into the program flash/EE memory.

In figure 8, the dotted line shows the logical link between the `DAT_Write.x.Teds()` and `DAT_Read.x.Teds()` functions, and the actual writing and reading to/from the TEDS data flash/EE area. These TEDS_ function calls are designed to be logically transparant to the end user, and the method of implementation is not important.

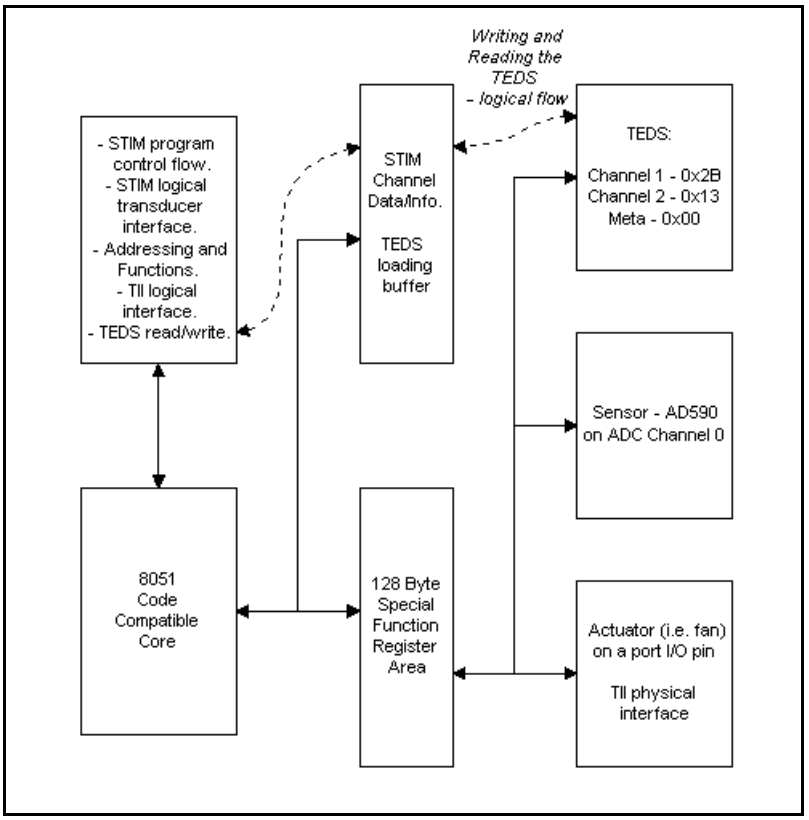

Figure 8. 1451.2 Implementation mapped into the ADuC812 Programming Model

Design Details:

The details of the TII and transducer connections are shown back in figure 5.

1. The TII

The TII interface specifies 10 lines, namely:

Group:	Line:	Abbreviation:	Description:
Data	DATA_OUT	DOUT	Data transport from STIM to NCAP
	DATA_IN	DIN	Address and data transport from NCAP to STIM
	DATA_CLOCK	DCLK	Positive-going edge latches data on both DIN and DOUT
	N_IO_ENABLE	NIOE	Signals that the data transport is active and delimits data transport framing.
Triggering	N_TRIGGER	NTRIG	Performs triggering function

Support	POWER COMMON N_ACKNOWLEDGE N_STIM_DETECT	POWER COMMON NACK NSDET	Nominal 5V power supply Signal common or ground Serves two functions: 1. trigger acknowledge 2. data transport acknowledge Used by the NCAP to detect the presence of the STIM
Interrupt	N_IO_INTERRUPT	NINT	Used by the STIM to request service from the NCAP

Table 3. TII definition

The lines defined in Table 3 are realised as outlined in Table 4, below:

TII Line:	**ADuC812 pin:**	**EVAL-ADuC812QS :**	**Description:**
DOUT	MISO	J3, pin 5	'Master In, Slave Out' data I/O pin. Part of the SPI port interface.
DIN	MOSI	J3, pin 3	'Master Out, Slave In' data I/O pin. Part of SPI.
DCLK	SCLOCK	J3, pin 1	Serial Clock I/O. Part of SPI.
NIOE	Slave Select	J3, pin 7	'Slave Select Input Pin'. Part of SPI. When it is used, it indicates that the SPI is in 'slave mode' - i.e. the serial clock will be driven externally.
NTRIG	Port 3.2 (INT0)	J7, pin 3	A general I/O port pin, configured as an input. The reason for wiring NTRIG into this particular pin is that it may be configured (through software) as an external interrupt line. [5]
POWER	Vcc	Power Rail	Power supply for the STIM – by definition power for the TII must be supplied by the NCAP.
COMMON	Gnd	Ground Rail	Ground rail for the STIM.
NACK	Port 3.4	J7, pin 5	General purpose I/O port pin, configured as an output.
NSDET	Not Connected	Connect through a pull-down resister (i.e. 10k) to ground rail.	The NSDET line is active low, and so must be pulled down on the STIM side.
NINT	Port 3.5	J7, pin 6	General purpose I/O port pin, configured as an output.

