TU Graz

Software Interface for the Configuration of an AD1835A Audio Codec on an ADSP-21369 Processor

Bachelor Thesis

by

Matthias Hotz

Graz University of Technology
Institute of Broadband Communications

Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Gernot Kubin
Advisor: Dipl.-Ing. Dr.techn. Werner Magnes

Graz, June 2010

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, June 28, 2010

Matthias Hotz

Abstract

During the laboratory "Digital Audio Engineering" the configuration of the AD1835A audio codec, itself connected to an ADSP-21369 signal processor, exhibited erratic behavior. The objective of this thesis has been to thoroughly analyze the involved hardware to develop from scratch a reliable, convenient and enhanced software interface for the configuration of the codec. From the physical connections on the printed circuit board over the utilized protocol and the configuration of the communication interface of the processor the entire framework required for the configuration of the codec is discussed in-depth. The process of configuration including all its particular characteristics is investigated and the foundations of the new software interface are exposed. Two exemplary applications, a volume control and an input level meter, depict the practical utilization of the software interface. Concluding, the acquired knowledge is applied to identify the weaknesses and causes of error of the code used during the laboratory.

Zusammenfassung

Während der Laborübung "Digitale Audiotechnik" zeigte sich der Konfigurationsvorgang für den über den Signalprozessor ADSP-21369 angesprochenen Audio-Codec AD1835A mehrfach als fehlerhaft. Ziel dieser Arbeit war die tiefgründige Analyse der dabei involvierten Hardware, um anschließend von Grund auf eine zuverlässige, komfortable und erweiterte Software-Schnittstelle für die Konfiguration des Codecs zu entwickeln. Es werden von den physikalischen Verbindungen auf der Leiterplatte über das verwendete Protokoll bis zur Konfiguration der Kommunikationsschnittstelle des Signalprozessors alle Voraussetzungen umfassend erörtert, der Vorgang zur Konfiguration des Codecs eingehend analysiert und die grundlegenden Elemente der neu entwickelten Software diskutiert. Abschließend wird die Verwendung der Software-Schnittstelle anhand zweier Anwendungen, einer Lautstärkeregelung und einer Aussteuerungsanzeige, demonstriert und das fundierte Wissen über die Konfiguration für die Analyse des Quellcodes der Laborübung herangezogen, um dessen Fehlerquellen aufzuzeigen.

Contents

1 Introduction **1**

2 Utilized Hardware and Objective of the Thesis **1**

2.1 ADSP-21369 Digital Signal Processor . . . 1
2.2 AD1835A Audio Codec . . . 2
2.3 ADSP-21369 EZ-KIT Lite® Evaluation Board . . . 4
2.4 Objective of the Thesis . . . 5

3 Essence of the Software Interface **5**

3.1 Connections on the Printed Circuit Board . . . 5
3.2 Signal Routing . . . 6
3.2.1 Architecture of the Serial Peripheral Interface . . . 6
3.2.2 Configuration of the Pin Buffers . . . 8
3.2.3 Routing Signals between the Pin Buffers and SPI Port . . . 10
3.3 SPI Port Configuration on the DSP . . . 10
3.3.1 SPI Control Register . . . 10
3.3.2 SPI Baud Rate Register . . . 11
3.3.3 SPI Port Flag Register . . . 12
3.3.4 Configuration Process . . . 12
3.4 SPI Communication . . . 13
3.5 Representation of the Configuration . . . 15
3.6 Configuration of the Codec . . . 17
3.7 Summary . . . 18

4 Applications **18**

4.1 Volume Control . . . 18
4.2 Input Level Meter . . . 20

5 Conclusion **21**

Bibliography **22**

Appendix A Excerpts of the AD1835A Data Sheet **23**

A.1 Data Frames associated with the Registers . . . 23
A.2 SPI Transfer Diagram . . . 24

Appendix B Excerpts of the ADSP-21369 Documentation **25**

B.1 SPI Transfer Diagram . . . 25
B.2 Clock Relationship to the Input Clock . . . 25

Appendix C Source Code **25**

C.1 Embedding the Software Interface into a Project . . . 25
C.2 Signal Routing, SPI Configuration and Communication . . . 26
C.2.1 `spi.h` . . . 26

C.2.2 `spi.asm` 27
C.3 Codec Configuration Representation and Modification 31
C.3.1 `ad1835a.h` 31
C.3.2 `ad1835a.asm` 37
C.4 LED Routing and Control 39
C.4.1 `led.h` 39
C.4.2 `led.asm` 40

List of Figures

1 ADSP-21369 SHARC® processor block diagram 2
2 AD1835A functional block diagram 3
3 ADSP-21369 EZ-KIT Lite® system architecture block diagram 4
4 Objective of the thesis — from the hardware to the configuration 5
5 DPI system design — pins, signal routing unit and interfaces 7
6 SPI bus architecture 8
7 Schematic diagram of the pin buffer 8
8 Internal structure of the primary SPI port in core-driven master mode 14
9 Structure of a data frame for configuration 16
10 Control register map 23
11 Data frame: DAC Control 1 23
12 Data frame: DAC Control 2 23
13 Data frame: DAC Volume Control 23
14 Data frame: ADC Peak 23
15 Data frame: ADC Control 1 24
16 Data frame: ADC Control 2 24
17 Data frame: ADC Control 3 24
18 SPI transfer diagram of the AD1835A control port 24
19 SPI transfer diagram for `CPHASE = 0` 25
20 Clock relationship to the input clock 25

List of Tables

1 Configurable features of the AD1835A codec 3
2 Physical connections on the evaluation board 6
3 SPI bus signal naming convention 6
4 SPI bus signals of the AD1835A control port 8
5 Function table of the pin buffer enable state 9
6 Structure of the configuration buffer 16

Listings

1 Syntax of the signal routing macro provided by Analog Devices 9
2 Configuration of the pin buffers 9
3 Routing of the SPI signals 10
4 Configuration of the primary SPI port of the DSP 13
5 Transmission of a data word 13
6 Waiting for SPI transfer completion 14
7 Realization of the minimum wait time between successive transfers 15
8 Receiving a data word via SPI with transfer initiation mode `TIMOD = 01` . . . 15
9 Transmission of the content of the configuration buffer to the codec 17
10 Configuration of the interrupt inputs $\overline{\texttt{IRQ0}}$ and $\overline{\texttt{IRQ1}}$ 19
11 Interrupt service routine for the push button `PB1` 19
12 Routing of the LEDs and configuration of the timer 20
13 Interrupt service routine for the timer interrupt 21

1 Introduction

The information age brought an omnipresent need for computing power. *Digital signal processors* (DSP) play an important role in satisfying those needs, being the driving force in many devices such as video players, audio devices and mobile phones.

At Graz University of Technology, a laboratory on digital signal processors, "Digital Audio Engineering"[1] (DAL), provides a first insight into this fascinating technology. During the course the processor *ADSP-21369* from Analog Devices is studied and put into operation using the associated evaluation board *ADSP-21369 EZ-KIT Lite®*. The primary focus of the course is to familiarize the student with the general architecture, arithmetic operations and elements of flow control to subsequently implement some audio applications. Due to the complexity of the hardware its detailed configuration is not considered.

The evaluation board features the audio codec chip *AD1835A* from Analog Devices which is the bridge between the analog audio signals and the digital world of the DSP. During the course some problems regarding the configuration of the codec chip emerged, occasionally configured settings did not take effect.

This circumstance offered a motivation and an opportunity to dive into the depths of the involved hardware, given the aim for this thesis to rework and expand the configuration of the codec while supplementing the gained knowledge during the course with a deeper understanding of the hardware.

2 Utilized Hardware and Objective of the Thesis

The definition of the objective, the configuration of the codec chip AD1835A, may sound rather vague. In order to put it more concretely this chapter provides an introduction to the involved hardware. Starting point is the signal processor, followed by the codec and the evaluation board. Finally, this information is put together to formulate a tangible definition of the established aim.

2.1 ADSP-21369 Digital Signal Processor

The ADSP-21369, in the remainder of this thesis referred to as DSP, is a high performance floating-point processor with a 400 MHz core instruction rate featuring the Super Harvard Architecture (*SHARC®*) of Analog Devices which provides a separate program and data memory bus[2]. The DSP possesses a single-instruction, multiple-data (SIMD) computational architecture with two processing elements (PE) supporting 32-bit fixed-point and 32-bit/40-bit floating-point operations. It provides 2 Mbit SRAM and 6 Mbit ROM on-chip memory and has a very rich interface to communicate with peripheral devices including an S/PDIF transceiver, serial ports (SPORT), Universal Asynchronous Receiver Transmitters (UART) and Serial Peripheral Interfaces (SPI) amongst others [5].

Figure 1[3] depicts a block diagram of the DSP where the relevant parts are displayed in black. By reason of the complexity of this processor only the blocks which are fundamental to the problem at hand are further considered. It is needless to say that the core processor, the on-chip memory and the program and data bus are essential for program execution, but as they covered by the DAL course mentioned in Chapter 1 they are not considered here. Besides the course, continuative information can be found in [5, 7, 9].

Signal processing applications often need the processor to extensively communicate with off-chip devices. To disburden the core processor the ADSP-21369 contains an *I/O-processor*

[1] Course number 441.055, taught by DI Dr. Werner Magnes and DI David Fischer.
[2] The program memory bus may also be used for data transfers [7, ch. 1].
[3] This diagram was taken from a past revision of the programming reference as, in the opinion of the author, it provides a more comprehensive overview than the diagram contained in the current revision.

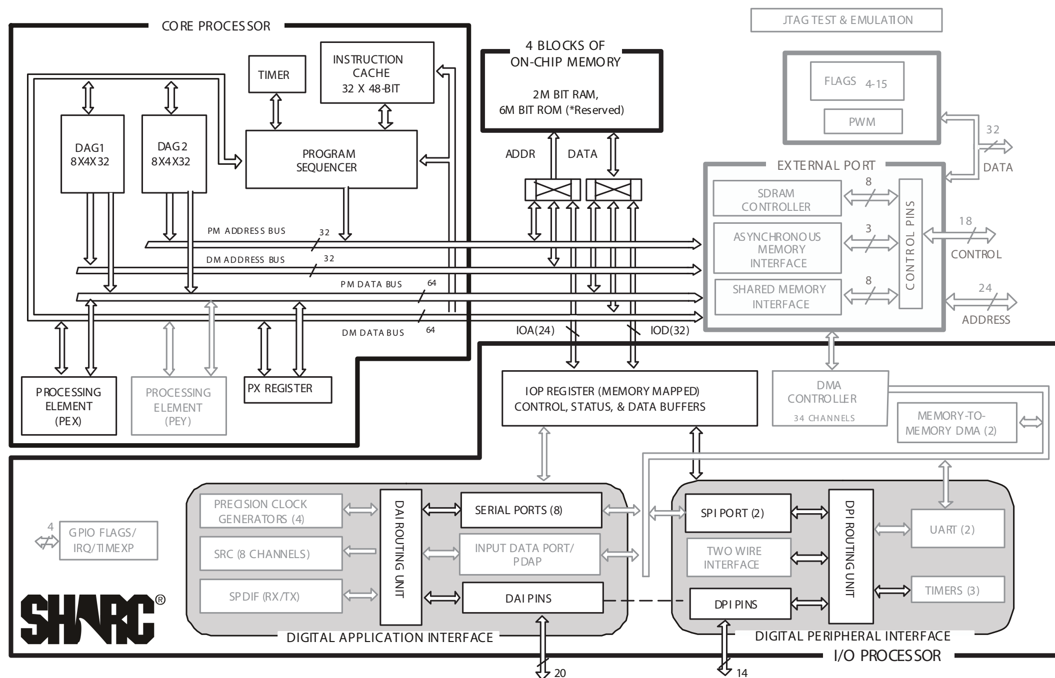


Figure 1: ADSP-21369 SHARC® processor block diagram [3, p. 1-4]

that handles data transfers to peripherals managing all details of the communication. The I/O processor includes several direct memory access (DMA) channels which enable direct access to the memory without involvement of the core processor. The offered interfaces are divided into two groups named the *digital application interface* (DAI) and the *digital peripheral interface* (DPI), each having a specific number of assigned physical pins. These pins are connected to an individual *signal routing unit* (SRU) for each group which allows a flexible assignment of the physical pins to the interfaces (cf. Figure 1) [9, ch. 2 and 6].

Three serial ports of the DAI are used to receive data from the *analog-to-digital converters* (ADC) and send data to the *digital-to-analog converters* (DAC) of the AD1835A audio codec. Anyway, this topic is beyond scope and will not be discussed. More information on the serial ports of the DSP is available in [9, ch. 7].

The configuration of the AD1835A, being the focus of this thesis, is performed using a *Serial Peripheral Interface* (SPI) port of the DPI. SPI is a bus system that was introduced by Motorola with the MC68HC11 microcontroller. Unfortunately, there is no official SPI standard but, however, the original reference manual of the MC68HC11 might serve as a specification while it is hard to find. The current revision of the manual still contains the chapter on the SPI and is available from Motorola's spin-off Freescale Semiconductors [10, ch. 8]. Before more details about the SPI port and DPI are revealed in Chapter 3, the audio codec and evaluation board are introduced.

2.2 AD1835A Audio Codec

The audio codec chip AD1835A provides a conversion between analog and digital audio signals through one stereo Σ–Δ ADC and four stereo Σ–Δ DACs which are accessed via *serial ports*. The converters can operate at different sample rates and word lengths and many additional features are offered. The key features of this codec chip are summarized in Table 1. All of these features are configurable through a dedicated *control port* realized as SPI [1].

Figure 2 depicts a functional block diagram of the AD1835A where all blocks affected by

ADC Features	**DAC Features**
16-bit, 20-bit and 24-bit word length	
48 kHz and 96 kHz sample rate	48 kHz, 96 kHz and 192 kHz sample rate (192 kHz: Only one DAC)
Clickless mute for every channel	
8 different serial data output modes	6 different serial data input modes
Power-down mode	
Optional digital high-pass filter	Optional de-emphasis filter at 32.0 kHz, 44.1 kHz or 48 kHz
Peak level information	1024-step linear volume control for every channel

Table 1: Configurable features of the AD1835A codec

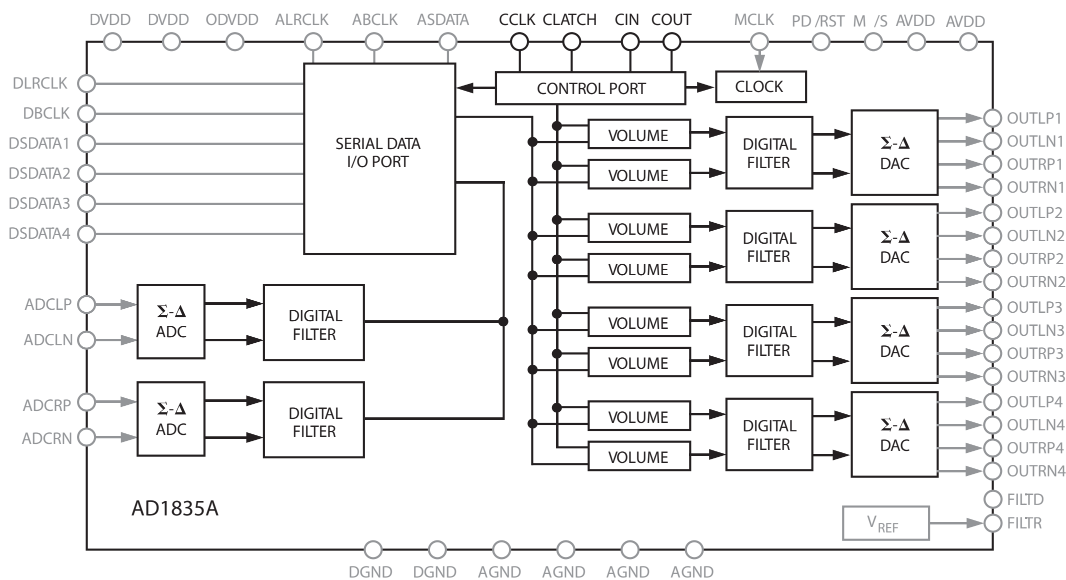


Figure 2: AD1835A functional block diagram [1, p. 1]

the configuration are colored black.

