Features

- **Software Module Dedicated to Telephony Signaling**
- **Optimized for the AT75 Series Smart Internet Appliance Processor (SIAP™)**
- **Includes Several Run-time Configurable Independent Algorithms**
  - **Caller ID Type I (On-hook) and Type II (Off-hook)**
  - **DTMF Generator**
  - **DTMF Detector**
  - **Arbitrary Tone Generator**
- **Compliant with ITU-T V.23 and Bell 202 Standards**
- **Available With a uClinux® Device Driver**

Description

The AT75C1010 is a software module designed to run on the OakDSPCore® subsystem of the AT75 Series Smart Internet Appliance Processor. It implements commonly used telephony algorithms:

- A DTMF generator to dial phone numbers.
- A DTMF detector to decode incoming DTMF signaling.
- An arbitrary tone generator that can be used to generate any frequency during a programmable duration.
- A Caller ID decoder that supports Type I and Type II, and two standard modulation schemes (Bellcore and V.23).

All these algorithms have a number of parameters that 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 AT75C to cope with many non-standard situations often encountered on private telephone networks.

The AT75C1010 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 AT75C1010 interface into most operating systems.

For developers using uClinux, a specific device driver is supplied. It allows the uClinux capabilities to be extended to the complete functionality of the AT75C1010 module in a seamless manner.

This datasheet is made up of three sections:

- A functional description of the supported algorithms.
- A description of the low-level software interface.
- A description of the uClinux device driver.

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

Smart Internet Appliance Processor (SIAP™)

AT75C1010 – Telephony Software Module

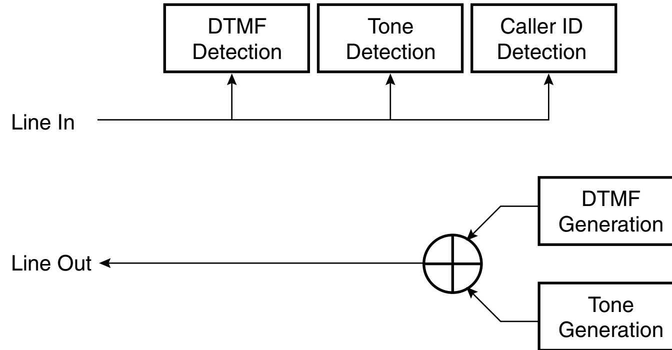
Rev. 1786A–11/01

Functional Description

A functional block diagram of the AT75C1010 module is given in Figure 1.

The various algorithms are independent. They can be enabled, disabled or programmed individually.

Figure 1. AT75C1010 Block Diagram



DTMF Generator

The DTMF is based on two individually programmable harmonic oscillators. Each of the oscillators is assigned a DTMF band. The low-band oscillator can produce 697 Hz, 770 Hz, 852 Hz or 941 Hz. The high-band oscillator can produce 1209 Hz, 1336 Hz, 1477 Hz, and 1633 Hz.

A DTMF signal is the sum of one frequency in the low group and one frequency in the high group, thus leading to a total of sixteen different signals. Figure 2 shows a block diagram of the DTMF generator. The power level for each tone, the tone duration and the silence between tones are individually programmable to comply with the ITU-T standard. These parameters are depicted in Figure 3.

The DTMF generator can be used for dialing, calling line identification, ASCII data transmission or remote control operations over the telephone network.

Figure 2. DTMF Generation Block Diagram

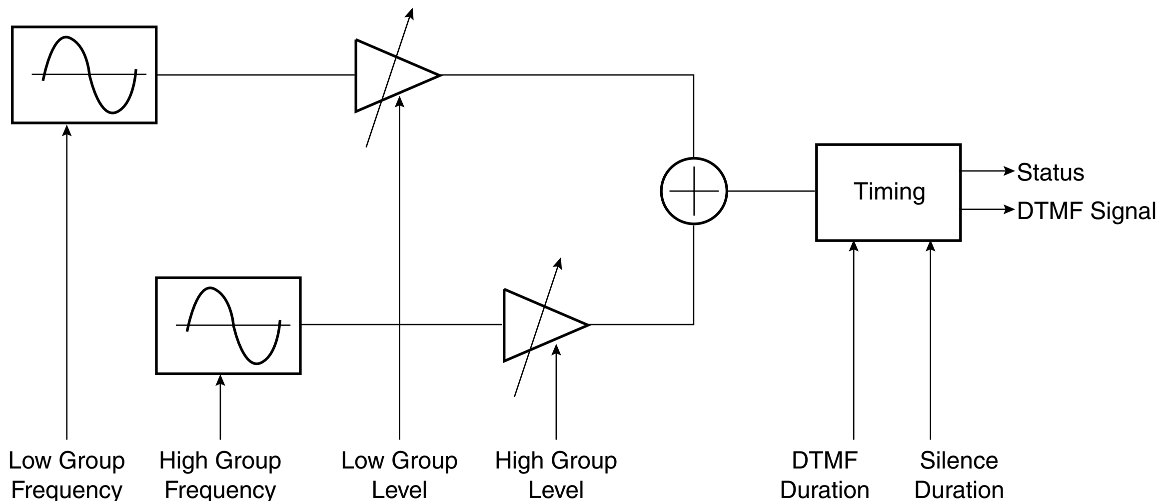
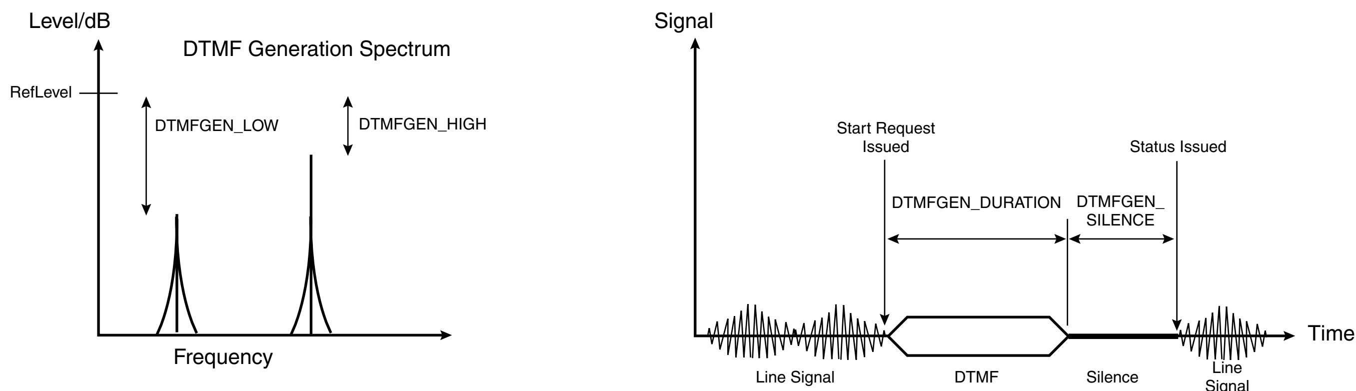