Table 4. Actual implementation of TII interface on the EVAL-ADuC812QS

To be TII compatible, the SPI must be initialised for use as 'slave mode', with the 'clock idle' polarity set to 'high' and the clock set to latch on the rising edge (see IEEE 1451.2, sections 6.2 and 6.3.7).

[5] Although the 1451 is best implemented without using an interrupt for NTRIG, there may be occasions when this scenario would be advantageous.

This is achieved by writing to the SFR registers SPICON and SPE and enabling the SPI interrupt (held within IE2) :

```
SPICON = 0x0C;    // slave mode: CPOL=1, CPHA=1, SPIM=0
   IE2 = 0x01;    // enable SPI interrupt
   SPE = 1;       // enable SPI
```

The TII lines that use port pins (i.e. NINT, NACK, NTRIG, NIOE[6]) must be configured (for either input or output) at the beginning of the module. Their physical mapping is defined first :-

```
// STIM Inputs...
//
sbit NIOE   = P1 ^ 5;
sbit NTRIG  = P3 ^ 2;

// STIM Outputs...
//
sbit NACK   = P3 ^ 4;
sbit NINT   = P3 ^ 5;
```

and their direction is defined within the `TII_Initialise()` function :-

```
// Set up Port 1.5 as a digital input port (NIOE pin, 'SS')
//    - this is done by writing 0 to that SFR bit [7] .
//
   P1 = P1 & 0xDF;

// Set up Ports 3.4 and 3.5 as digital outputs (NACK and NINT)
//    - this is done by writing 0 to those SFR bits.
// Set up Port 3.2 as a digital input (i.e. NTRIG)
//    - this is done by writing 1 to that SFR bit.
//
   P3 = P3 & 0xCF;
   P3 = P3 | 0x04;
```

2. The Sensor and Actuator

The sensor is an AD590 (temperature sensor), and it is input on the ADuC812 ADC channel 0. It is defined and initialised in `STIM_InitCh1()` :

```
// Set Port 1.0 as an analog input port (Sensor Input on ADC0)
//    - this is done by writing 1 to that SFR bit.
//
P1 = P1 | 0x01;

ADCCON1 = 0x6C;   // ADCCON1  01101100
ADCCON3 = 0x00;   // ADCCON3  00000000
ADCCON2 = 0x00;   // Don't enable any data conversion mode (yet)
```

[6] Note that NIOE uses the 'slave select' line - the second function on port 1.5. 'Slave select' is confingured by writing '0' to port 1.5, and thereafter reading port 1.5 shows the status of 'slave select'.
[7] See the ADuC812 datasheet

Every time a data sample on channel 1 is requested[8], a 'single sample' conversion sequence is started. When this sequence completes, a data sample is available. The data conversion must wait for an ADC interrupt bit to toggle before the sample is considered to be valid. See `STIM_GetCh1Sample()`.

```
// Set the 'single sample' bit in ADCCON2 to initiate a 'single
// conversion cycle'
//
SCONV = 1;

// Wait for the ADC interrupt (i.e. sample acquired) bit ...
//
while( !ADCI )
   ;

<< The code here (not shown) acquires and holds the sample. When
   that's done, reset the 'single conversion' bit and the 'ADC
   interrupt' bit, so that the ADC is ready for the next
   triggered sample. >>

SCONV = 0;
ADCI = 0;
```

The actuator is a simple fan, that is controlled by port I/O pin 2.7. It is defined at the beginning of the 'stim.c' module, and it exists on 1451.2 channel 2. It is initialised in `STIM_InitCh2()`, and manipulated from `STIM_SetCh2State()`.

```
// Define the pin that the Actuator (i.e. the Fan) on channel 2
// is controlled by.
//
sbit FAN_CH2  = P2 ^ 7;
```

3. The TEDS

The first thing to be done with the TEDS was to define how many were needed, and where they should be located in memory.
The only TEDS that are compulsory (as defined in the standard) are the Channel- and Meta- TEDS. That meant that one for each channel (Channel 1 TEDS and Channel 2 TEDS) and one that describes the entire system as a whole (the Meta TEDS) were required. There are no other TEDS defined in this implementation.

