Lucent Speech Processing Solutions

Software Development Kit

3.1

Release 3.1
October 2000
Official Edition

Lucent Technologies
Bell Labs Innovations



Copyright 1994 - 2000
Lucent Technologies All Rights Reserved

Lucent Speech Solutions Department
Lucent Technologies
2000 N. Naperville Road
P.O. Box 3033
Naperville, IL 60566-7033

Release 3.1
API Interface Version 40.21.6

First Edition -- October 2000

The latest version of this document can be accessed on-line:
Lucent (internal): *http://ihweb2.ih.lucent.com/lsps/documentation.html*
External: *http://www.lucent.com/speech* (under Services & Support)

Contents

1. Introduction to the LSPS Software Development Kit (SDK) 3
1.1 LSPS Application Program Interface (API) 3
1.2 System Architecture 3
1.3 Call Handling Architecture 4
1.4 API Glossary 5
1.5 Naming Conventions 6

2. SDK Examples, Tools and Test Programs 9
2.1 LSPS API Examples 9
2.2 Test Programs and Tools Using the LSPS Speech Processing API 10

3. LSPS Software Packages - UNIX 13
3.1 The LSPS API Package lspsapi 14
3.2 The LSPS SDK Package lspssdk 14
3.3 LSPS SDK Installation Directories - UNIX 15
3.4 LSPS Packages Installation - Unix 15
3.4.1 Special Considerations When Installing on Solaris 386 OS 16
3.4.2 Special Consideration When Installing and Running On SCO Unixware 2.1.3 17
3.4.3 Installing Some LSPS Tools on Non-Native Operating Systems 17
3.5 Compiling Application Code on Unix 17
3.5.1 Compiling and Linking on SCO Unixware 17
3.5.2 Compiling and Linking on Solaris 386 18

4. General API Information for Programmers 19
4.1 API Function Description 19
4.2 Channel and Resource Allocation, Configuration and Commands 19
4.3 Channels, Resources and Timeslots 20
4.4 Modes of Operation 21
4.4.1 Asynchronous Event Mode 21
4.4.2 Asynchronous Signal Mode 23
4.4.3 Synchronous Mode 24
4.5 Timeslots 25
4.5.1 HTS Timeslots 26
4.5.2 ETS Timeslots 27
4.5.3 VTS Timeslots 27
4.6 Error Codes, Completion Codes and All Other Codes 28
4.7 Error Messages 29
4.8 Events 30
4.9 Signal Handling (Unix Only) 31
4.10 Using API in a Multi-Threaded Program Environment 32
4.10.1 Free Threading Mode 32
4.10.2 Single Threaded Apartment Mode 33
4.11 Resource Allocation Conflicts 34
4.12 Cleanup After an Ungraceful Process Termination 35

4.13 Channel Handoff 36
4.14 Debugging 36
4.14.1 Debugging Versions of API libraries 36
4.14.2 User Application Trace Macro - USR_Trace 37
4.14.3 Internal API Trace Macros 37

5. The Speech Processing Interface 39

5.1 Speech Processing Boards - Boot and Initialization 39
5.1.1 Summary of Prepackaged Board Types and Their Resources 40
5.1.2 Creating Custom Board Types 42
5.1.3 Loading (Booting) Speech Boards 42
5.1.4 Generating the H.110 Bus Clock 43
5.2 Speech API Interface Summary 43
5.3 Speech Processing Channel Resources 44
5.4 Initialization of the Speech Processing Interface 44
5.5 Configuration Requests 45
5.6 Command Requests 47
5.7 Run-Time Control (RTC) 50
5.8 Completion of a Request on a Speech Processing Channel 51
5.8.1 Relevant Variables 51
5.8.2 API Operation 53
5.9 Timer Control of Speech Processing Resources 53
5.10 Speech Processing Channel Data 54
5.11 Speech Processing Event Data 56
5.12 Speech Processing Board Control Channel (BCC) 59
5.12.1 Control of 6UB5 board LEDs 59
5.12.2 Board Hot Swap 60
5.12.3 Control of the H.110 Bus Clock 60
5.12.4 6UB5 Board CPU Performance Measurements 60
5.12.5 BCC Configuration 61
5.12.6 BCC Commands 62
5.12.7 BCC Channel State 64
5.12.8 BCC Code Example 66
5.13 Play Resource 69
5.13.1 Play Resource Configuration 69
5.13.2 Play Resource Commands 70
5.13.3 Play Resource State 72
5.13.4 Play Example 73
5.14 Record Resource 74
5.14.1 Record Resource Configuration 74
5.14.2 Record Resource Commands 77
5.14.3 Record Resource State 78
5.14.4 Recording WAVE Files 78
5.14.5 Record Example 79
5.15 DTMF Digit Detector Resource 80
5.15.1 DTMF Detector Resource Configuration 80
5.15.2 DTMF Detector Resource Commands 82
5.15.3 DTMF Detector Resource State 83
5.15.4 DTMF Example 84
5.16 TDD Digit Detector Resource 85
5.16.1 TDD Detector Resource Configuration 85
5.16.2 TDD Resource Commands 86
5.16.3 TDD Resource State 87
5.16.4 TDD Detector Example 88

5.17 Tone Generator Resource 89
5.17.1 Tone Generator Configuration 90
5.17.2 Tone Generator Commands 91
5.17.3 Tone Generator State 91
5.17.4 Tone Generator Example 93
5.18 Call Progress Tone Detector Resource 94
5.18.1 Tone Detector Resource Configuration 94
5.18.2 Call Progress Tone Detector Commands 96
5.18.3 Call Progress Tone Detector Resource State 96
5.18.4 Call Progress Tone Detector Example 97
5.19 Connect Resource 98
5.19.1 Connect Resource Configuration 98
5.19.2 Connect Resource Commands 99
5.19.3 Connect Resource State 99
5.19.4 Connect Example 100
5.20 Transcoder Resource 101
5.20.1 Transcoder Resource Configuration 101
5.20.2 Supported Speech Coding Types 102
5.20.3 Transcoder Resource Commands 103
5.20.4 Transcoder Resource State 103
5.20.5 Transcoder Example 103
5.21 Text to Speech (TTS) Resource 108
5.21.1 TTS Resource Configuration 109
5.21.2 TTS Resource Commands 112
5.21.3 TTS Resource State 114
5.21.4 TTS Events 115
5.21.5 TTS Example 116
5.22 The Automatic Speech Recognition (ASR) Resource 117
5.22.1 ASR Resource Configuration 117
5.22.2 Passing the Drop-In Grammar Information to the ASR 122
5.22.3 In-Line Grammars 123
5.22.4 ASR Resource Commands 124
5.22.5 ASR Resource State 125
5.22.6 Results of the Recognition 126
5.22.6.1 Unparsed (Raw) Results 126
5.22.6.2 Parsed Results 127
5.22.7 JSGF Path 129
5.22.8 ASR-CD Example 130

6. Virtual Timeslots 131
6.1 Introduction 131
6.1.1 Description of Virtual Timeslots (VTS) 131
6.1.2 Drain Speech and Discard Speech 132
6.1.3 API Interface Summary 132
6.2 VTS Data Structures 133
6.2.1 VTS Equipment Configuration V5_EQP 133
6.2.2 VTS State Data Structure V5_VTS 134
6.2.3 VTS Event Data Structure V5_EVT 135
6.3 VTS Example 136

7. The Echo Canceller Interface 139
7.1 Introduction 139
7.2 EC Channels on 6UB5 Board 139
7.3 Initialization of the EC Interface 140
7.4 EC Channel Configuration Requests 141
7.4.1 Configuration Requests for EC Board Control Channel (BCC) 142

7.4.2 Configuration Requests for Echo Cancellation Channel 142
7.4.3 Configuration Values Structure E53_CFGVAL 144
7.5 Channel Data 145
7.6 Event Data 145

8. LSPS API Function Reference 147
A5_DestrThrInfo....148
A5_MakeHtsTid, A5_MakeEtsTid, A5_MakeVtsTid, A5_GetTid*....149
A5_SignalHandler, A5_SignalCheckpoint....150
A5_StaThread....151
A5_StrBdState, A5_BdState2Str....152
A5_Strerror, A5_Errno2Str....153
A5_Zerout....154
E53_CfgChan....155
E53_CloseChan....156
E53_EventChan....157
E53_Init....158
E53_MakeCid, E53_GetBd, E53_GetPort, E53_GetType....159
E53_Term....160
S51_CfgChan....161
S51_CloseChan....162
S51_CmdChan....163
S51_EventChan....164
S51_Init....165
S51_MakeCid, S51_GetBd, S51_GetPort, S51_GetType....166
S51_Term....167
USR_Trace, USR_PrintTrace, USR_TraceSw....168
V5_AbortVts....169
V5_CloseVts....170
V5_CreateVts....171
V5_DataEndVts....172
V5_DestroyVts....173
V5_EventVts....174
V5_InitVts....175
V5_OpenVts....176
V5_ReadVts....177
V5_TermVts....178
V5_WriteVts....179

9. LSPS SDK Commands and Tools Reference 181
a5dsm - display LSPS API timeslot and channel allocation tables....182
asr_capacity - estimate maximum number of ASR channels on a speech board....183
bdcmd - send a command to an LSPS speech board....184
bdstate - show & monitor the state of the LSPS speech and echo canceller boards....185
debug51- interactive debugging shell....186
ear - listen to local or remote timeslot using local speech device....187
ftcomp - compiler for EN-TTS text preprocessor rules....188
ftcdump - dump the contents of compiled EN-TTS text proprocessor rules....189
gc_make - ASR grammar compiler....190
genini - regenerate speech board initialization files....192
lspsinst - LSPS software installation tool for UNIX....193
lspsspmon - monitor speech board performance and load....194
mkboardtype - create a new type of speech board....195

mpron - create pronunciation for a list of words - multiple languages 196
pron - create pronunciation for a list of words - North American English 198
rear - listen to remote timeslot using local speech device 200
reboot51 - reboot LSPS speech board 201
s51boot - reboot LSPS speech board 202
status51 - display state of all LSPS speech boards in the host 203
xplot - display and manipulate speech data 204

Read Me First - What is New in 3.1

This is a short summary of significant differences between LSPS Release 3.0 and Release 3.1. The most noticeable difference is the removal of API interfaces for the LSPS AYC51, AYC52 and AYC53 boards, removal of the support for the ASR GSL grammars, and removal of the interfaces for Remote API (RAPI). Although the documentation for these interfaces has been removed in 3.0, the actual API interface still existed. This was to allow the users to migrate from these interfaces by giving them a grace period in which they can update their application software. In addition, there are several new features in 3.1:

- Increase of the maximum number of plug-in (drop-in) grammars from 8 to 108.
- Provide new ASR parameters to enable the application to select which part of the recognized ASR result should be delivered to the application.
- Remove a disconnect timeslot request (S51_DISCTS_CFGREQ) - this was never implemented, and cannot be implemented correctly.

1. **Interface to AYC-Series of LSPS Boards removed from the API**.
   Starting with 3.1 the only API interfaces are to the 6UB5 cPCI speech board. That means that all API interfaces (functions, macros, etc) which were specific to the AYC boards have been removed. The applications which plan on using the AYC51, AYC52 or AYC53 boards must stay with Release 2.3 which will continue to be supported in the foreseeable future.

2. **ASR GSL Grammar Format No Longer Supported**.
   The support for the GSL grammar has been completely removed starting with Release 3.1. The applications which plan on migrating to Release 3.1 need to rewrite their GSL grammars to the JSGF standard. Luckily, the differences between these grammars are mostly syntactic.

3. **Deletion of the LSPS Remote API (RAPI).**
   Starting with Release 3.1, the LSPS Remote API interface (RAPI) has been completely removed.

4. **Increase of the Maximum Number of Plug-In Grammars from 8 to 108.**
   The application can now specify up to 100 additional drop-in grammars. This is an increase from the previous limit of 8. The interface to specifying drop-in grammars remains source-code upward compatible, which means that only a recompilation is necessary.

5. **ASR Filtering Parameters.**
   The application has an option to request that only tagged words be delivered in the ASR result. In addition, the application can provide a list of words which are considered "content-less" and which will not be delivered as a part of recognizer result. There are usually words such as "the", "please", "at", etc.

6. **Remove Disconnect Timeslot Request (S51_DISCTS_CFGREQ)**
   The constant S51_DISCTS_CFGREQ has been removed from 3.1. It was actually never implemented and it did not do anything. The board has no concept of "disconnect". Once a timeslot is assigned to a resource, the board considers it "connected", and it is "disconnected" only when the resource is given a different timeslot, or the resource is deallocated. To truly disconnect the timeslot, the application must deallocate the resource.

1. Introduction to the LSPS Software Development Kit (SDK)

This document describes the Software Development Kit (SDK) and the Application Program Interface (API) for the "Lucent Speech Processing Solutions" (LSPS) 6UB5 board. The SDK contains the header files and the libraries for the API, numerous code examples (sources and executables), number of tools to help with code development and testing and documentation (this document).

The information provided in this document includes an overview of the system architecture which includes a description of how the software and firmware work together, description of the contents of the SDK package, the general API design philosophy, a description of each API data structure, and details of the API function calls. Additional chapters provide details covering the more complex subjects, such as ASR and TTS. A set of reference pages for the function calls and for the tools are also included.

The intended audience is programmers evaluating the functionality of the 6UB5 board and application programmers who are using the SDK to develop applications.

1.1 LSPS Application Program Interface (API)

This API can be thought to consist of three parts (subsystems) - one for the speech-related operations on the speech board, second one for the hardware echo canceller operations on the speech board and the third part which is used to control Virtual Timeslots (VTS). In addition to this document, the API can be best learned and understood by examining the source code for the many example applications provided with the SDK, and the various header files referenced in this document.

