Features

- **Software Module Dedicated to Voice Processing and Multi-way Conferencing**
- **Optimized for the AT75 Series Smart Internet Appliance Processor (SIAP™)**
- **Includes Several Run-time Configurable Stand-alone Algorithms**
  - **G.723.1 Dual-rate Vocoder (5.3 Kbps/6.4 Kbps)**
  - **VAD/CNG Silence Compression (Annexe A of G.723.1)**
  - **G.711 µ-law or A-law Compression (64 Kbps)**
  - **Arbitrary Tone Generator**
  - **DTMF Detector**
  - **Echo Canceller**
- **ITU-T G.723.1 and G.711 Standard-compliant**
- **Either up to Two Decode Channels with G.723.1 Standard, or up to Four Decode Channels with G.711 Standard**
- **Available with a uClinux® Device Driver**

Overview

The AT75C1212 Multi-way Conferencing Software Module is designed to run on the OakDSPCore® subsystem of the AT75 series Smart Internet Appliance Processor. It implements commonly-used voice processing algorithms:

- Low bit-rate G.723.1 vocoder for multimedia communication.

- Silence compression algorithm to efficiently handle periods of silence during communication.

- High bit-rate voice compression algorithm.

- Arbitrary tone generator that can be used to generate any dual-tone or single-tone frequency during a programmable duration.

- DTMF detector to decode incoming DTMF signaling.

- An echo canceller that eliminates the near-end echo.

The implemented algorithms have a number of parameters which can be programmed at run time. These parameters modify the behavior of the DSP algorithms in such a manner that they comply with the applicable standards under most situations. They also allow the AT75C1212 to cope with many non-standard situations often encountered on private telephone infrastructures.

Moreover, the AT75C1212 module is able to perform multi-way conferencing. Either up to two independent decode channels with G.723.1 or up to four decode channels with either µ-law or A-law compression are available with no penalty on the voice quality.

The AT75C1212 takes advantage of the AT75 mailbox to exchange data with the on-chip ARM7TDMI® core. The organization of the data communication channel makes it easy to integrate the AT75C1212 interface into most operating systems.

For developers using uClinux, a specific device driver is supplied, thereby assuring the extension of uClinux capabilities to the complete functionality of the AT75C1212 module in a seamless manner.

This document is made up of three sections:

1. Functional description of the supported algorithms.

2. Description of the low level software interface.

3. Description of the uClinux device driver and full integration of AT75C1212 functionality.

Note: Mixing low-level and driver-level programming should be avoided.

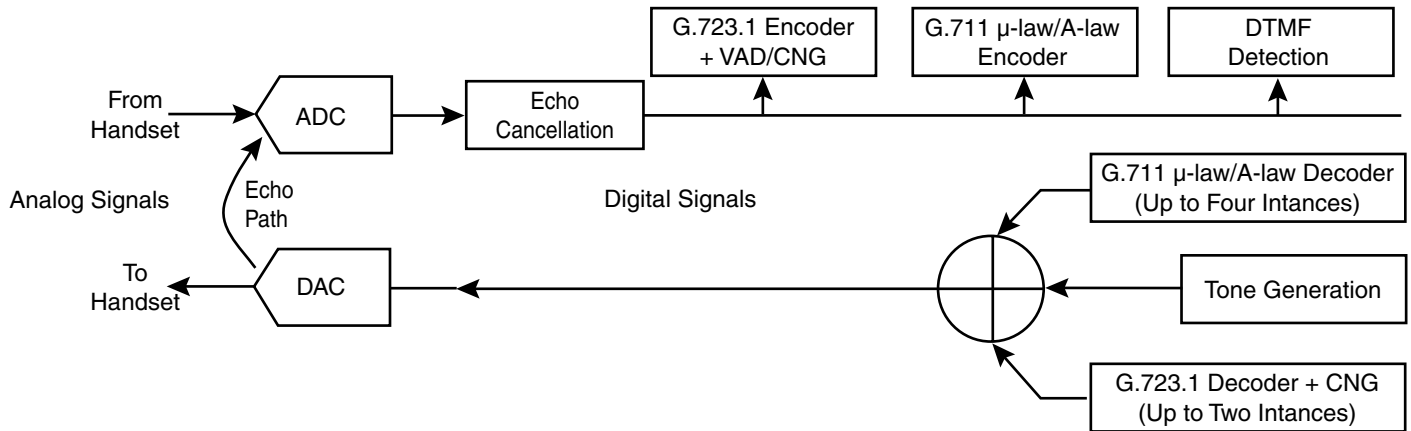
Smart Internet Appliance Processor (SIAP™)

AT75C1212 Multi-way Conferencing Software Module

Functional Description

A functional block diagram of the AT75C1212 Multi-way Conferencing Software Module is given in Figure 1.

Figure 1. Block Diagram

G.723.1 Dual Rate Vocoder

This algorithm can be used for compressing the speech or other audio signal components of a multimedia service at a very low bit rate. This coder has two bit rates associated with it: 5.3 and 6.4 Kbps. The higher bit rate has better quality; it is based on Multi Pulse Maximum Likelihood Quantization (MP-MLQ) technique. The lower bit rate gives good quality and provides system designers with additional flexibility. This rate is based on an Algebraic Code Linear Pre-diction (ACELP) technique.

This coder operates on 30 ms speech frames of 16-bit linear PCM samples (sampling frequency is 8 kHz). An algorithmic delay of 7.5 ms is to be taken into account before getting an encoded voice data frame. That leads to a total delay of 37.5 ms. The resulting encoded frames are 20 bytes long for 5.3 Kbps rate and 24 bytes long for 6.4 Kbps rate. The encoding rate can be chosen by means of a configuration command sent to the DSP. (See "Request Notification Messages" on page 9.)

VAD/CNG

Voice Activity Detection (VAD) and Comfort Noise Generator (CNG) algorithms are designed to work hand-in-hand with the G.723.1 vocoder. Silence compression techniques are used to reduce the transmitted bit rate during silent intervals of speech. The VAD side detects those silent intervals. CNG is used to produce a noise that matches the actual background noise. CNG uses information provided by VAD to encode silent intervals into Silence Insertion Descriptor (SID) frames that are 4 bytes long. It also re-synthesizes 16-bit linear PCM samples of background noise with a SID frame input. The VAD/CNG feature can be enabled or not by means of a configuration command sent to the DSP. (See "Request Notification Messages" on page 9.)

G.711 µ-law and A-law Voice Compression

µ-law and A-law are logarithmic compression techniques applied to speech signals. They are done by simple operations that give no delay and excellent quality of speech. However, the bit rate is very high (each 16-bit linear PCM speech sample gives an 8-bit compressed sample leading to 64 Kbps) making this feature useful only for broadband data networks. The compression/decompression algorithm can be chosen by means of a configuration command send to the DSP. (See "Request Notification Messages" on page 9.)

Echo Cancellation Operation

The AT75C1212 contains an echo cancellation unit to eliminate near-end echo. This unit is based on an adaptive FIR filter, which computes the expected echo and a subtractor, which removes it from the transmitted signal. Since the echo characteristics can slowly vary with time, an adaptive algorithm continuously updates the echo model.

A block diagram of the echo canceller is shown in Figure 2 below.

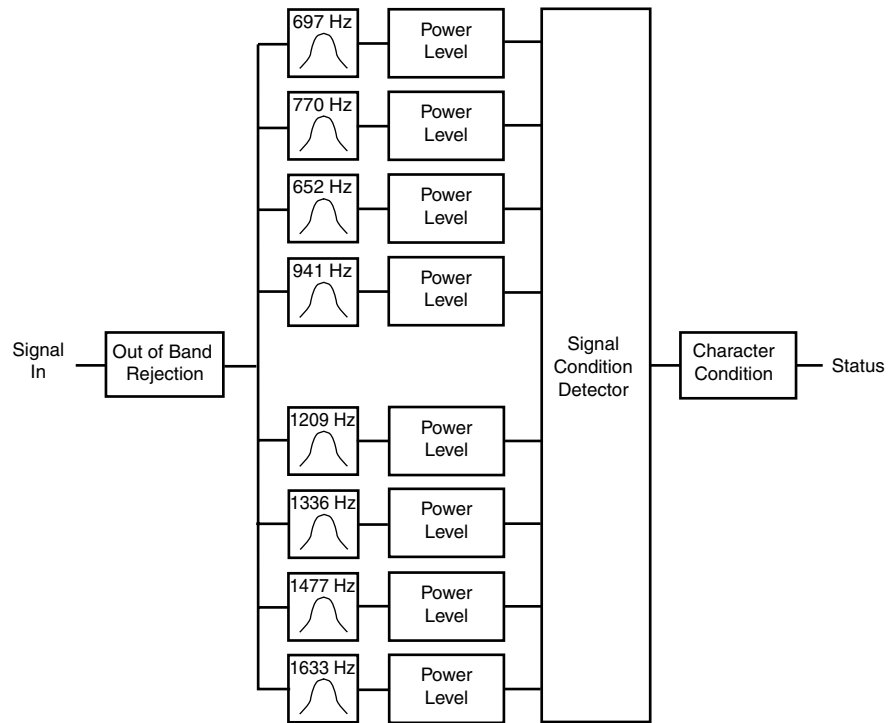**Figure 2.** Block Diagram of Echo Canceller
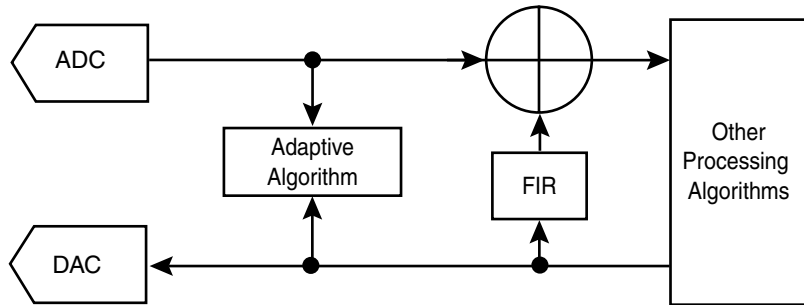
Multi-way Conferencing

AT75C1212 module allows several decoding channels to be active at the same time (up to two decode channels with G.723.1 standard or up to four with either PCM µ-law or PCM A-law).