The Meta- TEDS for this system is 76 bytes in total, and the Channel TEDS are 96 bytes each. They are mapped into the 640 byte data flash/EE area :-

TEDS:	Data flash/EE Address:[9]	TEDS Length:
Meta	0x00	76 bytes
Channel 1	0x13	96 bytes
Channel 2	0x2B	96 bytes
	Total:	268 bytes

Table 5: TEDS locations in data flash/EE.

[8] A sample request comes in the form of a 'trigger', which is an assertion of NTRIG
[9] Note that each address location represents 4 bytes.

There is one function defined for loading each of the TEDS from program space into the data flash/EE area.

Each function defines the specific TEDS data structure first[10], then allocates a pointer so that it may be accessed as a byte-array. The checksum for the set of data specified is calculated and automatically filled into the 'checksum' field of the TEDS. The TEDS is then written to flash/EE (call `TEDS_WriteTEDSToFlash()`).
Note that the TEDS address in flash/EE specific to the TEDS being written must be specified and passed to this function call.

```
boolean TEDS_SetupMetaTEDS(void) {
  stMetaTeds idata MetaTEDS = {
  (U32L)sizeof(stMetaTeds)- sizeof(U32L), // Meta-TEDS length
  2,                                      // 1451 Working Grp Num
  1,                                      // TEDS Version Num
  // Universally unique ID:
  // This UUID represents: 53 deg 30 minutes North,
  // --------------------  08 deg  0 minutes  West,
  //                       Manufacturers Code: 1,
  //                       Year: 1999,
  //                       Time: 00:00:00 on May 4th.
  //

  0x97,0x82,0xC0,0x1C,0x20,0x05,0xF3,0xD0,0x59,0x00,

  0,                                  // Calib TEDS ext key
  0,                                  // NonVolatile Data ext key
  0,                                  // Industry TEDS ext key
  0,                                  // End User App-Spec ext key
  NUM_CHANNELS,                       // Num of implemented chans
  2,                                  // Meta Chan Data Model Len
  0,                                  // Meta Chan Data Reps
  0L,                                 // CH_0 writable TEDS len
  0.0025,                             // Meta Chan Update Time
  0.001,                              // Global Write Setup Time
  0.001,                              // Global Read Setup Time
  0.0005,                             // Chan Samp period
  0.0005,                             // Chan Warmup Time
  0.5,                                // Command Response Time
  0.0003,                             // STIM handshake Time
  0.04,                               // EOF Detection Latency
  0.03,                               // TEDS hold off Time
  0.03,                               // Operational hold off Time
  2000000,                            // Max Data Rate
  0,                                  // Chan Grp Data Sub-Block
  //
  // There are no Channel Groupings Sub-Blocks
  //  => no iterations for fields 25-28 exist.
  //
  0                                   // Checksum (not filled yet)
  };

  // The TEDS Data area must be programmed one byte at a time
  // => we need to access the TEDS structure as a byte array.
  //
```

[10] The data presented here is correct for this implementation.

```
unsigned char *pTEDSByteArray = &MetaTEDS;

// Calculate the Checksum for the Meta-TEDS, and fill it into
//  the Meta-TEDS data structure.
//
MetaTEDS.Checksum = DAT_CalcChecksum(pTEDSByteArray);

// Program the Meta-TEDS structure into the TEDS Flash data
// area.
//
return TEDS_WriteTEDSToFlash( META_TEDS_ADDRESS,
                              pTEDSByteArray,
                              MetaTEDS.Length+sizeof(U32L) );
}
```

Together with the Channel TEDS setup functions (`TEDS_SetupCh1TEDS()` and `TEDS_SetupCh2TEDS()`) , this is all that is required to set the TEDS for this system.

To get the TEDS data back out[11] of the data flash/EE area, there is a need for another function. This function is `TEDS_ReadTEDSfromFlash()`. When reading a TEDS back from the flash/EE area, it must be stored into RAM. Therefore, a RAM buffer that is sufficiently big[12] for the TEDS must be reserved and passed to this function.

These functions are designed to be modular, and to allow expandibility for customisation.

Note that if a user's application maps the TEDS to an external EEPROM[13] (rather than to the internal flash/EE), just one function call would need to change in the 'TEDS_Setup…()' functions. It would be a simple matter of replacing the existing `TEDS_WriteTEDSToFlash()` function call with a `TEDS_WriteTEDSToEeprom()` call. By extension, a `TEDS_ReadTEDSfromEeprom()` matching function would also be required.