Figure 3. DTMF Signal Characteristics

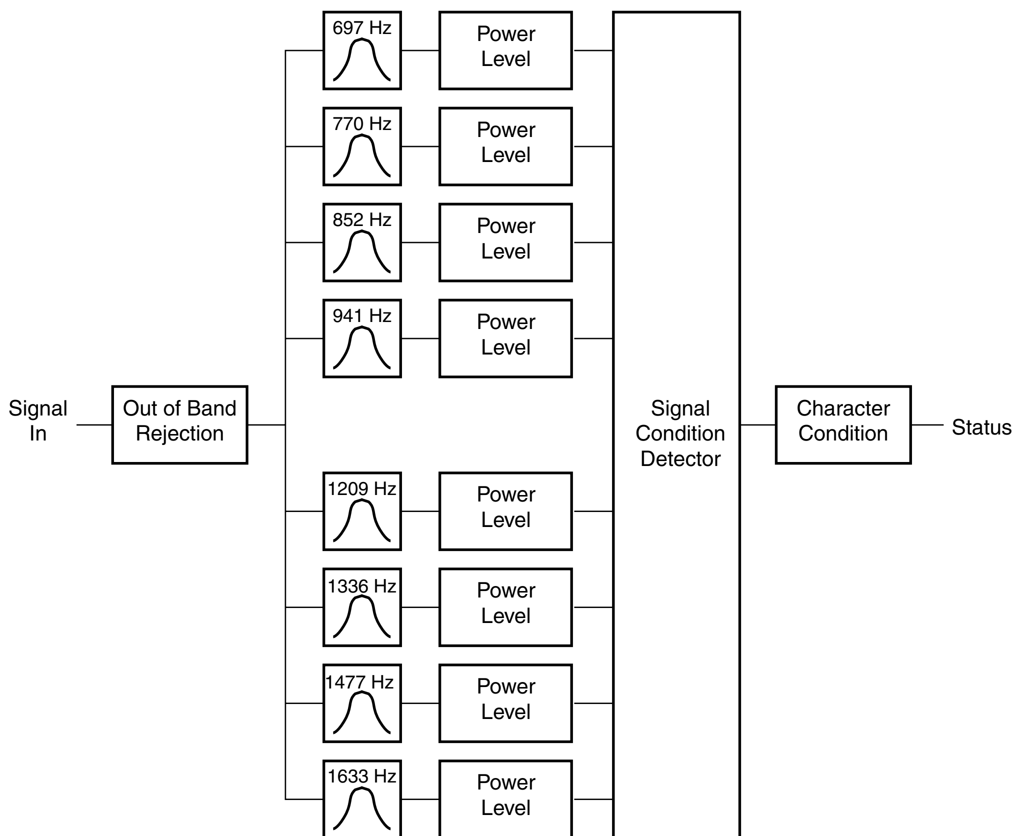


DTMF Detector

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 4 illustrates a block diagram of the DTMF detector.

Figure 4. DTMF Detection Block Diagram



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 with programmable frequency, amplitude and duration.

Caller ID Decoder

This task, when activated, supervises the telephone line-in input signal and looks for a Caller ID message.

The functions included in this task are:

- Detection of the “Dual Tone Alerting Signal”
- Detection of the “Channel Seizure Signal”
- Detection of the loss of carrier event
- Detection of the message bytes and framing verification
- Timing supervision

Two modulation standards are used for the Caller ID function: ITU-T V.23 and Bell 202.

