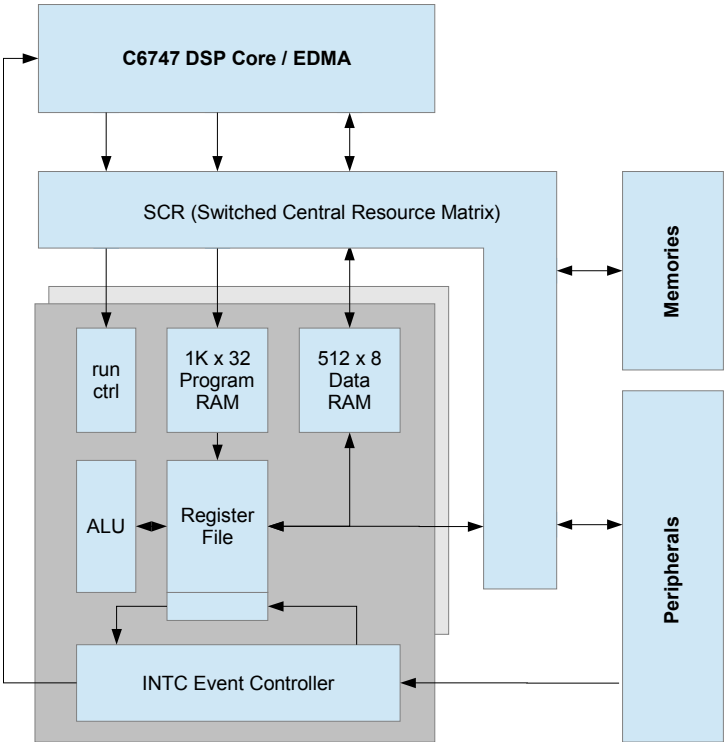
UniDAQ2 Application Note PRU

Document Revision 1.0 **Jul 9, 2018**

Background

The "Programmable Realtime Unit" subsystem is one of the most overlooked features of the TMS320C6747 DSP. Although this first generation implementation offers only limited capabilities compared to the up-to-date PRU-ICSS in the Sitara and Keystone processor families, the PRU can significantly offload the DSP and contribute to building robust and well structured programs. Unfortunately information and examples on C6747 PRU usage is rare.

The PRU subsystem consists of two 32-bit Harvard-architecture RISC processors, operating at 228 MHz, each with its own dedicated program and data RAM. The PRUSS also provides an event controller to route system events into and out of the PRU. The PRU has access to all peripheral subsystems and memories through the C6747 switched central resource matrix (SCR). The DSP loads, starts, halts or alters the PRU program code at runtime. DSP and EDMA both have access to the PRU data RAM for data exchange.



The PRU subsystem operates completely independent of the DSP core. This enables the PRU to execute hard real-time tasks reliably, even if the DSP code disables interrupts. Interrupt disabling is used by many of the C6x signal processing libraries to allow software pipelining optimizations. On C674x processors this may for example cause issues with the McASP peripherals: if an McASP transmit buffer is not serviced in time and an underrun occurs, the McASP switches to error mode and from then on only sends zero data words. There is no way to recover except resetting and re-initializing the McASP. In systems having to cope with multiple external interrupt events such issues may occur only sporadically and can hardly be replicated with test cases for debugging. Problems arise at times when multiple interrupt events accumulate while a library function has simultaneously disabled the interrupt system.

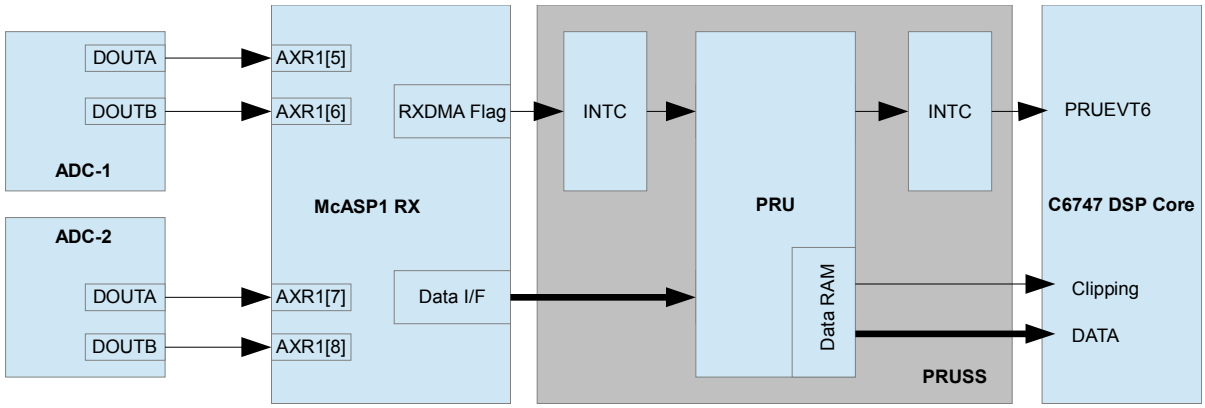
An RTOS (real-time operating system) does not cure this situation, it is locked out itself while interrupts are disabled. The use of DMA to service the McASP is a proven method to avoid buffer underruns. The C6747 EDMA controller operates independent of the DSP core and is not affected by interrupt locks. But DMA suffers from low efficiency if the transfer buffers are short, as it is often required by control and audio algorithms to minimize latency. Usually DMA is implemented with a ping-pong buffer scheme: while one buffer is processed by the DSP, the other is filled or unloaded by the DMA. The DSP program must always keep track which buffer to process.