The API provides the following benefits to the application programmer:

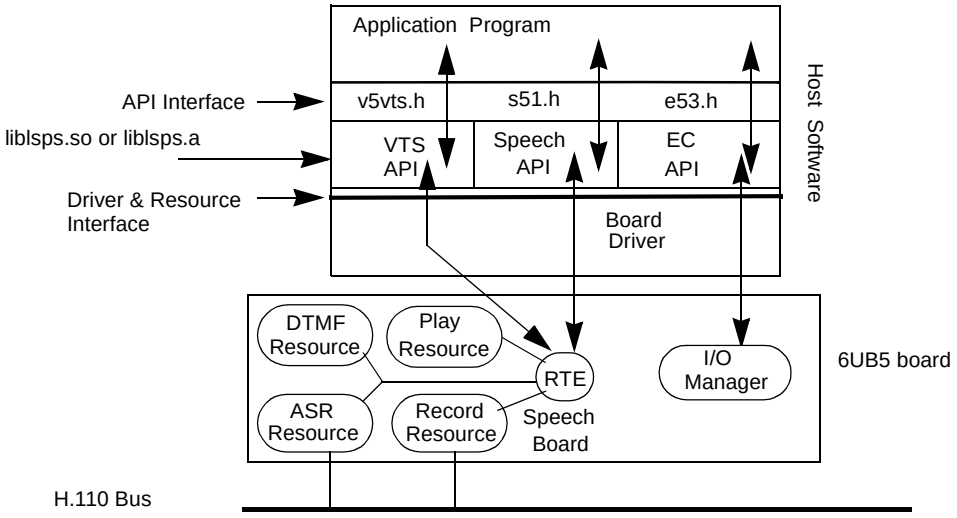
- Error checking. The API performs extensive checking of parameters and requests issued by the application and provides specific error messages to help the application programmer to quickly identify any problems.
- Simple and clean application interface. For example, adding *Play, Record, and DTMF* Resources to a given channel requires only one API call which is translated into many messages exchanged between the API and the various resources on the board (RTE, Resource Manager, Play, Record, and DTMF resources).
- The application can choose to use the API either in asynchronous or in synchronous mode. The direct interface to the board depends on the board type and is either asynchronous or synchronous (but not both). The API provides both asynchronous and synchronous modes of operation for all types of boards.
- Hide changes in the low-level resource and firmware software from the application and provide upward compatibility between old and new versions of the firmware.

1.2 System Architecture

The picture on the next page shows the software and firmware architecture below the application program level.

When the application program makes a function call to the **Speech Processing Board Interface** API library, the API translates it into appropriate board resource messages and sends them to the board (via the driver). The driver only routes and forwards the messages to the RTE[1] on the appropriate board. The RTE then routes the messages to the appropriate resource[2]. After performing the requested action, the resource sends a response message, which is routed by the RTE to the driver, and then routed by the driver back to the API. The API then returns an event to the application. The Echo Canceller and the Virtual Timeslot (VTS) API works in a similar fashion.

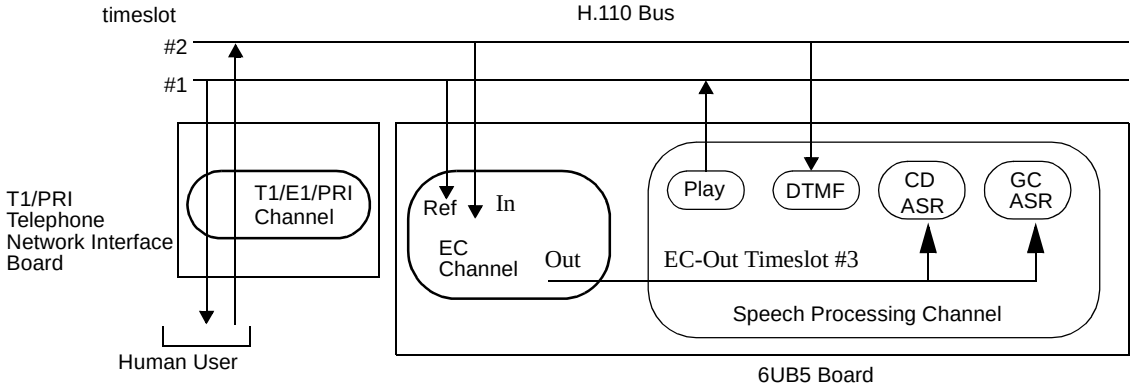
1. RTE is the operating system running on the Lucent Speech Processing Board.
2. For clarity, the picture shows only a few resources. In reality many resources run on the Speech Processing Board.



When the application program makes function calls to the **Echo Canceller Board Interface** library, the API handles the call in the same way as a call to the *Speech Processing Board Interface* described previously (API uses messages to and from the board).

1.3 Call Handling Architecture

The picture below shows an example of how to configure and connect resources and channels to handle incoming calls which require playing an announcement and detecting either DTMF, connected digit ASR, or grammar-based ASR.



The picture shows the following configuration:

- One T1/E1/PRI channel is connected to two H.110 timeslots: #1 (out to caller) and #2 (in from caller).
- One Echo Canceller channel is connected to three timeslots: #1 (reference), #2 (input), and EC-Out #3 (output to internal speech board timeslot).
- One Speech Processing channel is connected to three timeslots: #1 (out to caller), #2 (in from caller) and EC-Out #3 (internal input from the echo canceller). This speech channel is configured for a Play resource to play on timeslot #1, for the DTMF detector to “listen” on timeslot #2 and for the Connected Digit ASR and Grammar-Based ASR resources to "listen" on the internal EC-Out timeslot #3.

Usually such configuration would be set up before a call arrives. When a call comes in, the software commands the Play Resource to start playing an announcement on timeslot #1. This announcement is sent to the T1 board and then out to the caller, and is also used as the reference speech signal by the Echo Canceller channel. At the same time, the DTMF Detector is listening on timeslot #2, and the Connected Digit ASR (CD ASR), and Grammar-Based ASR (GC ASR) are listening to the "echo cancelled" caller voice signal on EC-Out timeslot #3. When the caller sends DTMF digits or speaks, this signal propagates through the T1 board to timeslot #2, then through the echo canceller to the EC-Out timeslot #3. As soon as one of the listening resources detects callers voice or DTMF signal, the Play resource may[3] automatically stop playing. And as soon as one of the resources recognizes a valid callers request, the other resources may be configured to also automatically stop.

1.4 API Glossary

Callback Function....This is a function optionally provided by the application and called by the API at appropriate time to deliver each event. If provided, the API calls one of the callback functions exactly once for every event created by the API, passing it a pointer to a single event structure. Notice that this is true even in the synchronous mode, in which the application does not usually see the events associated with requests, unless it provides a call-back function. The application can specify different callback functions on different levels, for example a global (default) callback function or a different callback function for each channel or for each resource type. The call-back function should normally return 1 (one). Returning 0 (zero) from a callback function signals to the API to unblock any currently blocked API function call and to return prematurely.

Channel..... This is a "virtual" API object and is represented by a *data structure*. The channel can be associated with zero or more ports or with zero or more resources. The channel capabilities and behavior can be defined using a channel configuration function. In addition, commands can be sent to and events can be received from a channel. Each channel is addressable using a *Channel ID* which is created using one of the *MakeCid()* macros. Each channel (data structure) describes the current state of the channel and all of the resources or ports that are allocated for that channel. The channel structure does not provide any information about the real-time events, that is, how the channel or its resources got to the state that it is in. In other words, do not try to deduce the real time events from the contents of the channel structure.

Completion Code.....This is the indication given to the application about the current state of the last request issued to the channel and/or to its resources. It is delivered with each event. Some of the possible values are: A5_UNDEF (no previous request), A5_OK (request completed successfully), A5_INPROG (request is still being processed) or one of the many other error codes. In case of an error, the channel and resource structure also contain a pointer to a text message describing the error.

Event ..... This is a *data structure* created by the API and given to the application. The event structure only describes a real-time event on a channel and does not describe the current channel state. For this it contains a pointer to the corresponding channel structure. There are three general classes of events:

1. *Completion Event* signals to the application that the requested asynchronous operation has completed (successfully or unsuccessfully) and there will be no more events associated with this request. For example when a file play completes, the application gets a play completion event.
2. *In-Progress Event* provides interim "report" about a request that is still being processed. For example during DTMF digit detection, the application may get an in-progress event(s), reporting each digit as it is detected. Once all digits are detected, the application gets a completion event.
3. *Unsolicited Event* provides event data that are not associated with a previous explicit request. For example an incoming call results in an unsolicited event.

Note that the API always generates all possible events that may be needed by any possible application. A particular application usually needs only a few of the events (usually the operation completion events) to implement it's functionality and it should ignore the rest. In other words, the application should not handle events just because they are there, but only if it actually needs them.

3. Depending on how the resources were initially set up.

Request ..... This is a particular operation to be performed on a given channel and on one or more of its resources. There are two types of requests: configuration and command. Configuration requests set up the channel parameters, they take a short time to complete (less than two seconds). Command requests are associated with the speech or call processing and there may be no time limit on their duration. The requests are initiated by an *API function call* made by the application. Some requests are inherently *synchronous* and always complete when the API function returns (they are always non-blocking). Other requests are *asynchronous* and may or may not complete when the function returns (depending on the operation). An asynchronous request always generates a *completion event*.

Resource ..... This is the smallest addressable virtual entity on the boards. There are a number of resource types, each type can perform a well defined function (for example play recorded speech, generate tones, etc). An instance of a resource is created by allocating it to a channel, and is destroyed by deallocating it. There can be several resources allocated to one channel. In other words, there is a hierarchy of a channel which contains/owns/controls zero or more of resources or ports.

Returned Value[4] ..... This is an indication given to the application about the validity of the API function call from the application. Some of the function calls return only an integer value (success or failure). Other functions return a pointer to the *channel structure* (valid or NULL), pointer to a *text error message* (describing the error if any) and an external *X5X_Errno* variable (S51_Errno, E53_Errno or V5_Errno) further describing the cause of failure. It is important to note that the returned values are associated only with the immediate function call. In other words, they only indicate whether the function call was valid and whether a request was initiated. Some examples of when returned values would indicate a failure are: an invalid parameter in the function call or the API could not communicate with the driver. On the other hand, the function returns success (A5_OK) even if the board firmware reported that the requested operation failed or the request eventually timed out. For these failures, the application must check the completion code(s) inside of the channel structure. Another way of looking at this is to see each API call as a sending of a message. The local API function returned values then only show whether the message was sent, but they do not give indication about the eventual result. To determine the current (or final) state of the request itself, the application must refer to the *Completion Code* which is delivered with the completion event and also stored in the channel structure.

Timeslot .... This is a mechanism used to send and receive data (usually encoded speech) between resources or between a resource and the host. Generally there can be only one resource transmitting to a timeslot at a time, but any number of resources can be receiving the data from the same timeslot. There is a separate section in this chapter which describes timeslots in more detail (See “Timeslots” on page 25.)

1.5 Naming Conventions

The following is a summary of the conventions used in naming the externally visible API variables, functions, macros, header files, etc. Each such name starts with a 3-letter or a 2-letter prefix. This helps to prevent naming conflicts with other software (applications) and also helps the programmer to identify and classify the name. The prefix is formed as follows:

1. Single letter designating the API subsystem:
   'A' or 'a'..... common to all API subsystems (“all”),
   'S' or 's'..... speech processing subsystem (“speech”),
   'E' or 'e' ..... echo cancellation subsystem (“echo”),
   ‘V’ or ‘v’ ..... virtual timeslot subsystem (“virtual timeslot”).
2. Followed by a single (arbitrary) digit '5' (the most random single digit number).
3. Followed by an optional single digit (used only for *S* and *E* subsystems):
   '1': the speech processing subsystem,
   '3': the echo cancellation subsystem.

4. These are the values returned directly from an API function call.

Note that this 3rd digit intentionally duplicates the information provided by the 1st letter. This is meant to prevent programming errors of referring to wrong functions or variables when simply mistyping the 1st letter. This is possible because many functions and variables in the 'S' 'and 'E' subsystems have exactly the same names except for the prefix. Thus it would be easy to unintentionally type *S5_ErrNum* instead of *E5_ErrNum*, but unlikely to type *S51_ErrNum* instead of *E53_ErrNum*.

Here are some examples:

`A5_OK` .... common constant representing “no error”

`S51_Init()` ..... speech processing init function

`E53_TraceSw` ..... echo canceller debug/trace flag

`v5vts.h` ... . global header file for VTS subsystem

2. SDK Examples, Tools and Test Programs

The *Software Development Kit (SDK)* includes **source code** for a large number of examples, tools and test programs. These are provided for the application programmer to help with learning the LSPS API. The examples denoted by '*' use the Dialogic *D41ESC* four channel tip/ring board and require the Dialogic API libraries and the files *d41esc.c* and *dti_tr.c* to compile. The compiled and ready to run examples and tools that do not need the Dialogic API libraries (the ones NOT denoted by '*') are included in the LSPS SDK. Only the programs which come with the source code are included in the lists below. Some of the tools come only in compiled versions (with no source), these are documented in the "LSPS SDK Commands and Tools Reference" on page 181.

2.1 LSPS API Examples

The *examples* include programs which do not usually perform anything of much use. They are written as simply as possible and are meant to be only as coding examples. Most of these programs provide a usage summary or a help page which can be accessed by specifying "-?" (dash question mark) as the only command line parameter to the program. For example: `example2 -?`

asy_playd.... Simple coding example, functionally useless. Plays prompt using Play and detects DTMF digits. Uses Play and Dtmf Detector resources and runs in asynchronous (non-blocking) mode.

asy_tones..... Simple coding example, can be used for testing DTMF generation/detection. Plays DTMF digits, detects them and verifies them. Uses Dtmf Detector and Tone Generator resources and runs in asynchronous (non-blocking) mode with callback.

example1*... This example uses the Dialogic *D41ESC* board for telephony interface. This example uses the API in blocking (synchronous) mode to perform the following:

- Waits for an incoming call.
- Answers an incoming call and plays a prompt.
- Detects DTMF digits dialed by the caller, also handles "barge-in", that is, digits dialed during the prompt.
- Plays the digits back as spoken words.

example2..... This program generates a sequence of DTMF digits, detects them and verifies that they were detected correctly. It uses asynchronous mode, so it can run multiple channels at the same time. It does not need and does not use any telephony interface.

example3..... This program plays a file containing digit recording while recognizing the recorded digits using CD ASR. After the digits are recognized, it compares the recognized digits with the played digits. The program uses asynchronous mode, so it can run multiple channels at the same time. It does not need and does not use any telephony interface.

example4..... This program plays call progress tones using the Tone Generator resource and recognizes the tones by using the Call Progress Tone Detector resource. The program uses asynchronous mode, so it can run multiple channels at the same time. It does not need and does not use any telephony interface.

example6..... This example initializes and connects one echo canceller channel, and then quits leaving the channel allocated and configured.

syn_playd.... Simple coding example, functionally useless. Plays prompt using Play and detects DTMF digits. Uses Play and Dtmf Detector resources and runs in blocking (synchronous) mode. Similar to *asy_playd*, except it uses blocking mode.