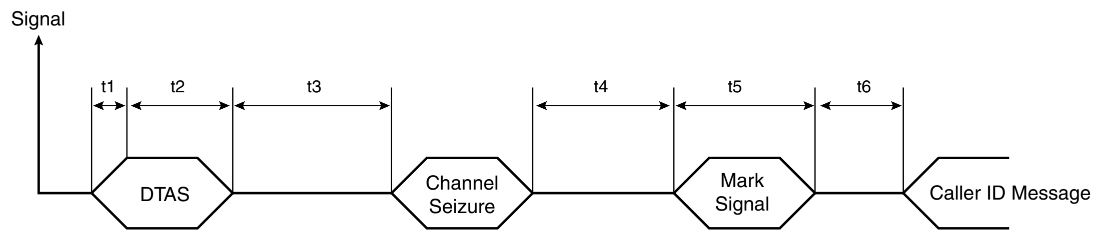
The Caller ID can be programmed in the following three modes:

- Optimized for V.23 demodulation
- Optimized for Bell 202 demodulation
- Compromise demodulation

In the first case, the system is fully optimized for the demodulation of the V.23 scheme, and it can be used with a slight degradation to detect Bell 202 modulation. Conversely, the same degradation is suffered when a Bell 202 demodulator is used to detect a V.23 modulation. In the third mode, a sub-optimal demodulator for both modulations is used.

Figure 5, Figure 6 and Figure 7 depict the different tunable parameters of the Caller ID algorithm.

Figure 5. Caller ID Algorithm – 1 [1]



Note: 1. Where:
t1 = min time for recognize DTAS
t2 = max time allowed to DTAS
t3 = timeout after loss of DTAS, waiting Channel Seizure
t4 = timeout after Channel Seizure, waiting Mark signal
t5 = validation time of Mark signal
t6 = timeout for the reception of the first character (Caller ID message type)

Figure 6. Caller ID Algorithm – 2

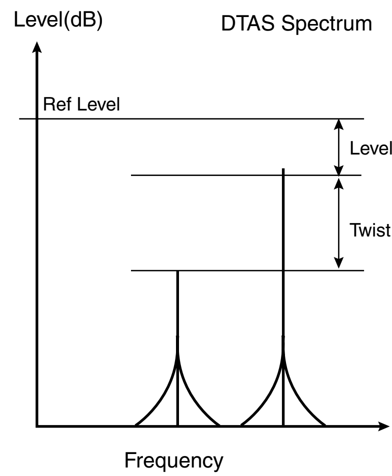
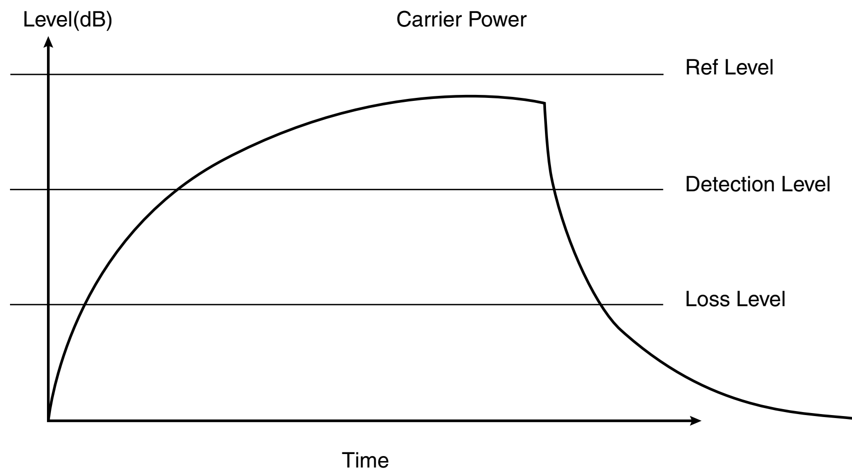


Figure 7. Caller ID Algorithm – 3



Low-level Interface

This section describes how the AT75C1010 software is uploaded into the DSP subsystem program memory. It also describes how the application software running on the ARM® and the AT75C1010 running on the DSP subsystem exchange information through the mailboxes.

This section assumes an in-depth knowledge of the ARM/DSP subsystem interface mailbox system (DPMB).

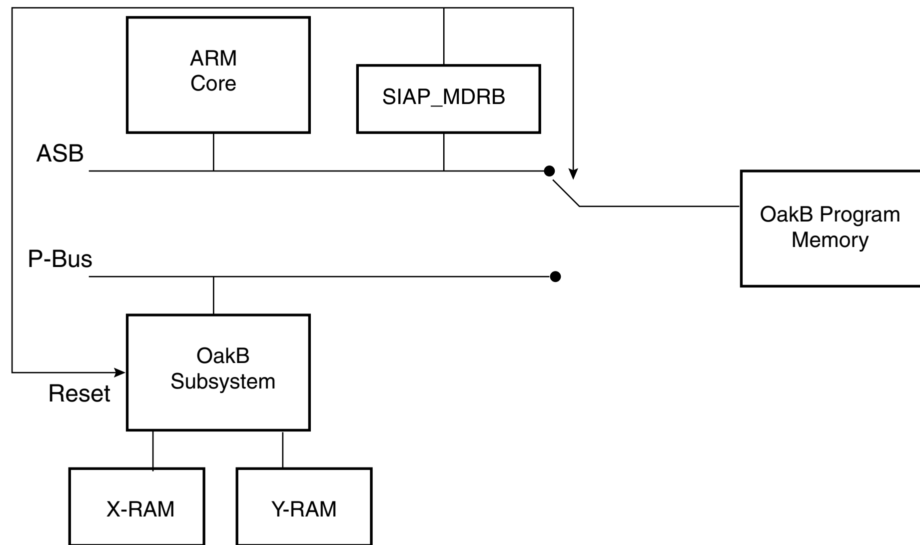
Telephony Program Upload

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 8.

Figure 8. Telephony Program Upload



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.

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 AT75C1010 module is stored in the ARM memory space, possibly in nonvolatile memory. The module contains the AT75C1010 flat binary image.

DPMB Configuration

The DPMB is programmed in configuration 2 (see AT75C DSP subsystem datasheet, literature number 1368), which leads to the configuration specified in Table 1.