The AD1835A is configured by the content of a set of 10 bit wide control registers, i.e. three registers for DAC related settings, three registers for ADC related settings and eight registers for the DAC output volume settings. Additionally, two 6 bit wide read-only status registers provide information about the peak input level of the left and right ADC if peak readback is enabled. In order to write to a control register or read from a peak level register a 16 bit data word is sent to the codec via the control port. The interface of the control port complies to the SPI specification and consists of four wires, serial clock (`CCLK`), device select (`CLATCH`), data input (`CIN`) and data output (`COUT`) visible in Figure 2.

Further details about the AD1835A codec regarding its configuration will be introduced in Chapter 3, whereas the next section presents the evaluation board which connects the AD1835A to the DSP.

2.3 ADSP-21369 EZ-KIT Lite® Evaluation Board

The ADSP-21369 EZ-KIT Lite® evaluation board, in the remainder of this thesis named "evaluation board", is a printed circuit board designed to provide a cost-efficient method to evaluate the ADSP-21369 signal processor. Figure 3 depicts a block diagram of its system architecture where the significant blocks are emphasized. The core of the evaluation board is the ADSP-21369 to which different interfaces, connectors and various types of memory are connected to enable a comprehensive assessment of the diverse features of the DSP [6].

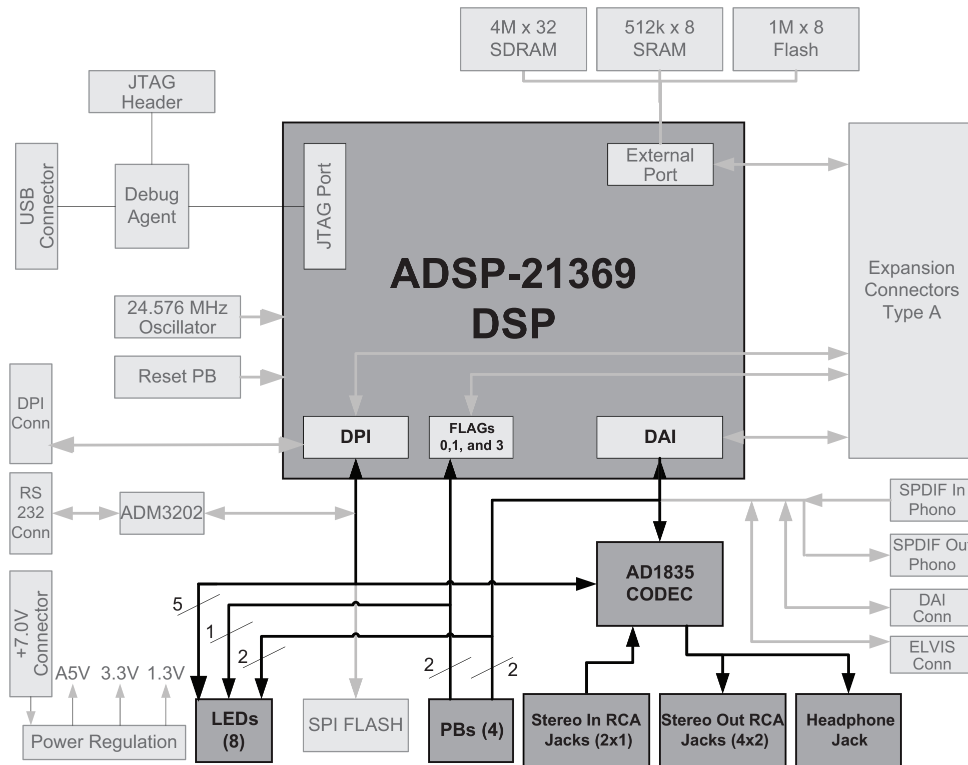


Figure 3: ADSP-21369 EZ-KIT Lite® system architecture block diagram [6, p. 2-2]

The audio codec chip AD1835A introduced in Section 2.2 is part of the evaluation board as well. While the serial ports of the codec are connected to the DAI of the DSP, the control port of the codec is connected to the DSP's DPI. Connection to the inputs and outputs of the codec is provided by cinch (RCA) connectors. Additionally, one stereo DAC (DAC 4) is connected to a 3.5 mm TRS connector for use with headphones.

The evaluation board offers eight general-purpose light-emitting diodes (LED) and four general-purpose push buttons connected to the DAI and DPI of the DSP. They will be used in Chapter 4 to demonstrate the capabilities of the configuration routines.

Analog Devices offers a special integrated software development and debugging environment (IDDE) named *VisualDSP++* for software development in C/C++ and assembler for Analog Devices' signal processors[4]. Through the Universal Serial Bus (USB) of the personal computer and the debug agent on the evaluation board, VisualDSP++ is connected to the ADSP-21369 and provides convenient methods to download and test programs on the DSP. Furthermore, libraries and example programs are included [4].

[4]The ADSP-21369 EZ-KIT Lite® includes an evaluation suite of VisualDSP++.

2.4 Objective of the Thesis

Chapter 1 mentioned that the code[5] used to configure the AD1835A codec during the DAL course frequently caused problems by means of settings not taking effect. This erratic behavior was best noticed when DAC volumes were changed.

The objective of this thesis was to rebuild the whole software from scratch which is involved in the configuration of the AD1835A codec, i.e. all software required to alter any setting of the codec including the necessary communication framework. Based on the information about the hardware previously communicated in this chapter, the goal can be concretized as illustrated in Figure 4.

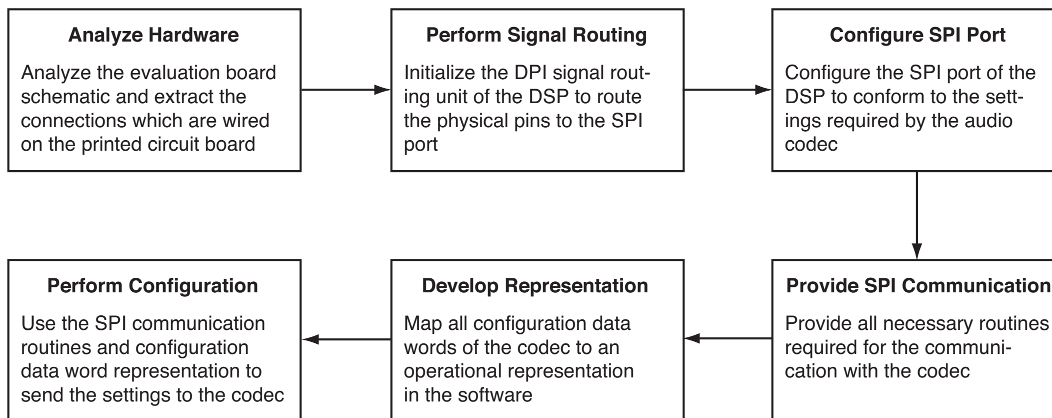


Figure 4: Objective of the thesis — from the hardware to the configuration

3 Essence of the Software Interface

The hardware overview of Chapter 2 was utilized to concretize the goal, yielding the road map for the process towards an implementation of the software interface depicted in Figure 4. In this chapter the implementation of the software interface for the configuration of the AD1835A codec will be discussed, following the aforementioned road map.

The implementation will be done using the assembler language of the DSP, being predestined and illustrative for this task since it is tied very closely to the hardware of the DSP. However, a description of the assembler language would be beyond scope of this thesis. More information about the instruction set of the ADSP-21369 is available in [7, ch. 9 – 11], programming in assembler using VisualDSP++ is described in [8].

3.1 Connections on the Printed Circuit Board

As it was mentioned in Chapter 2 the physical pins of the DSP are not connected directly to its internal SPI port. Instead, the physical pins are connected to a signal routing unit (SRU) that can be configured to route the signals between the physical pins and the internal interfaces, e.g. an SPI port. In order to perform the routing the physical pins of the DSP which are conntected to the control port of the AD1835A have to be determined.

The schematic of the evaluation board is available in [6, app. B]. The AD1835A audio codec chip (`U31`) is shown on sheet 5 of the schematic. By tracing the signal labels of the AD1835A's pins `CIN`, `COUT`, `CCLK` and `CLATCH`[6] to the ADSP-21369 (`U44`) on sheet 2 of the schematic, the physical pins of the DSP are investigated. Using the pin assignment table from the DSP's

[5] This code was extracted from one of Analog Devices' example programs included with VisualDSP++.
[6] `CLATCH` is connected through switch `SW15` which is assumed to be in its on-position.

data sheet [5, p. 51] the names of the pins in Analog Devices' software library are uncovered. These observations are summarized in Table 2.

Signal of AD1835A	Physical pin at ADSP-21369	Pin's name in software library
CIN	B15	DPI_P01
COUT	A16	DPI_P02
CCLK	A15	DPI_P03
CLATCH	B14	DPI_P04

Table 2: Physical connections on the evaluation board

3.2 Signal Routing

The overview of the hardware in Chapter 2 revealed that the AD1835A codec offers an SPI port, i.e. the control port for its configuration. The DSP contains two SPI ports, entitled primary (SPI) and secondary (SPIB) SPI port, whereas the *primary SPI port* is used for communication with the codec. The SPI ports are connected to the signal routing unit of the DPI, named SRU2, as depicted in Figure 5[7].

In order to enable the configuration of the AD1835A, the control port of the codec, wired to the DPI pins of the DSP, and the primary SPI port of the DSP need to be connected through the SRU. Routing a physical DPI pin to a "virtual" pin of the primary SPI port by means of the SRU requires the

- configuration of the pin buffer of the physical pin and
- routing of the internal pin buffer connectors to the virtual pin of the SPI port.

These steps depend on architectural information about the SPI. Therefore, the architecture of the SPI is discussed beforehand, followed by the configuration of the pin buffers and the routing to the primary SPI port.

3.2.1 Architecture of the Serial Peripheral Interface

The SPI is a *bus system*[8] enabling serial data transfer between a *master* and one or more *slave* devices. The master controls and initiates all transfers, whereas the addressed slave device, chosen via a dedicated select signal, only responds to the master.The physical connection between the master and slave devices consists of four wires specified in Table 3. Figure 6a illustrates the wiring of an SPI bus consisting of a master and three slave devices.

Signal	Full name
SCLK or CLK	(Serial) Clock
MOSI	Master Output, Slave Input
MISO	Master Input, Slave Output
SS or DS	Slave Select or Device Select

Table 3: SPI bus signal naming convention

[7]This diagram was taken from a past revision of the hardware reference as, in the opinion of the author, it provides a more comprehensive overview than the depictions contained in the current revision.

[8]Only the typical SPI bus wiring is considered since "daisy-chaining" of slave devices is not relevant here.

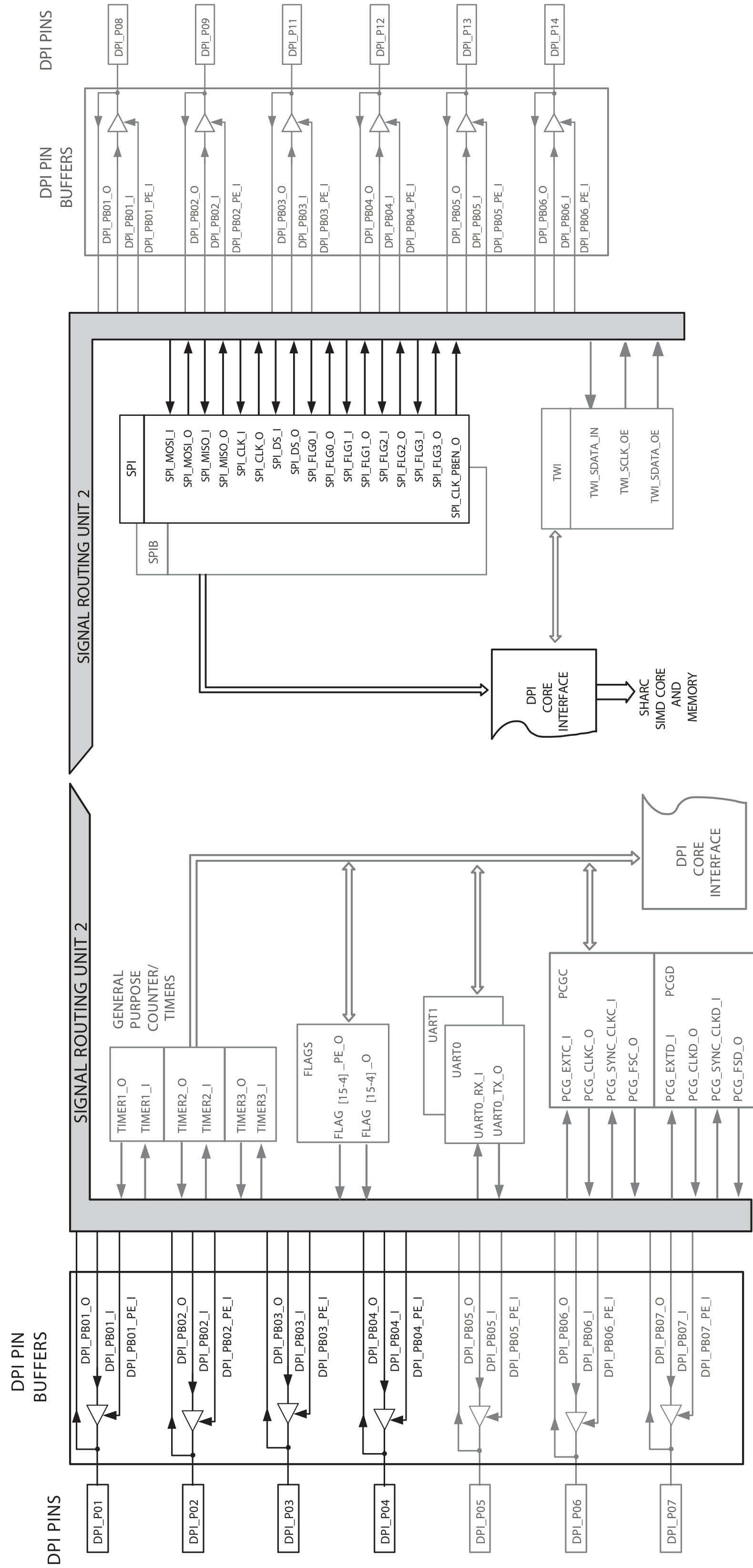
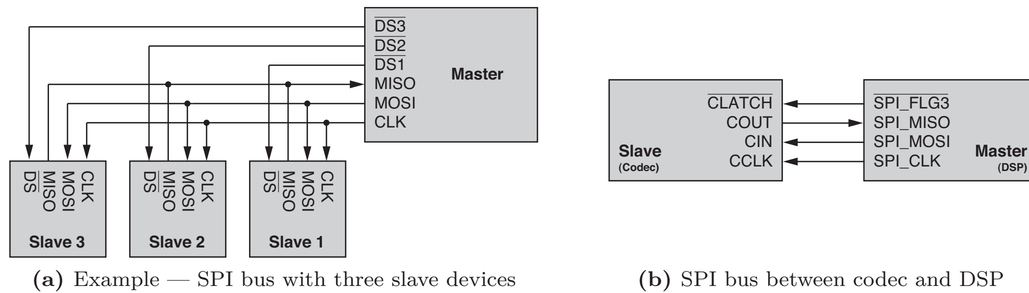


Figure 5: DPI system design — pins, signal routing unit and interfaces [2, p. 4-6 and 4-7]

Control port signal name	Conventional signal name	Description
`CCLK`	`CLK`	Clock Input
`CIN`	`MOSI`	Slave Input
`COUT`	`MISO`	Slave Output
`CLATCH`	`DS`	Device Select Input

Table 4: SPI bus signals of the AD1835A control port



(a) Example — SPI bus with three slave devices **(b)** SPI bus between codec and DSP

Figure 6: SPI bus architecture

At the SPI bus for the configuration of the codec, the DSP acts as master and the AD1835A is a slave[9]. The SPI port of the DSP mostly follows the naming convention of Table 3, except the four available the slave select outputs are named "flag" [9, p 12-3]. The control port of the codec, however, uses completely different names. Based on the information from the data sheet of the AD1835A [1, p. 12], the correspondance to the conventional signal names is established in Table 4. Figure 6b summarizes the determined SPI bus connections.