DTMF Detector

The DTMF detection task detects and decodes the 16 standard DTMF signals, in compliance with the ITU-T Q.24 recommendation, with programmable threshold levels. The application program, to comply with special (i.e. non-standard) situations, can tune some parameters of the algorithm. In order to detect the DTMF signal, a bank of eight resonant band pass filters is used. The central frequency of each filter corresponds to one of the eight nominal values employed by standard DTMF generators. The power level at each filter output is used to check for signal presence, signal condition requirements, and character condition requirements.

Figure 3. DTMF Detector Block Diagram

697 Hz — Power Level

770 Hz — Power Level

652 Hz — Power Level

941 Hz — Power Level

Signal In — Out of Band Rejection

1209 Hz — Power Level

1336 Hz — Power Level

1477 Hz — Power Level

1633 Hz — Power Level

Signal Condition Detector — Character Condition — Status

The eight band pass filters are centered on the eight frequencies defined in the ITU-T Q.24 specification. The bandwidth is specified according to the tolerance established in this standard. Each filter rejects at least 20 dB of the other seven frequencies. The power level is obtained by averaging the instantaneous energy during a window of 2 ms for each of the eight filtered signals.

The detection of a DTMF signal requires that the following conditions be met:

- One frequency of each group is above a specified level.
- The power level difference between the low group tone and the high group tone is within a given interval (twist).
- The power level of the highest tone of each group is above a specified level above the other frequencies of the same group.

The character condition is fulfilled when:

- The signal condition is preceded by a different character recognition condition or by the continuous non-existence of a signal condition for a specified duration (silence).

- The signal condition for the same two tones exists continuously for a specified duration. When the signal condition is satisfied for less than a specified duration, the character is rejected. Once the character condition exists, it is unaffected by an interval shorter than a specified duration.

Tone Generator

The tone generation task generates a pure sine wave or a dual tone with programmable frequencies, amplitudes and duration.

Low Level Interface

This section describes how the AT75C1212 software is uploaded into the DSP subsystem program memory. It also describes how the application software running on the ARM and the AT75C1212 running on the DSP Subsystem exchange information through Dual Port Mailboxes (DPMB).

This section assumes in-depth knowledge of the ARM/DSP Subsystem interface mailbox system.

Voice Module Upload

While the DSP subsystem is held in reset, its program memory is made visible in the ARM memory space. This allows the ARM application to write a binary image of the DSP software very easily.

When the DSP subsystem is taken out of reset, its program memory is switched from the ARM memory space back to the DSP program space just before the first instruction is fetched. This process is illustrated in Figure 4.

Figure 4. Voice Module Upload
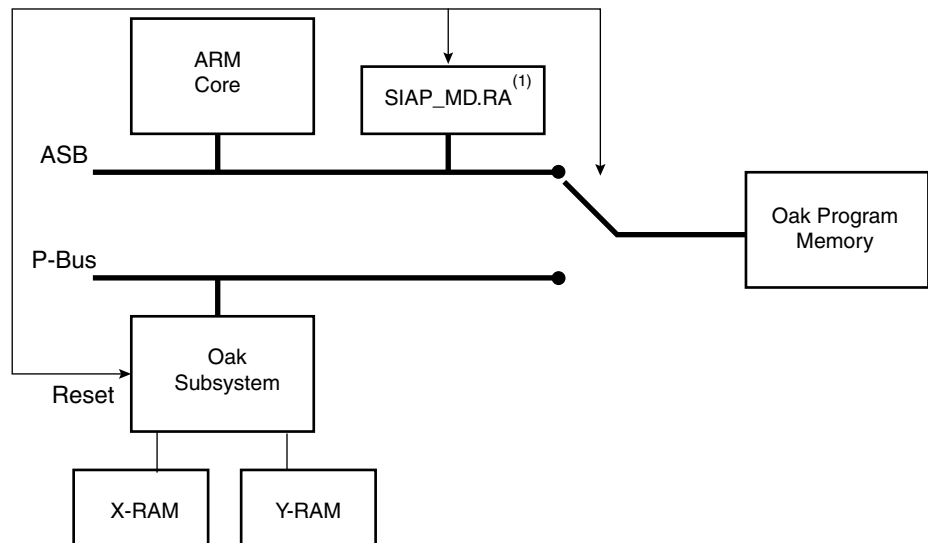
Note: 1. Bit RA in Register SIAP_MD.

Upload Process

A typical DSP program uses a number of initialized variables. Typically, the initial values are stored in the program space, and copied into their RAM location by the DSP start-up routine. This leads to the following statements:

- Just after the boot routine has initialized the variables, the DSP subsystem exhibits high redundancy since the same values exist in both program and data memories.

- The initial values stored in the program memory waste space and are not used during operation.

- To improve the program memory usage, the software is loaded in two consecutive steps.

- A small data initialization program is first loaded and executed. This program just initializes the X- and Y-RAM to the values expected by the audio decoder software. When the initialization is done, the program sends a DATA_INIT_DONE status message to the ARM application through the status mailbox.

- Then, the DSP subsystem is put in reset and the program itself is loaded. This code has no data init start-up routine. It assumes the RAMs are already initialized, which saves program space. When the software is ready to work, it sends a SW_INIT_DONE status message through the status mailbox.

The mailbox operation and status messages are described in the section "Mailbox Usage" on page 7.

Binary Image Format

When the system is idle, the AT75C1212 module is stored in the ARM memory space, possibly in non-volatile memory. The module contains the data initialization code, the application code, and additional formatting data. The various fields of the AT75C1212 binary image are described in Table 1.

Table 1. Binary Image Fields

Field Name	Offset from Start of File (Bytes)	Length (Bytes)	Description
INIT_OFFSET	0	4	Defines the position of the data initialization code from the beginning of the module image.
INIT_LENGTH	4	4	Defines the length of the data initialization code (16-bit words). Valid between 0 and 24576.
SW_OFFSET	8	4	Defines the position of the audio decoder program from the beginning of the module image.
SW_LENGHTH	12	4	Defines the length of the audio decoder code (16-bit words). Valid between 0 and 24576.
INIT_CODE	16	2*INIT_LENGTH	Binary code of the data initialization program.
SW_CODE	16 + 2*INIT_LENGTH	2*SW_LENGTH	Binary code of the application program.

Dual-port Mail Box Configuration

The dual-port mail box (DPMB) is programmed in configuration 0 (as defined in the AT75C DSP Subsystem Datasheet) and gives the configuration shown in Table 2 on page 7. All the mailboxes allow read/write access from both sides. Arbitration is done using the semaphores.

Table 2. DPMB Configuration

Mailbox #	Offset from Base[1]	Length	Direction	Semaphore Address[1]	Usage
0	0x000	0x80	ARM -> Oak	0x200	RX Voice Data
1	0x040	0x80	ARM <- Oak	0x204	TX Voice Data 1
2	0x080	0x40	ARM -> Oak	0x208	TX Voice Data 2
3	0x0C0	0x40	ARM -> Oak	0x20C	TX Voice Data 3
4	0x100	0x40	ARM -> Oak	0x210	TX Voice Data 4
5	0x140	0x40	ARM <- Oak	0x214	DSP Memory Access
6	0x180	0x40	ARM -> Oak	0x218	Request Notification
7	0x1C0	0x40	ARM <- Oak	0x21C	Status Notification

Note: 1. Base address is 0xFA000000.

Mailbox Access

ARM-to-Oak Mailboxes

Before accessing the ARM-to-Oak mailboxes, the ARM must check that the corresponding semaphore is cleared to 0. Then it can read or write the mailbox data. When the data access is done, it must set the semaphore to 1 to notify the Oak that new data has arrived.

Oak-to-ARM Mailboxes

The ARM is notified that new data is available in a mailbox when the corresponding semaphore is raised to 1, possibly triggering an interrupt. Then the ARM can access the mailbox. When the access is finished, the ARM must clear the semaphore to release the mailbox.

Mailbox Usage

This section describes the specific purpose of each mailbox. The exchanged information is formatted in structured messages. The message format and semantics are described in sections "Request Notification Messages" on page 9 and "Status Notification Messages" on page 15.

Mailbox 0: RX Encoded Voice Data

Used by the ARM to get from the OAK encoded speech frames (either G.711 data or G.723.1 data)

Mailbox 1 to 4: TX Encoded Voice Data

Used by the ARM to provide to the OAK encoded speech frames (either G.711 data or G.723.1 data).

Mailbox 5: Oak Memory Access

The ARM has the ability to send requests to read or write any location of the DSP memories, either in program or data space. This is useful for two purposes:

• DSP software debug

• Programming of the DSP peripherals under the ARM application control

Mailbox 6: Request Notification

This mailbox is used by the ARM to pass requests to the DSP. These requests trigger specific tasks in the DSP software. For example, request notification messages are used to start or to stop the telephony algorithms.

Mailbox 7: Status Notification

This mailbox is used by the DSP software to send status information. For example, a status notification message is sent by the DSP software at the end of the data initialization to notify the ARM application that the data has been initialized.

TX/RX Encoded Voice Data

The first two mailboxes deal with speech compressed frames. Each byte sent through the mailbox is put in a 16-bit word where low byte is the original byte value and in high byte are flags.

Assuming the data to be transmitted is in "char buf[0..N-1]", it is formatted in the mailbox as shown in Table 3 (otherwise the frame is ignored).

Table 3. Speech Frame Format[1]

Word 0	...	Word i (i = 1... N - 2)	...	Word N - 1
FRAME_START	buf[0]	...	0x0000	buf[i]	...	FRAME_END	buf[N - 1]

Note: 1.   With FRAME_START = 0x8000 and FRAME_END = 0x4000

Delivered frames are of variable length:

- The length is encoded within the first two bits of buf[0] for G.723.1:
    - buf[0] & 0x3 = 0 -> 24 bytes at a rate of 6.3 Kbits per second
    - buf[0] & 0x3 = 1 -> 20 bytes for a rate of 5.3 Kbits per second
    - buf[0] & 0x3 = 2 -> 4 bytes for silence compression frames
    - buf[0] & 0x3 = 3 -> 1 byte: it follows a 4-byte frame while the silence scheme is unchanged
- If the system is in G.711 mode, frames are 32 bytes long, independent of the contents of buf[0].