Table 1. DPMB Configuration

Mailbox #	Offset from Base[(1)]	Length	Direction	Semaphore Address[(1)]	Usage
0	0x000	0x80	ARM -> Oak	0x200	Unused
1	0x080	0x80	ARM <- Oak	0x204	Unused
2	0x100	0x40	ARM -> Oak	0x208	DSP memory access
3	0x140	0x40	ARM -> Oak	0x20C	Unused
4	0x180	0x20	ARM -> Oak	0x210	Unused
5	0x1A0	0x20	ARM <- Oak	0x214	Unused
6	0x1C0	0x20	ARM -> Oak	0x218	Request notification
7	0x1E0	0x20	ARM <- Oak	0x21C	Status notification

Note: 1. Base address is 0xfa000000 for OakA, 0xfb000000 for OakB.

All the mailboxes allow read/write access from both sides. Arbitration is done by using the semaphores.

Mailbox Access

ARM to Oak Mailboxes

Before accessing the ARM->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 14.

Mailbox 2: 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 DSP software initialization to notify the ARM application that the software is ready to execute tasks.

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 2 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 subsequent): 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 OakB:

1. Wait for *(0xfb000208) = 0, i.e., the semaphore is cleared
2. *(0xfb000100) = 0x0004 // data write command
3. *(0xfb000102) = 0xabcd // this is the address
4. *(0xfb000104) = 0x0001 // only one word to write
5. *(0cfb000106) = 0x1234 // this is the value
6. *(0xfb000208) = 1 // notify the OakB

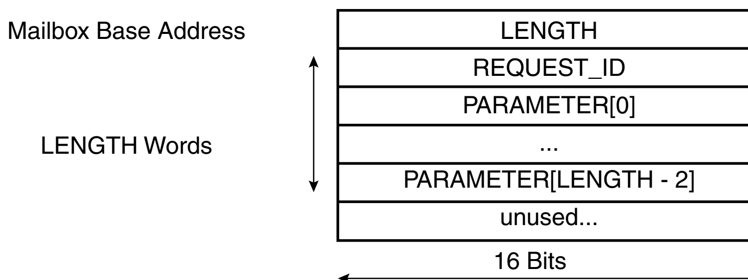
Example of use: Read data locations 0xabcd and 0xabce from OakB:

1. Wait for *(0xfb000208) = 0, i.e. the semaphore is cleared
2. *(0xfb000100) = 0x0003 // data read command
3. *(0xfb000102) = 0xabcd // this is the first address to read
4. *(0xfb000104) = 0x0002 // two words to read
5. *(0xfb000208) = 1 // notify the OakB
6. Wait for the semaphore to be back to 0
7. Read 0xfb000106 and 0xfb000108 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 9 describes this format.

Figure 9. 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 gracefully.

Tone Generation Configuration Request

Table 2. Tone Generation Configuration Request

Word 0	0x0007	Message length = 0x0007
Word 1	0x0800	Request ID = 0x0800
Word 2	32768 * cos (pi* TONE_FREQ /4000)	Words 2 and 3 define the frequency of the generated tone
Word 3	32768 * sin (pi* TONE_FREQ /4000)	
Word 4	TONE_LEVEL = 32768 * 10E(dB/20)	Level of the generated tone
Word 5	TONE_DURATION	Duration of the generated tone in milliseconds 0x0000 means unlimited duration
Word 6	SILENCE_DURATION	Duration of the silence following the tone in milliseconds 0x0000 means unlimited duration
Word 7	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 causes the generation to start immediately Bit 1: 0: the tone is added to all other signals emitted on the speaker; 1: all other signals are blocked while the tone is emitted.

Example: 0x0007 0x0801 0x5A82 0x5A83 0x4000 0x0080 0x0080 0x0003

This message configures the generator to emit a 1024 Hz tone 6 dB below the reference level. 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 messageis emitted.

Tone Generation Start Request

Table 3. Tone Generation Start Request

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

The tone starts as soon as the DSP unit receives this 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.

Tone Generation Stop Request

Table 4. Tone Generation Stop Request

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

The tone stops as soon as the DSP unit receives this 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).

DTMF Generation Configuration Request

Table 5. DTMF Generation Configuration Request

Word 0	0x0005	Message length = 0x0005
Word 1	0x0820	Request ID = 0x0820
Word 2	DTMF_GEN_LOW= 32768 * 10E(dB/20)	Low Group Level
Word 3	DTMF_GEN_HIGH= 32768 * 10E(dB/20)	High Group Level
Word 4	DTMF_GEN_DURATION	Duration of signal in milliseconds. 0x00 means unlimited duration.
Word 5	DTMF_GEN_SILENCE	Duration of silence in milliseconds. 0x00 means unlimited duration.

Example: 0x0005 0x0820 0x2414 0x2D6B 0x0064 0x0064

This message configures the DTMF generator with a Low Group level 11 dB below the reference level and with a High Group level 9 dB below the reference level. The signal and the silence last for 100 ms.

DTMF Generation Start Request

Table 6. DTMF Generation Start Request

Word 0	0x0002	Message length = 0x0002
Word 1	0x0821	Request ID = 0x0821
Word 2	DTMF_DIGIT_CODE	DTMF digit code wanted

Digit Codes of DTMF Tones

Table 7. Digit Codes of DTMF Tones

	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

Table 7 shows the attribution of the digit codes to each of the sixteen possible DTMF tones.

Example: 0x0002 0x0821 0x000A generates DTMF “0”.