3.2.2 Configuration of the Pin Buffers

Before the information about the bus architecture is utilized, another concept needs to be introduced. Within the context of the SRU a physical DPI pin is replaced by a logical interface named *pin buffer*. Its internal interface to the SRU consists of three connectors, enable (`PBENxx_I`), input (`DPI_PBxx_I`) and output (`DPI_PBxx_O`), as depicted in Figure 7. Depending

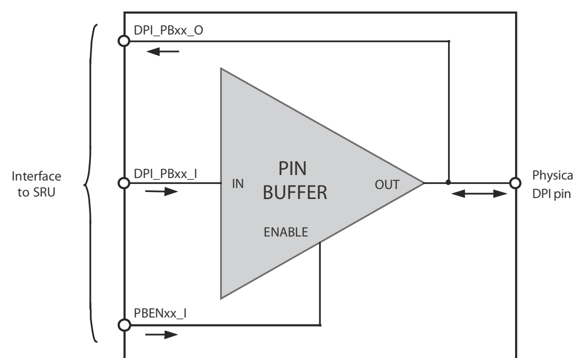


Figure 7: Schematic diagram of the pin buffer [9, p. 6-8]

[9]The SPI flash memory shown in Figure 3 is also connected to this SPI bus as slave device. This was neglected to avoid unnecessary complexity.

on the logic level at the enable connector, the physical pin acts as input or output as shown in Table 5. It might be confusing that, for example, the output `DPI_PBxx_O` yields the state of the physical pin if it is configured as input. This naming convention becomes clear when the flow of information in reference to the pin buffer is considered; the pin buffer *outputs* the state of the physical pin to the SRU.

Pin enable (`PBENxx_I`)	Physical pin function	Active pin buffer connector
`HIGH`	Output	`DPI_PBxx_I`
`LOW`	Input	`DPI_PBxx_O`

Table 5: Function table of the pin buffer enable state

This simplified description of pin buffers, where its mode of operation is "hard-wired", is sufficient for the problem at hand. More information on the function of the pin buffers, e.g. how to use the the output signal of an interface to control the enable signal of a pin buffer, is available in [9, ch. 6].

The principle behind the configuration of the SRU, and this includes the pin buffers, is that each output has an assigned identifier and every input a dedicated configuration register[10].In order to connect an output to an input, the identifier of the output has to be written to the configuration register of the input. This task is supported by the macro `SRU` provided by Analog Devices with VisualDSP++ in `SRU.H` [9, p. 6-42 ff.].

Listing 1: Syntax of the signal routing macro provided by Analog Devices

```
1 SRU(Output_Signal, Input_Signal);
```

The signal names follow the naming convention

`PERIPHERAL_FUNCTION_DIRECTION`

where e.g. "peripheral" is `SPI`, "function" is `CLK` and "direction" is `O` for output [9, p. 6-7].

Finally, the information gathered since the beginning of this chapter is combined. Table 2 specifies which physical pins of the DAI need to be configured and Figure 6b shows the direction of data flow for each pin. Considering the naming convention for pin buffers visible in Figure 7 and the function of the enable signal depicted in Table 5, the code to configure the pin buffers using the `SRU`-macro from above can be derived.

Listing 2: Configuration of the pin buffers

```
1  // DPI_P01: Output (MOSI)
2  SRU(HIGH, DPI_PBEN01_I);
3
4  // DPI_P02: Input (MISO)
5  SRU(LOW, DPI_PBEN02_I);
6
7  // DPI_P03: Output (Clock)
8  SRU(HIGH, DPI_PBEN03_I);
9
10 // DPI_P04: Output (Device select for AD1835A)
11 SRU(HIGH, DPI_PBEN04_I);
```

[10] Additionally, the inputs and outputs are arranged in groups [9, p. 6-16 ff.]. This will not be considered to avoid unnecessary complexity.

3.2.3 Routing Signals between the Pin Buffers and SPI Port

Figure 5 illustrates that by configuring the pin buffers the physical pin is made available to the “core” of the signal routing unit. All that is left to connect the control port of the codec to the DSP is to route the signals from the pin buffers to the primary SPI port of the DPI. The involved signals of the SPI port of the DSP are depicted in Figure 6b. Using the information on pin buffers and the `SRU`-macro from the previous section, the code to route the signals can be composed.

Listing 3: Routing of the SPI signals

```
// MOSI: SPI_MOSI_O -> DPI_P01 -> CIN
SRU(SPI_MOSI_O, DPI_PB01_I);

// MISO: COUT -> DPI_P02 -> SPI_MISO_I
SRU(DPI_PB02_O, SPI_MISO_I);

// CLK: SPI_CLK_O -> DPI_P03 -> CCLK
SRU(SPI_CLK_O, DPI_PB03_I);

// DS: SPI_FLG3_O -> DPI_P04 -> CLATCH
SRU(SPI_FLG3_O, DPI_PB04_I);
```

3.3 SPI Port Configuration on the DSP

Previously the connection between the DSP’s primary SPI port and the codec’s control port was established. In addition to the bus architecture the SPI specification defines three attributes for a connection, the *clock rate*, *word length* and *mode*. Therefore, the SPI port of the DSP needs to be configured to conform to the settings required by the codec. Additionally, some DSP–related settings of the SPI port have to be set. The complete configuration of the primary SPI port is described by the content of three registers, the *SPI control register*, *SPI baud rate register* and *SPI port flag register*. In the following the settings provided by these registers are disussed and eventually the code for the configuration is presented.

3.3.1 SPI Control Register

The main part of the configuration of the primary SPI port is accessible through the SPI control register `SPICTL` [9, p. A-140 ff.]. It consists of 18 different settings why only the essential ones are discussed.

Each setting of the register has a short name. Along with VisualDSP++ Analog Devices provides an ADSP-21369 specific header file `def21369.h` that defines constants of the same name to be used for configuration. These short names will be introduced along with the description of each setting for the code at the end of the section.

SPI Port Enable If the bit `SPIEN` is set, the SPI port is enabled, otherwise disabled.

SPI Master Select As mentioned in Section 3.2.1 the DSP has to act as master on the SPI bus. Setting the `SPIMS` bit configures the primary SPI port as master.

Word Length In order to modify the content of a control register of the codec a 16 bit data word has to be sent to its control port as described in Section 2.2. Therefore, the 2 bit wide SPI bus word length code `WL` in `SPICTL` has to be set to `WL16`, specifying 16 bit word length.

Clock Polarity and Clock Phase The SPI specification defines four modes by means of the two-valued settings *clock polarity* and *clock phase*. The clock polarity specifies whether the clock signal is active-low or active-high and the clock phase defines when the data is put on the bus. Figure 19 in Appendix B provides the SPI transfer diagram for clock phase `CPHASE = 0` and clock polarity `CLKPL = 0` and `CLKPL = 1`. By comparision with the SPI transfer diagram of the AD1835A depicted in Figure 18 in Appendix A, the `SPICTL` settings `CLKPL = 0` and `CPHASE = 0` corresponding to SPI mode 0 are investigated[11]. Thus, both bits, `CLKPL` and `CPHASE`, are *not* set.

Most Significant Byte First The setting `MSBF` specifies if the most significant byte of the data word is sent first. Figure 18 in Appendix A depicts that the control port of the codec assumes that the byte order is big-endian, thus this bit has to be set.

Transfer Initiation Mode Basically there are two alternatives to manage the SPI communication, direct memory access (DMA) provided by the I/O processor and core driven operation. As the configuration of the codec only requires the occasional transfer of a handful of data words, DMA is not purposeful due to the additional configuration overhead and negligible unloading of the core processor because of the small amount of transmitted data. Therefore, core driven operation is used and DMA is not discussed further.

The transfer initiation mode setting `TIMOD` consists of two bits, whereas the more significant bit selects between DMA (`TIMOD2`) and core driven (`0`) operation and the second bit defines how a transfer is initiated in the core driven mode, by a request to send (`TIMOD1`) or receive (`0`) a data word[12]. The control registers of the codec are modified by sending data words to its control port, therefore, the `TIMOD1` mode is suited best for this task.

3.3.2 SPI Baud Rate Register

The SPI baud rate register `SPIBAUD` configures the clock rate of the SPI bus [9, p. A-148]. The bits 15 – 1 of `SPIBAUD` are named `BAUDR` and specify the divisor by which the SPI baud rate f_{SPI} is related to the peripheral clock rate f_{PCLK} of the I/O processor. In master mode this relation is given by Equation (1) [9, p. 12-6].

$$f_{\mathsf{SPI}} = \frac{f_{\mathsf{PCLK}}}{8 \cdot \mathsf{BAUDR}} \tag{1}$$

In order to determine `BAUDR` for a specific SPI baud rate f_{PCLK} needs to be known. The peripheral clock rate f_{PCLK} is half the core clock rate f_{CCLK}, whereas the core clock is the output of the phase-locked loop (PLL) (cf. Figure 20 in Appendix B).

$$f_{\mathsf{PCLK}} = \frac{f_{\mathsf{CCLK}}}{2} \tag{2}$$

The PLL is controlled via the power management control register `PMCTL` where `INDIV` configures the input pre divider, `PLLM` the multiplier and `PLLD` the post divider [9, ch. 16]. The details of the PLL configuration are beyond scope, thus only the necessary setting values for the calculation of f_{CCLK} are considered. However, the PLL is configured with `PLLM = 27`, `PLLD = 2` and `INDIV = 0` which deactivates input pre divisor. The input clock rate f_{CLKIN} of the DSP is 24.576 MHz as shown on sheet 2 of the evaluation board schematic [6, app. B]. Utilizing this information and applying the equation for the core clock rate given in [5, p. 19] (with $f_{\mathsf{CLKIN}} = f_{\mathrm{INPUT}}$) yields f_{CCLK} as depicted in Equation (3).

$$f_{\mathsf{CCLK}} = f_{\mathsf{CLKIN}} \cdot \frac{\mathsf{PLLM}}{\mathsf{PLLD}} = 24.576\,\mathrm{MHz} \cdot \frac{27}{2} = 331.776\,\mathrm{MHz} \tag{3}$$

[11] More information about the SPI transfer modes is found in [10, ch. 8] and [9, ch. 12].
[12] Details on how the sending or receiving of a data word is requested is discussed in Section 3.4.

By substituting Eq. (2) in Eq. (1), BAUDR can be expressed in dependence of f_{SPI} and f_{CCLK}. Considering that f_{SPI} specifies an upper limit for the SPI baud rate the relation in Equation (4) is derived.

$$f_{\mathsf{SPI}} > \frac{f_{\mathsf{PCLK}}}{8 \cdot \mathsf{BAUDR}} = \frac{f_{\mathsf{CCLK}}}{16 \cdot \mathsf{BAUDR}} \qquad \Rightarrow \qquad \mathsf{BAUDR} > \frac{f_{\mathsf{CCLK}}}{16 \cdot f_{\mathsf{SPI}}} \tag{4}$$

In the data sheet of the AD1835A a maximum serial bit clock frequency ($\rightarrow f_{\mathsf{SPI}}$) of 12.5 MHz is specified for the control port [1, p. 13]. With Eq. (3) and Eq. (4) the valid range of values for BAUDR is determined.

$$\mathsf{BAUDR} > \frac{f_{\mathsf{CCLK}}}{16 \cdot f_{\mathsf{SPI}}} = \frac{331.776\,\mathrm{MHz}}{16 \cdot 12.5\,\mathrm{MHz}} = 1.66 \qquad \Rightarrow \qquad \mathsf{BAUDR} \geq 2 \tag{5}$$

An additional "safety margin" is added and the value $(4)_{\mathrm{dec}} = (4)_{\mathrm{hex}}$ is chosen for BAUDR. Considering that BAUDR begins at bit 1, the value $(8)_{\mathrm{hex}}$ is written to SPIBAUD.

3.3.3 SPI Port Flag Register

In Section 3.2.1 it was already mentioned that the device select signals are called "flags" in the context of the DSP's SPI port. The flags of the primary SPI port are controlled by the SPI port flag register SPIFLG, where each of the four flags has an assigned device select enable (DS0EN – DS3EN) and device select control bit (SPIFLG0 – SPIFLG3) [9, p. A-150 f.]. Therefore, e.g. to enable the SPI port signal SPI_FLG3_O (cf. Figure 5), i.e. the device select output "flag 3", the bit DS3EN in SPIFLG has to be set. The state of this device select signal is then controlled by the bit SPIFLG3. Considering that the device select signal is active low, the slave is selected when the SPIFLG3 bit is cleared.

If CPHASE = 0 in SPICTL, which is the case in this configuration, all enabled device select signals are controlled by the internal SPI hardware of the DSP. If a SPI transfer is initiated, independent of the SPIFLGx bit all enabled device select signals are pulled low for the duration of the transfer [9, p. 12-15 ff.].

Figure 6b depicts that flag 3 is used as device select signal for the AD1835A codec, consequently, DS3EN needs to be set in SPIFLG. Additionally, for the sake of formality all SPIFLGx bits are set to achieve a save initial state.

3.3.4 Configuration Process

Figure 1 shows that the registers of the I/O processor are memory-mapped, thus the previously discussed SPI port configuration registers are accessed via the data memory bus. Before performing the configuration it has to be considered that the SPI configuration may only be changed safely when some preconditions are fulfilled [9, p. 12-22].

1. There must not be any data transfer active.
2. No slaves may be selected.
3. The SPI port is disabled.

The means to check if a transfer is active and block execution until its completion are presented in Section 3.4 and, therefore, it is only noted in the code shown below. How the other prerequisites are accomplished was already implicitly mentioned. The device select signals are disabled by clearing the DSxEN bits in SPIFLG and the SPI port is disabled by clearing the SPIEN bit in SPICTL[13]. The code of the complete configuration incorporating all

[13] When the SPI port is disabled the transmit and receive buffer is cleared, leading to a defined basis for subsequent transfers.

discussed settings and aspects of the execution sequence is listed below. Attention should be paid to the modification of the SPI control register, enabling and disabling the SPI port has to be separated from changing other settings in `SPICTL`.

Listing 4: Configuration of the primary SPI port of the DSP

```
// Wait for active data transfer to finish
//    --> Discussed in the next section

// Disable SPI port: Clear SPIEN bit
r0 = dm(SPICTL);
r1 = ~SPIEN;
r0 = r0 and r1;
dm(SPICTL) = r0;

// Set the SPI baud rate
r0 = 0x8;
dm(SPIBAUD) = r0;

// Disable all slave select lines (Clear DSxEN and set SPIFLGx)
r0 = SPIFLG0 | SPIFLG1 | SPIFLG2 | SPIFLG3;
dm(SPIFLG) = r0;

// Configure the primary SPI port
r0 = TIMOD1 | // Trigger transfer by write request
     WL16   | // 16 bit word length
     MSBF   | // Send most significant byte first
     SPIMS;   // Set to SPI master
dm(SPICTL) = r0;

// Enable the device select signal for the AD1835A
r0 = DS3EN | SPIFLG0 | SPIFLG1 | SPIFLG2 | SPIFLG3;
dm(SPIFLG) = r0;

// Enable the SPI port: Set SPIEN bit
r0 = dm(SPICTL);
r1 = SPIEN;
r0 = r0 or r1;
dm(SPICTL) = r0;
```

3.4 SPI Communication

The SPI port is routed, configured and, consequently, ready for data transfers. In order to initiate a transfer, an insight into the internal structure of the SPI port is necessary.

The primary SPI port contains a transmit and a receive *shift register* which are *not* directly accessible. These registers serially transmit, respectively receive data synchronously with the SPI clock signal. A shift register is written, respectively read by the associated *transmit data buffer* `TXSPI` and *receive data buffer* `RXSPI` via the data memory bus as depicted in Figure 8.

In Section 3.3 the transfer initiation mode was set to `TIMOD = 01`, defining core-driven operation of the SPI port whereas a transfer is initiated by the request to send a data word. A request to send a data word is deposited by simply writing the data word into the transmit data buffer `TXSPI`.

Listing 5: Transmission of a data word

```
// Example: Transmit the data word 0x1234
r0 = 0x1234;
dm(TXSPI) = r0;
```

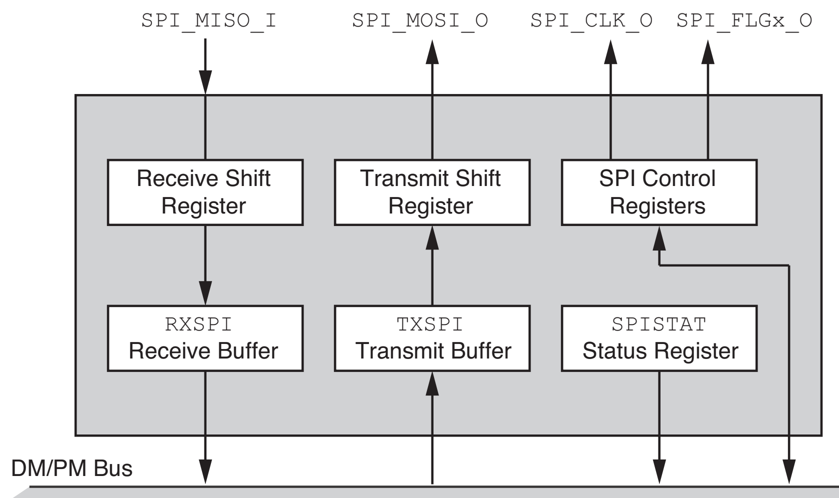


Figure 8: Internal structure of the primary SPI port in core-driven master mode

In core-driven SPI transfers, if a data word is written to `TXSPI` when another data transfer is already active, the current buffer content is overwritten and the new data word is transferred, leading to the corruption of the preceding data word. Therefore, a transmission may only be initiated if no other transfer is active.