Oak Memory Access

The ARM has the ability to send requests to read or write any location of the Oak memories, either in program or data space. To achieve this, the mailbox 5 is divided into four fields:

- Command field (mailbox base + 0): This is a request ID that tells what kind of operation is to be performed. Valid codes are:
    - 0x0001: Program memory read
    - 0x0002: Program memory write
    - 0x0003: Data memory read
    - 0x0004: Data memory write
- Address field (base + 1 16-bit word): Should be written with the address location to be accessed. This is the value of the address as it is seen by the Oak.
- Length field (base + 2 16-bit words): Should be written with the number of consecutive locations to access.
- Data field (base + 3 16-bit words and following): For write access, should be filled with the values to write. For read access, contains the read values requested by the previous command.

Example of use: Write 0x1234 into data location 0xabcd of the 0ak:

1. Wait for *(0xfa000214) == 0, i.e., the semaphore is cleared
2. *(0xfa000140) = 0x0004 // data write command
3. *(0xfa000142) = 0xabcd // this is the address
4. *(0xfa000144) = 0x0001 // only one word to write
5. *(0cfa000146) = 0x1234 // this is the value
6. *(0xfa000214) = 1 // notify the Oak

Example of use: Read data locations 0xabcd and 0xabce from Oak:

1. Wait for *(0xfb000214) == 0, i.e. the semaphore is cleared
2. *(0xfa000140) = 0x0003 // data read command
3. *(0xfa000142) = 0xabcd // this is the first address to read
4. *(0xfa000144) = 0x0002 // two words to read
5. *(0xfa000214) = 1 // notify the Oak
6. Wait for the semaphore to go back to 0.
7. Read 0xfa000146 and 0xfa000148 to get the requested values.

Request Notification Messages

Request messages are used by the ARM to trigger specific tasks running on the DSP. These messages are always formatted in the same way. Figure 5 describes this format.

Figure 5. Request Notification Message Format
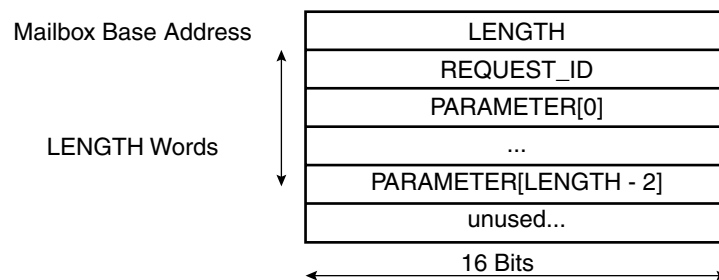
A message always begins with a LENGTH field. This field contains the number of words of the message, excluding the LENGTH field itself.

The REQUEST_ID field is uniquely defined to designate the type of request. Each request can be followed by a variable but well-defined number of PARAMETER fields. These fields contain additional data needed to handle the request.

The description of the supported request messages is listed below. It is forbidden for the ARM application to issue unsupported messages. However, should the ARM application issue an unsupported or malformed request, the Oak software must recover correctly.

Decoding Configuration Request

The decoding configuration request is sent to the Oak before any voice decoding operation.

Table 4. Decoding Configuration Request

Word 0	0x0003	Message length = 0x0003
Word 1	0x0201	Request ID = 0x0201
Word 2	TX_ID	TX Channel's ID. Valid: 0 to 3
Word 3	TX_TYPE	TX Channel's type: G.723.1: TX_TYPE = 0x0004 μ-law: TX_TYPE = 0x0000 A-law: TX_TYPE = 0x0008

Example: 0x0003 0x0201 0x0000 0x0004

This request will set up Decode channel 0 with G.723.1 standard.

Encoding Configuration Request

The encoding configuration request is sent to the Oak before any voice data encoding operation.

Table 5. Encoding Configuration Request

Word 0	0x0002	Message length = 0x0002
Word 1	0x0202	Request ID = 0x0202
Word 2	RX_TYPE	RX Channel's type: G.723.1: RX_TYPE = 0x0004 μ-law: RX_TYPE = 0x0000 A-law: RX_TYPE = 0x0008

Example: 0x0002 0x0202 0x0004

This request will set up Record channel with G.723.1 standard.

Volume Configuration Request

The volume configuration request is sent to the Oak to set volume parameters.

Table 6. Volume Configuration Request

Word 0	0x0003	Message length = 0x0003
Word 1	0x0300	Request ID = 0x0300
Word 2	MICR_GAIN = 0x1000 * 10E(dB/20)	Gain for the microphone input. Valid: 0x0040 (- 36 dB) to 0x7FFF (+18 dB)
Word 3	SPKR_GAIN = 0x1000 * 10E(dB/20)	Gain for the speakerphone input. Valid: 0x0040 (- 36 dB) to 0x7FFF (+18 dB)

Volume Up Request

The volume up request is sent to the Oak to increase the speakerphone volume. It can be sent at anytime.

Table 7. Volume Up Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0301	Request ID = 0x0301

Volume Down Request

The volume down request is sent to the Oak to lower the speakerphone volume. It can be sent at anytime.

Table 8. Volume Down Request

Word 0	0x0003	Message length = 0x0001
Word 1	0x0300	Request ID = 0x0302

G.723.1 Configuration Request

The G.723.1 configuration request should be sent to the Oak before enabling any G.723.1 encoding operation, otherwise the default configuration will be used (6.4 Kbits/s rate, VAD/CNG disabled).

Table 9. G.723.1 Configuration Request

Word 0	0x0003	Message length = 0x0003
Word 1	0x0400	Request ID = 0x0400
Word 2	WORKRATE	Work rate for encoding, valid values: 0: 6.4 Kbits/s rate 1: 5.3 Kbits/s rate
Word 3	USEVX	0: disable VAD for encoding 1: enable VAD for encoding

Echo Cancellation Configuration Request

The echo cancellation configuration request should be sent to the Oak before enabling any echo cancellation operation, otherwise the default configuration will be used.

Table 10. Echo Cancellation Configuration Request

Word 0	0x0006	Message length = 0x0006
Word 1	0x0860	Request ID = 0x0860
Word 2	ECHO_SIZE	Filter size in blocks of 16 – valid: 1 to 15 (between 16 and 240 filter coefficients) Default: 4 (64 coefficients: cancellation up to an echo path of 8ms)
Word 3	ECHO_UPDATE	Number of coefficients updated at each sample Interval. It must be a sub-multiple of the filter size (otherwise an invalid parameter status is generated) Default: 64 (all the filter is updated)
Word 4	ECHO_STEPSZ	Step size Default: 0x7F00
Word 5	ECHO_TIMECST	Time constant of power estimation filter (ms) Default: 32 ms
Word 6	ECHO_MUCALC	Time interval between calculations of normalized step size in milliseconds. Default: 1 ms

Echo Cancellation Step-size Adjust Request

The echo cancellation step-size adjust request can be sent to the Oak at any time during the echo cancellation operation in order to enhance convergence characteristics.

Table 11. Echo Cancellation Step-size Adjust Request

Word 0	0x0002	Message length = 0x0002
Word 1	0x0863	Request ID = 0x0863
Word 3	ECHO_STEPSZ	Step size

Start Decoding Request

The start decoding request is sent to the Oak to start decode operations as configured by the decoding configuration request.

Table 12. Start Decoding Request

Word 0	0x0002	Message length = 0x0002
Word 1	0x0401	Request ID = 0x0401
Word 2	TX_ID	TX Channel's ID. Valid: 0 to 3

Stop Decoding Request

The stop decoding request is sent to the Oak to stop one of the active decode channels as configured by the decoding configuration request.

Table 13. Stop Decoding Request

Word 0	0x0002	Message length = 0x0002
Word 1	0x0402	Request ID = 0x0402
Word 2	TX_ID	TX Channel's ID. Valid: 0 to 3

Start Record Request

The start record request is sent to the Oak to start the recording operation as configured by encoding configuration request.

Table 14. Start Record Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0403	Request ID = 0x0403

Stop Record Request

The stop record request is sent to the Oak to stop the record channel as configured by multi-way configuration request.

Table 15. Stop Record Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0404	Request ID = 0x0404

Start Echo Cancellation Request

The start echo cancellation request is sent to the Oak to start echo cancellation operations on one channel.

Table 16. Start Echo Cancellation Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0861	Request ID = 0x0861

Stop Echo Cancellation Request

The stop echo cancellation request is sent to the Oak to stop echo cancellation on one channel.

Table 17. Stop Echo Cancellation Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0862	Request ID = 0x0862

DTMF Detection Configuration Request

Table 18. DTMF Detection Configuration Request

Word 0	0x0009	Message length = 0x0009
Word 1	0x0830	Request ID = 0x0830
Word 2	DTMFDET_LOWTHRES = 65535 * 10E(dB/10)	Low Group Power Detection Threshold (default: -40dB)
Word 3	DTMFDET_HIGHTHRES = 65535* 10E(dB/10)	High Group Power Detection Threshold (default: -40dB)
Word 4	DTMFDET_LOWREL = 65535* 10E(dB/20)	Minimum difference level between the strongest frequency in the low group and the other of the same group (default: -20dB)
Word 5	DTMFDET_HIGHREL = 65535* 10E(dB/20)	Minimum difference level between the strongest frequency in the high group and the other of the same group (default: -20dB)
Word 6	DTMFDET_POSTWIST = 65535* 10E(dB/10)	Maximum Low to High twist (default: -8dB)
Word 7	DTMFDET_ NEGTWIST = 65535* 10E(dB/10)	Maximum High to Low twist (default: -8dB)
Word 8	DTMFDET_DURATION	Duration of signal condition for character recognition in milliseconds (default: 30 ms).
Word 9	DTMFDET_SILENCE	Duration of silence condition for character recognition in milliseconds (default: 30 ms).