[11] When the TEDS are being 'read' by the NCAP, they are individually read out of the flash/EE area and transmitted from RAM, one byte at a time. See the DAT_Read…TEDS() functions.

[12] As only 256 bytes of RAM exist, and the buffer must co-exist with the runtime data and stack requirements, the maximum size available is around 150 bytes.

[13] An external EEPROM solution might be required in a situation where there are more TEDS than can be fitted into the on-chip data flash/EE. In this case, another viable solution would be to embed the non-writable TEDS into the program flash/EE, along with the code. Whichever solution taken, the code-changes required to access the re-mapped TEDS would be minor.

Appendix 2:

Example of how to add a 'Meta-ID TEDS' to the downloaded Software Implementation

See IEEE 1451.2 section 5.4 'Meta-Identification TEDS Data Block'.

The only software modules that will need to be modified are 'teds.h', 'teds.c' and 'stim.c'.

Modifications to 'teds.h' :

- the following definitions must be added, so that a Meta-Id TEDS can be defined. These definitions define the "number of languages", and the relevant "language enumerations" (see IEEE 1451.2 sections 3.3.7.1 - 3.3.7.3) supported in the Meta-Id TEDS:

```
#define TEDS_LANGUAGES      1
#define ENGLISH            25
#define ASCII              10
#define ONE_OCTET           0
```

- the following definitions describe the string–lengths that are used within the Meta-Id TEDS data structure, and the actual corresponding strings that the structure contains:

```
#define MANU_ID_LEN        16
#define MANU_ID_STRING     "Analog Devices  "
#define MODEL_NUM_LEN      10
#define MODEL_NUM_STRING   "ADuC812 v1"
#define VERSION_LEN        10
#define VERSION_STRING     "Demo Ver  "
#define SERIAL_NUM_LEN      6
#define SERIAL_NUM_STRING  "123456"
#define DATE_CODE_LEN       6
#define DATE_CODE_STRING   "130599"
#define PROD_DESC_LEN      20
#define PROD_DESC_STRING   "STIM EVAL-ADuC812QS "
```

- define a structured data-type that defines the "Identification Related Data Sub-Block" (i.e. the language specific sub-block) that will be contained within the Meta-Id TEDS:

```
typedef struct {
      U16L     LangSubBlkLength;
      LANG     StringSpec;
      U8L      ManuIdLength;
      STRING   ManuId[MANU_ID_LEN];
      U8L      ModelNumLength;
      STRING   ModelNum[MODEL_NUM_LEN];
      U8L      VersionCodeLength;
      STRING   VersionCode[VERSION_LEN];
      U8L      SerialNumLength;
      STRING   SerialNum[SERIAL_NUM_LEN];
      U8L      DataCodeLength;
      STRING   DataCode[DATE_CODE_LEN];
      U16L     ProdDescLength;
      STRING   ProdDesc[PROD_DESC_LEN];
```

```
} stMetaIDTedsIdDataSubBlk;
```

- define the 'Meta-Id TEDS' structure data-type:

```
typedef struct {
   U32L     Length;
   U8C      NumLangs;
   U8E      LangCodeList[TEDS_LANGUAGES];

   // Identification Related Data Sub-Block
   //
   stMetaIDTedsIdDataSubBlk LangData[TEDS_LANGUAGES];

   // Note that the 'identification related data sub-block'
   // would be repeated for each language implemented.
   // We are only implementing 1 language - English.

   // Channel Grouping Data Sub-Block
   //
   U16L     ChanGroupDataSubBlk;          // field 18

   // In this implementation there are no "Channel Groupings
   // Data Sub-Blocks" => no iterations for fields 19-21 exist

   // Data Integtrity Data Sub-Block
   //
   U16C     LanguageBlkChecksum;          // field 22

   // Data Integrity Data Sub-Block
   //
   U16C     Checksum;                     // field 23
} stMetaIdTeds;
```

- Add the function prototype for '`TEDS_SetupMetaIdTEDS()`'.