Information about the status of the primary SPI port is available through the read-only SPI port status register `SPISTAT` [9, p. A-148 ff.]. Two bits thereof indicate the transmit transfer status.

- `TXS` describes the `TXSPI` buffer status. This bit is set if the buffer is full and cleared if the buffer is empty, i.e. it was emptied into the transmit shift register.
- `SPIF` describes the state of the transfer. This bit is cleared if a transfer is active, i.e. data is shifted out of the transmit shift register[14], and set if the transfer is finished, i.e. the transmit shift register is empty[14].

In order to assure that no transfer is active, `TXS` needs to be cleared and subsequently `SPIF` to be set [9, p. 12-36 f.]. The program execution is blocked by subsequently polling these flags until no transfer is active anymore[15].

Listing 6: Waiting for SPI transfer completion

```
// Wait for the TXS bit to be cleared
//   => Transmit data buffer was emptied
testTXS:
    ustat1 = dm(SPISTAT);
    bit tst ustat1 TXS;
if TF jump testTXS;

// Wait for the SPIF bit to be set
//   => Transmit shift register was emptied
testSPIF:
    ustat1 = dm(SPISTAT);
    bit tst ustat1 SPIF;
if not TF jump testSPIF;
```

[14] Assuming that the transfer initiation mode is set to `TIMOD = 01`.
[15] The loops for polling were implementing using the `JUMP` instruction as the `DO`/`UNTIL` instruction does not support bottom–controlled loops [7, p. 9-55].

Commonly, SPI slave devices require that a minimum wait time between successive word transfers is maintained. Unfortunately, this is not specified in the data sheet of the AD1835A. An indication for the magnitude of the wait time is given by the frame delay required by the DSP's SPI port if it is configured as a slave device, which is specified with 2 SPI clock periods T_{SPI} [9, p. 12-16]. On the basis of this specification, it is assumed that 4 SPI clock periods are a sufficient wait time for the control port of the AD1835A codec.

This minimum wait time is achieved by a loop of no-operation instructions (`NOP`), whereas each `NOP` takes one core clock cycle to execute. The number of core clock cycles to establish the required wait time is determined by the ratio of the core clock rate f_{CCLK} and the SPI clock rate f_{SPI}. In conjunction with Eq. (1) and Eq. (2) from Section 3.3, the number N_{CCLK} of required core clock cycles is derived as shown in Equation (6), whereas $T = \frac{1}{f}$ denotes the periodic time of the respective clock signal.

$$
\begin{aligned}
N_{\mathsf{CCLK}} &= \frac{2 \cdot T_{\mathsf{SPI}}}{T_{\mathsf{CCLK}}} = \frac{2 \cdot f_{\mathsf{CCLK}}}{f_{\mathsf{SPI}}} = \frac{4 \cdot f_{\mathsf{PCLK}}}{f_{\mathsf{SPI}}} = \frac{32 \cdot \mathsf{BAUDR} \cdot f_{\mathsf{SPI}}}{f_{\mathsf{SPI}}} = \\
&= 32 \cdot \mathsf{BAUDR} = 32 \cdot 4 = 128
\end{aligned} \tag{6}
$$

Consequently, the wait time between successive SPI data word transmissions is realized with the code shown below.

Listing 7: Realization of the minimum wait time between successive transfers

```
1 // Wait minimum wait time between successive transfers
2 lcntr = 128, do waitTimeLoop until LCE;
3 waitTimeLoop: nop;
```

These are all means required for the successive transmission of data words. Additionally, the transmission of a data word may also be used to receive a data word. Every time a data word is transmitted, the data is sent through the ("virtual") `SPI_MOSI` pin of the SPI port. Synchronously, the data from the `SPI_MISO` pin is shifted into the receive shift register of the SPI port. Therefore, every time a data word is transmitted, simultaneously a data word is received. Analog to the `TXS` bit, the `RXS` bit in the SPI port status register `SPISTAT` indicates when the receive buffer is full. In order to read the data word that was received synchonously to the transmission of a data word on the primary SPI port, the `RXS` is polled until it is set and afterwards the data word is read from the receive buffer `RXSPI`.

Listing 8: Receiving a data word via SPI with transfer initiation mode `TIMOD = 01`

```
1 // Here: Transmit data word and wait for transfer completion
2
3 // Wait for the RXS bit to be set
4 //   => Receive data buffer is full
5 testRXS:
6     ustat1 = dm(SPISTAT);
7     bit tst ustat1 RXS;
8 if not TF jump testRXS;
9
10 // Read the data word from the primary SPI port receive data buffer
11 r0 = dm(RXSPI);
```

3.5 Representation of the Configuration

The communication channel to the control port of the AD1835A via the SPI is established. For a convenient modification of the configuration of the codec some kind of representation thereof is necessary. The approach taken is to keep a mirrored version of the codec's configuration in the internal memory of the DSP.

As mentioned in Section 2.2, a control register of the codec is modified by sending a specific

data frame to the codec's control port. The basic structure of such a data frame is shown in Figure 9. The four most significant bits contain the address of the target register, followed by one bit declaring the type of operation (read or write). The ten remaining bits specify the configuration which should be set, or, if a read operation is performed, they are ignored. The exact composition of all data frames is available in Appendix A.

Figure 9: Structure of a data frame for configuration

Instead of only storing the control register content in the memory of the DSP, a buffer labeled `_ad1835a_config_buffer` is created in the data segment which contains a data word for each data frame. The assignment of the data words of this configuration buffer to the respective data frame is depicted in Table 6, where the offset from the base address of the buffer is shown as well[16]. Therefore, if a setting is changed in the configuration buffer, the associated data frame is instantly available through the related data word and can be sent to the codec.

Number	Offset from base address	Associated control register
1	0	DAC Control 1
2	1	DAC Control 2
3	2	DAC Volume-Left 1
4	3	DAC Volume-Right 1
5	4	DAC Volume-Left 2
6	5	DAC Volume-Right 2
7	6	DAC Volume-Left 3
8	7	DAC Volume-Right 3
9	8	DAC Volume-Left 4
10	9	DAC Volume-Right 4
11	10	ADC Control 1
12	11	ADC Control 2
13	12	ADC Control 3

Table 6: Structure of the configuration buffer

The configuration buffer is initialized by setting the address-, $R/\overline{W}$- and configuration-field of each data word as depicted in Figure 9. The settings of the AD1835A codec are concisely described in [1, p. 18] and a replication thereof is probably not worthwhile. However, in the following some remarks will be given.

After a basic configuration is sent to the codec, access to its configuration is probably mostly necessary to use the mute, volume, peak level information and maybe the power-down feature. The DAC Control 2 and ADC Control 2 register enable the individual muting of every DAC output and ADC input, whereas the eight DAC Volume Control registers provide an individual volume control for every DAC output which will be utilized in Chapter 4. Further, the AD1835A codec provides information about the peak input level of each ADC via the two ADC Peak registers if peak readback is enabled in ADC Control 3. The ADC Peak registers are somewhat special, as they are the only registers which are read. This is the reason, why they do not appear in Table 6. How these registers are read will be discussed in Section 3.6, whereas Chapter 4 presents an example of use.

In DAC Control 1 and ADC Control 2 the data format and word length for the serial ports for the DAC and ADC data of the codec is configured. It should be kept in mind that these

[16] The data memory is addressed in 32-bit words. More information is available in [7, ch. 7].

settings have to conform to the settings of the DSP's serial ports and, therefore, may not be changed without further consideration.

3.6 Configuration of the Codec

Section 3.5 introduced the configuration buffer, which contains the data frames with the desired configuration of the codec. In order to apply the configuration, these data frames are sent to the codec's control port by the means derived in Section 3.4.

This is accomplished by iterating through all data frames in the configuration buffer using a data address generator (DAG) [7, ch. 6] of the processor, whereas each data frame is synchronously sent and the minimum wait time between successive transfers is maintained.

1. Initiate the transmission by writing the data word into the transmit data buffer `TXSPI`.
2. Wait for the transfer to finish by polling the `TXS` and `SPIF` status bit.
3. Maintain the minimum wait time using a loop of `NOP` instructions.

Considering the configuration buffer name `_ad1835a_config_buffer` and the code fragments from Section 3.4, the code to send the configuration can be formulated as shown below. Individual data frames are sent similarly by only using the code from the loop body.

Listing 9: Transmission of the content of the configuration buffer to the codec

```
// Use the index and modify register 0 of DAG1 for buffer access
i0 = _ad1835a_config_buffer;  // Base address of the buffer
m0 = 1;                       // Increment address in steps of 1

// Loop through all data frames and send them to the AD1835A
lcntr = @_ad1835a_config_buffer, do sendCalibrationBufferLoop until LCE;

    // Read and transmit a data frame
    r0 = dm(i0, m0);
    dm(TXSPI) = r0;

    // Wait for the TXS bit to be cleared
    testTXS:
        ustat1 = dm(SPISTAT);
        bit tst ustat1 TXS;
    if TF jump testTXS;

    // Wait for the SPIF bit to be set
    testSPIF:
        ustat1 = dm(SPISTAT);
        bit tst ustat1 SPIF;
    if not TF jump testSPIF;

    // Wait minimum wait time between successive transfers
    lcntr = 128, do waitTimeLoop until LCE;
    waitTimeLoop: nop;

sendCalibrationBufferLoop: nop;
```

The considerations and the code presented at the end of Section 3.4 is used to read the content of an ADC Peak register[17]. The sequence of actions required to achieve the reading of a peak register with the transfer initation mode configured in Section 3.3 is as follows.

[17] Assuming that peak readback is enabled in ADC Control 3.

1. Send a data frame to the codec which contains the address of the desired ADC Peak register in the address-field and where the R/$\overline{\text{W}}$ bit set. The configuration field of the data frame is ignored. Thereupon the codec transfers the value of the addressed peak register into its transmit shift register and resets the peak register for a new peak level determination.
2. Send a dummy data frame, e.g. an arbitrary data frame from the configuration buffer, to the codec to shift the data from the codec's transmit shift register into the receive shift register of the primary SPI port of the DSP.
3. Poll the `RXS` status bit until it is set and, therefore, the receive data buffer is full.
4. Read the receive data buffer `RXSPI` to obtain the peak value.

As the clue of reading the peak registers lies in the sequence of the individual steps but does not introduce any new aspects regarding the code, no listing is included.

3.7 Summary

This chapter described the fundamental steps required to configure an AD1835A codec chip on an ADSP-21369 signal processor brought together via an ADSP-21369 EZ-KIT Lite® evaluation board and illustrated the principles of the involved hardware. Starting from the connections on the printed circuit board, the signals of the control port of the codec were traced to the DPI of the DSP and routed to the DSP's primary SPI port by programming the SRU. The SPI port was configured to comply to the requirements of the codec and the means for communication were established, incorporating the issues which evolve when successive transfers are performed. Subsequently, a representation of the codec's configuration was introduced to finally carry out the configuration, bringing together all developed components.

The actual software interface uses the knowledge presented above as building blocks for a set of constants, macros and subroutines to provide convenient means to alter the configuration of the codec and read the content of its peak level registers. In Appendix C the essential source code is listed and its integration is explained. A detailed discussion of the internal structure of the software interface would go beyond the scope, however, with the information in this chapter its functionality should be comprehensible.

4 Applications

Chapter 3 discussed the configuration of the AD1835A codec using the primary SPI port of the DSP in-depth. The chapter concluded with the introduction of the software interface presented in Appendix C which provides convenient means to configure the codec. This chapter concisely demonstrates two examples of use to illustrate the practical aspect of the configuration as well.

The starting point for these examples is the project "Talkthrough" provided in the DAL course referred to in Chapter 1. That code in turn is derived from the project "Talkthru Analog In-Out (ASM)" provided by Analog Devices with VisualDSP++ included in the ADSP-21369 EZ-KIT Lite®.

4.1 Volume Control

The evaluation board offers four general-purpose push buttons, whereas the push buttons `PB1` and `PB2` are connected to the flag-pins `FLAG1` and `FLAG0` of the DSP which can be configured as interrupt inputs $\overline{\texttt{IRQ1}}$ and $\overline{\texttt{IRQ0}}$ [6, p. 1-12], [9, p. 17-27 ff.], [7, p. 4-29 ff.]. These two push buttons are utilized to realize a linear volume control. The output volume of all DACs

is increased (`PB1`) and decreased (`PB2`) by modifying the DAC Volume Control registers of the codec using the software interface discussed in Chaper 3. Therefore, the interrupt vector table (IVT) is supplemented with two additional interrupt service routines (ISR) [7, app. C], `_push_button_2_isr` for $\overline{\texttt{IRQ1}}$ and `_push_button_1_isr` for $\overline{\texttt{IRQ0}}$, which perform the reconfiguration of the codec.

Instead of a detailed discussion some code excerpts are listed below that illustrate the underlying principles. Nonetheless, two aspects should be emphasized. Sheet 5 of the evaluation board's schematic [6, app. B] depicts the circuit of the push buttons which reveals that a logical low is ouput in the unpressed state. Considering that the interrupt inputs are active-low, their interrupt sensitivity has to be configured for edge-sensitivity, as level-sensitivity would implicate a constant triggering of the interrupt. Further, interrupt nesting has to be disabled [7, p. 4-43 ff.], as otherwise the push button ISRs and the audio sample ISR, which implements the talkthrough by reading a sample from the ADCs and outputting them on the DACs, might interfere.

Listing 10: Configuration of the interrupt inputs $\overline{\texttt{IRQ0}}$ and $\overline{\texttt{IRQ1}}$

```
// Disable nested interrupts
bit clr mode1 NESTM;

// Enable interrupt mode for FLAG0 and FLAG1
ustat1 = dm(SYSCTL);
bit set ustat1 IRQ0EN | IRQ1EN;
dm(SYSCTL) = ustat1;

// Select edge-sensitivity for IRQ0 and IRQ1
bit set mode2 IRQ0E | IRQ1E;

// Clear the interrupt latch register for IRQ0 and IRQ1
bit clr irptl IRQ0I | IRQ1I;

// Unmask the interrupt IRQ0 and IRQ1
bit set imask IRQ0I | IRQ1I;

// Enable interrupts (globally)
bit set mode1 IRPTEN;
```

In the interrupt service routines of the push buttons the volume of *all* DACs is adjusted, whereas the representative volume setting is taken from DAC1 left. The following listing depicts the ISR for the push button `PB1`. The ISR for push button `PB2` is similar, except that the volume is decremented and the lower bound is checked.

Listing 11: Interrupt service routine for the push button `PB1`

```
_push_button_1_isr:

    // Get current volume setting from configuration buffer
    r14 = dm(AD1835A_REG_DACVOL1L);
    r15 = AD1835A_DACVOL_MAX;
    r14 = r14 and r15;

    // Increase volume by 1024/10 ~= 102
    r13 = 102;
    r14 = r14 + r13;

    // Check if volume is above maximum
    r13 = r15 - r14;
    if GE jump vol_below_max;
        r14 = AD1835A_DACVOL_MAX;
    vol_below_max:
```

```
17
18     // Set volume on all outputs
19     ad1835aSetVolume(DACVOL1L, r14);
20     ad1835aSetVolume(DACVOL1R, r14);
21     ad1835aSetVolume(DACVOL2L, r14);
22     ad1835aSetVolume(DACVOL2R, r14);
23     ad1835aSetVolume(DACVOL3L, r14);
24     ad1835aSetVolume(DACVOL3R, r14);
25     ad1835aSetVolume(DACVOL4L, r14);
26     ad1835aSetVolume(DACVOL4R, r14);
27
28 _push_button_1_isr.end:
29     rti;
```

4.2 Input Level Meter

The evaluation board provides eight general-purpose light-emitting diodes (LED) connected to the DAI, DPI and `FLAG3` pin of the DSP [6, p. 1-12 f.], which allure to realize an ADC input level meter using the peak level information provided by the codec. Therefore, the LEDs are routed and a timer interrupt is set up, whereas in the timer ISR the peak level of both ADCs is read and visualized using the LEDs.