When emitting several DTMF tones, it is important to wait for the previous DTMF generation status before the next one is started (Status ID = 0x8821). Otherwise the timing specified in the DTMF generation configuration request is not respected.

DTMF Generation Stop Request

Table 8. DTMF Generation Stop Request

Word 0	0x0001	Message length = 0x0001
Word 1	0x0822	Request ID = 0x0822

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

DTMF Detection Configuration Request

Table 9. DTMF Detection Configuration Request

Word 0	0x0009	Message length = 0x0009
Word 1	0x0830	Request ID = 0x0830
Word 2	DTMF_DET_LOWTHRES = 65535 * 10E(dB/10)	Low Group Power Detection Threshold
Word 3	DTMF_DET_HIGHTHRES = 65535* 10E(dB/10)	High Group Power Detection
Word 4	DTMF_DET_LOWREL = 65535* 10E(dB/20)	Minimum difference level between the strongest frequency in the low group and the other of the same group
Word 5	DTMF_DET_HIGHREL = 65535* 10E(dB/20)	Minimum difference level between the strongest frequency in the high group and the other of the same group
Word 6	DTMF_DET_POSTWIST = 65535* 10E(dB/10)	Maximum Low-to-high twist
Word 7	DTMF_DET_ NEGTWIST = 65535* 10E(dB/10)	Maximum High-to-low twist
Word 8	DTMF_DET_DURATION	Duration of signal condition for character recognition in milliseconds.
Word 9	DTMF_DET_SILENCE	Duration of silence condition for character recognition in milliseconds.

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 other of the same group must be at least of 18 dB. The maximum difference level between the two groups must be at most of 12 dB. The signal must at least last for 30 ms and so on for silence, in order to recognize a character.

DTMF Detection Start Request

Table 10. DTMF Detection Start Request

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

The DTMF detection is started as soon as the DSP unit receives this request.

DTMF Detection Stop Request

Table 11. DTMF Detection Stop Request

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

The DTMF detection is stopped as soon as the DSP unit receives this request.

Caller ID Configuration Request

Table 12. Caller ID Configuration Request

Word 0	0x000F	Message length = 0x000F
Word 1	0x0810	Request ID = 0x0810
Word 2	CALLER_ID_STD	Demodulation standard (0: V.23 ITU-T; 1: Bell 202; 2: compromise)
Word 3	CALLER_ID_DTAS_OK	Minimum time for recognize DTAS [hundredths of second]
Word 4	CALLER_ID_DTAS_LENGTH	Maximum time allowed to DTAS [hundredths of second]
Word 5	CALLER_ID_WCSZ	Timeout after loss of DTAS waiting channel seizure [hundredths of second]
Word 6	CALLER_ID_WMS	Timeout after channel seizure waiting the Mark signal [hundredths of second]
Word 7	CALLER_ID_VALID_MS	Validation time of the mark signal [hundredths of second]
Word 8	CALLER_ID_TIMEOUT_CHAR	Timeout for reception of first character (message type) [hundredths of second]
Words 9:10	CALLER_ID_DTAS_THRES = 536870912 * 10E(dB/10)	Threshold level of each tone of DTAS (power)
Words 11	CALLER_ID_TWIST = 65535 * 10E(dB/10)	Twist between the two tones (power)
Words 12:13	CALLER_ID_CARRIER_THRES = 822083584 * 10E(dB/10)	Threshold for carrier detection (power)
Words 14:15	CALLER_ID_ CARRIER_LOSS = 822083584 * 10E(dB/10)	Threshold for carrier loss (power)

Caller ID Start Request

Table 13. Caller ID Start Request

Word 0	0x0002	Message length = 0x0002
Word 1	0x0811	Request ID = 0x0811
Word 2	CALLER_ID_START	Bit 0: - Bit 1: 1 for enable DTAS detection; 0 for disable

The Caller ID detection is started as soon as this request is received by the DSP unit. Setting the proper value in Word 2 (i.e. 0x0002) enables DTAS detection.

Caller ID Stop Request

Table 14. Caller ID Stop Request

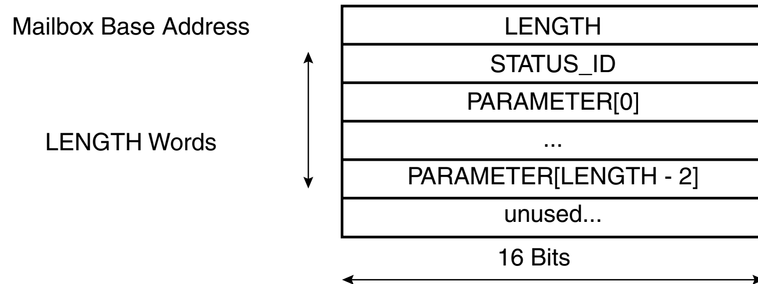
Word 0	0x0001	Message length = 0x0001
Word 1	0x0812	Request ID = 0x0812

The Caller ID detection is stopped as soon as this request is received by the DSP unit.

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. Those messages are always formatted in the same way. Figure 10 describes this format.

Figure 10. 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 gracefully.

Telephony Module Initialization Status

This message is issued when the telephony 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 15. Telephony Module Initialization Status

Word 0	LENGTH	Message length = 0x0001
Word 1	SW_INIT_DONE_ID	Status ID = 0x8002

Bad Format Status

The Oak issues this 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 16. Bad Format Status

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

Unknown Request Status

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

Table 17. Unknown Request Status

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

Bad Parameter Status

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

Table 18. Bad Parameter Status

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

Tone Generation Status

This 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 19. Tone Generation Status

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

DTMF Generation Status

This message is issued (DTMF_GEN_DURATION + DTMF_GEN_SILENCE) milliseconds after a DTMF generation start request has been emitted. It is not issued if the DTMF generation is stopped by a stop request.