2.2 Test Programs and Tools Using the LSPS Speech Processing API

The test programs and tools perform functions which are (more or less) useful either for testing or for determining or changing system state. They also come with the source code and so can be used as examples, except that the code is not necessarily always simple or easy to read or understand. Most of these programs provide a usage summary or a help page which can be accessed by specifying "-?" (dash question mark) as the only command line parameter to the program. For example: `bdcmd -?`

a5dsm .....This tool displays the current allocation of the channels and allocation of the HTS/VTS transmit timeslots to different processes.

alloc.....This tool tests allocation of speech resources and tests orphan cleanup by allocating and deallocating a channel.

bdcmd.....This tool uses the board control channel to issue command(s) to the board. The commands include: boot board, take board out of service, return board back to service, ping the board.

bdstate .....This tool uses the board control channel to continuously monitor the current state of all the equipped speech boards. It prints the initial state, and then runs forever printing any changes in board states.

buftest.....This tool tests the API operations when using memory buffers (instead of files) when sending or receiving data for Play, TTS and Record operations.

cust .....This tool tests the customization capabilities of the API. The API can be customized in limited ways for different projects by using a special add-on project-specific libraries. This program is normally not compilable because the custom add-on library that it uses is not a part of the API or SDK.

dtmf_str* .....This tool detects DTMF digits in streaming mode or in burst mode and prints the digits as they are detected. It also has many options to test different features, such as dial pulse detection, ignoring tones present at start and others.

dtmf_tst .....This tool is similar to example2, except that is uses synchronous mode and therefore can only run on one channel at a time. This example also provides an option to test MF digits.

ear.....This tool makes it possible to use the local host speech hardware (SoundBlaster for example) to listen to an arbitrary timeslot on either the local machine or on any remote machine which is on the network and which is running the LSPS listen daemon (part of the LSPS API). Related tool is rear (remote-only ear) which can listen only to timeslots on remote machines.

multasr.....This tool is very similar to example3. This program plays a file containing spoken digit recording while recognizing the recorded digits using CD-ASR. After the digits are recognized, it compares the recognized digits with the played digits. The program uses asynchronous mode, so it can run multiple channels at the same time. It is stand-alone and does not use any telephony interface.

ntn*.....This tool is very similar to example1. This example also has the option to test the Connect resource. This is done by playing the prompt into one timeslot and redirecting it with the Connect resource to another timeslot (where the announcement is expected). This tool does the following:

- Waits for an incoming call. It uses the Dialogic *D41ESC* to connect to the telephone network.
- Answers an incoming call and plays a prompt.
- Detects DTMF digits dialed by the caller, also handles “barge-in”, that is, digits dialed during the prompt.
- Plays the digits back as spoken words.

ntnplus*.....This tool is really an enhancement of the ntn tool in that it uses not only the DTMF detector to detects digits, but also the connected digit ASR. It also uses echo canceller and allows several options:

- record the signal being detected (speech or DTMF) into a file,
- record the signal being detected (speech or DTMF) into a memory buffer,
- reset and initialize the echo canceller board,
- barge in on ASR recognition,

- explicitly and gracefully stop all resources before termination,
- option to use or not to use echo canceller,

In summary, it does the following:

- Waits for an incoming call. It uses the Dialogic D/41ESC to connect to the telephone network.
- Answers an incoming call and plays a prompt.
- Detects DTMF or spoken digits, also handles “barge-in”, that is, digits dialed or spoken during the prompt.
- Plays the digits back as spoken words.

playts..... This tool plays a set of play files into the specified timeslot on the H.110 bus.

recd41* ..... This tool records speech into a file or a memory buffer (implemented as a memory-mapped file). It uses the Dialogic D/41ESC for telephony interface, and the keyboard to start and stop the recording. It can be used as a primitive tool to record announcements. It does the following.

- Waits for an incoming call.
- Answers an incoming call.
- Prompts (on the screen) for the file name into which to record.
- Starts recording when given a keyboard command.
- Stop recording when given a keyboard command.
- Plays back the recording.
- Reprompts for recording into another file.

recdts..... This tool is similar to *recd41* except that the recording source is an H.110 Bus timeslot.

tts* ..... This tool uses Text To Speech (TTS) to play text files or memory buffers (using memory mapped files). Optionally it can also play speech files, and/or record the generated speech. It does the following:

- Waits for an incoming call. It uses a Dialogic *D41ESC* to connect to the telephone network.
- Answers an incoming call.
- Prompts (on the screen) for the file name(s) to play,
- Plays the files using either TTS or Play,
- Reprompts for another set of files to play.

vts_loop ..... This tool reads an arbitrary input file and sends the data to a virtual timeslot which is in turn read by a connect resource on the board. The connect resource then writes the data into a second virtual timeslot, which is in turn read back by the vts_loop program which then writes the data into another file.

vts_test..... This tool plays a file using TTS and send the resulting coded speech into a virtual timeslot. The virtual timeslot is read back by vts_test which then writes the coded speech into a file. It uses asynchronous mode and can handle several channels at the same time

vts_tts..... This tool uses TTS to play a text file into a virtual timeslot and then reads back the resulting coded speech and writes it to a file on the host. Functionally it is similar to *vts_test*, but internally it is a very different creature: it uses mix of synchronous/aynch modes, handles only one channel and is coded in a simpler and an easier to understand style.

3. LSPS Software Packages - UNIX

To develop LSPS software on Unix you need to install two packages on the development machine:

1. The API package ***lspsapi***, which is required to develop applications and also to run the applications,
2. The SDK Package ***lspssdk***, which is required to develop applications but is not necessary when running the applications.

At this time these packages are available for the following **Unix** operating systems:

- SCO Unixware (versions 2.1.3, 7.0 and 7.1) for i386, the installable packages are located in *uw386/ and common/* subdirectories on the installation CD-ROM.
- Solaris (version 2.6 and 2.7) for i386, the installable packages are located in *sol386/ and common/* subdirectories on the installation CD-ROM.

To run the LSPS application software on Unix you need to install the following packages on the application server machine in the order listed:

Speech board driver package ***lspssp***..... ..... contains the speech board driver, needed to boot the 6UB5 speech board and run speech applications on the host.

Run Time Environment package ***lspsrte***. ..... contains the basic firmware for the speech board, needed to boot the speech boards and run speech applications on the host.

API package ***lspsapi***..... ..... contains the API dynamic libraries and host daemons (server programs). It is needed both, to run application on the host and for application software development.

If you intend to also develop (compile) application software on this Unix host, you must also install the following package:

SDK package ***lspssdk***..... ..... the LSPS Software Development Kit contains the header files, archive (static) API libraries, documentation, coding examples, etc.

In addition the following optional package may be installed as needed by the application:

Text to Speech packages. Needed only when application is using TTS resource(s) on the speech boards. Note that the multi-lingual packages are common for all Unix systems, and they can be found on the installation CD-ROM in the subdirectory named "*common*", while the single language (old) TTS module is in an OS-dependent subdirectory.

- **mlttsenna** Multi-lingual TTS module: North American English,
- **mlttsfr** Multi-lingual TTS module: Continental French (Rel 2.2),
- **mlttsfrca** Multi-lingual TTS module: Canadian French (Rel 2.2),
- **mlttsge** Multi-lingual TTS module: German (Rel 2.2),
- **mlttsit** Multi-lingual TTS module: Italian,
- **mlttsspca** Multi-lingual TTS module: Castilian Spanish (Rel 2.2),
- **mlttsspna** Multi-lingual TTS module: North American Spanish,
- **ttsenna** Single language (old) TTS module: North American English.

ASR packages. Needed only when application is using ASR resource(s) on the speech boards. Note that these package are common for all Unix systems, and they can be found in the subdirectory named "*common*".

- **asrenna** Automatic Speech Recognition for North American English,
- **asrfrca** Automatic Speech Recognition for Canadian French,
- **asrge** Automatic Speech Recognition for German,
- **asrit** Automatic Speech Recognition for Italian,
- **asrspca** Automatic Speech Recognition for Castilian Spanish,
- **asrspna** Automatic Speech Recognition for North American Spanish.

3.1 The LSPS API Package *lspsapi*

This package must be installed before the SDK package *lspssdk*. It is required for application development and also to run the applications. It contains:

- API dynamic libraries *liblsps.so* and *liba5api.so,* installed in */usr/lib*. These are used by application programs which were compiled and linked with the dynamic libraries (this is the default under Unix).
- Remote timeslot listening daemon and server *eardmn* and *earsrv* installed in */usr/lib/lsps*. These programs are needed only if the developer wishes to "listen-in" on what is going on the H.110 timeslots. The *eardmn* is the program which runs in the background and is the destination for listen-in connection requests. To service these requests, the daemon starts up a separate *earsrv* process for each listen-in request.

The installation process gives the installer a choice whether to install and/or enable (start automatically) certain programs and software (tools, daemons and servers):

- The remote timeslot listening daemon should be enabled if you plan on using this machine for debugging your application. It enables you to remotely listen to the audio played on the timeslots (HTS or VTS) on this machine. The default is *disabled*.

In Unix hosts, enabling these servers causes the installation process to add appropriate entries to the */etc/inittab* file in order to start these processes automatically.

3.2 The LSPS SDK Package *lspssdk*

This package is required for application development. It is installed under the */usr/lsps* directories.You must install the *lspsapi* package (described previously) before installing this package. This package contains:

- API header files installed in */usr/include/lsps*.
- API archive libraries *liblsps.a* and *liba5api.a* installed in */usr/lib*.
- Documentation, installed in */usr/lsps/doc*.
- Source code for examples and tools, installed in */usr/lsps/src/**.
- Some compiled examples and tools, installed in */usr/lsps/bin*.
- Source for the TTS preprocessor compiler *ftcomp* and *ftcdump* in */usr/lsps/src/pars*.
- TTS preprocessor external rules (source) in */usr/lsps/src/pars*.
- ASR Grammar Compiler software needed to create grammar files for grammar-based ASR, installed in */usr/lsps/bin*.
- Other miscellaneous item considered useful in application development.
- "Non-native operating system" tools (WinNt, Solaris Sparc, etc). These are usually the tools which can be reasonably run in a "desktop" personal computer and which do not need the LSPS hardware. For example an installation on a Unixware host also provides the ASR grammar compiler (*gc_make*) and the remote timeslot listener (*rear*) for the WinNt OS. The user then must copy these tools from the Unixware host to the host running the appropriate operating system.

During the installation process, the system queries the installer whether to install the "**lspsti tools**" (default is "yes"). The installer should respond with "no" (do NOT install the *lspsti* tools).

3.3 LSPS SDK Installation Directories - UNIX

Once the LSPS SDK package is installed under the */usr/lsps* top directory, there are the following subdirectories:

lsps/src/ex/ .. .. This directory contains source code for examples. The examples are meant to be simple and well-documented code to help the programmer get familiar with the basic philosophy and methods of writing the API code.

lsps/src/s51/ ..... This directory contains source code for some of the tools used to test the speech processing API. Some of them may not be that well documented nor clearly written. But in some cases the source code may provide some help for programmers already somewhat familiar with the API. Also some useful tools can be found here, for example "*bdcmd*" found here can be used to reboot the speech processing boards.

lsps/src/e53/ ..... This directory contains source code for one of the tools which was used for testing the echo canceller API.

lsps/src/pars/ ..... This directory contains the source for the TTS preprocessor external rules and also the source for the compiler needed to compile the rules.

lsps/bin/..... ..... This directory contains the executable (compiled) versions of development tools and also some compiled examples. Examples which use Dialogic boards (and thus Dialogic API) were not compiled to avoid any licensing issues.

lsps/data/ ..... This directory contains the data files (speech files etc.) which are needed to run some of the tools and examples in the bin directory). There are several ways to enable the programs to find these data files:

- provide the directory or file name as a command-line option (if there is such an option),
- link/copy the data files to the current directory (for example to *lsps/bin*),
- use the *data/* as the current directory and execute programs with the *../bin* pathname.

lsps/doc/pdf/ ..... Documentation converted to the PDF format,[5] including this document.

3.4 LSPS Packages Installation - Unix

On the UNIX operating systems, packages are installed using the standard UNIX installation command *pkgadd*. The CD-ROM contains three groups of packages, each group is in a separate subdirectory:

common ..... packages that are common to Unixware 386 and Solaris 386, these are all the ASR packages,

uw386 ..... packages that are installable on the Unixware 386 system,

sol386..... packages that are installable on the Solaris 386 system.

Before you can run the **pkgadd** command to add a package, you must mount the LSPS CD-ROM. Here is an example for one Unixware system[6]:

```
/etc/mount -Fcdfs -r /dev/cdrom/c0b0tXl0 /mnt
```

After installation, once you are done with the CD-ROM, unmount the CD-ROM:

```
/etc/umount /mnt
```

It is recommended to NOT remove the already installed packages, but to install the new packages over the existing ones, although there are occasional exceptions. If the *pkgadd* refuses to install a package because the "package is already installed". In such case the already installed package must be removed first. This happens when the already installed package has different version than the new package, in other words when the two packages are not compatible.