Modifications to 'teds.c' :

- Add a definition for the location of the Meta-Id TEDS within the data flash/EE

```
#define  METAID_TEDS_ADDRESS  0x43
```

- Add a new function, `TEDS_SetupMetaIdTEDS`, which will setup the Meta-Id TEDS into it's address in flash/EE, i.e.:

```
boolean TEDS_SetupMetaIdTEDS(void)
{
   // Define The MetaID-TEDS Contents at Declaration Time
   // ---------------------------------------------------
stMetaIdTeds idata MetaIdTEDS = {

         // Data Structure Related Data Sub-Block
         // -------------------------------------

(U32L)sizeof(stMetaIdTeds)- sizeof(U32L), // MetaID-TEDS length
TEDS_LANGUAGES,                           // Num of Langsuages
ENGLISH,                                  // Languages. . .

         // Identification Related Data Sub-Block
         // -------------------------------------
                                   // Lang Sub Blk Len
(U16L)sizeof(stMetaIDTedsIdDataSubBlk)-sizeof(U16L),
ASCII,                             // LANG1: Char Set
ONE_OCTET,                         // LANG2: Char Code Format
```

```
ENGLISH,                              // LANG3: Str Lang Code
MANU_ID_LEN,                          // Manu Id Len
MANU_ID_STRING,                       // Manufacturer Id
MODEL_NUM_LEN,                        // Model Num Length
MODEL_NUM_STRING,                     // Model Num
VERSION_LEN,                          // Ver Code Length
VERSION_STRING,                       // Ver Code
SERIAL_NUM_LEN,                       // Serial # Length
SERIAL_NUM_STRING,                    // Serial #
DATE_CODE_LEN,                        // Date Code Length
DATE_CODE_STRING,                     // Date Code
PROD_DESC_LEN,                        // Product Desc Len
PROD_DESC_STRING,                     // Product Description

        // Channel Grouping Data Sub-Block
        // -------------------------------
0,                                    // Chan Grp Data Sub Blk Len

        // Data Integrity Data Sub-Block (1)
        // ---------------------------------
0,                                    // Checksum for Lang Sub-Blk

        // Data Integrity Data Sub-Block (2)
        // ---------------------------------
0                                     // Checksum for MetaId TEDS
};

   // The TEDS Data area must be programmed one byte at a time.
   // Therefore, we need to access the TEDS structure as a byte
   // array.
   //
unsigned char *pTEDSByteArray = &MetaIdTEDS;

   // Calculate the Checksum for the Language Sub-Block section
   // of the Meta ID Teds, and fill it into the Meta ID TEDS
   // 'data integrity  data sub-block (1).
   //
MetaIdTEDS.LanguageBlkChecksum = TEDS_CalcLangSum(
   MetaIdTEDS.LangData[0].LangSubBlkLength + sizeof(U16L),
   (unsigned char *)&MetaIdTEDS.LangData[0] ) ;

   // Calculate the Checksum for the overall Meta-ID TEDS, and
   // fill it into the Meta-ID TEDS data structure.
   //
MetaIdTEDS.Checksum = DAT_CalcChecksum(pTEDSByteArray);

   // Now program the Meta-TEDS structure into the TEDS Flash
   // data area.
   //
return TEDS_WriteTEDSToFlash( METAID_TEDS_ADDRESS,
                              pTEDSByteArray,
                              MetaIdTEDS.Length+sizeof(U32L) );
}
```

- Note that in the above function there are **two** checksums required and calculated, one for the 'language sub-block' and one for the overall 'Meta-Id TEDS'. The TEDS generic checksum calculation function (`DAT_CalcChecksum()`) will only work for the latter checksum calculation, therefore a local function

(`TEDS_CalcLangSum()`) must be defined to calculate the checksum for a specified number of bytes within a specified data block:

```
U16C TEDS_CalcLangSum(U16L NumBytes,
                      unsigned char *pStartOfLangBlk)
{
U16C Offset=0;
U16C ChkSum=0;

for(Offset=0; Offset<NumBytes; Offset++)
{
   ChkSum += *(pStartOfLangBlk+Offset);
}

// The actual checksum is the 1's complement => negate bits.
//
return ~ChkSum;
}
```

- Remember to put the module-level function prototype for `TEDS_CalcLangSum()` at the beginning of this file.

- In the `TEDS_GetTEDSHandle()` function, add the following case:

```
case(TEDS_META_ID):  ucTEDSAddress = METAID_TEDS_ADDRESS;
break;
```

Modifications to 'stim.c' :

- In the initialisation section of the 'main' routine (see figure 6), add a call to `TEDS_SetupMetaIdTEDS()`

That's all the modifications required. Now just build the code and send it to the evaluation board, as described in sections 2.4 and 2.5.

This code, together with the example described in section 3.2 of the application note, is available for download at www.analog.com/microconverter/example1