Due to the fact that the LEDs are connected to diverse interfaces of the DSP, their routing and control is not that trivial. `LED8` is connected to the flag 3 pin of the DSP which is controlled through the Flag I/O register if it is configured as an output in "flag-mode" [9, p. 17-27 ff.], [7, p. B-18 ff.]. `LED1` to `LED5` are connected to the DPI and, therefore, may be routed to the I/O flags as well, which is visible in Figure 5. `LED6` and `LED7` are connected to the DAI, whereas the DAI does not offer a general-purpose register. For this reason these two LEDs are controlled by routing the desired logical level to their pin buffer input using the SRU. The code to accomplish the control of the LEDs is shown in Appendix C.

The timer of the DSP, described in [7, ch. 5], is configured by setting the timer count `TCOUNT` and the timer period `TPERIOD`. `TCOUNT` is decremented by one during each clock cycle. If `TCOUNT` reaches zero, an interrupt is generated and `TCOUNT` is reset to `TPERIOD`. For example, by utilizing Equation (3) from Section 3.3, the number of clock cycles for `TPERIOD` to set up a timer interrupt with an interval time of $T_{\text{Interval}} = 0.1\,\text{s}$ is determined as shown by Equation (7).

$$\texttt{TPERIOD} = T_{\text{Interval}} \cdot f_{\texttt{CCLK}} = 0.1\,\text{s} \cdot 331.776\,\text{MHz} = 33177600 \tag{7}$$

The timer generates a low priority (`TMZLI`) and a high priority (`TMZHI`) interrupt, whereas the latter is used[18]. A timer ISR `_timer_isr` was created and assigned to `TMZHI` in the interrupt vector table.

Concluding, some code excepts are depicted that illustrate the essence of the input level meter. The steps listed hereafter are performed during the startup to route and initialize the LEDs, as well as to configure and start the timer. The macros `ledInitialize()` and `ad1835aModifySetting()` are part of the software interface listed in Appendix C[19].

Listing 12: Routing of the LEDs and configuration of the timer

```
1 // Initialize and route the general-purpose LEDs
2 ledInitialize();
3
4 // Enable peak level readback on AD1835A
5 ad1835aModifySetting(ADCPEAKRB, ON);
```

[18]The low priority timer interrupt frequently did not get serviced, why the high priority interrupt was chosen.

[19]Prior to these steps the PLL, serial ports, SPI port and the AD1835A codec have to be initialized.

```
6
7 // Disable nested interrupts
8 bit clr mode1 NESTM;
9
10 // Counter start value (start with immediate interrupt)
11 tcount = 0;
12
13 // Counter period in core clock cycles
14 tperiod = TIMER_PERIOD;
15
16 // Enable timer interrupt
17 bit set mode2 TIMEN;
18
19 // Clear latched high priority timer interrupt
20 bit clr irptl TMZHI;
21
22 // Unmask high priority timer interrupt
23 bit set imask TMZHI;
24
25 // Enable interrupts (globally)
26 bit set mode1 IRPTEN;
```

In the following the ISR for the timer interrupt is shown. The macro `ad1835aUpdateLevelMeter()` reads the peak level registers of the codec and sets the LEDs of the evaluation board accordingly, whereas the LED5 – LED8 is used for the left ADC and LED1 – LED4 for the right ADC. Its code is listed in Appendix C[20].

Listing 13: Interrupt service routine for the timer interrupt

```
1 _timer_isr:
2
3     // Update the ADC input level meter
4     ad1835aUpdateLevelMeter();
5
6 _timer_isr.end:
7     rti;
```

5 Conclusion

In this thesis the configuration of the AD1835A audio codec chip in the context of the ADSP-21369 EZ-KIT Lite® evaluation board was discussed in-depth, whereas the involved hardware was thoroughly explored to reason and prove the correctness of every taken step. Further, two examples of use were introduced to illustrate the practical aspects of the configurable features of the codec. Moreover, these examples provided a platform for the comprehensive testing of the software interface, which exhibited a faultless operation.

This brings up the question why the code used in the DAL course mentioned in Chapter 1 occasionally exhibited erroneous behavior. That code is very rudimental and partially omits the checking of prerequisites. Considering that it is only executed once immediately after startup, this is not that dramatic. However, when its loop for sending the data frames is compared to the one presented in Section 3.6 a subtle difference can be noticed. The completion of a transfer is checked by only polling the `SPIF` bit which indicates if the transmit shift register is empty, whereas the `TXS` bit, specifying if the content of the transmit data buffer was emptied into the transmit shift register, is ignored. The exact moment when the transmit data buffer is emptied into the transmit shift register is not specified in the hardware reference, while it probably depends on the current status of the SPI clock. Anyway, it may

[20] The thresholds for the LEDs are specified by the constant `LEVEL_THRES_STEP` defined in `ad1835a.h`.

occur that the first check of the `SPIF` bit is performed *before* the transmit data buffer was emptied into the transmit shift register and, consequently, the program assumes that the current transfer was finished and continues to transmit the next data frame. Therefore, the previous data frame is overwritten and lost, ending in the fact that those modifications of the configuration do not take effect.

Concluding, this thesis provided an introduction to the rather complex, yet flexible and efficient mechanism of peripheral communication using the ADSP-21369 digital signal processor. The aim of the thesis, the configuration of the AD1835A audio codec chip, yielded a convenient tool which was able to eliminate the existing problems and will hopefully find its way into practical use, being a helpful assistant in the utilization of the ADSP-21369 EZ-KIT Lite® evaluation board.

Bibliography

[1] Analog Devices Inc., Norwood. *AD1835A Data Sheet*, Revision A, Dec. 2003. http://www.analog.com/static/imported-files/data_sheets/AD1835A.pdf.

[2] Analog Devices Inc., Norwood. *ADSP-21368 SHARC® Processor Hardware Reference*, Revision 1.0, Sept. 2006.

[3] Analog Devices Inc., Norwood. *ADSP-2136x SHARC® Processor Programming Reference*, Revision 1.1, Mar. 2007.

[4] Analog Devices Inc., Norwood. *VisualDSP++ 5.0 User's Guide*, Revision 3.0, Aug. 2007. http://www.analog.com/static/imported-files/software_manuals/719705850_ug.pdf.

[5] Analog Devices Inc., Norwood. *ADSP-21367/ADSP-21368/ADSP-21369 Data Sheet*, Revision E, Jul. 2009. http://www.analog.com/static/imported-files/data_sheets/ADSP-21367_21368_21369.pdf.

[6] Analog Devices Inc., Norwood. *ADSP-21369 EZ-KIT Lite® Evaluation System Manual*, Revision 2.2, Sept. 2009. http://www.analog.com/static/imported-files/eval_kit_manuals/ADSP-21369%20EZ-KIT%20Lite%20Manual%20Rev%202.2.pdf.

[7] Analog Devices Inc., Norwood. *SHARC® Processor Programming Reference*, Revision 2.0, Jun. 2009. http://www.analog.com/static/imported-files/processor_manuals/ADSP_2136x_PGR_rev2-0.pdf.

[8] Analog Devices Inc., Norwood. *VisualDSP++ 5.0 Assembler and Preprocessor Manual*, Revision 3.3, Sept. 2009. http://www.analog.com/static/imported-files/software_manuals/50_asm_man_3.3.pdf.

[9] Analog Devices Inc., Norwood. *ADSP-2137x SHARC® Processor Hardware Reference*, Revision 2.1, May 2010. http://www.analog.com/static/imported-files/processor_manuals/ADSP-21367_hwr_rev2-1.pdf.

[10] Freescale Semiconductor Inc. *M68HC11 Reference Manual*, Revision 6.1, 2007. http://www.freescale.com/files/microcontrollers/doc/ref_manual/M68HC11RM.pdf.

A Excerpts of the AD1835A Data Sheet

A.1 Data Frames associated with the Registers

Register Address	Register Name	Description	Type	Width	Reset Setting (Hex)
0000	DACCTRL1	DAC Control 1	R/$\overline{W}$	10	000
0001	DACCTRL2	DAC Control 2	R/$\overline{W}$	10	000
0010	DACVOL1	DAC Volume–Left 1	R/$\overline{W}$	10	3FF
0011	DACVOL2	DAC Volume–Right 1	R/$\overline{W}$	10	3FF
0100	DACVOL3	DAC Volume–Left 2	R/$\overline{W}$	10	3FF
0101	DACVOL4	DAC Volume–Right 2	R/$\overline{W}$	10	3FF
0110	DACVOL5	DAC Volume–Left 3	R/$\overline{W}$	10	3FF
0111	DACVOL6	DAC Volume–Right 3	R/$\overline{W}$	10	3FF
1000	DACVOL7	DAC Volume–Left 4	R/$\overline{W}$	10	3FF
1001	DACVOL8	DAC Volume–Right 4	R/$\overline{W}$	10	3FF
1010	ADCPeak0	ADC Left Peak	R	6	000
1011	ADCPeak1	ADC Right Peak	R	6	000
1100	ADCCTRL1	ADC Control 1	R/$\overline{W}$	10	000
1101	ADCCTRL2	ADC Control 2	R/$\overline{W}$	10	000
1110	ADCCTRL3	ADC Control 3	R/$\overline{W}$	10	000
1111	Reserved	Reserved	R/$\overline{W}$	10	Reserved

Figure 10: Control Register Map [1, p. 19]

			Function				
Address	R/$\overline{W}$	RES	De-emphasis	DAC Data Format	DAC Data-Word Width	Power-Down Reset	Sample Rate
15, 14, 13, 12	11	10	9, 8	7, 6, 5	4, 3	2	1, 0
0000	0	0	00 = None 01 = 44.1 kHz 10 = 32.0 kHz 11 = 48.0 kHz	000 = I^2S 001 = RJ 010 = DSP 011 = LJ 100 = Packed 256 101 = Packed 128 110 = Reserved 111 = Reserved	00 = 24 Bits 01 = 20 Bits 10 = 16 Bits 11 = Reserved	0 = Normal 1 = Power-Down	00 = 8 × (48 kHz) 01 = 4 × (96 kHz) 10 = 2 × (192 kHz) 11 = 8 × (48 kHz)

Figure 11: DAC Control 1 [1, p. 19]

				Function								
					MUTE DAC							
Address	R/$\overline{W}$	RES	Reserved	Stereo Replicate	OUTR4	OUTL4	OUTR3	OUTL3	OUTR2	OUTL2	OUTR1	OUTL1
15, 14, 13, 12	11	10	9	8	7	6	5	4	3	2	1	0
0001	0	0	0	0 = Off 1 = Replicate	0 = On 1 = Mute	0 = On 1 = Mute	0 = On 1 = Mute	0 = On 1 = Mute	0 = On 1 = Mute	0 = On 1 = Mute	0 = On 1 = Mute	0 = On 1 = Mute

Figure 12: DAC Control 2 [1, p. 19]

			Function
Address	R/$\overline{W}$	RES	DAC Volume
15, 14, 13, 12	11	10	9, 8, 7, 6, 5, 4, 3, 2, 1, 0
0010 = DACL1 0011 = DACR1 0100 = DACL2 0101 = DACR2 0110 = DACL3 0111 = DACR3 1000 = DACL4 1001 = DACR4	0	0	0000000000 = Mute 0000000001 = 1/1023 0000000010 = 2/1023 1111111110 = 1022/1023 1111111111 = 1023/1023

Figure 13: DAC Volume Control [1, p. 20]

			Function	
Address	R/$\overline{W}$	RES	Six Data Bits	Four Fixed Bits
15, 14, 13, 12	11	10	9, 8, 7, 6, 5, 4	3, 2, 1, 0
1010 = Left ADC 1011 = Right ADC	1	0	000000 = 0.0 dBFS 000001 = –1.0 dBFS 000010 = –2.0 dBFS 111111 = –63.0 dBFS	0000 These four bits are always zero.

Figure 14: ADC Peak [1, p. 20]

Address	R/$\overline{W}$	RES	Function				
			RES	Filter	ADC Power-Down	Sample Rate	Reserved
15, 14, 13, 12	11	10	9	8	7	6	5, 4, 3, 2, 1, 0
1100	0	0	0	0 = All Pass 1 = High-Pass	0 = Normal 1 = Power-Down	0 = 48 kHz 1 = 96 kHz	0, 0, 0, 0, 0, 0 0, 0, 0, 0, 0, 0

Figure 15: ADC Control 1 [1, p. 20]

Address	R/$\overline{W}$ RES	RES	Function					
			Master/Slave Aux Mode	ADC Data Format	ADC Data-Word Width	Reserved	ADC MUTE	
							Right	Left
15, 14, 13, 12	11	10	9	8, 7, 6	5, 4	3, 2	1	0
1101	0	0	0 = Slave 1 = Master	000 = I^2S 001 = RJ 010 = DSP 011 = LJ 100 = Packed 256 101 = Packed 128 110 = Auxiliary 256 111 = Auxiliary 512	00 = 24 Bits 01 = 20 Bits 10 = 16 Bits 11 = Reserved	0, 0	0 = On 1 = Mute	0 = On 1 = Mute

Figure 16: ADC Control 2 [1, p. 20]

Address	R/$\overline{W}$ RES	RES	Reserved	Function			
				IMCLK Clocking Scaling	ADC Peak Readback	DAC Test Mode	ADC Test Mode
15, 14, 13, 12	11	10	9	8, 7, 6	5	4, 3, 2	1, 0
1110	0	0	0, 0	00 = MCLK × 2 01 = MCLK 10 = MCLK × 2/3 11 = MCLK × 2	0 = Disabled Peak Readback 1 = Enabled Peak Readback	000 = Normal Mode All others reserved	00 = Normal Mode All others reserved

Figure 17: ADC Control 3 [1, p. 20]

A.2 SPI Transfer Diagram

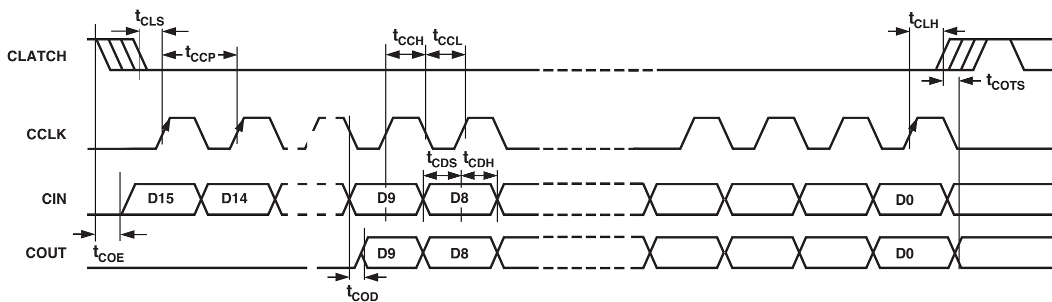


Figure 18: SPI transfer diagram of the AD1835A control port [1, p. 12]

B Excerpts of the ADSP-21369 Documentation

B.1 SPI Transfer Diagram

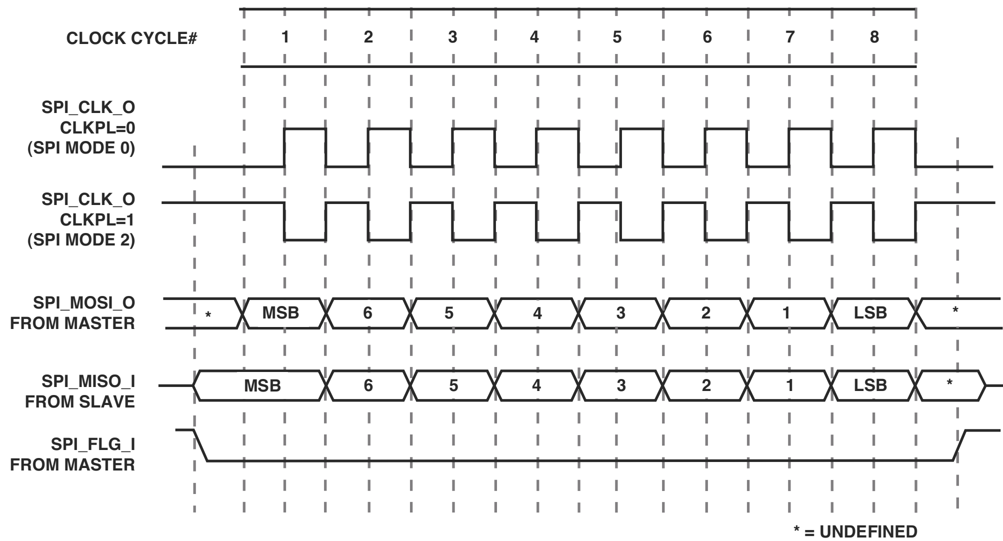


Figure 19: SPI transfer diagram for `CPHASE = 0` [9, p. 12-14]