The PRU can replace or complement the DMA and offer additional features like data sorting, ping-pong buffer management, apply offset calibration, clipping detection and more. The PRU may even provide watchdog features and gracefully shut down DAC outputs if a DSP core hang is detected. An additional advantage of PRU-driven McASP service shows during debug sessions: While the DSP is halted the PRU continues to transfer data to and from McASP and prevents buffer over- and underruns. Of course the data will not be updated during DSP halts, but as soon as execution is resumed the program will continue to work as expected. If the McASP is serviced directly by the DSP, an emulation halt will result in a buffer underrun and stop the McASP. If DMA is used, the buffer assignment may get lost during a processor halt.

UniDAQ2 PRU Usage

On the D.SignT UniDAQ2 data acquisition and processing system the PRU handles the following tasks:

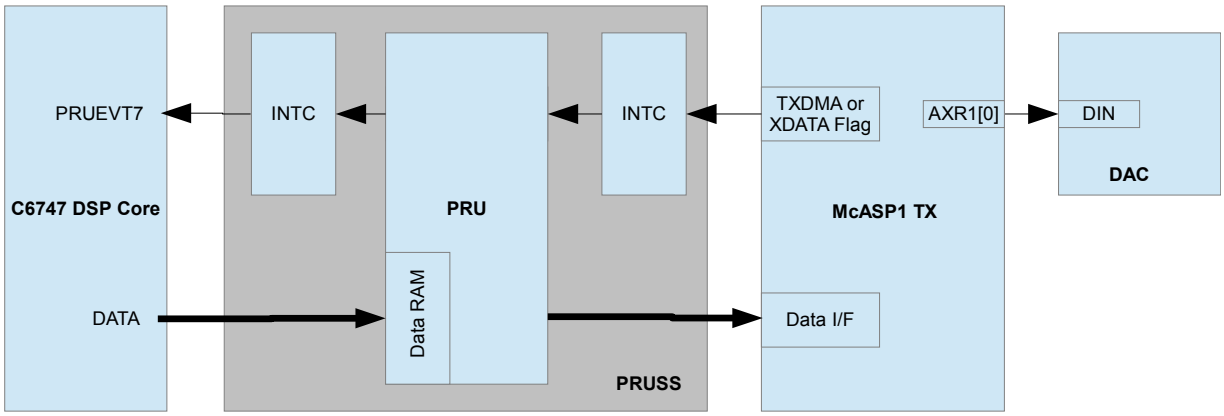
ADC



Two 8-channel A/D converters transmit their data via four McASP serializers. The data arrives in an unordered channel sequence: 1-5-9-13-2-6-10-14... The PRU reads the McASP receiver channels and stores them in the PRU data RAM, sorted in ascending channel order. An event is generated to inform the DSP about the new data set. The DSP now retrieves the ADC data from this fixed PRU memory buffer and does not need to care about ping-pong buffer management. The DSP is also free to read data in random order or read the ADC buffer only partially. Doing so in direct McASP reads would cause receive buffer overruns.

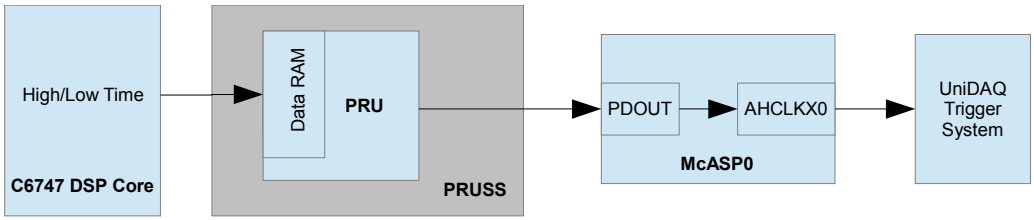
Additionally the PRU monitors the ADC channels for clipping and reports the results in a sticky register: clipping events remain set until cleared by the DSP. This is a useful feature to detect sensor faults, as an indicator to reduce the pre-amplifier gain in audio processing, and for control algorithms where clipping may cause loop instabilites. For block processing algorithms a DMA transfer can be triggered by the PRU event. The EDMA controller then transfers the data from PRU data RAM to a larger buffer memory and the EDMA data sorting capability can be fully exploited to format the data buffer in a manner suitable for the processing algorithm.