5. Requires the Adobe Acrobat reader 4.0 or higher.
6. In this command, the *'X'* should be replaced with the SCSII ID (1-6) and */mnt* is an available mount point. Note that the command you should use to mount the CD-ROM may vary from the example because it depends on the CD-ROM device and on the type of Unix system that you are using.

If the *lspssp* driver packages must be removed, you must first remove the *lspsrte* package. When removing several packages, they should be removed in the reverse order in which they are normally installed (see below).

To remove an existing package on SCO/Unixware:

```
pkgrm pkgname
```

To remove an existing package on Solaris:

```
pkgrm -a lspsadm pkgname
```

On the **run-time** (execution) machine, the following system-dependent (uw386 or sol386) packages need to be installed in the order listed. The $OS symbol should be replaced with either uw386 or sol386 depending on the type of UNIX OS being used.

- Speech board driver:
  `pkgadd -d /mnt/$OS lspssp`,
- Reboot the machine
- Runtime environment:
  `pkgadd -d /mnt/$OS lspsrte`,
- API and API servers:[7]
  `pkgadd -d /mnt/$OS lspsapi`,
- Text to speech software (optional, see the complete list of TTS packages in “LSPS Software Packages - UNIX” on page 13):
  `pkgadd -d /mnt/$OS ttsenna` or
  `pkgadd -d /mnt/common tts_pkg_name`
- ASR software (optional, see list of TTS packages in “LSPS Software Packages - UNIX” on page 13):
  `pkgadd -d /mnt/common asr_pkg_name`,

On the **development** machine (which can be same as the run-time machine), the following system-dependent (uw386 or sol386) packages need to be installed in the order listed:

- C/C++ SDK (commercial package, not part of LSPS)
- LSPS Application Program Interface (API):[8] `pkgadd -d /mnt/$OS lspsapi`
- LSPS Software Development Kit:[9] `pkgadd -d /mnt/$OS lspssdk`

If you are using the dynamic libraries *liblsps.so* or *liba5api.so* to link the executable, the *same version* of the API must be installed on the machine where the application will run. A run-time check for API version compatibility is built into the API. The application will fail if the run-time and the link-time versions of the API differ. In such case either the application must be recompiled using the version installed on the run-time host, or the *lspsapi* package on the runtime host must be reinstalled to match the version used to compile the application.

3.4.1 Special Considerations When Installing on Solaris 386 OS

Before installing or removing LSPS packages on the **Solaris 386 OS**, manually install[10] the **installation administrative file *lspsadm*** on the host machine in the directory */var/sbin/install/admin*. This eliminates the need to type a full pathname for this file and minimizes the amount of typing and the chance of errors. The example below assumes that the LSPS CD-ROM has been mounted in */mnt*. You must be root to do this.

```
cp /mnt/sol386/lspsadm /var/sadm/install/admin/lspsadm
```

7. Refer to a previous section on the lspsapi package to help with answers to questions posed by the installation program.
8. Refer to a previous section on the lspsapi package to help with answers to questions posed by the installation program.
9. Refer to a previous section on the lspssdk package to help with answers to questions posed by the installation program.
10. This needs to be done only once on each host machine.

The *lspsadm* file must be used in the -a option when installing or removing packages on the Solaris 386 system:

```
pkgadd -a lspsadm ...
pkgrm -a lspsadm ...
```

3.4.2 Special Consideration When Installing and Running On SCO Unixware 2.1.3

Before installing the LSPS packages on the SCO/Unixware version 2.1.3, you must first install the Binary Compatibility Module (BCM) called **UW2compat**:

- Check if the package is already installed:
  ```
  pkginfo | grep UW2compat
  ```
- Download the *UW2compat.txt* and *UW2compat.pkg* files from the following SCO ftp site:
  ```
  ftp://ftp.sco.com/udk/unixware21
  ```
- Read the information in the *UW2compat.txt* file.
- Install the UW2compat package (... represents the directory where UW2compat.pkg is located):
  ```
  pkgadd -d .../UW2compat.pkg
  ```

Also the applications must be (re)compiled using the dynamic API libraries *liblsps.so* or *liba5api.so*. This is the default linking mode on SCO Unixware. So the existing applications should be OK unless the application link command explicitly specifies static linking. Application statically linked to the API libraries *liblsps.a* or *liba5api.a* will not work on the SCO Unixware 2.1.3.

3.4.3 Installing Some LSPS Tools on Non-Native Operating Systems

If you wish to run certain LSPS tools on a **MS-Windows**-based or **Sparc**-based desktop machine, copy the contents of the appropriate directories to the machine running the non-native OS:

- Windows-based tools are in */usr/lsps/nt386*,
- Solaris-on-Sparc tools are in */usr/lsps/solsparc*,[11]
- SunOs-on-Sparc tools are in */usr/lsps/sunos*.[12]

3.5 Compiling Application Code on Unix

An application must be linked with one of the LSPS API libraries:

- The production library *liblsps* contains the optimized code which does not contain the symbol tables and does not contain the API debugging traces.
- The debugging library *liba5api* contains a non-optimized code, the symbol tables and the API debugging traces. Also it logs internal API inconsistencies, suspicious events or internal errors (which are mostly harmless) either to *stdout* or to */tmp/lsps_api.log*.

Both of these libraries come in an archive (.a) version and in a dynamic (.so) version. Therefore there are four LSPS API libraries: *lspsapi.a*, *lspsapi.so*, *liba5api.a* and *liba5api.so*.

3.5.1 Compiling and Linking on SCO Unixware

To **compile and link** with the dynamic library *liblsps.so*:

```
cc appl.c -Kthread -D_REENTRANT -I/usr/include/lsps -llsps -lthread -o foo
```

11. Available only in LSPS Release 2.2 and higher.
12. Available only in LSPS Release 2.1.

To **compile and link** with the static library *liblsps.a*:[13]

```
cc appl.c -Kthread -D_REENTRANT -I/usr/include/lsps -Bstatic -llsps -Bdynamic \
                                                        -lthread -o foo
```

To **compile and link** with the dynamic debugging library *liba5api.a*:

```
cc application.c -I /usr/include/lsps -la5api -lthread -o foo
```

Note that because the LSPS API libraries are built using multi-threaded C libraries, the compiler command line requires *-D_REENTRAN*T and *-Kthread* parameters, and the linker command line requires *-lthread* library.

Also note that the -I argument is not necessary if your code includes the API header files as follows:

```
#include <lsps/a5api.h>
#include <lsps/s51.h>
(...etc...)
```

3.5.2 Compiling and Linking on Solaris 386

To **compile and link** with the dynamic library *liblsps.so*:

```
cc appl.c -I/usr/include/lsps -llsps -o foo
```

To **compile and link** with the static library *liblsps.a*:[14]

```
cc appl.c -I/usr/include/lsps -Bstatic -llsps -Bdynamic -o foo
```

To **compile and link** with the dynamic debugging library *liba5api.a*:

```
cc application.c -I /usr/include/lsps -la5api -o foo
```

Note that if your application does NOT use multithreading, no special multi-threading parameters are needed on the compiler or on the linker command lines.

If your application DOES use multi-threading, the compiler command line requires *-D_REENTRAN*T and either *-Kthread* parameter (for C programs) or *-mt* parameter (for C++ programs). You will also have to include the thread library directive *-lthread* on the linker command line. This is identical to compiling and linking on the SCO Unixware operating system.

Also note that the -I argument is not necessary if your code includes the API header files as follows:

```
#include <lsps/a5api.h>
#include <lsps/s51.h>
(...etc...)
```

13. Do not use static libraries on SCO Unixware 2.1.3.

14. Do not use static libraries on SCO Unixware 2.1.3.

4. General API Information for Programmers

4.1 API Function Description

The API consist of three closely related but for the most part separate subsystems: 1) speech processing, 2) echo cancellation, and 3) Virtual Timeslot subsystem. The interface for the first two subsystems of API consists of the following six **functions**[15]:

Init() .... initializes or re-initializes the API and returns the board information, can be called repeatedly any number of times.

Term() .... shuts down the API.

CfgChan() .. .. allocates and sets up a channel, allocates resources to the channel, and configures the resource parameters before call handling (can be called any number of times).

CmdChan() ..... requests an action (command) on one or more channel resources, usually during call handling.

EventChan() ..... gets event(s) from any and all channels and any and all of their resources.

CloseChan() ..... closes and deallocates one channel and all its resources.

The *Init()* and the *CfgChan()* functions allow the application to define callback functions. These callback functions are called by the API either from the *EventChan()* function or from the SIGPOLL signal handler (depending on the mode) when the application has a pending event.

The third subsystem (virtual timeslot, VTS) uses a different function set simply because the underlying functionality is inherently very different. But except for the API function set, the modes of operation, the error codes, modes of operation and most other issues covered in this chapter also apply to the VTS subsystem.

4.2 Channel and Resource Allocation, Configuration and Commands

Before a process can use a channel and any resources allocated to it (to send commands to the resources), it must first allocate the channel and its resources. The allocation of the channel is done (implicitly) by the first channel configuration request via *CfgChan()* function. Once a channel is allocated to a process, it stays allocated until explicitly deallocated via the *CloseChan()* function call. While the channel is allocated, no other process is allowed to allocate the same channel. This sanity check can be disabled by setting the API into the “no resource checking mode” (A5_NORESCHK_MOD) via the appropriate *Init()* function call. Keep in mind that if two processes attempt to control the same channel, pandemonium results.

The same configuration request for one channel can include parameters for zero or more resources. If the parameters are for a not (yet) allocated resource, the resource is allocated automatically. Parameters for an already allocated resource simply change its configuration.

This means that the application must first call the *CfgChan()* function to allocate the channel and its resources and then set some resource configuration parameters. Once the channel and its resources is allocated and configured, the application can call the *CmdChan()* function to send a command together with command parameters to the resources. The application can also call *CfgChan()* at any time to change the resource configuration parameters.

As suggested in the previous paragraph, the parameters that are sent to the channel and its resources are divided[16] into two general groups: Configuration and Commands.

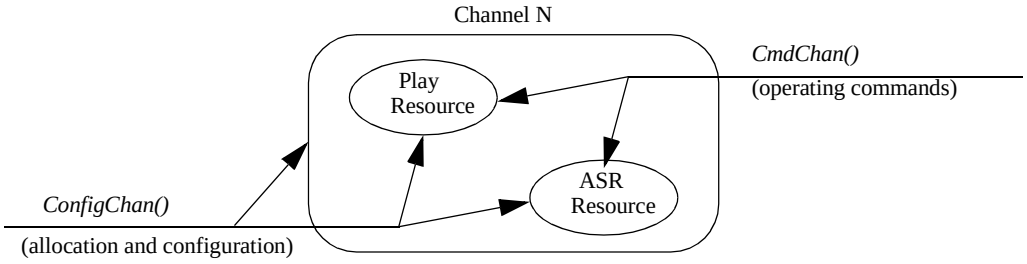
15. The actual function names are prefixed by S51_ or E53_ depending on the API type (speech, or echo canceller). Also, the echo canceller API does not need and does not provide the *CmdChan()* function.

16. In few cases the division is somewhat arbitrary, but most of the parameters are configuration parameters.

Configuration parameters are assumed to stay more-less constant between calls.[17] They are changed via the *CfgChan()* functions which are called during program initialization or before starting the processing of the calls. A configuration request always completes in a finite and short time.[18] Also, a configuration request does not have to contain all of the possible configuration parameters. If a parameter is not specified via the API, it will have some "reasonable" default. These defaults are coded into the firmware and are not listed in this document.

Command parameters are assumed to usually vary from phone call to phone call. Commands are issued via the *CmdChan()* functions. The commands are usually issued during call processing, that is, once the call has been answered. Some command requests can take a long (possibly unlimited) time to complete. A command request has to contain all of its command parameters and there are no defaults. Unused or non-applicable parameters must be set to zero or NULL. A command request can also optionally specify a "last second" configuration request as a part of the command. Such configuration request is completed first, before the actual command is started.

The picture below illustrates the relationship between a channel and its resources.



4.3 Channels, Resources and Timeslots

The *channels* and *resources* are "virtual" entities assigned dynamically. They must be created (allocated) and they must be destroyed. Only the process which created a channel or a resource can control it. A channel or a resource is local to a specific board and is of specific type. For example speech channel 1 on board 0 is different than speech channel 1 on board 1 or a echo canceller channel 1 on board 0. In other words, each channel is identified by a three-tuple: type, board and number. If any of these is different, it is a different channel. A resource is always associated a particular channel. A resource is identified by a two-tuple: channel and resource-type. If any of these are different, it is a different resource.

Channels and resources are created (allocated) dynamically (via **_CfgChan()*) and must be destroyed (deallocated via **CfgChan()* or **CloseChan()* or **_Term()*). If the resources are not deallocated, they remain even after the process which created them has terminated. This eventually causes you to run against the limit on number of simultaneously existing resources of a certain type. There are ways to prevent this from happening - refer to "Cleanup After an Ungraceful Process Termination" on page 35.

There are built-in limits on a number of channels and number of resources which can be allocated simultaneously on a given speech board. The resource limits are assigned to the board in the board initialization files. That means, for example, that although you can allocate 128 channels on one speech board, you may be only able to allocate 64 DTMF resources, or 36 TTS resources, or 12 ASR resources on one board. The limits depend on the type of resources (different types of ASR have different limits, for examples) and on the limits defined by the board type. When creating a new board type via "mkboardtype", you get to assign your own limits on number of simultaneously allocated resources.

17. Actually, the configuration parameters can be changed at any time, with the only side effect being an additional slight delay (less than 1 second) introduced by the additional configuration request.