B.2 Clock Relationship to the Input Clock

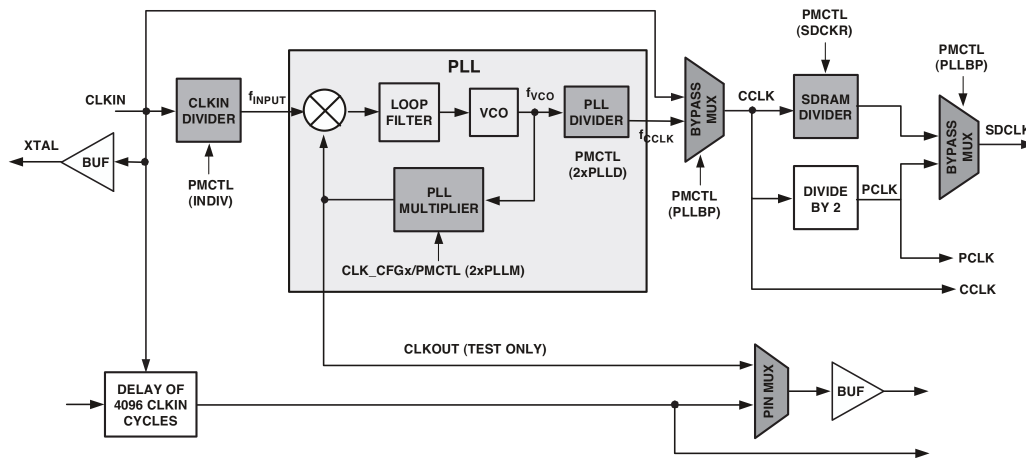


Figure 20: Clock relationship to the input clock [5, p. 19]

C Source Code

C.1 Embedding the Software Interface into a Project

The software interface is embedded into a project by adding the files `spi.h`, `spi.asm`, `ad1835a.h` and `ad1835a.asm` to the project directory. Both header files have to be included in the assembler file of the startup routine (e.g. `main.asm`) and the initialization macros must be called in the startup routine as shown below.

```
// Initialize the primary SPI port and SRU for its DPI pins
spiInitPrimary();

// Initialize the AD1835A codec
ad1835Initialize();
```

The configuration of the codec may then be accessed using the macros provided by `ad1835a.h`. If the configuration has to be permantently changed, this may be done by altering the initial configuration defined in `ad1835a.asm`.

In order to utilize the input level meter discussed in Chapter 4, the files `led.h` and `led.asm` have to be added to the project directory. The LEDs are routed and initialized by applying the macro shown below.

```
// Initialize and route the general-purpose LEDs
ledInitialize();
```

C.2 Signal Routing, SPI Configuration and Communication

C.2.1 spi.h

```
/****************************************************************
 * File: spi.h
 *
 * This file provides macros for SPI port initialization,
 * configuration and data transfer.
 * Currently, all macros work on the *PRIMARY* SPI port.
 *
 * Revisions:
 *   2010-05-03, Matthias Hotz: Initial version
 *   2010-06-01, Matthias Hotz: Added receive data buffer readout
 *
 ****************************************************************/
#ifndef _SPI_H_
#define _SPI_H_

#include <def21369.h>
#include <sru.h>

.extern _spi_init_primary;
.extern _spi_configure;
.extern _spi_send_data_buffer;
.extern _spi_send_data_word;
.extern _spi_send_data_word_and_wait;
.extern _spi_read_recv_data_buffer;

/****************************************************************
 * MACROS
 ****************************************************************/

//--------------------------------------------------------------
// Macro for initialization of the primary SPI port
#define spiInitPrimary() \
    call _spi_init_primary;


//--------------------------------------------------------------
// Macro for configuration of the primary SPI port
//
// Parameters:
//   ctl_reg .... Data word for SPICTL register (SPIEN must not be set)
//   baud_reg ... Data word for SPIBAUD register
//
#define spiConfigure(ctl_reg, baud_reg) \
    r5 = ctl_reg;                       \
    r6 = baud_reg;                      \
    call _spi_configure;


//--------------------------------------------------------------
// Macro to transmit the content of a buffer via the primary SPI
// port (Synchronous transfer)
```

```
//
// Parameters:
//   buf_addr ... Start address of the buffer
//   buf_len .... Numer of data words in the buffer
//
#define spiSendDataBuffer(buf_addr, buf_len) \
    r5 = buf_addr;                           \
    r6 = buf_len;                            \
    call _spi_send_data_buffer;


//--------------------------------------------------------------
// Macro to transmit one data word via the primary SPI port. The
// call is blocking, it does not return until the data word was
// completely transmitted.
//
// Parameters:
//   dw ... Data word to transmit
//
#define spiSendDataWord(dw) \
    r0 = dw;                \
    call _spi_send_data_word;


//--------------------------------------------------------------
// Macro to transmit one data word via the primary SPI port
// (blocking) and wait the minimum wait time between successive
// transfers. (This subroutine can be used if some data words
// should be transmitted successively).
//
// Parameters:
//   r0 ... Data word to transmit
//
#define spiSendDataWordAndWait(dw) \
    r0 = dw;                       \
    call _spi_send_data_word_and_wait;


//--------------------------------------------------------------
// Macro to read one data word from the receive data buffer of
// the primary SPI port. Program execution is blocked until the
// receive data buffer is filled before readout.
//
// Return Value:
//   r0 ... Received data word
//
#define spiReadRecvDataBuffer() \
    call _spi_read_recv_data_buffer;


#endif
```

C.2.2 spi.asm

```
/*****************************************************************
 * File: spi.asm
 *
 * This file provides subroutines for SPI port initialization,
 * configuration and data transfer.
 * Currently, all subroutines work on the *PRIMARY* SPI port.
 * However, except the init- and route-subroutines all sub-
 * routines would be adaptable to support both SPI ports (e.g.
 * by passing the register addresses SPICTL/SPICTLB, etc. using
 * a processor register). Due to the fact that only the primary
 * SPI port is in use, this generalization would only degrade the
 * readability of the code, hence it was not implemented.
 *
 * Revisions:
 *   2010-05-03, Matthias Hotz: Initial version
 *   2010-06-01, Matthias Hotz: Added receive data buffer readout
 *
 *****************************************************************/

#include <def21369.h>
#include <sru.h>

.global _spi_init_primary;
.global _spi_configure;
.global _spi_send_data_buffer;
.global _spi_send_data_word;
.global _spi_send_data_word_and_wait;
```

```
28 .global _spi_read_recv_data_buffer;

30 // Between two successive transfers on the SPI port a minimum
   // wait time is required. This is defined here as four SPI clock
32 // periods which is for this configuration the below specified
   // number of core clock cycles
34 #define WAIT_CCLK_CYCLES 128

36
   /****************************************************************
38  * GLOBAL SUBROUTINES
    ****************************************************************/
40 .section/pm seg_pmco;

42 //--------------------------------------------------------------
   // Initialize the primary SPI port: Route signals, set
44 // configuration and baud rate
   _spi_init_primary:
46
       // Initialize the signal routing unit regarding the primary
48     // SPI port
       call _spi_route_primary;
50
       // Configure the primary SPI port
52     r5 = TIMOD1 | // Trigger transfer by write to TXSPI
            WL16   | // 16 bit word length
54          MSBF   | // Send most significant byte first
            SPIMS;   // Set to SPI master
56     r6 = 0x8;
       call _spi_configure;
58
   _spi_init_primary.end:
60     rts;

62
   //--------------------------------------------------------------
64 // Configure the primary SPI port
   //
66 // Parameters:
   //   r5 ... Data word for SPICTL register (SPIEN must not be set)
68 //   r6 ... Data word for SPIBAUD register
   //
70 _spi_configure:

72     // Disable the SPI port before changing its configuration
       call _spi_disable;
74
       // Set the SPI baud rate
76     dm(SPIBAUD) = r6;

78     // Backup the SPI port flag register and disable all devices
       r2 = dm(SPIFLG);
80     r0 = SPIFLG0 | SPIFLG1 | SPIFLG2 | SPIFLG3;
       dm(SPIFLG) = r0;
82
       // Configure the SPI port via its control register
84     dm(SPICTL) = r5;

86     // Enable the SPI port
       r1 = SPIEN;
88     r0 = r5 or r1;
       dm(SPICTL) = r0;
90
       // Restore the SPI port flag register
92     dm(SPIFLG) = r2;

94 _spi_configure.end:
       rts;
96

98 //--------------------------------------------------------------
   // Send all data words of a specified buffer via the primary SPI
100 // port
   //
102 // Parameters:
   //   r5 ... Start address of the buffer
104 //   r6 ... Numer of data words in the buffer
   //
106 _spi_send_data_buffer:

108     // Use the index and modify reg. 0 of DAG1 for buffer access
       i0 = r5;
```

```
110     m0 = 1;

112     // Loop through all data words and send them
        lcntr = r6, do spiSendDataBufferLoop until LCE;
114
            // Read and transmit the data word (synchronous)
116         r0 = dm(i0, m0);
            call _spi_send_data_word;
118
            // Wait minimum wait time between successive transfers
120         lcntr = WAIT_CCLK_CYCLES, do intermedWaitTimeSDB until LCE;
            intermedWaitTimeSDB: nop;
122
        spiSendDataBufferLoop: nop;
124
    _spi_send_data_buffer.end:
126     rts;

128
    //---------------------------------------------------------------
130 // Send one data word via the primary SPI port. The subroutine
    // is blocking, it does not return until the data word was
132 // completely transmitted.
    //
134 // Parameters:
    //   r0 ... Data word to transmit
136 //
    _spi_send_data_word:
138
        // Transmit the data word
140     dm(TXSPI) = r0;

142     // Wait for transmit tranfer to be finished
        call _spi_wait_for_transmit_completion;
144
    _spi_send_data_word.end:
146     rts;

148
    //---------------------------------------------------------------
150 // Send one data word via the primary SPI port synchronously and
    // wait minimum wait time between successive transfers afterwards
152 // (This subroutine can be used if some data words should be
    // transmitted successively).
154 //
    // Parameters:
156 //   r0 ... Data word to transmit
    //
158 _spi_send_data_word_and_wait:

160     // Send data word synchronously
        call _spi_send_data_word;
162
        // Wait minimum wait time between successive transfers
164     lcntr = WAIT_CCLK_CYCLES, do intermedWaitTimeSDW until LCE;
        intermedWaitTimeSDW: nop;
166
    _spi_send_data_word_and_wait.end:
168     rts;

170
    //---------------------------------------------------------------
172 // Wait until the receive data buffer of the primary SPI port is
    // full and store the received data word in the register R0.
174 // (Attention: This call blocks program execution if the receive
    // buffer is empty and no SPI transfer is active)
176 //
    // Return Value:
178 //   r0 ... Received data word
    //
180 _spi_read_recv_data_buffer:

182     // Wait for the RXS bit to be set
        //  => Receive data buffer is full
184     testRXS:
            ustat1 = dm(SPISTAT);
186         bit tst ustat1 RXS;
        if not TF jump testRXS;
188
        // Read the data word from the primary SPI port receive buffer
190     r0 = dm(RXSPI);
```

```
_spi_read_recv_data_buffer.end:
    rts;


/****************************************************************
 * LOCAL SUBROUTINES
 ****************************************************************/

//--------------------------------------------------------------
// Route the primary SPI port to DPI_P01 - DPI_P03 and the device
// select signal for the AD1835A to DPI_P04.
_spi_route_primary:

    // Master data output
    SRU(SPI_MOSI_O, DPI_PB01_I);
    SRU(HIGH, DPI_PBEN01_I);

    // Master data input
    SRU(DPI_PB02_O, SPI_MISO_I);
    SRU(LOW, DPI_PBEN02_I);

    // Master clock output
    SRU(SPI_CLK_O, DPI_PB03_I);
    SRU(HIGH, DPI_PBEN03_I);

    // Device select output for AD1835A
    SRU(SPI_FLG3_O, DPI_PB04_I);
    SRU(HIGH, DPI_PBEN04_I);

    rts;


//--------------------------------------------------------------
// Disables the primary SPI port. If a transfer is active the
// subroutine waits for it to finish before disabling the port.
_spi_disable:

    // If any transmit transfer is active wait for it to finish
    call _spi_wait_for_transmit_completion;

    // Disable primary SPI port (and implicitly clear the
    // transmit and receive buffer as well as the DMA FIFO buffer
    // status)
    r0 = dm(SPICTL);
    r1 = ~SPIEN;
    r0 = r0 and r1;
    dm(SPICTL) = r0;

    rts;


//--------------------------------------------------------------
// Wait until a transmit transfer on the primary SPI port is
// finished
_spi_wait_for_transmit_completion:

    // Check if the DMA transfer mode is active and if so, wait
    // for the DMA FIFO buffer to be emptied
    r0 = dm(SPICTL);        // Read SPI control register
    r1 = TIMOD2 | TIMOD1;   // Mask for transfer mode
    r0 = r0 and r1;         // Extract transfer mode
    r1 = TIMOD2;            // TMOD setting for DMA
    r0 = r0 xor r1;         // Check for equality
    if NE jump notDMA;

        // Transfer initiation mode is DMA, so wait for DMA FIFO
        // to empty. Status 00 indicates that the DMA FIFO is empty.

        r1 = SPIS1 | SPIS0;     // Mask for DMA FIFO status

        testFIFO:
            r0 = dm(SPIDMAC);   // Read SPI DMA config. register
            r0 = r0 and r1;     // Extract FIFO status
        if NE jump testFIFO;

    notDMA:

    // Wait for the TXS bit to be cleared
    //  => Transmit data buffer was emptied
    testTXS:
        ustat1 = dm(SPISTAT);
        bit tst ustat1 TXS;
```

```
    if TF jump testTXS;

    // Wait for the SPIF bit to be set
    //  => Transmit shift register was emptied
    testSPIF:
        ustat1 = dm(SPISTAT);
        bit tst ustat1 SPIF;
    if not TF jump testSPIF;

    rts;
```