Example: 0x0009 0x0830 0x0020 0x0020 0x2000 0x2000 0x4000 0x4000 0x001E 0x001E:

This message configures the DTMF detector with a detection level of 33 dB below the reference level for each group. The minimum difference level between the strongest frequency in a group and the others of the same group must be at least of 18 dB. The maximum difference level between the two groups must be at most 12 dB. In order to recognize a character, the signal must last for a minimum of 30 ms and then be followed by 30 ms of silence.

DTMF Detection Start Request DTMF detection is started as soon as the DSP unit receives the DTMF detection start request.

Table 19. DTMF Detection Start Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0831	Request ID = 0x0831

DTMF Detection Stop Request DTMF detection is stopped as soon as the DSP unit receives the DTMF detection stop request.

Table 20. DTMF Detection Stop Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0832	Request ID = 0x0832

Tone Generation Configuration Request

Table 21. Tone Generation Configuration Request

Word 0	0x000A	Message length = 0x000A
Word 1	0x0800	Request ID = 0x0800
Word 2	32768 * cos (pi* TONE_FREQ /4000)	Words 2 and 3 define the frequency of the first generated tone.
Word 3	32768 * sin (pi* TONE_FREQ /4000)	
Word 4	TONE_LEVEL = 32768 * 10E(dB/20)	Level of the first generated tone.
Word 5	32768 * cos (pi* TONE_FREQ /4000)	Words 5 and 6 define the frequency of the second component of generated tone.
Word 6	32768 * sin (pi* TONE_FREQ /4000)	
Word 7	TONE_LEVEL = 32768 * 10E(dB/20)	Level of the second generated tone.
Word 8	TONE_DURATION	Duration of the generated tone in milliseconds 0x0000 means unlimited duration.
Word 9	SILENCE _DURATION	Duration of the silence following the tone in milliseconds 0x0000 means unlimited duration.
Word 10	TONE_START	**Bit 0**: 0: causes the generator to wait for a tone generation start request (request ID 0x0801) before the tone is generated. 1: The generation starts immediately. **Bit 1**: 0: adds the tone to all other signals emitted on the speaker. 1: All other signals are blocked while the tone is generated. **Bit 2**: 0: single tone generation, parameters of the second component are ignored. 1: Dual tone generation.

Example: 0x000A 0x0801 0x6D4B 0x429F 0x4000 0x3FC4 0x6EFB 0x3000 0x0080 0x0080 0x0007

This message configures the generator to emit a dual tone with:

- First component. 697 Hz tone 6 dB below the reference level.
- Second component. 1336 Hz tone 8.5 dB below the reference level.

DTMF digit "2" will have been recognized.

The tone is emitted as soon as the DSP unit receives the request. After 128 ms of signal and 128 ms of silence, a tone generation done status message will be emitted.

Tone Generation Start Request

The tone starts as soon as the DSP unit receives tone generation start request. A tone generation configuration request (request ID 0x0800) should be issued before the tone generation start request is sent. If not, the behavior of the tone generator is unpredictable.

Table 22. Tone Generation Start Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0801	Request ID = 0x0801

Tone Generation Stop Request

The tone stops as soon as the DSP unit receives the tone generation stop request. This request can be used to stop an unlimited tone generation, or to halt the generator before the predefined duration has elapsed (early termination).

Table 23. Tone Generation Stop Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0802	Request ID = 0x0802

Status Notification Messages

Status messages are used by the Oak to inform the ARM application that a specific event has occurred, or to respond to an earlier request. These messages are always formatted in the same way. Figure 6 describes this format.

Figure 6. Status Notification Message Format
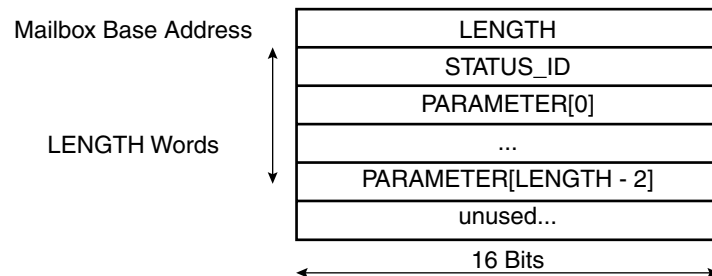
A status message always begins with a LENGTH field. This field contains the number of words of the message, excluding the LENGTH field itself.

The STATUS_ID field is uniquely defined to designate the type of status. Each status can be followed by a variable but well-defined number of PARAMETER fields. These fields contain additional status information.

The description of the supported status messages is listed below. It is forbidden for the Oak program to issue unsupported status messages. However, should the Oak program issue an unsupported or malformed status message, the ARM application must recover correctly.

AT75C1212 Module Initialization Status

This initialization status message is issued when the AT75C1212 module has finished initializing itself and is ready to accept request messages. The ARM should not issue any request messages before this status message has been received.

Table 24. AT75C1212 Module Initialization Status

Word 0	0x0001	Message length = 0x0001
Word 1	0x8002	Request ID = 0x8002

Bad Format Status

The Oak issues a bad format status message when it has received a request message in which the LENGTH field is not compatible with the request type. The Oak ignores the corresponding malformed request.

Table 25. Bad Format Status

Word 0	0x0002	Message length = 0x0002
Word 1	0x80FF	Status ID = 0x80FF
Word 2	BAD_FORMAT_VALUE	Contains the request ID of the malformed request message.

Unknown Request Status

The Oak issues an unknown request status message when it has received a request message with an unsupported request ID field.

Table 26. Unknown Request Status

Word 0	0x0002	Message length = 0x0002
Word 1	0x80FE	Status ID = 0x80FE
Word 2	UNKNOWN_REQ_VALUE	Contains the request ID of the malformed request message.

Bad Parameter Status

The Oak issues a bad parameter status message when it has received a request message with a parameter having an invalid value.

Table 27. Bad Parameter Status

Word 0	0x0002	Message length = 0x0002
Word 1	0x80FD	Status ID = 0x80FD
Word 2	UNKNOWN_REQ_VALUE	Contains the request ID of the malformed request message.

G.723.1 Encoding Stopped Status Message

The G.723.1 encoding stopped status message is issued if the encode task was stopped by a stop encoding request (request ID 0x0404).

Table 28. G.723.1 Encoding Stopped Status Message

Word 0	0x0001	Message length = 0x0001
Word 1	0x8404	Status ID = 0x8404

G.711 Encoding Stopped Status Message

The G.711 encoding stopped status message is issued if the encode task was stopped by a stop encoding request (request ID 0x0404).

Table 29. G.711 Encoding Stopped Status Message

Word 0	0x0001	Message length = 0x0001
Word 1	0x8414	Status ID = 0x8414

G.723.1 Decoding Stop Status Message

This status message is issued if one G.723.1 decoding task was stopped by a stop decoding request (request ID 0x0402).

Table 30. G.723.1 Decoding Stop Status Message

Word 0	0x0002	Message length = 0x0002
Word 1	0x8402	Status ID = 0x8402
Word 2	TX_N	G.723.1 decode channel which has been stopped. Valid: 0 to 3

G.711 Decoding Stop Status Message

This status message is issued if one G.711 decoding task was stopped by a stop decoding request (request ID 0x0402).

Table 31. G.711 Decompression Stopped Status Message

Word 0	0x0002	Message length = 0x0002
Word 1	0x8412	Status ID = 0x8412
Word 2	TX_N	G.711 decoding channel which has been stopped. Valid: 0 to 3

DTMF Detection Status

The DTMF detection status message is issued each time a valid DTMF digit is detected on the line-in input signal.

Table 32. DTMF Detection Status

Word 0	0x0002	Message length = 0x0002
Word 1	0x8831	Status ID = 0x8831
Word 2	DTMFDET_DIGIT	Detected DTMF digit

The association of the digit codes to each of the sixteen possible DTMF tones is shown in Table 33.

Table 33. Map of DTMF Tones and Associated Digit Code

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	**1** 0x01	**2** 0x02	**3** 0x03	**A** 0x0D
770 Hz	**4** 0x04	**5** 0x05	**6** 0x06	**B** 0x0E
852 Hz	**7** 0x07	**8** 0x08	**9** 0x09	**C** 0x0F
941 Hz	***** 0x0B	**0** 0x0A	**#** 0x0C	**D** 0x00

Tone Generation Status

The tone generation status message is issued when the tone duration has elapsed. It is not issued if the tone was stopped by a tone generation stop request (request ID 0x0802).

Table 34. Tone Generation Status

Word 0	0x0001	Message length = 0x0001
Word 1	0x8802	Status ID = 0x8802

AT75C1212 Device Driver

The AT75C1212 software module is supplied with a device driver for uClinux. This device driver integrates all the AT75C1212 functionalities into the uClinux kernel. The features of the AT75C1212 modules can be accessed through the standard uClinux API. This API is documented in the following section.

AT75C1212 Device Driver Overview

Under uClinux, the device drivers are accessed through file system entries. The AT75C1212 device driver is a character type driver. The associated virtual file can be opened, read from, written to and closed like any regular file. The major role of the device driver is to redefine the file access methods, so that the application can interact with the underlying device, as if it were a file, through the standard file manipulation functions. It provides the application with an abstraction layer which hides the low level interface on top of which it sits.

The AT75C1212 device driver is operated through several file systems:

- /dev/g723encoder0 which is used for G.723.1 encoding operations,
- /dev/g723decoder0 to /dev/g723decoder3 which are used for G.723.1 decoding operations,
- /dev/g711encoder0 which is used for G.711 µ-law/A-law encoding operations,
- /dev/g711decoder0 to /dev/g711decoder3 which are used for G.711 µ-law/A-law decoding operations,
- /dev/tones which is used for DTMF detection and arbitrary tone generation.