18. In less than 2 seconds, usually in about 20 to 100 milliseconds.

On the other hand, the *timeslots* are using are "physical entities", that is, they always exist and are not "created" or "go away". They are independent of boards, that is, they are outside of the boards. Any board can access any timeslot, and boards (and channels) can share timeslots, providing that a certain rules are followed. There is only one kind of "speech bus" timeslots. These are the H.110 timeslots which are numbered in the range 0-4095 and in this document they are designated as "HTS" (H.110 timeslots)

There are also two other types of timeslots:

Virtual Timeslots.... .....VTS, created dynamically, see “VTS Timeslots” on page 27 and “Virtual Timeslots” on page 131, and

EC-Out timeslots.... .....ETS, hard-wired to EC hardware on the 6UB speech board, see “ETS Timeslots” on page 27 and “EC Channels on 6UB5 Board” on page 139.

In summary, the HTS and ETS timeslots always exist and do not need to be allocated or deallocated. The application must follow the certain rules when sharing a timeslot between different resources when transmitting speech to the timeslot. There is no restriction when sharing timeslots for receiving speech from an HTS or ETS timeslot. As long as the rules are followed, these timeslots can be shared between resources on the same board, and in the case of HTS, between different channels and on different boards.

4.4 Modes of Operation

The API provides **three basic modes** of operation: Asynchronous Event Mode, Asynchronous Signal Mode and Synchronous (Blocking) Mode. The mode is set by calling the appropriate API *Init()* function and it can be changed at any time[19] and also can be set independently for each of the three API interface types (Speech Processing, Echo Cancellation and Virtual Timeslots).

4.4.1 Asynchronous Event Mode

This mode (A5_ASYNC_MOD) is the default mode, which means that the API uses this mode when no mode is specified in the *Init* function. In this mode, the API functions never block but always return immediately. Whether an API function has completed the request at the time it returns depends on the type of the function call, this can also be determined at any time by looking at the completion code. If an API function could not complete the operation on return, it returns success (valid channel pointer or an A5_OK integer value) with an *in-progress* completion code (stored in the channel structure as A5_INPROG). In this case the application will get a completion event later, when the request completes. The application must call the *EventChan()* API function at appropriate time[20] to get the event(s).

When application is using this mode, it should use the following **general paradigm**:

- Initialize the interface via *Init()*, optionally providing a callback function.
- Allocate and configure the channel(s) via *CfgChan()* and optionally provide alternate callback function(s). Note that in this mode the configuration request may not be completed on return from the *CfgChan()* function, but the application must wait for a completion event.
- During call handling, send command(s) to the channel via *CmdChan()*. Note that in this mode the command does not usually complete on return from the *CmdChan()* function, but the application must wait for a completion event.
- Although the application can have only one request outstanding on each resource per channel[21], it can have many requests outstanding from many channels at the same time.

19. The channel must be idle, that is, no resources may have an in-progress completion state at that time.

20. Whenever there is data pending on the API’s file descriptor (or a file handle).

21. Except for Stop, Cancel or Abort requests, which can be issued at any time.

- While in this mode, the application must at all times monitor the API's file descriptor(s) via *poll()* or via *EventChan()* and explicitly call *EventChan()* function to get the event(s) when the *poll()* indicates that data is present on the API file descriptor. It is the *EventChan()* function that creates the event data structure(s), calls the appropriate application callback function(s) (if defined) and then returns the event data to the application. The event data is returned to the application either by the callback function, or by the *EventChan()* function, but not by both. In other words, unless the application calls the *EventChan()* function,[22] it will not get events and its callback function(s) will not be called.

Programmers Note: Timeout Events and a Completion Due to a Timeout:

If the board fails to respond to a request/message from the API, there is no data received on the API file descriptor to break out of blocking *poll()* or *EventChan()* functions. At the same time there is a timeout event ready to be delivered to the application reporting that the requested operation completed with a timeout error. Thus an application which waits for data/events in *poll()* or *EventChan()* with an unlimited time could lock up. The suggested way to code the application is as follows.

- If the application is using only *EventChan()*, do not set the wait time in *EventChan()* to more than 3000 milliseconds. Here is an example code fragment:

```
/*wait for speech events for a limited time only*/
if (SpeechEvents = S51_EventChan(3000, &SpeechEventCnt, &ErrMsg))
{
    /*handle speech events*/
    ...
```

- If the application is using *poll()* together with *EventChan()*, do not set the wait time in *poll()* or *EventChan()* to more than 3000 milliseconds, and always call *EventChan()* after returning from *poll()*, even if *poll()* indicates that there is no data present on the file descriptor. Here is an example code fragment:

```
/*wait for limited time, ignore return from poll()*/
(void) poll(FdArray, FdArrSize, 3000);

/*always check for speech events*/
if (SpeechEvents = S51_EventChan(0, &SpeechEventCnt, &ErrMsg))
{
    /*handle speech events*/
    ...
```

Programmers Note: Event Processing in Asynchronous Modes:

The application should read and process all the pending events before issuing a configuration or command request. If this is not the case, the events could back up and could potentially overflow internal queues and be lost. Some of the implications of this rule are:

- If issuing configuration or command requests in a loop to a numbers of channels, a call to *CfgChan()* or *CmdChan()* must be done only after all pending events are read. In other words, a *EventChan()* call must also be in the loop. The following abbreviated example assumes that a callback function is used to handle events:

```
...
for (ChNum = 0; ChNum < ChanMax; ChNum++)
{
    (void) S51_EventChan(40, NULL, NULL);
    Chan[ChNum] = S51_CfgChan(S51_MakeCid(BdNum, ChNum), &CfgReq, &CfgVal, &Errmsg);
    ....
}
```

22. At the appropriate time when an event is pending.

Note that the *S51_EventChan()* is called with **40 msec wait period**. Although theoretically no wait period should be necessary, the current implementation of the board firmware occasionally cannot tolerate such a "full-speed loop" and on some occasions could lose a message. This is true especially if the application runs on a fast host or is starting a large number of channels. The 40 msec delay parameter then introduces a small delay (usually less than 40 msec) which is sufficient to prevent overwhelming the board firmware and prevent message loss.

Programmers Note: API Function Calls from Callbacks in Asynch Modes:

When issuing *CfgChan()* or *CmdChan()* from a callback or from a signal handler, there is no way to assure that all of the pending events have been read and processed. This is because a call to *EventChan()* at this point is not allowed because it would be recursive. There are also a number of other possible problems which can be triggered by such a callback call which are too complex to describe here. As a consequence of this, issuing configuration or command requests from a callback function or a signal handler is **not recommended**. Also, the default behavior in the next LSPS Release (3.0) will be to reject such calls, and the application will have to explicitly disable this protection to be able to make these types of callback function calls.

4.4.2 Asynchronous Signal Mode[23]

This mode (A5_SIGPOLL_MOD) is similar to the asynchronous event mode (above) with one main difference: the application gets its events directly via its callback function without a need of explicitly calling *EventChan()*[24]. This mode is set by setting A5_SIGPOLL_MOD in the *Mode* argument. Again, in this mode the API functions never block but always return immediately. Whether the API function completes the request on return depends on the type of the function call. Completion can also be determined at any time by looking at the completion code. If an API function could not complete the operation on return, it returns success (valid channel pointer or A5_OK) with an *in-progress completion code* (stored in the channel structure as A5_INPROG). In this case, the application will get an event later (when the request completes) by a direct call from the API's SIGPOLL signal handler to the application's callback function.

When application is using this mode, it should use the following **general paradigm**:

- Initialize the interface via *Init()* and provide a default callback function (which is required in this mode).
- Allocate and configure the channel(s) via *CfgChan()* and optionally provide a per-channel callback function. Note that in this mode the configuration request may not be completed on return from the *CfgChan()* function, and the application must wait for a completion event.
- During call handling, send command(s) to the channel via *CmdChan()*. Note that in this mode the command may not be completed on return from the *CmdChan()* function, and the application must wait for a completion event.
- Although the application can have only one request outstanding on each channel, it can have many requests outstanding from many channels at the same time.
- In this mode the application will get its events (completion and otherwise) by automatic calls to its callback function and it can not call the *EventChan()* function. In other words, in this mode the application can get its events only through the callback function.

To implement this mode the API is using the **SIGPOLL signal**. So if the application is using the same signal for its own purposes[25] it must be **careful to avoid conflicts** by not changing the handling of the SIGPOLL signal in any way as long as the asynchronous signal mode is in effect. This implies the following rules:

- Set up the application's SIGPOLL handling before changing the API to the asynchronous signal mode.
- While the application is using the API in asynchronous signal mode, the application must not change the handling of the SIGPOLL signal in any way.

23. Implemented only for Unix OS.

24. In this mode a callback function is required.

25. For example because it is using another unrelated API (such as Dialogic API) which also uses SIGPOLL.

- When there is a SIGPOLL signal, the API's signal handler first performs its own data processing (including creating events and calling the callback functions) and then calls the application's SIGPOLL signal handler. In DOS-speak, the API "hooks into the SIGPOLL interrupt". This way, both the API's and the application's signal handlers get a chance to run.
- The application can change the handling of the SIGPOLL signals only after the API is no longer in this mode.

Programmers Note: Using the Asynchronous Signal Mode:

This writer feels that a warning is in order here. Signal handler functions should only perform trivial setting of a simple global variable (flag). They should not, for example, make any function calls. Violating this rule of thumb may result in undeterministic but occasionally catastrophic software problems which are difficult to detect and correct. This is mostly because of the inherent complexity of such design and because many programmers are not familiar with how to implement such design correctly.[26] Therefore this writer feels that the use of this mode should be discouraged.

4.4.3 Synchronous Mode

This mode (A5_BLOCK_MOD) is also known as the **Blocking Mode** and is set by setting A5_BLOCK_MOD in the *Mode* argument. In this mode, the API functions block and all return only after the requested operation has been completed.[27] Therefore in this mode, the application calls the *EventChan()* function only to get unsolicited events. But even in this mode the application can get all the events by specifying a callback function. In such case the API calls the callback function to deliver each event in the same way as in an asynchronous mode. In addition, a callback function can force a return from the currently blocked function call by returning a zero value. Actually, there are two ways to "interrupt" a blocked API function and force it to return before the operation is completed:

- If there is an event and the API calls one of the application's callback functions and the callback function returns a zero (0) value. In this case the blocked API function will be forced to return prematurely with a failure and with error code of A5_INPROG. To complete the operation, the application must either call the API function again with the same arguments or abort or stop the request. This "premature return due to callback" behavior can be inhibited by always returning a non-zero value from the callback function.[28]
- If a signal is received while executing API code, the application must also decide whether this signal should force a premature return from a blocked API function. To force a return, the application must call the *A5_SignalHandler()* function from its own signal handler for this signal. In such case the API function returns prematurely with a failure and with an error code of A5_EINTR. To complete the operation, the application must either call the API function again with the same arguments or abort, cancel or stop the request. If the application does not call *A5_SignalHandler()*, the API call remains blocked until it completes.

In either of these two cases, the application must either complete the request by calling the API function again with the same arguments, or must terminate the incomplete operation by a stop or an abort request to the same channel. In no case can the application start another request on this or on another channel before the current request is completed, cancelled, stopped or aborted. In other words, in this mode there can be only one request on one channel being processed at any one time.

When application is using this mode, it should use the following **general paradigm**:

- Initialize the interface via *Init()* and provide a default callback function (optional).
- Allocate and configure the channel(s) via *CfgChan()* and optionally provide a per-channel callback function.

26. For example access to program data structures which are written in the signal handler and read in the main code (or vice-versa) usually need to be controlled by locks and "critical sections", so the program does not "step on its own toes".

27. Completed successfully, unsuccessfully or because of a timeout.

28. This is the only use of the value returned by the callback function.

- During call handling send command(s) via *CmdChan()*. Note that in this mode, the *CmdChan()* may block (possibly forever) until the command is completed. If a callback function was specified and an event occurs while the *CmdChan()* is blocked, the API calls the callback function so the application can check out the event and decide whether to force a premature return from the blocked call by returning a zero value.
- When the application has nothing to do and is interested in unsolicited events, it should block using *poll()* or *EventChan()*, wait for unsolicited events, and get them by calling *EventChan()*.

Programmers Note: API Discarding Possibly Important Events in Synch Mode:

If any subsystem is blocked in the synchronous mode and an event is received for another LSPS API subsystem in the same process, that event can at that time be delivered to the application only via a callback function. If no callback function is defined, **the event is discarded without any notification to the application**. Therefore if the application is using more than one API subsystem (for example both the S51 and the E53 subsystems) and at least one of the subsystems operates in a synchronous mode, the application must define callback functions to catch these events. Even if all subsystems operate in synchronous mode, callback functions are necessary to deliver unsolicited events while the application is blocked in one of the API function calls. Therefore *it is highly recommended that, if the application operates in blocking mode and it is interested in events (even unsolicited events), the application defines callback functions for each of the API subsystems.*

Programmers Note: API Function Calls from Callbacks in Synch Mode:

It is never a good idea to call an API function from a callback function in synchronous mode. The recommended way to handle events in synchronous mode which require a LSPS API function call is to set an external state variable or flag, force a return of the blocked API function by returning 0 from the callback function and handle the event outside of the LSPS API.

4.5 Timeslots

Timeslots are a medium used to send and receive data (usually encoded speech) between resources on the same board or different boards, or between a resource on the board and a host process. In LSPS the term "timeslot" has come to be used to describe a general concept of a data bus that can have one writer and many readers. In other words there can be only one resource or a host process transmitting to a timeslot at a time, but any number of resources can be receiving the data from the same timeslot.

A timeslot is uniquely identified by its Timeslot ID (TID), which is a kind of handle to a timeslot and it has a data type of A5_TID.[29] To create a TID, you must use one of the TID creation macros. These are defined in the header file *a5api.h* and are described below. You must use a TID every time when you are identifying a timeslot to the LSPS API. Although there are several types of timeslots, they share the same type of TID which can be used anywhere where the API requires that a timeslot be specified. In other words, the resources use the different types of timeslots transparently to the application. But there are a number of restrictions to this, they are described later.