C.3 Codec Configuration Representation and Modification

C.3.1 ad1835a.h

```
/*****************************************************************
 * File: ad1835a.h
 *
 * This file provides macros and constants for the configuration
 * of the AD1835A codec via the primary SPI port of the DSP.
 *
 * Revisions:
 *   2010-05-03, Matthias Hotz: Initial version
 *   2010-06-01, Matthias Hotz: Added peak level register readout
 *   2010-06-02, Matthias Hotz: Added level meter
 *
 *****************************************************************/

#ifndef _AD1835A_H_
#define _AD1835A_H_

#include "spi.h"

.extern _ad1835a_config_buffer;
.extern _ad1835a_init;
.extern _ad1835a_update_level_meter;

/*****************************************************************
 * MACROS
 *****************************************************************/

//---------------------------------------------------------------
// Initialise the AD1835A: Set the SPI device-select signal and
// send the complete configuration
#define ad1835Initialize() \
    call _ad1835a_init;


//---------------------------------------------------------------
// Modify a setting in one of the control registers and transmit
// the changes to the AD1835A.
//
// Supported registers of the AD1835A:
//  *) DAC control register 1 and 2
//  *) ADC control register 1, 2 and 3
//
// Parameters:
//  s_type .... Type of the setting. This is the second part of
//              constants defined in the section "CONSTANTS".
//  s_value ... Value of the setting. This is the third part of
//              the constants defined in the section "CONSTANTS".
//
//  All constants for the control register settings conform to
//  the following naming scheme:
//
//    AD1835A_Type_ValuePt1_ValuePt2_...._ValuePtN
//    \-----/ \--/ \---------------------------/
//    Prefix  Type           Value name
//            name
//
// Example:
//  The right channel of DAC4 should be muted. The corresponding
//  constant is AD1835A_DACOUT4R_MUTE, thus the type is DACOUT4R
//  and the value is MUTE. The macro call to change this
//  setting is:
//
//  ad1835aModifySetting(DACOUT4R, MUTE);
//
#define ad1835aModifySetting(s_type, s_value)    \
    /* Read the corresponding register */        \
```

```
    r0 = dm(AD1835A_##s_type##_REG);              \
    /* Get the mask for this setting */           \
    r1 = AD1835A_##s_type##_MASK;                 \
    /* Clear the setting */                       \
    r0 = r0 and r1;                               \
    /* Get the new setting */                     \
    r1 = AD1835A_##s_type##_##s_value;            \
    /* Set the new setting */                     \
    r0 = r0 or r1;                                \
    /* Write the modification to the buffer */    \
    dm(AD1835A_##s_type##_REG) = r0;              \
    /* Send the data word to the AD1835A */       \
    spiSendDataWordAndWait(r0);

//-------------------------------------------------------------
// Set the volume level of an output
//
// Parameters:
//  dac_reg ..... Name of the DAC volume register:
//                DACVOL1L ... DAC 1 left
//                DACVOL1R ... DAC 1 right
//                DACVOL2L ... DAC 2 left
//                DACVOL2R ... DAC 2 right
//                DACVOL3L ... DAC 3 left
//                DACVOL3R ... DAC 3 right
//                DACVOL4L ... DAC 4 left
//                DACVOL4R ... DAC 4 right
//  vol_level ... Volume level in 1024 steps:
//                Either one of the AD1835A_DACVOL_x constants
//                or explicit value (direct or register).
//
// Example:
//  Set the volume of the right channel of DAC4 to low:
//
//  ad1835aSetVolume(DACVOL4R, AD1835A_DACVOL_LOW);
//
#define ad1835aSetVolume(dac_reg, vol_level)     \
    /* Read the corresponding register */         \
    r0 = dm(AD1835A_REG_##dac_reg);               \
    /* Create mask for volume setting */          \
    r1 = ~AD1835A_DACVOL_MAX;                     \
    /* Clear the volume setting */                \
    r0 = r0 and r1;                               \
    /* Get the new setting */                     \
    r1 = vol_level;                               \
    /* Set the new setting */                     \
    r0 = r0 or r1;                                \
    /* Write the modification to the buffer */    \
    dm(AD1835A_REG_##dac_reg) = r0;               \
    /* Send the data word to the AD1835A */       \
    spiSendDataWordAndWait(r0);


//-------------------------------------------------------------
// Read the content of a peak level register of the AD1835A. The
// peak level register is reset after it was read.
//
// Parameters:
//  peak_reg ... Name of the peak level register to read:
//               1L ... ADC left
//               1R ... ADC right
//
// Return Value:
//   r0 ... Received peak level (contained in bit 5 to 0)
//
// Example:
//  Read the peak level register of the left input channel:
//
//  ad1835aReadPeakRegister(1L);
//
#define ad1835aReadPeakRegister(peak_reg)                   \
    /* Generate read command data word */                   \
    r0 = AD1835A_REGADDR_ADCPEAK##peak_reg | AD1835A_READ;  \
    /* Instruct AD1835A to send the peak register content */ \
    spiSendDataWordAndWait(r0);                             \
    /* Send dummy word to read the result */                \
    r0 = dm(AD1835A_REG_ADCCTRL3);                          \
    spiSendDataWordAndWait(r0);                             \
    /* Read the data word from the receive data buffer */   \
    spiReadRecvDataBuffer();                                \
    /* Mask the peak level and shift it to bit 0 */         \
    r1 = AD1835A_ADCPEAK_MASK;                              \
```

```
    r0 = r0 and r1;                                         \
149    r0 = lshift r0 by AD1835A_ADCPEAK_SHIFT;

151
//--------------------------------------------------------------
153 // Update the level meter for the AD1835A ADC inputs. This
// requires that the AD1835A is initialized, peak level readback
155 // is activated (ADC Control 3 register) and that the LEDs are
// routed and initialized.
157 #define ad1835aUpdateLevelMeter()     \
    call _ad1835a_update_level_meter;
159

161 /**************************************************************
 * CONSTANTS
163 **************************************************************/

165 //--------------------------------------------------------------
// Aliases for the bits for better readability
167
#define AD1835A_ZERO                (0x0000)
169
#define AD1835A_BIT0                (0x0001)
171 #define AD1835A_BIT1                (0x0002)
#define AD1835A_BIT2                (0x0004)
173 #define AD1835A_BIT3                (0x0008)
#define AD1835A_BIT4                (0x0010)
175 #define AD1835A_BIT5                (0x0020)
#define AD1835A_BIT6                (0x0040)
177 #define AD1835A_BIT7                (0x0080)
#define AD1835A_BIT8                (0x0100)
179 #define AD1835A_BIT9                (0x0200)
#define AD1835A_BIT10               (0x0400)
181 #define AD1835A_BIT11               (0x0800)
#define AD1835A_BIT12               (0x1000)
183 #define AD1835A_BIT13               (0x2000)
#define AD1835A_BIT14               (0x4000)
185 #define AD1835A_BIT15               (0x8000)

187
//--------------------------------------------------------------
189 // DAC control 1 register settings

191 // De-emphasis filter:
//  1) none
193 //  2) at 44.1kHz
//  3) at 32.0kHz
195 //  4) at 48.0kHz
#define AD1835A_DEEMPH_NONE         AD1835A_ZERO
197 #define AD1835A_DEEMPH_44_1         AD1835A_BIT8
#define AD1835A_DEEMPH_32           AD1835A_BIT9
199 #define AD1835A_DEEMPH_48           (AD1835A_BIT9 | AD1835A_BIT8)

201 // DAC data format:
//  1) Inter-IC Sound, I2S
203 //  2) Right-Justified, RJ
//  3) DSP Serial Port Mode, DSP
205 //  4) Left-Justified, LJ
//  5) Packed 256, P256
207 //  6) Packed 128, P128
#define AD1835A_DACFO_I2S           AD1835A_ZERO
209 #define AD1835A_DACFO_RJ            AD1835A_BIT5
#define AD1835A_DACFO_DSP           AD1835A_BIT6
211 #define AD1835A_DACFO_LJ            (AD1835A_BIT6 | AD1835A_BIT5)
#define AD1835A_DACFO_P256          AD1835A_BIT7
213 #define AD1835A_DACFO_P128          (AD1835A_BIT7 | AD1835A_BIT5)

215 // DAC data word width:
//  1) 24 bits
217 //  2) 20 bits
//  3) 16 bits
219 #define AD1835A_DACWORD_24BIT       AD1835A_ZERO
#define AD1835A_DACWORD_20BIT       AD1835A_BIT3
221 #define AD1835A_DACWORD_16BIT       AD1835A_BIT4

223 // DAC power-down reset:
//  1) Normal operation
225 //  2) Power-down
#define AD1835A_DACPWR_NORM         AD1835A_ZERO
227 #define AD1835A_DACPWR_DOWN         AD1835A_BIT2

229 // DAC sample rate:
```

```
// 1) 48kHz
// 2) 96kHz
// 3) 192kHz (only 1/2)
#define AD1835A_DACRATE_48          AD1835A_ZERO
#define AD1835A_DACRATE_96          AD1835A_BIT0
#define AD1835A_DACRATE_192         AD1835A_BIT1


//--------------------------------------------------------------
// DAC control 2 register settings

// Stereo replicate:
// Data sent to 1/2 is also output at 3/4, 5/6 and 7/8
#define AD1835A_DACREPLIC_OFF       AD1835A_ZERO
#define AD1835A_DACREPLIC_ON        AD1835A_BIT8

// Mute DAC output
// ON -> normal operation, mute -> mute input
#define AD1835A_DACOUT1L_ON         AD1835A_ZERO
#define AD1835A_DACOUT1L_MUTE       AD1835A_BIT0
#define AD1835A_DACOUT1R_ON         AD1835A_ZERO
#define AD1835A_DACOUT1R_MUTE       AD1835A_BIT1
#define AD1835A_DACOUT2L_ON         AD1835A_ZERO
#define AD1835A_DACOUT2L_MUTE       AD1835A_BIT2
#define AD1835A_DACOUT2R_ON         AD1835A_ZERO
#define AD1835A_DACOUT2R_MUTE       AD1835A_BIT3
#define AD1835A_DACOUT3L_ON         AD1835A_ZERO
#define AD1835A_DACOUT3L_MUTE       AD1835A_BIT4
#define AD1835A_DACOUT3R_ON         AD1835A_ZERO
#define AD1835A_DACOUT3R_MUTE       AD1835A_BIT5
#define AD1835A_DACOUT4L_ON         AD1835A_ZERO
#define AD1835A_DACOUT4L_MUTE       AD1835A_BIT6
#define AD1835A_DACOUT4R_ON         AD1835A_ZERO
#define AD1835A_DACOUT4R_MUTE       AD1835A_BIT7


//--------------------------------------------------------------
// DAC volume control register settings

// The lower 10 bits are used to set the volume level. The volume
// may be specified (linearly) from zero (0) to full scale (1023):
//
// 3FF -> 1023/1023 ->   0.00 dBFS
// 3FE -> 1022/1023 ->  -0.01 dBFS
// ...
// 002 ->    2/1023 -> -54.18 dBFS
// 001 ->    1/1023 -> -60.20 dBFS
//
#define AD1835A_DACVOL_MIN          (0x0000)
#define AD1835A_DACVOL_MAX          (0x03FF)

#define AD1835A_DACVOL_LOW          (0x0100)
#define AD1835A_DACVOL_MED          (0x0200)
#define AD1835A_DACVOL_HI           (0x0300)


//--------------------------------------------------------------
// ADC peak register settings

// Mask to extract the data bits
#define AD1835A_ADCPEAK_MASK        (0x03F0)

// Number of bits to shift the data bits correctly
#define AD1835A_ADCPEAK_SHIFT       (-4)


//--------------------------------------------------------------
// ADC control 1 register settings

// Filter:
// 1) All pass
// 2) High pass
#define AD1835A_ADCFILT_ALLPASS     AD1835A_ZERO
#define AD1835A_ADCFILT_HIGHPASS    AD1835A_BIT8

// ADC power-down:
// 1) Normal operation
// 2) Power-down
#define AD1835A_ADCPWR_NORM         AD1835A_ZERO
#define AD1835A_ADCPWR_DOWN         AD1835A_BIT7

// ADC sample rate:
```

```
//  1) 48kHz
313 //  2) 96kHz
#define AD1835A_ADCRATE_48          AD1835A_ZERO
315 #define AD1835A_ADCRATE_96          AD1835A_BIT6

317
//-----------------------------------------------------------------
319 // ADC control 2 register settings

321 // Master/slave auxiliary mode
#define AD1835A_ADCAUXMD_SLAVE      AD1835A_ZERO
323 #define AD1835A_ADCAUXMD_MASTER     AD1835A_BIT9

325 // ADC data format:
//  1) Inter-IC Sound, I2S
327 //  2) Right-Justified, RJ
//  3) DSP Serial Port Mode, DSP
329 //  4) Left-Justified, LJ
//  5) Packed 256, P256
331 //  6) Packed 128, P128
//  7) Auxiliary 256, AUX256
333 //  8) Auxiliary 512, AUX512
#define AD1835A_ADCFO_I2S           AD1835A_ZERO
335 #define AD1835A_ADCFO_RJ            AD1835A_BIT6
#define AD1835A_ADCFO_DSP           AD1835A_BIT7
337 #define AD1835A_ADCFO_LJ            (AD1835A_BIT7 | AD1835A_BIT6)
#define AD1835A_ADCFO_P256          AD1835A_BIT8
339 #define AD1835A_ADCFO_P128          (AD1835A_BIT8 | AD1835A_BIT6)
#define AD1835A_ADCFO_AUX256        (AD1835A_BIT8 | AD1835A_BIT7)
341 #define AD1835A_ADCFO_AUX512        (AD1835A_BIT8 | AD1835A_BIT7 | AD1835A_BIT6)

343 // ADC data word width:
//  1) 24 bits
345 //  2) 20 bits
//  3) 16 bits
347 #define AD1835A_ADCWORD_24BIT       AD1835A_ZERO
#define AD1835A_ADCWORD_20BIT       AD1835A_BIT4
349 #define AD1835A_ADCWORD_16BIT       AD1835A_BIT5

351 // Mute ADC:
//  ON -> normal operation, mute -> mute input
353 #define AD1835A_ADCIN1L_ON          AD1835A_ZERO
#define AD1835A_ADCIN1L_MUTE        AD1835A_BIT0
355 #define AD1835A_ADCIN1R_ON          AD1835A_ZERO
#define AD1835A_ADCIN1R_MUTE        AD1835A_BIT1
357

359 //-----------------------------------------------------------------
// ADC control 3 register settings
361
// Internal master clock: Scaling of input master clock MCLK
363 //  1) IMCLK = MCLK * 2
//  2) IMCLK = MCLK * 1
365 //  3) IMCLK = MCLK * 2/3
#define AD1835A_ADCIMCLK_2MCLK      AD1835A_ZERO
367 #define AD1835A_ADCIMCLK_1MCLK      AD1835A_BIT6
#define AD1835A_ADCIMCLK_23MCLK     AD1835A_BIT7
369
// ADC peak level readback
371 #define AD1835A_ADCPEAKRB_OFF       AD1835A_ZERO
#define AD1835A_ADCPEAKRB_ON        AD1835A_BIT5
373

375 //-----------------------------------------------------------------
// ADC input level meter threshold step
377 #define LEVEL_THRES_STEP            10

379
/*****************************************************************
381  * CONSTANTS (Required by the macros and initial configuration)
 *****************************************************************/
383
//-----------------------------------------------------------------
385 // Register addresses inside the configuration buffer

387 #define AD1835A_REG_DACCTRL1        (_ad1835a_config_buffer)
#define AD1835A_REG_DACCTRL2        (_ad1835a_config_buffer + 1)
389 #define AD1835A_REG_DACVOL1L        (_ad1835a_config_buffer + 2)
#define AD1835A_REG_DACVOL1R        (_ad1835a_config_buffer + 3)
391 #define AD1835A_REG_DACVOL2L        (_ad1835a_config_buffer + 4)
#define AD1835A_REG_DACVOL2R        (_ad1835a_config_buffer + 5)
393 #define AD1835A_REG_DACVOL3L        (_ad1835a_config_buffer + 6)
```

```
#define AD1835A_REG_DACVOL3R        (_ad1835a_config_buffer + 7)
#define AD1835A_REG_DACVOL4L        (_ad1835a_config_buffer + 8)
#define AD1835A_REG_DACVOL4R        (_ad1835a_config_buffer + 9)
#define AD1835A_REG_ADCCTRL1        (_ad1835a_config_buffer + 10)
#define AD1835A_REG_ADCCTRL2        (_ad1835a_config_buffer + 11)
#define AD1835A_REG_ADCCTRL3        (_ad1835a_config_buffer + 12)


//-----------------------------------------------------------------
// Register addresses inside the AD1835A

#define AD1835A_REGADDR_DACCTRL1    (0x0000)
#define AD1835A_REGADDR_DACCTRL2    (0x1000)
#define AD1835A_REGADDR_DACVOL1     (0x2000)
#define AD1835A_REGADDR_DACVOL2     (0x3000)
#define AD1835A_REGADDR_DACVOL3     (0x4000)
#define AD1835A_REGADDR_DACVOL4     (0x5000)
#define AD1835A_REGADDR_DACVOL5     (0x6000)
#define AD1835A_REGADDR_DACVOL6     (0x7000)
#define AD1835A_REGADDR_DACVOL7     (0x8000)
#define AD1835A_REGADDR_DACVOL8     (0x9000)
#define AD1835A_REGADDR_ADCPEAK0    (0xA000)
#define AD1835A_REGADDR_ADCPEAK1    (0xB000)
#define AD1835A_REGADDR_ADCCTRL1    (0xC000)
#define AD1835A_REGADDR_ADCCTRL2    (0xD000)
#define AD1835A_REGADDR_ADCCTRL3    (0xE000)

// Aliases
#define AD1835A_REGADDR_ADCPEAK1L   AD1835A_REGADDR_ADCPEAK0
#define AD1835A_REGADDR_ADCPEAK1R   AD1835A_REGADDR_ADCPEAK1


//-----------------------------------------------------------------
// R/~W bit

#define AD1835A_READ                AD1835A_BIT11
#define AD1835A_WRITE               AD1835A_ZERO


//-----------------------------------------------------------------
// Bit masks to clear a setting in a control register

// DAC control 1 register
#define AD1835A_DEEMPH_MASK         (~(AD1835A_BIT9 | AD1835A_BIT8))
#define AD1835A_DACFO_MASK          (~(AD1835A_BIT7 | AD1835A_BIT6 | AD1835A_BIT5))
#define AD1835A_DACWORD_MASK        (~(AD1835A_BIT4 | AD1835A_BIT3))
#define AD1835A_DACPWR_MASK         (~AD1835A_BIT2)
#define AD1835A_DACRATE_MASK        (~(AD1835A_BIT1 | AD1835A_BIT0))

// DAC control 2 register
#define AD1835A_DACREPLIC_MASK      (~AD1835A_BIT8)
#define AD1835A_DACOUT1L_MASK       (~AD1835A_BIT0)
#define AD1835A_DACOUT1R_MASK       (~AD1835A_BIT1)
#define AD1835A_DACOUT2L_MASK       (~AD1835A_BIT2)
#define AD1835A_DACOUT2R_MASK       (~AD1835A_BIT3)
#define AD1835A_DACOUT3L_MASK       (~AD1835A_BIT4)
#define AD1835A_DACOUT3R_MASK       (~AD1835A_BIT5)
#define AD1835A_DACOUT4L_MASK       (~AD1835A_BIT6)
#define AD1835A_DACOUT4R_MASK       (~AD1835A_BIT7)

// ADC control 1 register
#define AD1835A_ADCFILT_MASK        (~AD1835A_BIT8)
#define AD1835A_ADCPWR_MASK         (~AD1835A_BIT7)
#define AD1835A_ADCRATE_MASK        (~AD1835A_BIT6)

// ADC control 2 register
#define AD1835A_ADCAUXMD_MASK       (~AD1835A_BIT9)
#define AD1835A_ADCFO_MASK          (~(AD1835A_BIT8 | AD1835A_BIT7 | AD1835A_BIT6))
#define AD1835A_ADCWORD_MASK        (~(AD1835A_BIT5 | AD1835A_BIT4))
#define AD1835A_ADCIN1R_MASK        (~AD1835A_BIT1)
#define AD1835A_ADCIN1L_MASK        (~AD1835A_BIT0)

// ADC control 3 register
#define AD1835A_ADCIMCLK_MASK       (~(AD1835A_BIT7 | AD1835A_BIT6))
#define AD1835A_ADCPEAKRB_MASK      (~AD1835A_BIT5)


//-----------------------------------------------------------------
// Map a setting to its control register

// DAC control 1 register
#define AD1835A_DEEMPH_REG          AD1835A_REG_DACCTRL1
```

```
#define AD1835A_DACFO_REG           AD1835A_REG_DACCTRL1
#define AD1835A_DACWORD_REG         AD1835A_REG_DACCTRL1
#define AD1835A_DACPWR_REG          AD1835A_REG_DACCTRL1
#define AD1835A_DACRATE_REG         AD1835A_REG_DACCTRL1

// DAC control 2 register
#define AD1835A_DACREPLIC_REG       AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT1L_REG        AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT1R_REG        AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT2L_REG        AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT2R_REG        AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT3L_REG        AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT3R_REG        AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT4L_REG        AD1835A_REG_DACCTRL2
#define AD1835A_DACOUT4R_REG        AD1835A_REG_DACCTRL2

// ADC control 1 register
#define AD1835A_ADCFILT_REG         AD1835A_REG_ADCCTRL1
#define AD1835A_ADCPWR_REG          AD1835A_REG_ADCCTRL1
#define AD1835A_ADCRATE_REG         AD1835A_REG_ADCCTRL1

// ADC control 2 register
#define AD1835A_ADCAUXMD_REG        AD1835A_REG_ADCCTRL2
#define AD1835A_ADCFO_REG           AD1835A_REG_ADCCTRL2
#define AD1835A_ADCWORD_REG         AD1835A_REG_ADCCTRL2
#define AD1835A_ADCIN1R_REG         AD1835A_REG_ADCCTRL2
#define AD1835A_ADCIN1L_REG         AD1835A_REG_ADCCTRL2

// ADC control 3 register
#define AD1835A_ADCIMCLK_REG        AD1835A_REG_ADCCTRL3
#define AD1835A_ADCPEAKRB_REG       AD1835A_REG_ADCCTRL3

#endif
```