G723.1 Encoder Driver Operations

The G.723.1 encoder driver redefines the following file manipulation functions:

- int open(const char *path, int flags, mode_t mode);
- int read(int fd, void *buf, int count);
- int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
- int close(int fd);

Additionally, the ioctl function controls additional features of the AT75C1212 which are not accessible with the other methods. These special commands are described below. The prototype of the ioctl function is:

- int ioctl(int fd, int request, char *argp);

Open Method

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags);
```

Description

The /dev/g723encoder0 virtual file must be opened prior to any encoding operation on the g723.1 encoder device driver. This is done with the open method, the same as for any regular file. The main operation performed by the open method of the device driver is to load and initialize the corresponding DSP software in the DSP subsystem.

When this initialization is successful, the open system call converts the file "path" name ("/dev/g723encoder0" in this case) into a file descriptor. This file descriptor is a non-negative integer which will be used in subsequent I/O operations such as read, ioctl, etc.

"flags" should be O_RDONLY which request opening the file in read-only mode.

"flags" may also be bitwise-or'd with O_NONBLOCK. In this case, neither the open nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait.

Return Values

Open() returns the new file descriptor, or -1 if an error occurred. In the latter case, the global variable errno is set appropriately to reflect the cause of error. Possible values of errno are:

- ENODEV: This indicates that the underlying hardware does not exist or is not supported. One reason can be corruption of the binary DSP software which could not be loaded into the DSP subsystem.
- EBUSY: The underlying hardware is busy. Most probably there is another process using the same resource.
- ENOMEM: A memory allocation requested by the driver failed. This happens when the system memory is full.

Example

```
int fd = open("/dev/g723encoder0", O_RDONLY | O_NONBLOCK);
```

This opens the G.723.1 encoder device driver in read mode. It selects non blocking I/O for read operations. The file descriptor is returned in "fd". If "fd" is positive, the g723.1 encoder device is readily available for read operations.

Close Method

Synopsis

```
#include <unistd.h>
int close(int fd);
```

Description

When the G.723.1 encoder device is no longer needed by the application, it can be closed to release system resources. This is done through the close method. The parameter is the file descriptor of the file to be closed.

Return Values

Close() returns 0 on success, or -1 if an error occurred. In the latter case the global variable is set appropriately to reflect the cause of error. The only possible value for errno is EBADF which means that "fd" is not a valid file descriptor.

Example

```
close(fd);
```

This closes the G.723.1 encoder device.

Read Method

Synopsis

```
#include <unistd.h>
int read(int fd, void *buf, int count);
```

Description

As for any file descriptor, the read() method attempts to read "count" bytes from "fd" into the buffer starting at "buf". When "fd" is a file descriptor attached to /dev/g723encoder0, the bytes read correspond to G.723.1 frames.

Both blocking and non-blocking reads are supported. In blocking mode, read() will return only when there are G.723.1 valid frames available to read. Although the process is blocked, it is safely put on a system wait queue and does not consume CPU time.

In non-blocking mode, the read function returns immediately even if no data is available. In this case the return value is -1 and errno is set to EAGAIN.

Return Values

On success, the number of bytes read is returned. It is not an error if this number is smaller than the number of bytes requested. This may happen, for example, because fewer bytes are actually available at that moment, or because read was interrupted by a signal.

On error, -1 is returned and errno is set appropriately. Possible values for errno follow:

- EAGAIN: non-blocking I/O has been selected using O_NONBLOCK and no data was immediately available.
- EBADF: "fd" is not a valid descriptor.
- EINVAL: the /dev/g723encoder0 file was not open for reading.
- EFAULT: "buf" is outside the accessible address space.

Example

```
ret = read(fd,buf,256);
```

This reads at most 256 bytes from file descriptor "fd" (assumed here to be related to /dev/g723encoder0), and stores them into the memory location pointed to by "buf".

Select Method

Synopsis

```
#include <sys/time.h>
#include <sys/types.h>
```

```
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

Description

Select() waits for a number of file descriptors to change status. The main usage of select() is to check if data (G.723.1 frames) are available for reading without having to actually read the data. In particular, when blocking operation is selected, it advises if a read access will block or not. This is similar to a polling operation.

Three independent sets of descriptors are watched.

1. Those listed in "readfds" will be watched to see if characters become available for reading.
2. Those in "writefds" will be watched to see if a write will not block (not used there).
3. Those in "exceptfds" will be watched for exceptions (not used there).

On exit, the sets are modified in place to indicate which descriptors actually changed status.

Four macros are provided to manipulate the sets.

• FD_ZERO will clear a set.
• FD_SET and FD_CLR add or remove a given descriptor from a set.
• FD_ISSET tests to see if a descriptor is part of the set. This is useful after select returns.

"n" is the highest-numbered descriptor in any of the three sets, plus 1.

"timeout" is an upper limit on the amount of time elapsed before select returns. It may be zero, causing select to return immediately. If timeout is NULL (no timeout), select can block indefinitely.

Return Values

On success, select returns the number of descriptors contained in the descriptor sets, which may be zero if the timeout expires before an event occurs.

On error, -1 is returned, and errno is set appropriately. The sets and timeout become undefined, therefore their contents are not to be relied upon after an error.

Example

```
fd_set rfds;
struct timeval tv;
int retval;
/* initialize file descriptor list */
FD_ZERO(&rfds);
FD_SET(df, &rfds);
/* define delay */
tv.tv_sec = 0;
tv.tv_usec = 50000;
retval = select(df+1, &rfds, NULL, NULL, &tv); /* df supposed to be a file
descriptor related to /dev/g723encoder0 */
if (retval > 0)
  printf("G.723.1 frame received.\n");
else
  printf("G.723.1 frame not received within 50 ms.\n");
```

This code checks if a G.723.1 frame has been received. The time-out is 50 ms.

IOCTL Method

Synopsis

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, char *argp);
```

Description

The ioctl() function manipulates the underlying device parameters of the g723.1 encoder device.

"fd" is the file descriptor upon which ioctl() will act. It should be related to the /dev/g723encoder0 virtual file.

"request" defines which predefined command to send to the G.723.1 encoder device. Some commands may require additional arguments which are stored or received in the buffer pointed to by "argp". The ioctl() requests supported by the G.723.1 device driver are described below:

- REC_CONFIG: This command is used to configure the encoding operation of the multiway driver with G.723.1 payload. This must be done before performing any other operation on the device. There is no additional argument.
- G723_CONFIG: This command is used to configure the characteristics of the G.723.1 encoder algorithm. An additional parameter is used as defined below:

```
struct g723cod_args {
unsigned short enc_rate;
unsigned short vad_cng;
};
```

The fields and the values to be written are those defined in the section "Low Level Interface" on page 5. This should be done before any other operation on the device is performed if default values are not appropriate.

- START_RECORD: This command is used to start the G.723.1 encoding operation. There is no additional argument.
- STOP_RECORD: This command is used to stop the G.723.1 encoding operation. There is no additional argument.
- ECHOCANCEL_CONFIG: This command is used to configure the characteristics of the echo canceller algorithm. An additional parameter is used as defined below:

```
struct echocancel_args {
    unsigned short echo_size;
    unsigned short echo_update;
    unsigned short echo_stepsz;
    unsigned short echo_timeconst;
    unsigned short echo_mucalc;
 };
```

- The fields and the values to be written are those defined in the "Request Notification Messages" section of this document. This should be done before performing any other operation on the device if default values are not appropriate.
- ECHOCANCEL_START: This command is used to start the echo cancelling operation. There is no additional argument.
- ECHOCANCEL_STOP: This command is used to stop the echo cancelling operation. There is no additional argument.
- OAKMEM_ACCESS: This command is used to read/write the memory space of the OAK, either program or data. It should be used with caution (primarily for OAK debug). An additional parameter is used as defined below:

```
struct oakmem_args {
  unsigned short command;
  unsigned short address;
  unsigned short length;
  unsigned short data[29];
};
```

The fields and the values to be written are those defined in the section: "Oak Memory Access" on page 8.

Example

```
struct g723cod_args {
unsigned short enc_rate;
unsigned short vad_cng;
} *g723_conf;
fd723_conf->enc_rate=1;//5.3 kbps rate for encoder
g723_conf->vad_cng=1;//VAD/CNG algorithm active
ioctl(fd, G723_CONFIG, g723_conf);
```

This configures the G.723.1 algorithm. "fd" refers to /dev/g723encoder0 virtual file.

G723.1 Decoder Driver Operations

The G.723.1 decoder driver redefines the following file manipulation functions:

- int open(const char *path, int flags, mode_t mode);
- int write(int fd, void *buf, int count);
- int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval timeout);
- int close(int fd);

Additionally, the ioctl function controls additional features of the AT75C1212 which are not accessible with the other methods. These special commands are described below. The prototype of the ioctl function is:

- int ioctl(int fd, int request, char *argp);

These methods apply to the four devices "/dev/g723decoderN" where N stands for 0,1,2,3. The four devices have the same MAJOR but have differing MINOR.

Open Method

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags);
```

Description

One /dev/g723decoderN virtual file must be opened prior to any decoding operation on the corresponding G.723.1 decoder device driver. This is made with the open() method, the same as for any regular file. The main operation performed by the open() method of the device driver is to load and initialize the corresponding DSP software in the DSP subsystem.

When this initialization is successful, the open system call converts the file "path" name ("/dev/g723decoder0" for example) into a file descriptor. This file descriptor is a non-negative integer which will be used in subsequent I/O operations such as write(), ioctl(), etc.

"flags" should be O_WRONLY which requests opening the file in write-only mode.

“flags” may also be bitwise-or'd with O_NONBLOCK. In this case, neither the open nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait.

Return Values

Open “return the new file descriptor”, or -1 if an error occurred. In the latter case, the global variable errno is set appropriately to reflect the cause of error. Possible values of errno are:

* ENODEV: This indicates that the underlying hardware does not exist or is not supported. One reason can be corruption of the binary DSP software which could not be loaded into the DSP subsystem.
* EBUSY: The underlying hardware is busy. Most probably there is another process using the same resource.
* ENOMEM: A memory allocation requested by the driver failed. This happens when the system memory is full.

Example

```
int fd = open("/dev/g723decoder0", O_WRONLY | O_NONBLOCK);
```

This opens the G.723.1 decoder device driver in write mode, decode channel chosen is 0. It selects non blocking I/O for write operations. The file descriptor is returned in “fd”. If “fd” is positive, the G.723.1 decoder device 0 is readily available for read operations.

Close Method

Synopsis

```
#include <unistd.h>
int close(int fd);
```

Description

When the G.723.1 decoder device is no longer needed by the application, it can be closed to release system resources. This is done through the close() method. The parameter is the file descriptor of the file to be closed.