Table 20. DTMF Generation Status

Word 0	0x0001	Message length = 0x0001
Word 1	0x8822	Status ID = 0x8822

DTMF Detection Status

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

Table 21. DTMF Detection Status

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

Table 7 on page 11 shows the attribution of the digit codes to each of the sixteen possible DTMF tones.

Caller ID Exit Status

This message is issued when the task ends due to an abnormal situation or at the end of reception. Note that when the ARM receives such a status, a Caller ID Stop Request should be sent to the Oak to free the DSP from Caller ID task.

Table 22. Caller ID Exit Status

Word 0	0x0002	Message length = 0x0002
Word 1	0x8813	Status ID = 0x8813
Word 2	CALLER_ID_EXIT	Exit code: 1 = loss of carrier 2 = time-out: no channel seize 3 = time-out: no message type 4 = framing error

DTAS Detected Status

This message is issued when the DTAS is detected if the feature is enabled (see Caller ID Start Request, ID 0x0811).

Table 23. DTAS Detected Status

Word 0	0x0001	Message length = 0x0001
Word 1	0x8811	Status ID = 0x8811

Received Character Status

This message is issued each time a character is received.

Table 24. Received Character Status

Word 0	0x0002	Message length = 0x0002
Word 1	0x8812	Status ID = 0x8812
Word 2	CALLER_ID_CHAR	Received character

AT75C1010 Device Driver

The AT75C1010 software module is supplied with the device driver for uClinux. This device driver enables the application developer to integrate all the AT75C1010 functionality into the uClinux kernel. All the features of the AT75C1010 modules can be accessed through the standard uClinux API. This section documents this API.

Under uClinux, the device drivers are accessed through filesystem entries. The AT75C1010 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 AT75C1010 device driver is operated through two filesystem entries, /dev/cid and /dev/dtmf. The former is used for Caller ID operations, the latter id used for DTMF operations. The tone generator can be used through both filesystem entries.

DTMF Driver Operations

The DTMF part of the 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 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 allows control of additional features of the AT75C1010 which are not accessible with the read or write methods. Those 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/dtmf virtual file must be opened prior to any operation on the DTMF device driver. This is done with the open method, just like 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/dtmf" in this case) into a file descriptor. This file descriptor is a non-negative integer that is used in subsequent I/O operations such as with read, write, etc.

Flags is one of O_RDONLY, O_WRONLY or O_RDWR which request opening the file read-only, write-only or read/write, respectively.

Flags may also be bitwise-or'd with O_NONBLOCK. In this case, neither the open nor any subsequent operation on the file descriptor which is returned causes the calling process to wait.

Return Values

Open returns the new file descriptor, or -1 if an error occured. 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/dtmf", O_RDWR | O_NONBLOCK);
```

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

Close Method

Synopsis

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

Description

When the DTMF device is not needed any longer 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 occured. In the latter case the global variable errno 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 DTMF device previously opened.

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/dtmf, the bytes read correspond to the DTMF digits recognized by the DTMF decoding device. Table 7 on page 11 gives the mapping between the value of each byte and the DTMF digit.

Both blocking and non-blocking reads are supported. In blocking mode, read returns only when there is a DTMF digit 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 the time, or because read was interrupted by a signal.

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

- 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/dtmf file was not open for reading.
- EFAULT: buf is ouside the accessible address space.

Example

```
readcount = read(fd, buf, 12);
```

This reads at most 12 bytes from file descriptor fd (assumed here to be related to /dev/dtmf), and stores them in the memory location pointed to by buf.

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/dtmf, the bytes written correspond to the DTMF digits which are to be emitted by the DTMF generation device. Table 7 on page 11 gives the mapping between the value of each byte and the DTMF digit.

Both blocking and non-blocking writes are supported. In blocking mode, write returns only when the DTMF 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. In most cases, the application retries to write until the entire data set is transferred.

Return Values

On success, the number of bytes written is returned. This corresponds to the number of DTMF digits 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 the time 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 are as follows:

- 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/dtmf file was not open for reading.
- EFAULT: buf is ouside the accessible address space.

Example

```
char Emergency[] = "\x09\x01\x01";
ret = write(fd,emergency, 3);
```

This dials the 911 emergency phone number.

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 use of select is to check if data (DTMF digits) is available for read (DTMF detected) or write (DTMF to be emitted) without having to actually read or write the data. In particular, when the blocking operation is selected, it allows the user to know if a read or write access will block or not. This is similar to a polling operation.

Three independent sets of descriptors are monitored. Those listed in readfds are watched to see if characters become available for reading, those in writefds are watched to see if a write will not block, and those in exceptfds are watched for exceptions. 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 clears 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 bound 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

If successful, select returns the number of descriptors contained in the descriptor sets, which may be zero if the timeout expires before anything of interest happens. On error, -1 is returned, and errno is set appropriately. The sets and timeout become undefined, so their contents should not be relied on 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 = 5;
tv.tv_usec = 0;
retval = select(df+1, NULL, &wfds, NULL, &tv); //df supposed to be a file
descriptor related to /dev/dtmf
if (retval > 0)
  printf("Ready to send DTMF.\n");
else
  printf("Couldn't send DTMF during 5 seconds.\n");
```

This code checks if a DTMF can be emitted. The timeout is 5 seconds.