Currently the API supports three types of timeslots:

- **H.110 timeslots** (HTS), which are "real" or "physical" timeslots. These timeslots are external to the boards and span all the boards in one host.
- **EC-Out timeslots** (ETS), which are implemented in the 6UB5 speech board hardware. Each one of these is "permanently assigned" (hard-wired) to one 6UB5 echo canceller (EC) channel. Each such timeslot is local to a particular 6UB5 speech board. ETS timeslot can only be written by the single specific 6UB5 EC channel, but can be read by any other speech resource(s) on the same 6UB5 board. In other words, the application may set the EC transmit timeslot, but the timeslot number must match the EC channel number.

29. This is different from LSPS Release 1.2, where timeslots were identified by integers in a specific range.

- **Virtual Timeslots** (VTS), which are implemented in software on the speech board (6UB5). They work very much like a physical timeslots, except that a virtual timeslot is accessible only from a single board. In addition to providing for data exchange between resources on the board, this type of timeslot also allows direct read or write access by applications on the host.

The default coding transmitted to and expected from any timeslot is 8000 samples per second, 8-bits per sample bit-reversed Mu-Law. ALthough the speed (sample size and rate) cannot be changed, the coding can be changed to A-Law. This can be changed (on a host-wide basis only) by changing the following:

- Add the following speech board boot parameter in the file */usr/lsps/cfg/customer/startup.ini* within the *BoardDefault* section.

  SPBusCoding=ALaw

This results in all of the LSPS speech boards (6UB5) using A-law when transmitting or receiving speech data to or from any type of timeslot (HTS, ETS, VTS).

One exception to this is the Transcoder Resource, which can generate speech data with various codings. Although it is physically impossible for the Transcoder to write such re-coded data into H.110 (HTS timeslot), it can write such data to the virtual timeslot (VTS). Such VTS cannot be then meaningfully read by any LSPS resource, but can be only read by the host and written into a file to be processed outside of the LSPS. In other words, the only meaningful use of the Transcoder resource is to:

- write specially-coded speech data into a VTS, then
- read this VTS from the host and write the data into a file.

4.5.1 HTS Timeslots

This is a physical timeslot on a H.100 bus. It can be used to transfer data between any of the resources on the boards in one host. It cannot be used to transfer data to a process (application) on the host. A unique HTS TID is created by using the following macro (*a5api.h*):

```
HtsTid = A5_MakeHtsTid(TsNumber);
```

TsNumber.... .....timeslot number in range of 0 to A5_HTSNUM_MAX.

There are two issues that an application writer should be aware when using HTSs:

HTS silence generators:... ..... If there is no transmitter connected to a particular HTS, the data received from such "transmitterless" timeslot is not silence. The data is "undefined" which means that it can be silence or it can be random noise. To assure that there is "silence" on a specific HTS, the application must allocate at least one "transmission-capable" resource (Play, Tone Generator, TTS) and set its timeslot to point to this HTS. Note that it is **not necessary** for the resource to be started, it is sufficient to simply allocate it. If there several "transmission-capable" resources allocated on the same speech board and connected to the same HTS, they are automatically coordinated so that only one of them generates the silence data, and if one starts transmitting speech data, the silence generation is automatically disabled. This is true even for the resources that are on different channels, as long as they are on the same board.

Accessing one HTS from several speech boards.....If two "transmission capable" resources are allocated on two different speech boards and are connected to the same HTS, they do not coordinate their silence generation. They either both attempt to transmit silence to the same HTS, or while one resource is transmitting speech signal, the other resource transmits silence into the same HTS. In either case, the HTS does not work correctly, and in the best case no speech signal can get through on this timeslot. Note that it is not necessary for the "transmission capable" resource to be actively started (that is, actively transmitting data) to cause this problem, simply by allocating the resource and giving it a specific timeslot causes that resource to transmit silence data on this HTS.

Based on the above, the following basic rules are recommended: 1) make sure that there is always at least one "transmission capable" resource allocated for each HTS used by the application, and 2) do not share access to any HTS between channels on different boards.

Programmers Note: Hardware Limitation of HTS Timeslots on 6UB5 Board

There are two limitation when connecting the HTS timeslots to 6UB5 speech board:

- If the hardware version of the 6UB5 board is less than 10 or 0xa (use the *bdstate* command to get the version), your application cannot use HTS timeslots 1, 255, 383 and 511.
- Any HTS timeslot which is connected as an input to an echo canceller on a 6UB5 board cannot be also connected to another listener resource on the same 6UB5 board.

Programmers Note: Disconnecting the H.110 Timeslots (HTS):

The constant S51_DISCTS_CFGREQ has been removed from 3.1. It was actually never implemented and it did not do anything. The board has no concept of "disconnect". Once a timeslot is assigned to a resource, the board considers it "connected", and it is "disconnected" only when the resource is given a different timeslot, or the resource is deallocated. Of course no data is actually transmitted or received until the resource is started. One exception is that if a one or more "transmitting" resources (on the same board) were assigned to the same timeslot, this timeslot will now have a guaranteed "silence" during the periods when no resource is transmitting into the timeslot. In other words, the resource can be in one of two states in relationship to the timeslots: connected (assigned via configuration request) and "transmitting" or receiving (the resource is actively transmitting or receiving data). To truly disconnect the timeslot, the application must deallocate the resource.

4.5.2 ETS Timeslots

This is a physical timeslot internal to a 6UB5 model speech board. Each ETS is hard-wired to an output of one echo canceller (EC) port on this board. So there is a direct one-to-one physical relationship between an EC port on 6UB5 board and one ETS timeslot. Although the transmitter into this timeslot is hard-wired, any resource on the board can listen to any of the ETSs on the same board. A unique ETS TID is created by using the following macro (*a5api.h*):

```
EtsTid = A5_MakeEtsTid(BdNum, TsNumber)
```

BdNumber... ..... is the number of the 6UB5 board where the timeslot is located (0 through A5_BD_MAX).

TsNumber ... . is the timeslot number in range of 0 through A5_ETSNUM_MAX. The relationship between the ETS timeslot number and the EC port number to which it is hardwired is:
`TsNumber = EcPortNumber - 1`
There is also a macro which creates the correct ETS TID given the EC channel ID (CID) on the 6UB5 board (*e53.h*):
EtsTid = E53_Cid2XmtEtsTid(EcCid);

EcCid .... is the channel ID (E53_CID) of the echo canceller channel on the 6UB5 board. The returned TID is of the timeslot to which the given EC channel is transmitting.

4.5.3 VTS Timeslots

This is a virtual timeslot (implemented in software) on the 6UB5 speech board. There is a separate group of API functions used to manipulate the VTSs (create, destroy, open, close, read, write, etc). As long as the VTS is created (on the speech board), any resource on the board can use it in the same way as any other type of timeslot, subject to certain restrictions which are described in the separate chapter covering VTSs (see “Virtual Timeslots” on page 131). A unique VTS TID is created by using the following macro (*a5api.h*):

```
VtsTid = A5_MakeVtsTid(BdNumber, TsNumber, VtsType);
```

BdNumber... .....is the number of the speech board where the timeslot is located (0 through A5_BD_MAX).

TsNumber.... .....is the timeslot number in range of 0 through A5_VTSNUM_MAX.

VtsType ....determines who can read and/or write this VTS. This can be one of two constants (*v5vts.h*):
(V5_HOST_READ | V5_SRES_WRITE) ....one host process can read and one resource can write,
(V5_HOST_WRITE | V5_SRES_READ) ....one host process can write and many resources can read.

The Virtual Timeslots are covered in much more detail in their own chapter later in this document (see page 131).

4.6 Error Codes, Completion Codes and All Other Codes

The API uses the same set of constants (defined in file *a5api.h*) in any and all types of variables that indicate success or failure of any kind. These variables are:

Function Error Codes..... ..... These indicate the immediate result of an API function calls and are stored in one of three global variables: *S51_Errno* and *E53_Errno*.

Completion Codes..... ..... These are provided to the application in two places: 1) in the event structures (*S51_EVT* and *E53_EVT)*, which shows the completion code at the time of the event, and 2) in the channel structures (*S51_CH* and *E53_CH*), which show the current state of channel, that is, the completion code of the last requested channel operation.

Board State .... These show the state of the hardware (boards) and are stored in the board state structures (*S51_BD and E53_BD*). The constants which are used to define board states are also redefined using some more descriptive mnemonics.

These universal codes can be separated into several categories: Non-Errors, Application-Caused errors, Port Errors, Non-Fatal LSPS Errors and Fatal LSPS Errors. To test for these errors, the application program must use the '= =' (equivalence) operator.

Non-Errors: these indicate the current status of the request.

A5_UNDEF .... not set or not applicable

A5_OK..... ..... success (copacetic)

A5_INPROG..... ..... in progress (keep waiting)

A5_EINTR_ERR..... ..... caught a signal (request still in progress), call again or request a stop

Application-Related Errors: these usually indicate a problem with the application code, with the current LSPS board configuration or with the current board state.

A5_NOTIMPL_ERR..... ..... capability not implemented

A5_BADCID_ERR..... ..... invalid channel ID (CID) or VTS ID (TID).

A5_BADARG_ERR..... ..... bad argument

A5_UNINIT_ERR..... ..... not initialized

A5_INCOMPL_ERR..... ..... previous request not completed (must either continue or request a stop)

A5_UNEQP_ERR..... ..... board not equipped

A5_OOS_ERR..... ..... board not in service

A5_BUSY_ERR..... ..... channel or transmit timeslot allocated to another process

A5_ALLOC_ERR..... ..... can not allocate resource - possibly the board is not configured properly

A5_NOCLK_ERR..... ..... board has no clock on its H.110 bus

A5_RECURS_ERR..... ..... API function called from a signal handler

A5_BADVERSION_ERR. ..... incompatible version (board or firmware)

Port Errors: these usually indicate a problem with one or more ports or a channels on the board. The application can usually recover by deallocating and reallocating the resources on the affected ports/channels. There are several cases, depending on which resources and/or ports are affected.

A5_PORT_ERR..... .....affects the single port/channel and all resources on it

A5_PORTREQ_ERR..... .....affects only the current API request (re-issue the request, no reallocation necessary)

A5_PORTRES_ERR..... .....affects all resources on the port/channel

A5_PORTBD_ERR..... .....affects all ports/channels on the board

A5_DEALLOC_ERR..... .....channel or resource spontaneously deallocated, affects the single port/channel

Non-Fatal LSPS Errors: these are errors in internal communication between the API and the board.

A5_RCVMSG_ERR..... .....received bad message

A5_LIB_ERR..... .....internal API library error (a.k.a. bug)

A5_TIMEOUT_ERR..... .....timeout

A5_BOARD_ERR..... .....board firmware detected error

A5_IO_ERR..... .....device write or read error

Fatal LSPS Errors: these are usually non-recoverable errors, and the application should act accordingly.

A5_DEV_ERR..... .....device driver error

A5_MALLOC_ERR..... .....cannot get memory

A5_SNDMSG_ERR..... .....cannot create or send message

To simplify error handling, each of these error codes is also associated with one of the **Error Categories** listed below. To test for these categories, the application must use the '&' operator.

A5_ANY_ERR..... .....any error (includes all categories below)

A5_REQUEST_ERR.....affects this API request, reissuing the request may recover from this error

A5_RESOURCE_ERR.....affects this one resource, deallocate & reallocate may recover from this error

A5_RESFATAL_ERR .....affects all resources on the board, board reboot may be necessary

A5_OTHER_ERR.. .....all other non-fatal errors

A5_FATAL_ERR... .....any fatal error

4.7 Error Messages

When an error occurs, the application gets two types of indications:

- an error code or a completion code (as described above) and
- a text description of the problem.

The **error code** is meant to enable the application software to make an intelligent decision at run-time about what to do after something goes wrong. The **text description** is meant for human consumption, to help the programmers or the maintenance people to pinpoint the problem. When a text message is generated by the API, it is stored inside of the API code and one or more pointers (which are accessible by the application) are set to point to it. The application should never write or copy into the memory pointed to by this pointer, otherwise some very bad things may happen. The pointers to the error message can be found in any and all of the following places:

Event Structure..... .....this *ErrMsg* pointer is set when an event completion code shows an error, otherwise it points to a NULL string.

Channel Structure... .....this *ErrMsg* pointer is set when the last completed request failed, otherwise it points to a NULL string.

Function Argument. .....this *ErrMsg* pointer is provided by the application and it is set by the API to point to an internally stored error message or to a NULL string. Notice that the application should provide only the pointer and not the actual storage for the text.

4.8 Events

Events are data structures that are made available to the application when something of interest happens on a channel. Each event (data structure) contains several identifying variables:

Cid .....channel ID which identifies the channel associated with this event,

Ch .....pointer to a channel structure associated with this event,

EventType ...contains two items of information which are OR-ed into a long: *event class* and *event type*,

ResType.....identifies the type of resource associated with this event (S51_*_RES constants),

ComplCode .completion code associated with the resource operation.[30]

The *EventType* variable contains two items of information: *Event class* which classifies the event into one of three classes (see below) and *Event type* which identifies the event in more detail (see the S51_*_EVT constants). To test for either one of these, the application must use a logical AND operation of the *EventType* variable with the appropriate constant. for example:

```
if ((Event->EventType & S51_COMPL_EVT)
&& (Event->EventType & S51_CMDREQ_EVT))
    ....
```

Event Class: there are three general classes of events:

Completion Event... .....Marks completion of the current request and delivers (among other things) the Completion Code (A5_OK or error code).

In-Progress Event... .....This event is used only for a few requests. It is an interim report about the progress of the request. Its Completion Code is always A5_INPROG.

Unsolicited Event.... .....Event which is not associated with a previous request. It's Completion Code is undefined (A5_UNDEF).

Each API group has its own event data structure: *S51_EVT, E53_EVT* or *V5_EVT* and its own *get event* function: *S51_EventChan(), E53_EventChan() and V5_EventChan()*. There are two general ways for the application to get its events: as an event list returned from the *EventChan()* function, or alternately one at a time through its callback function(s). *In any case, the application always gets a particular event only once*: either passed to its callback function or returned from a call to the *EventChan()*. The callback function (if defined) has the priority here. Also, how the application gets the events depends on the current API mode:

Asynchronous Event Mode..... ..... in this mode, the application must call the *EventChan()* function to get all of its events. If callback function is specified, it is called once for every event with pointers to each event. If there is no callback function, the application must get its events as returned from the *EventChan()* function. That means that the application sees each event only once: either returned from *EventChan()* or via the callback function.