Return Values

Close() returns 0 on success, or -1 if an error occurred. In the latter case the global variable is set appropriately to reflect the cause of error. The only possible value for errno is EBADF which means that fd is not a valid file descriptor.

Example

```
close(fd);
```

This closes the G.273 decoder device.

Write Method

Synopsis

```
#include <unistd.h>
int write(int fd, void *buf, int count);
```

Description

As for any file descriptor, the write() method attempts to write “count” bytes from the buffer starting at buf to the file descriptor “fd”. When “fd” is a file descriptor attached to /dev/g723decoderN, the bytes written correspond to the G.723.1 frames which are to be emitted by the G.723.1 decoder device.

Both blocking and non-blocking writes are supported. In blocking mode, write() will return only when the G.723.1 decoder device is ready to accept data. Although the process is blocked, it is safely put on a system wait queue and does not consume CPU time.

In non-blocking mode, the write function returns immediately even if no data is available. In this case the return value is -1 and errno is set to EAGAIN. Most often, the application will try again to write until the entire data is transferred.

Return Values

On success, the number of bytes written is returned. This corresponds to the number of G.723.1 bytes actually emitted. It is not an error if this number is smaller than the number of bytes requested. This may happen, for example, because fewer bytes are actually acceptable at that moment due to lack of memory, or because write was interrupted by a signal.

On error, -1 is returned and errno is set appropriately. Possible values for errno follow:

- EAGAIN: non-blocking I/O has been selected using O_NONBLOCK and no data was immediately available.
- EBADF: "fd" is not a valid descriptor.
- EINVAL: /dev/g723decoderN file was not opened for writing.
- EFAULT: "buf" is outside the accessible address space.

Example

```
ret = write(fd,buf,256);
```

This writes at most 256 bytes to file descriptor "fd" (assumed here to be related to one /dev/g723decoderN), from the memory location pointed to by "buf".

Select Method

Synopsis

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

Description

Select() waits for a number of file descriptors to change status. The main usage of select() is to check if data (G.723.1 frames) are available for writing without having to actually write the data. In particular, when blocking operation is selected, it indicates if a write access will block or not. This is similar to a polling operation.

Three independent sets of descriptors are watched.

1. Those listed in "readfds" (not used there) will be watched to see if characters become available for reading.
2. Those in "writefds" will be watched to see if a write will not block.
3. Those in "exceptfds" will be watched for exceptions (not used there).

On exit, the sets are modified in place to indicate which descriptors actually changed status.

Four macros are provided to manipulate the sets.

- FD_ZERO will clear a set.
- FD_SET and FD_CLR add or remove a given descriptor from a set.
- FD_ISSET tests to see if a descriptor is part of the set. This is useful after select returns.

"n" is the highest-numbered descriptor in any of the three sets, plus 1.

"timeout" is an upper limit on the amount of time elapsed before select returns. It may be zero, causing select to return immediately. If timeout is NULL (no timeout), select can block indefinitely.

Return

On success, select() returns the number of descriptors contained in the descriptor sets, which may be zero if the timeout expires before an event occurs.

On error, -1 is returned, and errno is set appropriately, the sets and timeout become undefined, therefore their contents are not to be relied upon after an error.

Example

```
fd_set wfds;
struct timeval tv;
int retval;
/* initialize file descriptor list */
FD_ZERO(&wfds);
FD_SET(df, &wfds);
/* define delay */
tv.tv_sec = 0;
tv.tv_usec = 50000;
retval = select(df+1, NULL, &wfds, NULL, &tv); /* df supposed to be a file
descriptor related to /dev/g723decoderN */
if (retval > 0)
  printf("G.723.1 frame requested by DSP.\n");
else
  printf("No G.723.1 frame requested within 50 ms.\n");
```

This code checks if a G.723.1 frame is requested by the DSP. The time-out is 50 ms.

IOCTL Method

Synopsis

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, char *argp);
```

Description

The ioctl() function manipulates the underlying device parameters of the G.723.1 decoder devices.

"fd" is the file descriptor upon which ioctl() will act. It is related to on of the /dev/g723decoderN virtual file.

"request" defines which predefined command to send to the G.723.1 decoder device. Some commands may require additional arguments which are stored or received in the buffer pointed to by "argp". The ioctl() requests supported by the G.723.1 decoder device driver are described below:

- DEC_CONFIG: This command is used to configure the decoding operation of the multiway driver with G.723.1 payload. This must be done before performing any other operation on the device. There is no additional argument.
- START_DECODE: This command is used to start the G.723.1 decoding operation. There is no additional argument.
- STOP_ DECODE: This command is used to stop the G.723.1 decoding operation. There is no additional argument.
- OAKMEM_ACCESS: This command is used to read/write the memory space of the OAK, either program or data. It should be used with caution, (primarily for OAK debug). An additional parameter is used as defined below:

```
struct oakmem_args {
  unsigned short command;
```

```
        unsigned short address;
        unsigned short length;
        unsigned short data[29];
    };
```

The fields and the values to be written are those defined in the section: "Oak Memory Access" on page 8.

Example

```
        ioctl(fd, DEC_CONFIG, NULL);
```

Assume that "fd" is referring to /dev/g723decoder1. This ioctl configures the decode channel 1 with the G.723.1 payload.

G.711 Encoder Driver Operations

The G.711 encoder driver redefines the following file manipulation functions:

- int open(const char *path, int flags, mode_t mode);
- int read(int fd, void *buf, int count);
- int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
- int close(int fd);

Additionally, the ioctl() function controls additional features of the AT75C1212 which are not accessible with the other methods. These special commands are described below. The prototype of the ioctl() function is:

- int ioctl(int fd, int request, char *argp);

Open Method

Synopsis

```
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>
        int open(const char *path, int flags);
```

Description

The /dev/g711encoder0 virtual file must be opened prior to any encoding operation on the G.711 encoder device driver. This is made with the open() method, the same as for any regular file. The main operation performed by the open() method of the device driver is to load and initialize the corresponding DSP software in the DSP subsystem.

When this initialization is successful, the open system call converts the file "path" name ("/dev/g711encoder0" in this case) into a file descriptor. This file descriptor is a non-negative integer which will be used in subsequent I/O operations such as read(), ioctl(), etc.

"flags" should be O_RDONLY which request opening the file in read-only mode.

"flags" may also be bitwise-or'd with O_NONBLOCK. In this case, neither the open nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait.

Return Values

Open return the new file descriptor, or -1 if an error occurred. In the latter case, the global variable errno is set appropriately to reflect the cause of error. Possible values of errno are:

- ENODEV: This indicates that the underlying hardware does not exist or is not supported. One reason can be a corruption of the binary DSP software which could not be loaded into the DSP subsystem.
- EBUSY: The underlying hardware is busy. Most probably there is another process using the same resource.

27

- ENOMEM: A memory allocation requested by the driver failed. This happens when the system memory is full.

Example

```
int fd = open("/dev/g711encoder0", O_RDONLY | O_NONBLOCK);
```

This opens the G.711 encoder device driver in read mode. It selects non blocking I/O for read operations. The file descriptor is returned in "fd". If "fd" is positive, the g711 encoder device is readily available for read operations.

Close Method

Synopsis

```
#include <unistd.h>
int close(int fd);
```

Description

When the G.711 encoder device is no longer needed by the application, it can be closed to release system resources. This is done through the close method. The parameter is the file descriptor of the file to be closed.

Return Values

Close() returns 0 on success, or -1 if an error occurred. In the latter case the global variable is set appropriately to reflect the cause of error. The only possible value for errno is EBADF which means that "fd" is not a valid file descriptor.

Example

```
close(fd);
```

This closes the G.711 encoder device.

Read Method

Synopsis

```
#include <unistd.h>
int read(int fd, void *buf, int count);
```

Description

As for any file descriptor, the read() method attempts to read "count" bytes from "fd" into the buffer starting at "buf". When "fd" is a file descriptor attached to /dev/g711encoder0, the bytes read correspond to G.711 frames recognized by the G.711 encoder device.

Both blocking and non-blocking reads are supported. In blocking mode, read() will return only when there are G.711 valid frames available to read. Although the process is blocked, it is safely put on a system wait queue and does not consume CPU time.

In non-blocking mode, the read function returns immediately even if no data is available. In this case the return value is -1 and errno is set to EAGAIN.

Return Values

On success, the number of bytes read is returned. It is not an error if this number is smaller than the number of bytes requested. This may happen, for example, because fewer bytes are actually available at that moment, or because read was interrupted by a signal.

On error, -1 is returned and errno is set appropriately. Possible values for errno follow:

- EAGAIN: non-blocking I/O has been selected using O_NONBLOCK and no data was immediately available.
- EBADF: fd is not a valid descriptor.
- EINVAL: /dev/g711encoder0 file was not open for reading.
- EFAULT: "buf" is outside the accessible address space.

Example

```
ret = read(fd,buf,256);
```

This reads at most 256 bytes from file descriptor "fd" (assumed here to be related to /dev/g711encoder0), and stores them into the memory location pointed to by "buf".

Select Method

Synopsis

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

Description

Select() waits for a number of file descriptors to change status. The main usage of select() is to check if data (G.711 frames) are available for reading without having to actually read the data. In particular, when blocking operation is selected, it advises if a read access will block or not. This is similar to a polling operation.

Three independent sets of descriptors are watched:

1. Those listed in "readfds" will be watched to see if characters become available for reading.

2. Those in "writefds" will be watched to see if a write will not block (not used there)

3. Those in "exceptfds" will be watched for exceptions (not used there).

On exit, the sets are modified in place to indicate which descriptors actually changed status.

Four macros are provided to manipulate the sets:

*   FD_ZERO will clear a set.

*   FD_SET and FD_CLR add or remove a given descriptor from a set.

*   FD_ISSET tests to see if a descriptor is part of the set. This is useful after select returns.

"n" is the highest-numbered descriptor in any of the three sets, plus 1.

"timeout" is an upper limit on the amount of time elapsed before select returns. It may be zero, causing select to return immediately. If timeout is NULL (no timeout), select can block indefinitely.

Return Values

On success, select returns the number of descriptors contained in the descriptor sets, which may be zero if the timeout expires before an event occurs. On error, -1 is returned, and errno is set appropriately, the sets and timeout become undefined, therefore their contents are not to be relied upon after an error.

Example

```
fd_set rfds;
struct timeval tv;
int retval;
/* initialize file descriptor list */
FD_ZERO(&rfds);
FD_SET(df, &rfds);
/* define delay */
tv.tv_sec = 0;
tv.tv_usec = 5000;
retval = select(df+1, &rfds, NULL, NULL, &tv); /* df supposed to be a file
descriptor related to /dev/g711encoder0 */
if (retval > 0)
```

29

```
    printf("G.711 frame received.\n");
else
    printf("G.711 frame not received within 5 ms.\n");
```

This code checks if a G.711 frame has been received. The time-out is 5 ms.

IOCTL Method

Synopsis

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, char *argp);
```

Description

The ioctl() function manipulates the underlying device parameters of the G.711 encoder device.