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 DTMF generator and detector. Additionally, the arbitrary tone generator is controlled trough an ioctl request.

fd is the file descriptor upon which ioctl acts. It is related to the /dev/dtmf virtual file.

Request defines which predefined command to send to the DTMF 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 DTMF device driver are described below:

- DTMF_DET_START: This command is sent to start a DTMF detection. There are no additional arguments.
- DTMF_DET_STOP: This command is sent to stop a DTMF detection. There are no additional arguments.
- DTMF_GEN_CONFIG: This command is used to configure the characteristics of the DTMF signals. An additional parameter is used as defined below:

```
struct t_dtmf_gen_args {
    unsigned short dtmf_gen_low;
    unsigned short dtmf_gen_high;
    unsigned short dtmf_gen_duration;
    unsigned short dtmf_gen_silence;
};
```

The fields and the values to be written are those defined in the section on “Low-level Interface” on page 6.

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

```
struct t_dtmf_det_args{
    unsigned short dtmf_det_lowthres;
    unsigned short dtmf_det_highthres;
    unsigned short dtmf_det_lowrel;
    unsigned short dtmf_det_highrel;
    unsigned short dtmf_det_postwist;
    unsigned short dtmf_det_negtwist;
    unsigned short dtmf_det_duration;
    unsigned short dtmf_det_silence;
};
```

The fields and the values to be written are those defined in the section on “Low-level Interface” on page 6.

TONE_GEN_CONFIG: This command is used to configure the characteristics of the arbitrary tone signals. An additional parameter is used as defined below:

```
struct t_tone_gen_args{
    unsigned short tone_cosw;
    unsigned short tone_sinw;
    unsigned short tone_level;
    unsigned short tone_duration;
    unsigned short silence_len;
    unsigned short tone_start;
};
```

The fields and the values to be written are those defined in the section on “Low-level Interface” on page 6.

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

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

Example

```
struct  t_dtmf_gen_args {
    unsigned short dtmf_gen_low;
    unsigned short dtmf_gen_high;
    unsigned short dtmf_gen_duration;
    unsigned short dtmf_gen_silence;
} * dtmf_gen_args;
dtmf_gen_args->dtmf_gen_low = 0x2414; // -11 dB below reference level
dtmf_gen_args->dtmf_gen_high = 0x2d6b; // -9 dB below reference level
dtmf_gen_args->dtmf_gen_duration = 100; // milliseconds
dtmf_gen_args->dtmf_gen_silence = 100; // milliseconds
ioctl(fd, DTMF_GEN_CONFIG, dtmefgen_args);
```

This configures the signal characteristics of the DTMF generator for a standard operation.

Caller ID Driver Operations

The Caller ID part of the driver redefines the following file manipulation functions:

- int open(const char *path, int flags, mode_t mode)
- int read(int fd, void *buf, size_t 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 allows control of additional features of the AT75C1010 which are not accessible with the read or write 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/cid virtual file must be opened prior to any operation on the Caller ID device driver. This is made with the open method, just like 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/cid” in this case) into a file descriptor. This file descriptor is a non-negative integer which is used in subsequent I/Os as with read, write, etc.

In particular case of Caller ID device opening flags is O_RDONLY which request opening the file in read-only mode, because there is no need to send Caller ID characters.

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

Open return the new file descriptor, or -1 if an error occured. 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/cid", O_RDONLY | O_NONBLOCK);
```

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

Close Method

Synopsis

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

Description

When the Caller ID device is not needed any more 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 errno 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 Caller ID device previously opened.

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/cid, the bytes read correspond to the Caller ID character received by the Caller ID device.

Both blocking and non-blocking reads are supported. In blocking mode, read returns only when there is a Caller ID character 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 present, 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/cid file was not open for reading.
- EFAULT: buf is ouside the accessible address space.
- ETIME: Timeout during Caller ID detection.

Example

```
readcount = read(fd, buf, 12);
```

This reads at most 12 bytes from file descriptor fd (assumed here to be related to /dev/cid), and stores them in 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 Caller ID characters are available for read without having to actually read the data. In particular, when the blocking operation is selected, it allows the user to know if a read access will block or not. This is similar to a polling operation.

Two independent sets of descriptors are monitored. Those listed in readfds are watched to see if characters become available for reading, and those in exceptfds are watched for exceptions. 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 clears 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 two sets, plus 1.

timeout is an upper bound 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 anything of interest happens. On error, -1 is returned, and errno is set appropriately. The sets and timeout become undefined, so their contents should not be relied on 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 = 5;
tv.tv_usec = 0;
retval = select(df+1, &rfds, NULL, NULL, &tv); //df supposed to be a file
descriptor related to /dev/cid

if (retval > 0)
  printf("Caller ID data are available now.\n");
else
  printf("No Caller ID data within 5 seconds.\n");
```

This checks during 5 second if Caller ID data is available or not.

Ioctl Method

Synopsis

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

The ioctl function manipulates the underlying device parameters of the Caller ID detector. Additionally, the arbitrary tone generator is controlled through an ioctl request.

fd is the file descriptor upon which ioctl acts. It is related to the /dev/cid virtual file.

request defines which predefined command to send to the Caller ID 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 Caller ID device driver are described below:

- CALLER_ID_START: This command starts the Caller ID detection. An additional parameter is used to select DTAS detection. Its type is unsigned short. This parameter is defined in the Low-level Interface section of this document, See "Low-level Interface" on page 6.
- CALLER_ID_STOP: This command is used to stop Caller ID detection. There is no additional argument.
- CALLER_ID_CONFIG: This command is used to configure the characteristics of the Caller ID detector. An additional parameter is used as defined below:

```
struct t_cid_args{
    unsigned short caller_id_std;
    unsigned short caller_id_dtas_ok;
    unsigned short caller_id_dtas_length;
    unsigned short caller_id_wcsz;
    unsigned short caller_id_wms;
    unsigned short caller_id_valid_ms;
    unsigned short caller_id_timeout_char;
    unsigned int caller_id_dtas_thres;
    unsigned short caller_id_twist;
    unsigned int caller_id_carrier_thres;
    unsigned int caller_id_carrier_loss;
};
```

The fields and the values to be written are those defined in the section on "Low-level Interface" on page 6.