Asynchronous Signal Mode..... ..... in this mode, the application must provide callback function(s) and should never call *EventChan()*. In this mode *EventChan()* always returns a failure (invalid function call). All events are delivered only through the callback function.

Synchronous (Blocking) Mode..... ..... in this mode, the completion and in-progress events can be delivered only through the callback function (if any) while the application is blocked in an API call. On the other hand, to get unsolicited events while not blocked in the API, the application must call *EventChan()*. The

30. Note that this is the completion code at the time that the event occurred which is not necessarily the same as the current (latest) completion code for the same resource.

unsolicited events are then delivered either via the callback function (if defined) or returned by *EventChan()*. If the application is using several API subsystems (for example S51), it must use callback function to get its events if it wants to be certain to receive events which arrive while blocked.

4.9 Signal Handling (Unix Only)

In the **Asynchronous Event Mode** and the **Asynchronous Signal Mode** all of the API calls are non-blocking and all of the signals are "transparent" to the API.[31] This means that in these two operational modes the application is completely responsible for handling its signals and the API will not interfere with any signals, nor will it give an indication that a signal occurred while the API code was being executed.

In the **Synchronous (Blocking) Mode** the application can control the API's signal behavior in one of the two ways:

1. The API ignores the signal[32] and continues to block, waiting for completion of the current operation. This is the default behavior.
2. If a signal occurs at any time while executing the API code, the API unblocks and immediately returns with a completion code of A5_INPROG and an error number A5_EINTR_ERR. The application then has a choice of completing the operation by repeating the API call, or of explicitly stopping or aborting the operation (via Stop, Cancel or Abort commands). The application can enable this behavior[33] by calling the *A5_SignalHandler()* function from its signal handler. In the most trivial case, the application can set up the *A5_SignalHandler()* as the direct signal handler for the particular signal in order to force return from a blocked API function call on signal.

Although in the second case the API guarantees that it will never block if the signal occurs while executing any of the API code, that may not be sufficient for the application to handle signals correctly. This is because there is a small chance that the following can take place:

- The application checks for a signal, but there was none.
- The signal occurs immediately after the signal check but before an API function is called.
- Immediately after an API function is called, it blocks and does not "see" the signal because the signal did not occur while executing the API code.

To be able to handle this situation reliably, the API provides the *A5_SignalCheckpoint()* function. Once the application calls this function, any signal that occurs between this call and the next return from an API function will be considered as to have occurred within the API function. So the application would handle the above situation as follows:

- The application calls *A5_SignalCheckpoint()* function.
- The application checks for signal, but there was none.
- The signal occurs immediately after the check but before an API function is called.
- An API function is called and because the signal occurred after the call to the *A5_SignalCheckpoint()*, the API treats the signal as if it has occurred within the API function and will return immediately due to a signal.

Programmers Note: Calling API functions from Signal Handlers:

The application must never call any API function from any signal handler. If this is the case and the signal occurs while the API code is being executed,[34] the API will reject such function call as recursive with A5_RECURS_ERR error code. In general, signal handling functions in a well written application should normally contain only a few lines of code, which simply set a global flag to be processed later in the main code.

31. Except for the special handling of the SIGPOLL signal in the Asynchronous Signal Mode which was described before.
32. Although the application's signal handler would be called as usual.
33. Per signal type and even per signal instance (for each signal).
34. This is nondeterministic.

Programmers Note: Calling API Functions From an API Callback Functions:

Calling API functions from API callback functions is allowed (although not recommended), providing that the callback occurred to deliver a completion event. By this time the internal API state is stable and deterministic, and if the delivered event is a completion event the current operation is considered to be completed. On the other hand, this is not recommended and strongly discouraged in asynchronous mode, because there is a possibility that the application may not process events in timely manner, causing the pending events to accumulate and back up and eventually some events may be lost. This is also covered before in “Asynchronous Event Mode” on page 21.

4.10 Using API in a Multi-Threaded Program Environment

The API provides two modes of multi-threaded operation: **free threading** mode (FT, default) and **single threaded apartment** mode (STA).

4.10.1 Free Threading Mode

This mode is backward compatible with the previous releases of the LSPS API (2.2 and before). It adds mutual exclusion locks when accessing the internal API structures, so that application does not need to worry about calling the API functions from different threads at the same time.

In synchronous (blocking) mode, the application process can only control one channel for each of the API subsystems, regardless of how many threads the application is using. This is because while any thread is blocked in an API function call, all other API function calls are prevented from executing because of a locked mutex (mutual exclusion lock). In asynchronous (non-blocking) mode the application can control any number of channels, but it must manage the API behavior where incoming events returned by the **_EventChan()* function could be for channels allocated and controlled from any thread.

This mode is the default behavior, and it has the following properties:

- The API initialization (calls to **_Init()* functions) must be done at least once (from any thread) before any other API calls are made from any thread. The API parameters set by this call (API mode, callback function, etc.) apply to all subsequent API channels and operations regardless of the thread.
- The API termination (calls to **_Term()* functions) must be done only once (from any thread), after the application is done with using all of the channels in all threads. This call closes all of the channels regardless of which thread they were allocated or controlled from.
- An API channel allocated in one thread can be safely controlled from any other thread at any time. Also, the application can safely call any API function from any thread at any time.
- The call to the **_EventChan()* function returns all available events for all threads.
- The API provides only one file descriptor for the entire process (all threads) for monitoring the incoming data via the *poll()* Unix system call.

The SDK contain an example of multi-threaded program using the API in the *free-threading* mode. This program is called *free_thr*, and it does the following:

- allocate speech channel and resources,
- run a loopback test of DTMF generator/detector,
- run a loopback test of play/ASR-CD,
- close speech channel,

- repeat the above N times.

This program has the following properties:

- It runs on a number of channels at the same time.
- It performs a number of tests (repeats) on each channel.
- It deallocates/reallocates resources after every test.
- It uses asynchronous mode with callback for event handler.
- It uses a number of threads as follows:

thread	function
1	main thread: initializes, reads & handle events, cleans up
2	determines which resources need to be allocated and requests allocation.
3	starts a DTMF test command when resource is ready.
4	starts an ASR-CD test when DTMF test completed
5	closes the channel when all the test are done.

This program modularizes the application by function calls, and not by channels. This may be a bit unusual way to write an application, but it does modularize the design and simplifies the code.

4.10.2 Single Threaded Apartment Mode

The single threaded apartment mode (STA mode) is enabled by calling the function *A5_StaThread()*. This function must be called only once, must be called from the main thread and must be called before any other API function is called. Once the mode is changed to the STA mode, it cannot be changed back.

In synchronous mode each thread can only control one channel per API subsystem. This means that an application process can control a number of channels while in blocking mode, as long as each channel is controlled by it's own thread. In asynchronous mode each thread can control any number of channels.

This mode has the following properties:

- The API initialization (calls to **_Init()* functions) must be done at least once in *each thread* before any other API calls are made by the thread. The API parameters set by this call (API mode, callback function, etc.) apply only to the thread which called the **_Init()* function.
- The API termination (calls to **_Term()* functions) must be done from *each thread* which called the API initialization function. This call only closes the channels which were allocated and controlled from the same thread.
- Any given API channel must always be allocated and controlled by the same thread. The API rejects attempts to control a channel (**_CfgChan()*, **_CmdChan()* function calls) by a thread which did not originally allocate the channel.
- The call to the **_EventChan()* function returns events only for the thread which called it. That means that each thread must call **_EventChan()* independently.
- The API provides one file descriptor for each thread for monitoring the incoming data via the *poll()* Unix system call.

The SDK contain an example of multi-threaded program using the API in STA mode. This program is called *apt_thr*. The functionality is the same as the *free_thr* program, except that the implementation is completely different. The *apt_thr* example program does the following:

- allocate speech channel and resources,
- run a loopback test of DTMF generator/detector,

- run a loopback test of play/ASR-CD,
- close speech channel,
- repeat the above N times.

The *apt_thr* program is implemented as follows:

- It uses synchronous mode without callback.
- It uses a separate thread to run each test on each channel.
- It runs on a number of channels at the same time.
- It performs a number of tests (repeats) on each channel.
- It deallocates/reallocates resources after every test.

This program modularizes the application by channel, that is, each channels runs in its own separate thread.

4.11 Resource Allocation Conflicts

There are three types of **resources that cannot be shared**: 1) the transmit access to timeslots,2) the access to timeslots which are read by the echo canceller, and the channels themselves.

Because each timeslot can have only one transmitter transmitting to it at one time and each channel can be controlled only by one application process at one time. The API has a built-in mechanism to prevent these type of resource conflicts from occurring. It keeps tables in shared memory of connected timeslot transmitters and allocated channels. The **default behavior** is to use the shared memory tables to reject any request that would result in more than one transmitter on the same timeslot, or would allocate the same channel to more than one process. The current **allocation of the transmit timeslots and channels** (that is, the contents of the shared memory tables) can be **displayed** by using the tool ***a5dsm***.

One **exception** to this is that an application is allowed to assign (configure) the same transmit timeslot to more than one resource as long as they are on the same speech processing channel. It is acceptable to do this as long as no two resources on the same channel are actually transmitting to the same timeslot at the same time. It is the application process' responsibility to assure that this does not happen.

The application can also get direct access to the shared memory tables by using a set of functions and structures defined in the header file *a5shmem.h.* It is recommended that the application not change these tables. However, if it becomes necessary, the changes should be made by calling the appropriate API functions, not by changing the structures directly.

These API resource **allocation checks can be inhibited** at any time by setting the API mode to a no resource check mode, that is, by adding *A5_NORESCHK_MOD* to the *Mode* argument when calling the API's **_Init()* function. When this mode is enabled, the API does not check for resource conflicts and does not mark any new transmit timeslot allocations or channel allocations in its shared memory tables. Regardless of the current mode, any transmit slot disconnects or channel deallocations will clear their corresponding entries in the shared memory.

Because of a hardware restriction, once an echo canceller channel is assigned to a HTS timeslot, no other resource can listen in on this timeslot - although it can still transmit. There are no automatic built-in API checks to prevent the application from doing this. If the application assigns an HTS to two listening resources and one of them is an echo canceller input or a reference, the application receives an error response from the board.

For additional information on conflicts in allocation of HST timeslots (H.110 bus), refer to "HTS Timeslots" on page 26.

4.12 Cleanup After an Ungraceful Process Termination

If a process terminates without closing its channels, there are two problems. One is that the transmit timeslots and the channels used by the dead process are still marked as used in the shared memory tables.[35] The other problem is that these transmitters may still be physically connected and the channel resources may still be physically allocated on the boards. All of this should be cleaned up before the transmit timeslots and channels can be reused by other processes. This is most critical for the speech processing channels on the Speech Processing Board.

This cleanup can be done in one of two different ways.

1. **Automatically** by the API by using the **resource cleanup mode**, that is, by adding *A5_CLEANUP_MOD* to the *Mode* argument when calling the API's **_Init()* function. By default, the cleanup mode is off, but it can be turned on or off at any time. When this mode is on, the API will automatically (behind the scenes) perform any actions needed to clean up orphaned transmitters or orphaned channels before reusing them in a new request. The API cleans up only the resources that have been orphaned by dead processes. If the resource is allocated to an existing live process, the API will not attempt to do any cleanup actions and the conflicting requests simply fail.
2. **Explicitly** by allocating[36] and then closing the orphan channel(s) before using them. This can be done only in the *no resource check* mode (A5_NORESCHK_MOD) and it is the application's responsibility to keep track of which channels need to be cleaned up, that is, which channels have been orphaned and where their transmitters are connected. This is because when doing this it is possible to allocate a channel currently allocated to another live process, ending up with deallocating the channel resources and disconnecting the transmit timeslots that are being used by another process without the other process knowing about it.

`Here` is an **example** how to explicitly clean up an orphaned speech processing channel (S51Cid):[37]

```
Eqp = S51_Init(Mode|A5_NORESCHK_MOD, CbFun, &ErrMsg);
...
A5_Zerout(Req);
(void) S51_CfgChan(S51Cid, &Req, NULL, NULL);
(void) S51_CloseChan(S51Cid, NULL));/*this clears the orphaned resources*/
Eqp = S51_Init(Mode, CbFun, &ErrMsg);/*restore the original mode*/
```

Note that the **board control channels** (port zero) do not need to be cleaned up, since they can be (and usually are) allocated to any number of processes at the same time.

The API keeps the information about the allocated and tracked resources in shared memory. The simplest way to display the contents of this shared memory is by using the command:

```
a5dsm
```

The orphaned resources can be identified by ".RIP" which follows the process ID (pid) of their owner.

If a program needs to access this information, it can include the *a5shmem.h* header file and use the functions:

- To get information about which channel (*XmitCh*) has allocated timeslot *Tid* for transmitting:
  `A5_GetXmitCh(Tid, 0, XmitCh, ErrMsg);  /*fills in XmitCh*/`
- To get information about which process allocated the given channel (Cid):
  `A5_GetAllocCh(Cid, AllocCh, ErrMsg);  /*fills in AllocCh*/`

The program can then determine whether the owner process is dead by:[38]

```
if (kill(A5_CHAN.Pid, 0))
{
   /*dead process, orphaned resource*/
```

35. Unless the *no resource check* mode was enabled.
36. First channel configuration request always automatically allocates the channel to the current process.
37. For simplicity, the example does not show error checking. For another example, refer to *alloc.c* file in the installable API package.
38. Unix OSs only.