"fd" is the file descriptor upon which ioctl() will act. It should be related to the /dev/g711encoder0 virtual file.

"request" defines which predefined command to send to the G.711 encoder device. Some commands may require additional arguments which are stored or received in the buffer pointed to by argp.

The ioctl() requests supported by the G.711 device driver are described below:

*   REC_CONFIG: This command is used to configure the encoding operation of the multiway driver with the G.711 payload. This must be done before performing any other operation on the device. An additional parameter is used as defined below:

    ```
    int payload;
    ```

This selects PCMU or PCMA.

The valid values to be written are those defined in the section "Low Level Interface" on page 5. This should be done before any other operation is performed on the device if default values are not appropriate.

*   START_RECORD: This command is used to start the G.711 encoding operation. There is no additional argument.
*   STOP_RECORD: This command is used to stop the G.711 encoding operation. There is no additional argument.
*   ECHOCANCEL_START: This command is used to start the echo cancelling operation. There is no additional argument.
*   ECHOCANCEL_STOP: This command is used to stop the echo cancelling operation. There is no additional argument.
*   OAKMEM_ACCESS: This command is used to read/write the memory space of the OAK, either program or data. It should be used with caution, (primarily for OAK debug). An additional parameter is used as defined below:

    ```
    struct oakmem_args {
      unsigned short command;
      unsigned short address;
      unsigned short length;
      unsigned short data[29];
    };
    ```

The fields and the values to be written are those defined in the section: "Oak Memory Access" on page 8.

Example

```
#define PCMA 8
ioctl(g711, G711_CONFIG, PCMA);
```

This configures the encoding channel with PCMA payload.

G711 decoder Driver Operations

The g711 decoder driver redefines the following file manipulation functions:

- int open(const char *path, int flags, mode_t mode);
- int write(int fd, void *buf, int count);
- int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
- int close(int fd);

Additionally, the ioctl function controls additional features of the AT75C1212 which are not accessible with the other methods. These special commands are described below. The prototype of the ioctl function is:

- int ioctl(int fd, int request, char *argp);

These methods apply to the four devices "/dev/g711decoderN" where N stands for 0,1,2,3. The four devices have the same MAJOR but have differing MINOR.

Open Method

Synopsis

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags);
```

Description

One /dev/g711decoderN virtual file must be opened prior to any decoding operation on corresponding G.711 decoder device driver. This is made with the open() method, the same as for any regular file. The main operation performed by the open() method of the device driver is to load and initialize the corresponding DSP software in the DSP subsystem.

When this initialization is successful, the open system call converts the file "path" name ("/dev/g711decoder0" for example) into a file descriptor. This file descriptor is a non-negative integer which will be used in subsequent I/O operations such as write, ioctl, etc.

"flags" should be O_WRONLY which requests opening the file in write-only mode.

"flags" may also be bitwise-OR'd with O_NONBLOCK. In this case, neither the open nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait.

Return values

Open return the new file descriptor, or -1 if an error occurred. In the latter case, the global variable errno is set appropriately to reflect the cause of error. Possible values of errno are:

- ENODEV: This indicates that the underlying hardware does not exist or is not supported. One reason can be a corruption of the binary DSP software which could not be loaded into the DSP subsystem.
- EBUSY: The underlying hardware is busy. Most probably there is another process using the same resource.
- ENOMEM: A memory allocation requested by the driver failed. This happens when the system memory is full.

Example

```
int fd = open("/dev/g711decoder0", O_WRONLY | O_NONBLOCK);
```

This opens the G.711 decoder device driver in write mode, decode channel chosen is 0. It selects non blocking I/O for write operations. The file descriptor is returned in "fd". If "fd" is positive, the G.711 decoder device 0 is readily available for write operations.

Close Method

Synopsis

```
#include <unistd.h>
int close(int fd);
```

Description

When the G.711 decoder device is no longer needed by the application, it can be closed to release system resources. This is done through the close() method. The parameter is the file descriptor of the file to be closed.

Return Values

Close() returns 0 on success, or -1 if an error occurred. In the latter case the global variable is set appropriately to reflect the cause of error. The only possible value for errno is EBADF which means that "fd" is not a valid file descriptor.

Example
```
close(fd);
```

This closes the G.711 decoder device.

Write Method

Synopsis

```
#include <unistd.h>
int write(int fd, void *buf, int count);
```

Description

As for any file descriptor, the write method attempts to write "count" bytes from the buffer starting at "buf" to the file descriptor "fd". When "fd" is a file descriptor attached to /dev/g711decoderN, the bytes written correspond to the G.711 frames which are to be emitted by the G.711 decoder device.

Both blocking and non-blocking writes are supported. In blocking mode, write() will return only when the G.711 decoder device is ready to accept data. Although the process is blocked, it is safely put on a system wait queue and does not consume CPU time.

In non-blocking mode, the write function returns immediately even if no data is available. In this case the return value is -1 and errno is set to EAGAIN. Most often, the application will retry to write as far as the entire data is transferred.

Return Values

On success, the number of bytes written is returned. This corresponds to the number of G.711 bytes actually emitted. It is not an error if this number is smaller than the number of bytes requested. This may happen, for example, because fewer bytes are actually acceptable at that moment due to lack of memory, or because write was interrupted by a signal.

On error, -1 is returned and errno is set appropriately. Possible values for errno follow:
- EAGAIN: Non-blocking I/O has been selected using O_NONBLOCK and no data was immediately available.
- EBADF: "fd" is not a valid descriptor.
- EINVAL: the /dev/g711decoderN file was not opened for writing.
- EFAULT: "buf" is outside the accessible address space.

Example

```
ret = write(fd,buf,256);
```

This writes at most 256 bytes to file descriptor "fd" (assumed here to be related to one /dev/g711decoderN), from the memory location pointed to by "buf".

Select Method

Synopsis
```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

Description
Select waits for a number of file descriptors to change status. The main usage of select is to check if data (G.711 frames) are available for writing without having to actually write the data. In particular, when blocking operation is selected, it indicates if a write access will block or not. This is similar to a polling operation.

Three independent sets of descriptors are watched:

1. Those listed in "readfds" (not used there) will be watched to see if characters become available for reading.
2. Those in "writefds" will be watched to see if a write will not block.
3. Those in "exceptfds" will be watched for exceptions (not used there).

On exit, the sets are modified in place to indicate which descriptors actually changed status.

Four macros are provided to manipulate the sets.
*   FD_ZERO will clear a set.
*   FD_SET and FD_CLR add or remove a given descriptor from a set.
*   FD_ISSET tests to see if a descriptor is part of the set. This is useful after select returns.

"n" is the highest-numbered descriptor in any of the three sets, plus 1.

"timeout" is an upper limit on the amount of time elapsed before select returns. It may be zero, causing select to return immediately. If timeout is NULL (no timeout), select can block indefinitely.

Return Values
On success, select() returns the number of descriptors contained in the descriptor sets, which may be zero if the timeout expires before an event occurs. On error, -1 is returned, and errno is set appropriately, the sets and timeout become undefined, therefore their contents are not to be relied upon after an error.

Example
```
fd_set wfds;
struct timeval tv;
int retval;
/* initialize file descriptor list */
FD_ZERO(&wfds);
FD_SET(df, &wfds);
/* define delay */
tv.tv_sec = 0;
tv.tv_usec = 5000;
retval = select(df+1, NULL, &wfds, NULL, &tv); /* df supposed to be a file
descriptor related to /dev/g711decoderN */
if (retval > 0)
```

33

```
                    printf("G.711 frame requested by DSP.\n");
                 else
                    printf("No G.711 frame requested within 5 ms.\n");
```

This code checks if a G.711 frame is requested by the DSP. The time-out is 5 ms.

IOCTL Method

Synopsis

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, char *argp);
```

Description

The ioctl function manipulates the underlying device parameters of the G.711 decoder devices.

"fd" is the file descriptor upon which ioctl will act. It is related to the /dev/g711decoderN virtual file.

"request" defines which predefined command to send to the G.711 decoder device. Some commands may require additional arguments which are stored or received in the buffer pointed to by "argp". The ioctl requests supported by the G.711 decoder device driver are described below:

- DEC_CONFIG: This command is used to configure the decoding operation of the multiway driver with G.711 payload. An additional parameter is used as defined below:

```
int payload; //
```

This selects PCMU or PCMA

The valid values to be written are those defined in the section "Low Level Interface" on page 5. This must be done before any other operation is performed on the device.

- START_DECODE: This command is used to start the G.711 decoding operation. There is no additional argument.
- STOP_ DECODE: This command is used to stop the G.711 decoding operation. There is no additional argument.
- OAKMEM_ACCESS: This command is used to read/write the memory space of the OAK, either program or data. It should be used with caution, (primarily for OAK debug). An additional parameter is used as defined below:

```
struct oakmem_args {
  unsigned short command;
  unsigned short address;
  unsigned short length;
  unsigned short data[29];
};
```

The fields and the values to be written are those defined in the section: "Oak Memory Access" on page 8.

Example

```
#define PCMU 0
ioctl(fd, DEC_CONFIG, PCMU);
```

Assume that "fd" is referring to /dev/g711decoder2. This configures decoding channel 2 with PCMU payload.

Tones & DTMF Driver Operations

The tones driver redefines the following file manipulation functions:

- int open(const char *path, int flags, mode_t mode);

- int read(int fd, void *buf, int count);
- int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
- int close(int fd);

Additionally, the ioctl() function controls additional features of the AT75C1212 which are not accessible with the other methods. These special commands are described below.

The prototype of the ioctl function is:

- int ioctl(int fd, int request, char *argp);

Open Method

Synopsis
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags);
```

Description
The /dev/tones virtual file must be opened prior to any operation on the tones device driver. This is made with the open() method, the same as for any regular file. The main operation performed by the open() method of the device driver is to load and initialize the corresponding DSP software in the DSP subsystem.