C.3.2 ad1835a.asm

```
/*****************************************************************
 * File: ad1835a.asm
 *
 * This file provides subroutines for the configuration of the
 * AD1835A codec via the primary SPI port of the DSP.
 *
 * Revisions:
 *   2010-05-03, Matthias Hotz: Initial version
 *   2010-06-02, Matthias Hotz: Added level meter
 *
 *****************************************************************/

#include <def21369.h>
#include "ad1835a.h"
#include "spi.h"
#include "led.h"

// Reset value of the SPI port flag register
#define SPIFLG_RSTVAL 0x0F80

.global _ad1835a_config_buffer;
.global _ad1835a_init;
.global _ad1835a_update_level_meter;


/*****************************************************************
 * INTERNAL MACROS
 *****************************************************************/

// Converts the peak level stored in r0 to a bit mask for the
// LEDs stored in r2. The threshold is assumed to be stored in
// r1.
#define levelToFlags(chname, lev_lo, lev_med1, lev_med2, lev_hi) \
    r0 = r0 - r1;                                                \
    if GT jump chname##_level2;                                  \
        r2 = lev_hi | lev_med2 | lev_med1 | lev_lo;              \
        jump chname##_end;                                       \
    chname##_level2:                                             \
    r0 = r0 - r1;                                                \
    if GT jump chname##_level3;                                  \
        r2 = lev_med2 | lev_med1 | lev_lo;                       \
        jump chname##_end;                                       \
    chname##_level3:                                             \
    r0 = r0 - r1;                                                \
    if GT jump chname##_level4;                                  \
```

```
        r2 = lev_med1 | lev_lo;                                  \
        jump chname##_end;                                       \
    chname##_level4:                                             \
    r0 = r0 - r1;                                                \
    if GT jump chname##_end;                                     \
        r2 = lev_lo;                                             \
    chname##_end:


/*****************************************************************
 * CONFIGURATION DATA
 *****************************************************************/
.section/dm seg_dmda;

// Buffer for the configuration of the AD1835A initialized with
// default configuration
.var _ad1835a_config_buffer[] =

    // DAC control 1 register
    AD1835A_REGADDR_DACCTRL1     | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DEEMPH_NONE      | // No de-emphasis filter
        AD1835A_DACFO_I2S        | // Inter-IC sound data format
        AD1835A_DACWORD_24BIT    | // 24 bit DAC word width
        AD1835A_DACPWR_NORM      | // Normal operation (not power-down)
        AD1835A_DACRATE_48,        // 48kHz sample rate

    // DAC control 2 register
    AD1835A_REGADDR_DACCTRL2     | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACREPLIC_OFF    | // No stereo replication
        AD1835A_DACOUT1L_ON      | // DAC 1 left not muted
        AD1835A_DACOUT1R_ON      | // DAC 1 right not muted
        AD1835A_DACOUT2L_ON      | // DAC 2 left not muted
        AD1835A_DACOUT2R_ON      | // DAC 2 right not muted
        AD1835A_DACOUT3L_ON      | // DAC 3 left not muted
        AD1835A_DACOUT3R_ON      | // DAC 3 right not muted
        AD1835A_DACOUT4L_ON      | // DAC 4 left not muted
        AD1835A_DACOUT4R_ON,       // DAC 4 right not muted

    // DAC 1 left volume control register
    AD1835A_REGADDR_DACVOL1      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // DAC 1 right volume control register
    AD1835A_REGADDR_DACVOL2      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // DAC 2 left volume control register
    AD1835A_REGADDR_DACVOL3      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // DAC 2 right volume control register
    AD1835A_REGADDR_DACVOL4      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // DAC 3 left volume control register
    AD1835A_REGADDR_DACVOL5      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // DAC 3 right volume control register
    AD1835A_REGADDR_DACVOL6      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // DAC 4 left volume control register
    AD1835A_REGADDR_DACVOL7      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // DAC 4 right volume control register
    AD1835A_REGADDR_DACVOL8      | // Register address
        AD1835A_WRITE            | // Write to register
        AD1835A_DACVOL_MAX,        // Maximum volume

    // ADC control 1 register
    AD1835A_REGADDR_ADCCTRL1     | // Register address
```

```
        AD1835A_WRITE             | // Write to register
        AD1835A_ADCFILT_ALLPASS   | // No high pass filter
        AD1835A_ADCPWR_NORM       | // Normal operation (not power-down)
        AD1835A_ADCRATE_48,         // 48kHz sample rate

    // ADC control 2 register
    AD1835A_REGADDR_ADCCTRL2      | // Register address
        AD1835A_WRITE             | // Write to register
        AD1835A_ADCAUXMD_SLAVE    | // Slave in auxiliary mode
        AD1835A_ADCFO_I2S         | // Inter-IC sound data format
        AD1835A_ADCWORD_24BIT     | // 24 bit ADC word width
        AD1835A_ADCIN1L_ON        | // ADC left active (not muted)
        AD1835A_ADCIN1R_ON,         // ADC right active (not muted)

    // ADC control 3 register
    AD1835A_REGADDR_ADCCTRL3      | // Register address
        AD1835A_WRITE             | // Write to register
        AD1835A_ADCIMCLK_2MCLK    | // Internal master clock = 2 * MCLK
        AD1835A_ADCPEAKRB_OFF;      // Peak level readback deactivated


/****************************************************************
 * GLOBAL SUBROUTINES
 ****************************************************************/
.section/pm seg_pmco;

//--------------------------------------------------------------
// Initialize the AD1835A CODEC
_ad1835a_init:

    // Enable the device select signal for the AD1835A
    r0 = SPIFLG_RSTVAL | DS3EN;
    dm(SPIFLG) = r0;

    // Send the complete configuration to the AD1835A
    spiSendDataBuffer(_ad1835a_config_buffer,
        length(_ad1835a_config_buffer));

_ad1835a_init.end:
    rts;

//--------------------------------------------------------------
// Update the level meter for the AD1835A ADC channels. This
// requires that the AD1835A is initialized, peak level readback
// is activated (ADC Control 3 register) and that the LEDs are
// routed and initialized.
_ad1835a_update_level_meter:

    // Read the left peak level register and generate its LED mask
    ad1835aReadPeakRegister(1L);
    r1 = LEVEL_THRES_STEP;
    r2 = 0;
    levelToFlags(left, LED5_ON, LED6_ON, LED7_ON, LED8_ON);
    r7 = r2;

    // Read the right peak level register and generate its LED mask
    ad1835aReadPeakRegister(1R);
    r1 = LEVEL_THRES_STEP;
    r2 = 0;
    levelToFlags(right, LED4_ON, LED3_ON, LED2_ON, LED1_ON);

    // Combine the LED masks and set the LEDs
    r0 = r2 or r7;
    ledSet(r0);

_ad1835a_update_level_meter.end:
    rts;
```

C.4 LED Routing and Control

C.4.1 led.h

```
/****************************************************************
 * File: led.h
 *
 * This file provides macros to route and control the general-
 * purpose LEDs of the ADSP-21369 EZ-KIT Lite board.
 *
 * Revisions:
 *   2010-06-02, Matthias Hotz: Initial version
```

```
 *
 ***************************************************************/
#ifndef _LED_H_
#define _LED_H_

.extern _led_init;
.extern _led_set;

/***************************************************************
 * MACROS
 ***************************************************************/

//-------------------------------------------------------------
// Macro to initialize and route the LEDs
#define ledInitialize() \
    call _led_init;


//-------------------------------------------------------------
// Macro set the LED status
//
// Parameters:
//   led_stat ... Status of the LEDs: Use the LEDx_x constants
//                and link them using a logical or operation.
//
// Example:
//   ledSet(LED1_ON | LED2_OFF | LED3_ON | LED4_OFF |
//       LED5_ON | LED6_OFF | LED7_ON | LED8_OFF);
//
#define ledSet(led_stat) \
    r5 = led_stat;       \
    call _led_set;


/***************************************************************
 * CONSTANTS
 ***************************************************************/

#define LED1_ON     0x01
#define LED2_ON     0x02
#define LED3_ON     0x04
#define LED4_ON     0x08
#define LED5_ON     0x10
#define LED6_ON     0x20
#define LED7_ON     0x40
#define LED8_ON     0x80

#define LED1_OFF    0x00
#define LED2_OFF    0x00
#define LED3_OFF    0x00
#define LED4_OFF    0x00
#define LED5_OFF    0x00
#define LED6_OFF    0x00
#define LED7_OFF    0x00
#define LED8_OFF    0x00

#endif
```

C.4.2 led.asm

```
/***************************************************************
 * File: led.asm
 *
 * This file provides subroutines to route and control the
 * general-purpose LEDs of the ADSP-21369 EZ-KIT Lite board.
 *
 * Revisions:
 *   2010-06-02, Matthias Hotz: Initial version
 *
 ***************************************************************/

#include <def21369.h>
#include <sru.h>
#include "led.h"

.global _led_init;
.global _led_set;


/***************************************************************
 * INTERNAL MACROS
```

```
 *****************************************************************/

// Macro to test the bit "led" in the USTAT1-register and set the
// bit "flgbit" in the FLAGS-register accordingly.
#define ledSetFlag(led, flgbit)     \
    bit tst ustat1 led##_ON;        \
    if not TF jump led##Off;        \
        bit set flags flgbit;       \
        jump led##End;              \
    led##Off:                       \
        bit clr flags flgbit;       \
    led##End:

// Macro to test the bit "led" in the USTAT1-register and set the
// pin buffer output "srupb" accordingly.
#define ledSetPinBufOut(led, srupb) \
    bit tst ustat1 led##_ON;        \
    if not TF jump led##Off;        \
        SRU(HIGH, srupb);           \
        jump led##End;              \
    led##Off:                       \
        SRU(LOW, srupb);            \
    led##End:


/*****************************************************************
 * GLOBAL SUBROUTINES
 *****************************************************************/
.section/pm seg_pmco;

//---------------------------------------------------------------
// Subroutine to initialize and route the LEDs
_led_init:

    // Route LEDs. Notes:
    //  * LED6 and LED7 are connected to DAI pins. The SRU1 of
    //    the DAI pins does not provide a connection to a general
    //    purpose (flag) register, for this reason the output is
    //    controlled by routing the pin buffer inputs to fixed
    //    logical levels.
    //  * LED8 is connected to the FLAG3 pin of the DSP.

    // LED1: Route to FLAG4 to the pin as output
    SRU(FLAG4_O, DPI_PB06_I);
    SRU(HIGH, DPI_PBEN06_I);

    // LED2: Route to FLAG5 to the pin as output
    SRU(FLAG5_O, DPI_PB07_I);
    SRU(HIGH, DPI_PBEN07_I);

    // LED3: Route to FLAG6 to the pin as output
    SRU(FLAG6_O, DPI_PB08_I);
    SRU(HIGH, DPI_PBEN08_I);

    // LED4: Route to FLAG7 to the pin as output
    SRU(FLAG7_O, DPI_PB13_I);
    SRU(HIGH, DPI_PBEN13_I);

    // LED5: Route to FLAG8 to the pin as output
    SRU(FLAG8_O, DPI_PB14_I);
    SRU(HIGH, DPI_PBEN14_I);

    // LED6: Configure as output and output logical low
    SRU(LOW, DAI_PB15_I);
    SRU(HIGH, PBEN15_I);

    // LED7: Configure as output and output logical low
    SRU(LOW, DAI_PB16_I);
    SRU(HIGH, PBEN16_I);

    // LED8: Set FLAG3-pin to "flag-mode"
    ustat1 = dm(SYSCTL);
    bit clr ustat1 TMREXPEN | MSEN;
    dm(SYSCTL) = ustat1;

    // Configure the FLAG-pins as output
    bit set flags FLG3O | FLG4O | FLG5O | FLG6O | FLG7O | FLG8O;

    // Initialize all LED outputs to logical low
    bit clr flags FLG3 | FLG4 | FLG5 | FLG6 | FLG7 | FLG8;

_led_init.end:
```

```
    rts;
105

107 //--------------------------------------------------------------
    // Subroutine to set the LED status
109 //
    // Parameters:
111 //   r5 ... Bit mask for the status of the LEDs
    //
113 _led_set:

115     // Use a universal status register for the bit tests
        ustat1 = r5;
117
        ledSetFlag(LED1, FLG4);
119     ledSetFlag(LED2, FLG5);
        ledSetFlag(LED3, FLG6);
121     ledSetFlag(LED4, FLG7);
        ledSetFlag(LED5, FLG8);
123     ledSetPinBufOut(LED6, DAI_PB15_I);
        ledSetPinBufOut(LED7, DAI_PB16_I);
125     ledSetFlag(LED8, FLG3);

127 _led_set.end:
        rts;
```