TONE_GEN_CONFIG: This command is used to configure the characteristics of the arbitrary tone signals. An additional parameter is used as defined below:

```
struct t_tone_gen_args{
    unsigned short tone_cosw;
    unsigned short tone_sinw;
    unsigned short tone_level;
    unsigned short tone_duration;
    unsigned short silence_len;
    unsigned short tone_start;
};
```

The fields and the values to be written are those defined in the section on "Low-level Interface" on page 6.

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

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

CALLER_ID_ACK_DIGIT: This command is sent to set the DTMF digit used to acknowledge DTAS (type II). The argument is the decimal value of the DTMF digit cast to unsigned short.

Example

```
struct t_cid_args{
    unsigned short CALLER_ID_std;
    unsigned short caller_id_dtas_ok;
    unsigned short caller_id_dtas_length;
    unsigned short caller_id_wcsz;
    unsigned short caller_id_wms;
    unsigned short caller_id_valid_ms;
    unsigned short caller_id_timeout_char;
    unsigned int caller_id_dtas_thres;
    unsigned short caller_id_twist;
    unsigned int caller_id_carrier_thres;
    unsigned int caller_id_carrier_loss;
} * cid_det_args;
cid_det_args -> caller_id_std = 0; // V.23 ITU-T demodulation standard
cid_det_args -> caller_id_dtas_ok = 6; // hundredths of second
cid_det_args -> caller_id_dtas_length = 50; // hundredths of second
cid_det_args -> caller_id_wcsz = 200; // hundredths of second
cid_det_args -> caller_id_wms = 50; // hundredths of second
cid_det_args -> caller_id_ valid_ms = 3; // hundredths of second
cid_det_args -> caller_id_ timeout_char = 50; // hundredths of second
cid_det_args -> caller_id_ dtas_thres = 536870; // threshold of -30dB under
reference
cid_det_args -> caller_id_ twist = 16461; // twist between the two tones of -6dB
under reference
cid_det_args -> caller_id_ carrier_thres = 822083; // threshold for carrier
detection of -30dB under reference
cid_det_args -> caller_id_ carrier_loss = 412017; // threshold for carrier loss of
-30dB under reference
ioctl(fd, CALLER_ID_CONFIG, cid_det_args);
```

This configures the signal characteristics of the Caller ID detector for a standard operation.

Installation

The installation of the AT75C1010 software is as follows:

Change directory to `siap-uClinux-1.x.y/` and launch `patch_AT75C1010`. It carries out the following actions:

- Add `line.bin` DSP binary in the `prods/dk020/romdisk/romdisk/lib/` directory.
- Add `teltest/` demo sources subdirectory in the `apps/` directory.
- Add `tel/` driver subdirectory in the `linux/arch/armnommu/driver/` directory.
- Modify various configuration files.

After it ends, change directory to `linux/` and type:

```
> make xconfig
```

This updates your configuration according to the file modifications. Verify that the “Telephony tools” item is correctly set to “y”. Afterwards clean and rebuild your uClinux distribution.

Application Example

Synopsis

```
#include <asm/messages.h>
```

The demo application delivered with AT75C1010 driver illustrates its capabilities.

Starting DTMF Generation

On the board type:

```
> teltest -dial 15
```

This opens the DTMF device, waits for a phone number of 15 digits from the keyboard and dials it.

Starting Caller ID Detection

On the board type:

```
> teltest -cidstart
```

This opens the Caller ID device, waits for Caller ID characters from the phone line and screens the different parts of the message.

Atmel Headquarters

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

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

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

Japan
Atmel Japan K.K.
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 Product Operations

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

Atmel Grenoble
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-7658-3000
FAX (33) 4-7658-3480

Atmel Heilbronn
Theresienstrasse 2
POB 3535
D-74025 Heilbronn, Germany
TEL (49) 71 31 67 25 94
FAX (49) 71 31 67 24 23

Atmel Nantes
La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 0 2 40 18 18 18
FAX (33) 0 2 40 18 19 60

Atmel Rousset
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-4253-6000
FAX (33) 4-4253-6001

Atmel Smart Card ICs
Scottish Enterprise Technology Park
East Kilbride, Scotland G75 0QR
TEL (44) 1355-357-000
FAX (44) 1355-242-743

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

© Atmel Corporation 2001.
Atmel Corporation makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in Atmel's Terms and Conditions located on the Company's web site. The Company assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of Atmel are granted by the Company in connection with the sale of Atmel products, expressly or by implication. Atmel's products are not authorized for use as critical components in life support devices or systems.

Atmel® is the registered trademark of Atmel; SIAP™ is the trademark of Atmel.

ARM® and ARM7TDMI® are registered trademarks of ARM Ltd.; OakDSPCore® is a registered trademark of DSP Group Inc.; uClinux® is the registered trademark of Lineo Inc. Other terms and product names may be the trademarks of others.

Printed on recycled paper.

1786A–11/01/0M