DAC



The 8-channel DAC supports two modes of operation: updates are either synchronized to a sampling clock or transparent, i.e. written on demand. In both cases the DSP interface is a buffer located in the PRU data memory. In synchronized mode an McASP transmit event causes the PRU to read the data buffer, reformat the data as required by the DAC serial protocol and transfer it to the McASP. The PRU then generates a notification event to the DSP or the EDMA controller to send new data. The DSP is not obliged to update all DACs as it is required on direct McASP access.In transparent mode the PRU continuously monitors the data buffer for changes.Each time one or more channels are changed, the PRU polls the McASP for a free transmit buffer and sends the new data with minimum delay. This mode assures the lowest latency for control loops and offloads the DSP from McASP transmitter polling or interrupts.

Clock Generation



Only one of the two PRU processors is used to handle ADCs and DACs. The other one is free for additional background tasks your application might benefit from. A typical example is adaptive ADC sampling clock generation for systems requiring coherent sampling, such as power line monitoring. The frequency resolution of the UniDAQ2 internal timers is limited by their 24 MHz master clock. The PRU however is clocked at 228 MHz and offers 9.5 times higher resolution. The PRU implements the high resolution timer by toggling the McASP AHCLKX0 pin, which is routable to the ADC conversion start in the UniDAQ2 trigger system.

Conclusion

Readily compiled PRU code is part of the UniDAQ2 board support package and easily integrated into application projects. Only three function calls are requried: PRU_init, PRU_load, and PRU_run. PRU usage guarantees in-time McASP service, facilitates a well-arranged programming interface, reduces the code size, ommitts the buffer housekeeping, adds clipping detection, and provides a high resolution programmable clock source for coherent sampling.

Appendix

UniDAQ2 loopback program using PRU

```
/*****************************************************************************
  includes
*****************************************************************************/
#include <BoardSupport/inc/unidaq2.h>       /**< Unidaq2-ADDA BIOS    */
#include <BoardSupport/inc/i2c1.h>          /**< Unidaq2-ADDA I2C     */
#include <BoardSupport/inc/adda.h>          /**< Unidaq2-ADDA ADC DAC */
#include <Common/pru.h>                     /**< PRU subsystem        */
#include <Common/pru/pru_adda_bin.h>        /**< PRU program code     */

/*****************************************************************************
  globals
*****************************************************************************/
volatile int32_t gNewDataFlag;


/*****************************************************************************//**
  @brief    PRU interrupt for ADC
  @param    -
  @return   -
*****************************************************************************/
__interrupt void adcInt (void)
{
    /*************************************************************************
      set "new data" flag
    *************************************************************************/
    gNewDataFlag = 1;
}


/*****************************************************************************//**
  main program
*****************************************************************************/
void main (void)
{
    /*************************************************************************
      locals
    *************************************************************************/
    PRU_addaOvly PRU_addaRegs = (PRU_addaOvly) PRU0_DRAM;
    int32_t idx;


    /*************************************************************************
      I2C Initialization
    *************************************************************************/
    I2C1_init (400000, 0x7E);

    /*************************************************************************
      Interrupt System Initialization
      ADCs and DACs operate synchonously, only the ADC int is required
    *************************************************************************/
    INT_init (&INT_vectorTable);
    INT_sel (15, INT_EVT_IIC1_INT);
    INT_instFunc (15, I2C1_int);
    INT_sel (4, INT_EVT_PRUEVT6);
    INT_instFunc (4, adcInt);
    INT_start();

    /*************************************************************************
      ADC Initialization:
        +/-5V range
        no oversampling
        sampling clock generated by Timer A, 100kHz
    *************************************************************************/
    ADC_config (RANGE_5, OS_NONE, ADCTRIG_TIMERA, 100000);

    /*************************************************************************
      DAC Initialization:
        +/-5V range
        synchronized to ADC
        no calibration
        no monitor output
    *************************************************************************/
    DAC_config (DAC_CFGREG_GAIN4, DACTRIG_ADC, 0, NULL, NULL, REF2V5, MON_OFF);
```

```
    /*********************************************************************
      enable PRU, load and execute PRU code
    *********************************************************************/
    PRU_init();
    PRU_load(0, (void *)PRUCode, sizeof(PRUCode));
    PRU_run(0);

    /*********************************************************************
      start data acquisition
    *********************************************************************/
    gNewDataFlag = 0;
    DAC_start();
    ADC_start();

    /*********************************************************************
      main program loop: move ADC data to DACs
    *********************************************************************/
    for (;;)
    {
        if (gNewDataFlag)
        {
            gNewDataFlag = 0;

            for (idx=0; idx<8; idx++) /* loopback ADC -> DAC */
            {
                PRU_addaRegs->dac[idx] = PRU_addaRegs->adc[idx];
            }

            if (PRU_addaRegs->clip) /* check ADC clipping bitfield */
            {
                /* handle clipping here, then clear the register */
                PRU_addaRegs->clip = 0;
            }
        }
    }
}
```

Disclaimer

This application note is provided to assist designers incorporating D.SignT products. The information is provided “as is” without any warranty for correctness or fitness for a specific application. The designer is obliged to perform his own analysis and assesments, to ensure compliance with applicable safety regulations and other requirements, and ensure no patents might be infringed.