When this initialization is successful, the open system call converts the file "path" name ("/dev/tones" in this case) into a file descriptor. This file descriptor is a non-negative integer which will be used in subsequent I/O as with read, ioctl, etc.

"flags" should be O_RDONLY which request opening the file in read-only mode.

"flags" may also be bitwise-or'd with O_NONBLOCK. In this case, neither the open nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait.

Return values
Open return the new file descriptor, or -1 if an error occurred. In the latter case, the global variable errno is set appropriately to reflect the cause of error. Possible values of errno are:

- ENODEV: This indicates that the underlying hardware does not exist or is not supported. One reason can be a corruption of the binary DSP software which could not be loaded into the DSP subsystem.
- EBUSY: The underlying hardware is busy. Most probably there is another process using the same resource.
- ENOMEM: A memory allocation requested by the driver failed. This happens when the system memory is full.

Example
```
int fd = open("/dev/tones", O_RDONLY | O_NONBLOCK);
```

This opens the tones device driver in read mode. It selects non blocking I/O for read operations. The file descriptor is returned in "fd". If "fd" is positive, the tones device is readily available for read operations.

Close Method

Synopsis
```
#include <unistd.h>
int close(int fd);
```

Description
When the tones device is no longer needed by the application, it can be closed to release system resources. This is done through the close() method. The parameter is the file descriptor of the file to be closed.

Return Values
Close returns 0 on success, or -1 if an error occurred. In the latter case the global variable is set appropriately to reflect the cause of error. The only possible value for errno is EBADF which means that "fd" is not a valid file descriptor.

Example
```
close(fd);
```
This closes the tones device.

Read Method

Synopsis
```
#include <unistd.h>
int read(int fd, void *buf, int count);
```

Description
As for any file descriptor, the read() method attempts to read "count" bytes from "fd" into the buffer starting at "buf". When "fd" is a file descriptor attached to /dev/tones, the bytes read correspond to either DTMF digits, if DTMF detection has been activated, or a tone_generation_done code, if TONE generation has been activated. The mapping between the value of each byte and the DTMF digit is as follows:

Table 35. DTMF Digit and Byte Value Map

Byte Value	DTMF Digit	Byte Value	DTMF Digit
0x01	'1'	0x09	'9'
0x02	'2'	0x0A	'0'
0x03	'3'	0x0B	'*'
0x04	'4'	0x0C	'#'
0x05	'5'	0x0D	'A'
0x06	'6'	0x0E	'B'
0x07	'7'	0x0F	'C'
0x08	'8'	0x00	'D'

Byte value of tone_generation_done = 0xBA

Both blocking and non-blocking reads are supported. In blocking mode, read will return only when there are tones valid frames available to read. Although the process is blocked, it is safely put on a system wait queue and does not consume CPU time.

In non-blocking mode, the read function returns immediately even if no data is available. In this case the return value is -1 and errno is set to EAGAIN.

Return Values
On success, the number of bytes read is returned. It is not an error if this number is smaller than the number of bytes requested. This may happen, for example, because fewer bytes are actually available at that moment, or because read was interrupted by a signal.

On error, -1 is returned and errno is set appropriately. Possible values for errno follow:
- EAGAIN: non-blocking I/O has been selected using O_NONBLOCK and no data was immediately available.
- EBADF: "fd" is not a valid descriptor.

- EINVAL: the /dev/tones file was not open for reading.
- EFAULT: "buf" is outside the accessible address space.

Example

```
ret = read(fd,buf,256);
```

This reads at most 256 bytes from file descriptor "fd" (assumed here to be related to /dev/tones), and stores them into the memory location pointed to by "buf".

Select Method

Synopsis

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

Description

Select waits for a number of file descriptors to change status. The main usage of select() is to check if data (DTMF digits or tone_generation_done code) are available for reading without having to actually read the data. In particular, when blocking operation is selected, it allows to know if a read access will block or not. This is similar to a polling operation.

Three independent sets of descriptors are watched.

1. Those listed in "readfds" will be watched to see if characters become available for reading.
2. Those in "writefds" will be watched to see if a write will not block (not used there),
3. Those in "exceptfds" will be watched for exceptions (not used there).

On exit, the sets are modified in place to indicate which descriptors actually changed status.

Four macros are provided to manipulate the sets.

- FD_ZERO will clear a set.
- FD_SET and FD_CLR add or remove a given descriptor from a set.
- FD_ISSET tests to see if a descriptor is part of the set. This is useful after select returns.

"n" is the highest-numbered descriptor in any of the three sets, plus 1.

"timeout" is an upper limit on the amount of time elapsed before select returns. It may be zero, causing select to return immediately. If timeout is NULL (no timeout), select can block indefinitely.

Return Values

On success, select returns the number of descriptors contained in the descriptor sets, which may be zero if the timeout expires before an event occurs. On error, -1 is returned, and errno is set appropriately, the sets and timeout become undefined, therefore their contents are not to be relied upon after an error.

Example

```
fd_set rfds;
struct timeval tv;
int retval;
/* initialize file descriptor list */
FD_ZERO(&rfds);
FD_SET(df, &rfds);
```

37

```
/* define delay */
tv.tv_sec = 5;
tv.tv_usec = 0;
retval = select(df+1, &rfds, NULL, NULL, &tv); //df supposed to be a file
descriptor related to /dev/tones
if (retval > 0)
  printf("DTMF digit detected.\n");
else
  printf("No DTMF digit detected within 5 s.\n");
```

This code checks if a DTMF digit has been detected. The time-out is 5 ms.

Ioctl Method

Synopsis
```
#include <sys/ioctl.h>
int ioctl(int fd, int request, char *argp);
```

Description
The ioctl() function manipulates the underlying device parameters of the tones device.

"fd" is the file descriptor upon which ioctl() will act. It should be related to the /dev/tones virtual file.

"request" defines which predefined command to send to the tones device. Some commands may require additional arguments which are stored or received in the buffer pointed to by "argp". The ioct() requests supported by the tones device driver are described below:

- DTMFDET_CONFIG: This command is used to configure the characteristics of the DTMF detector. An additional parameter is used as defined below:

```
struct dtmfdet_args{
    unsigned short lowthres;
    unsigned short highthres;
    unsigned short lowrel;
    unsigned short highrel;
    unsigned short postwist;
    unsigned short negtwist;
    unsigned short duration;
    unsigned short silence;
};
```

The fields and the values to be written are those defined in the section "Low Level Interface" on page 5.

- DTMFDET_START: This command is sent to start the DTMF detection immediately. There is no additional argument.
- DTMFDET_STOP: This command is sent to stop the DTMF detection immediately. There is no additional argument.
- TONEGEN_CONFIG: This command is used to configure the characteristics of the arbitrary tone signals. An additional parameter is used as defined below:

```
struct tonegen_args{
    unsigned short cosw1;
    unsigned short sinw1;
    unsigned short lev1;
    unsigned short cosw2;
    unsigned short sinw2;
```

```
                            unsigned short lev2;

                            unsigned short signal_len;

                            unsigned short silence_len;

                            unsigned short start;

                        };
```

The fields and the values to be written are those defined in the section "Request Notification Messages" on page 9.

- TONEGEN_START: This command is sent to start the generation of a tone immediately. There is no additional argument.

- TONEGEN_STOP: This command is sent to stop the generation of a tone immediately. There is no additional argument.

- ECHOCANCEL_CONFIG: This command is used to configure the characteristics of the echo canceller algorithm. An additional parameter is used as defined below:

```
struct echocancel_args {

    unsigned short echo_size;

    unsigned short echo_update;

    unsigned short echo_stepsz;

    unsigned short echo_timeconst;

    unsigned short echo_mucalc;

 };
```

- The fields and the values to be written are those defined in the "Request Notification Messages" section of this document. This should be done before any other kind of operation on the device if default values do not suit.

- ECHOCANCEL_START: This command is used to start the echo cancelling operation. There is no additional argument.

- ECHOCANCEL_STOP: This command is used to stop the echo cancelling operation. There is no additional argument.

- OAKMEM_ACCESS: This command is used to read/write the memory space of the OAK, either program or data. It should be carefully used, mainly for OAK debug purpose. An additional parameter is used as defined below:

```
struct oakmem_args {

  unsigned short command;

  unsigned short address;

  unsigned short length;

  unsigned short data[29];

};
```

The fields and the values to be written are those defined in the section:"Oak Memory Access" on page 8.

Example

```
struct tonegen_args{

    unsigned short cosw1;

    unsigned short sinw1;

    unsigned short lev1;

     unsigned short cosw2;

     unsigned short sinw2;

     unsigned short lev2;

     unsigned short signal_len;

    unsigned short silence_len;

    unsigned short start;
```

39

```
}* tone_args;

// 1st frequency component
tone_args-> cosw1 = 0x5a82; // 1kHz tone
tone_args-> sinw1 = 0x5a83; //
tone_args-> lev1 = 0x4000; // -6dB under full scale reference
// 2nd frequency component not used here
tone_args-> cosw2 = 0;
tone_args-> sinw2 = 0;
tone_args-> lev1 = 0;
tone_args-> signal_len = 500; // milliseconds
tone_args-> silence_len = 500; // milliseconds
tone_args-> start = 2; // wait for tone start request, single tone is
generated

ioctl(fd, TONEGEN_CONFIG, tone_args);
```

This configures the arbitrary tone characteristics for an usual operation.

![ATMEL logo]

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 487-2600

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-18
FAX (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-60-00
FAX (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-000
FAX (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

♻ Printed on recycled paper.