Some other ways (brute-force) to clean up the allocated resources:

- Reboot all the LSPS boards in a host (via *bdcmd* or *reboot51*) and remove the API shared memory segments via *ipcrm*. To remove these shared memory segments, you will need their segment IDs. First display all the shared memory IDs by using the *ipcs* command, and find the three API shared memory segments by looking for "7374" in the beginning or in the middle of the shared memory key.
- Reboot the host. This reboots all the boards and erases all allocation information in shared memory.

4.13 Channel Handoff

Channel handoff is the capability of one process to take over a channel previously allocated and used by another process and continue using it without a need to start from scratch. At this time the API **does not explicitly support** a channel handoff.

At this time the channel handoff is **not possible at all**. In other words, it is not possible for one process to take over a speech or echo canceller channel allocated (configured) previously by another process.

4.14 Debugging

The LSPS SDK provides a number of debugging aids to the programmer:

- debugging API libraries (liba5api),
- application (user) trace macro (USR_Trace),
- API (internal) trace macros (A5_Trace, S51_Trace, E53_Trace and V5_Trace).
- numerous tools: some are described in “Test Programs and Tools Using the LSPS Speech Processing API” on page 10 and other tools come with their own manual pages which are in the “LSPS SDK Commands and Tools Reference” on page 181 in this document.

4.14.1 Debugging Versions of API libraries

To help in debugging both the LSPS firmware and API and the application code, the API and SDK provides an alternate API libraries (both dynamic and archive) which have the equivalent functionality as the optimized production libraries, but also contain among other things: symbol tables, API internal traces, mutex sanity checking, logging API internal "suspicious" events and errors.

To use the debugging libraries, the application has several options:

- If using archive libraries, the application has to be relinked with the debugging library *liba5api.a* instead of *liblsps.so*.
- If using dynamic libraries, the application can be either relinked with the debugging library (liba5api.so). Or the user can temporary rename[39] the debugging library so the application uses it without the need for relinking (must be root to do this):

```
cd /usr/lib
mv liblsps.so liblsps.sav
ln liba5api.so liblsps.so
```

39. This is applicable only on the Unix OSs.

4.14.2 User Application Trace Macro - USR_Trace

The API provides a *USR_Trace* macro that can be used for application debugging. The capabilities of this macro are:

- It can be selectively turned on or off using one or more of 32 separate classes (bit fields) at run-time. To be able to selectively turn the tracing on and off, the application needs to implement a mechanism to set the global variable *USR_TraceSw* to any 32-bit hexadecimal value. Usually this is done via a command line argument to the application program.
- Overhead is minimal. Debugging statements that may be of future use can remain in the code. In addition the application can be recompiled completely without the trace code overhead (no debug tracing). This is done by recompiling the application with -DA5_NODEBUG (by defining the A5_NODEBUG macro).
- It automatically provides the following information: source file and line number, 10 millisecond-resolution time stamp and the version of the API library.
- The text output can be easily redirected to just about anywhere (for example into a messages sent via a local area network) For detail, see the *USR_Trace* manual page and the header file *a5api.h*.

The *USR_Trace()* macro requires a flag parameter and a *printf()*-like message. It is used like this:

```
USR_Trace(TraceFlag, (“printf-like specification”));/*uses USR_TraceSw global*/
```

TraceFlag .... A 32-bit value to test against the global trace flag.

Printf()-like specification. For example: *USR_Trace(0x10000, (“var x=%d”, x));*

Note the arcane double parentheses, they are required. Each trace can be then turned on or off at run-time by setting the global trace flag variable *USR_TraceSw* to the matching value. The macro works as follows.

- During run-time, the trace flag argument is AND-ed with a known global variable *USR_TraceSw.*
- If the result is not zero, the macro formats a debugging message (truncated after 4095 characters).
- The formatted output is passed to a function *A5_PrintTrace(char*)*, which prints the formatted message to the standard output. The application can redirect the message by supplying its own *A5_PrintTrace()* function as illustrated below.

This is an **example** of how to replace the default trace printing function with an application-specific trace printing function *MyTraceOut()*. Note that the generated trace text does NOT contain the terminating newline, therefore the terminating newline must be added by the *A5_PrintTrace()* function. The default *A5_PrintTrace()* function contained in the API library simply prints into the standard output.

```
#include <lsps/a5api.h>
extern    void         MyTraceOut(const char* Text);
...
USR_PrintTrace = MyTraceOut;
```

Here is an **example** of a trace. It assumes that the application has the capability to set the external global variable *USR_TraceSw* at start time (usually via a command line option):

```
USR_Trace(0x20000, (“error code=%d (%s)”, err_code, conv_err(err_code)));
```

The application can be compiled without the debug trace code by simply defining the A5_NODEBUG macro:

```
CC -DA5_NODEBUG ...
```

4.14.3 Internal API Trace Macros

First, these traces are not intended to be normally used by the application, but should be only used for debugging the LSPS API. They are described here because in some cases it is necessary to insert these traces into the application code to debug the interface between the API and the application.

The API itself uses several similar trace macros for its own debugging, and also makes them available to the application for API debugging. There are four other global 32-bit trace flags: *A5_TraceSw*, *S51_TraceSw*, *E53_TraceSw* and *V5_TraceSw*. As it is for the *USR_Trace*, to determine which traces are printed, the values contained in these variables are AND-ed with the appropriate trace flag during run-time. The trace message is printed only if the result of the AND is not zero. The application can turn on the API internal trace debugging macros only if it is linked using the debugging version of the API library (*liba5api.a* or *liba5api.so*). If the application is linked with the non-debugging API library (*liblsps.a or liblsps.so*), the internal API traces can NOT be turned on.

To be able to selectively turn the tracing on and off, the application needs to implement a mechanism to set the global variables mentioned above (**_TraceSw*) to any 32-bit hexadecimal value. Usually this is done via a command line argument to the application program. Both the internal traces and application level traces use the same underlying mechanism, so they work very much the same way.

The internal API trace macros are:

```
A5_Trace(TraceType, TraceFlag, (“printf()-like specification”));/*general macro*/
S51_Trace(TraceFlag, (“printf-like specification”));/*uses S51_TraceSw global*/
E53_Trace(TraceFlag, (“printf-like specification”));/*uses E53_TraceSw global*/
V5_Trace(TraceFlag, (“printf-like specification”));/*uses V5_TraceSw global*/
```

TraceType.... This must be one of: S51, E53 or V5, depending on which trace flag should be tested.

TraceFlag..... A value to test against the global trace flag.

Printf()-like specification... ..... For example: *A5_Trace(S51, 0x10000, (“var x=%d”, x));*

Some examples:

```
A5_Trace(S51, 0x10000, (“x=%d str=%s”, x, str));
S51_Trace(0x20000, (“error code=%d (%s)”, err_code, conv_err(err_code)));
```

5. The Speech Processing Interface

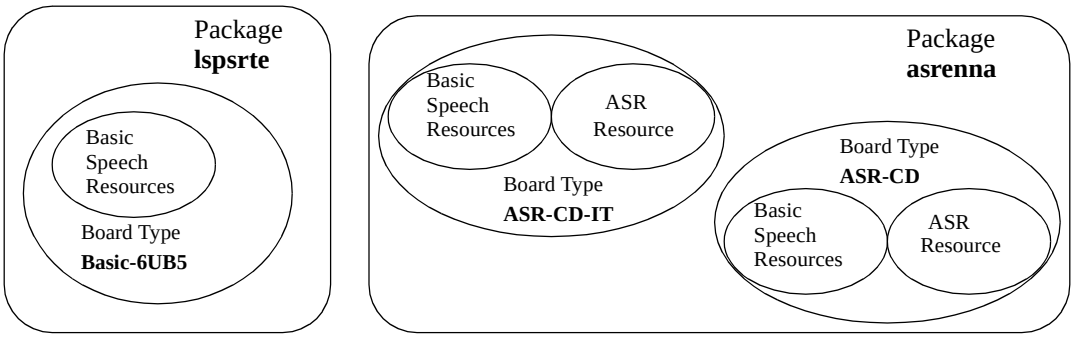
This portion of the API provides control of and interface to one or more 6UB5 Speech Processing boards. This introduction describes the structures and constants used as arguments to and returned values from the interface functions to these boards. They are defined in more detail in the *s51.h* header file.

5.1 Speech Processing Boards - Boot and Initialization

To be able to run an application using the API, the board to be used must be configured properly and must provide the necessary resources. The capability of each speech processing board in the system is determined by the software that is running on each board. Each board can potentially run a different set of software and thus can have different resources available for allocation. Usually the commands *status51* or *bdstate*[1] is used to determine the *Type* of each equipped board. This *Type* is actually an arbitrary label which gets translated using the board initialization files on the host into a particular set of board software that gets loaded during the board boot. If you examine the output of the *bdstate* command, you may see that different boards may have different *Type*. This does not mean that they have different hardware, it only means that they are currently running different software and so they have different capabilities. Some examples:

Board Type	Available Resources
Basic-6UB5.	Basic Resources (Play, Record, DTMF, TDD, Tone Generator, CP Tone Detector, Connect, TransCoder)
TTS	Text to Speech Resource only (no Basic Resources),
ASR-CD.....	Basic Resources, ASR Resource (Connected Digit ASR),
ASR-GC.....	Basic Resources, ASR Resource (Grammar-Compiled ASR).

To reiterate, the software on each board is loaded and started at the board boot time, and the set of software loaded during boot is determined from the board *Type*. There are a number of board types which come preconfigured and become available as a part of the LSPS packages. There is also a way to create a custom-designed board types based on the needs of an application. This is described in more detail elsewhere. The picture below illustrates relationships between software packages, board types and the speech resources.



When booting a board, the board *Type* must be specified. This is done either on the command line of the *s51boot* or re*boot51* command or interactively within the *bdcmd* command. The command *status51* or *bdstate*[2] will report the current type and status of the SP boards.

1. These are currently in */usr/lsps/bin* directory.
2. These are currently in */usr/lsps/bin* directory.

5.1.1 Summary of Prepackaged Board Types and Their Resources

There are number of speech **Unix** software packages, each containing a set of software that can be loaded on the speech boards. Each package comes with one or more different *board types*, that is, board software configurations that can be loaded on each board using it's specific name. Installing each package makes available an additional set of *board types* associated with that package.

The following shows *board types* and *resources* which become available for loading/allocation after the specific package is installed. Be aware that this list has a tendency to change between releases, and it also does not include the custom-created board types.[3]

The group of resources labeled ***Basic*** includes the following resources:

S51_PLAY_RES..... .....Play,
S51_REC_RES..... .....Record,
S51_DTMF_RES.... .....DTMF Detector,
S51_TDD_RES..... .....TDD Detector,
S51_TGEN_RES.... .....Tone Generator,
S51_TDET_REC.... .....Call Progress Detector,
S51_CON1_RES.... .....Timeslot Connect #1,
S51_CON2_RES.... .....Timeslot Connect #2,
S51_CODER_RES. .....Transcoder.

The ASR (S51_ASR1_RES and S51_ASR2_RES) and TTS (S51_TTS_RES) resources themselves can have different capabilities depending on the ASR/TTS ***Name*** which is specified during the allocation of the ASR/TTS resources. More details on this subject are in the "*ASR & TTS User's Guide*" which deals with the ASR and TTS resources.

The table below summarizes which board types are included in which packages, and which resources are included in each board type.

Pkg	Board Type	Description	Resources & ASR/TTS Names
lspsrte	Basic-6UB5	Basic resources only (6UB5)	Basic
ttsenna	TTS-6UB5	Rel 1.2 TTS (old): NA[a] English (6UB5)	Basic S51_TTS_RES: N/A[b]
mlttsenna	MLTTS-ENNA-6UB5	Multilingual TTS: NA English (6UB5)	Basic S51_TTS_RES "ENNA"
mlttsfr	MLTTS-FR-6UB5	Multilingual TTS: Continental French (6UB5)	Basic S51_TTS_RES: "FR"
mlttsfrca	MLTTS-FRCA-6UB5	Multilingual TTS: Canadian French (6UB5))	Basic S51_TTS_RES: "FRCA"
mlttsge	MLTTS-GE-6UB5	Multilingual TTS: German (6UB5)	Basic S51_TTS_RES: "GE"
mlttsit	MLTTS-IT-6UB5	Multilingual TTS: Italian (6UB5)	Basic S51_TTS_RES: "IT"

3. The list of valid board types on any system can be determined by examining the contents of the /usr/lsps/rte/BoardType directory, which contains one file for each board type of the same name.

Pkg	Board Type	Description	Resources & ASR/TTS Names
mlttsspca	MLTTS-SPCA-6UB5	Multilingual TTS: Castilian Spanish (6UB5)	Basic S51_TTS_RES: "SPCA"
mlttsspna	MLTTS-SPNA-6UB5	Multilingual TTS: North American Spanish (6UB5)	Basic S51_TTS_RES: "SPNA"
asrenna	ASR-CD-6UB5	ASR: NA English (6UB5)	Basic S51_ASR1_RES, S51_ASR2_RES: "CD_1" to "CD_16", "CD_UNKNOWN", "CD_PHONE", "CD_PHONE+" "IW_YESNO"
asrfrca	ASR-CD-FRCA-6UB5	ASR: Canadian French (6UB5)	Basic S51_ASR1_RES, S51_ASR2_RES: "CD_1_FRCA" to "CD_16_FRCA", "CD_UNKNOWN_FRCA", "CD_PHONE_FRCA", "CD_PHONE+_FRCA" "IW_OUINON_FRCA"
asrge	ASR-CD-GE-6UB5	ASR: German (6UB5)	Basic S51_ASR1_RES, S51_ASR2_RES: "CD_1_GE" to "CD_16_GE", "CD_UNKNOWN_GE", "CD_PHONE_GE", "CD_PHONE+_GE" "IW_OUINON_GE"
asrit	ASR-CD-IT-6UB5	ASR: Italian (6UB5)	Basic S51_ASR1_RES, S51_ASR2_RES: "CD_1_IT" to "CD_16_IT", "CD_UNKNOWN_IT", "CD_PHONE_IT", "CD_PHONE+_IT" "IW_SINO_IT"
asrspna	ASR-CD-SPNA-6UB5	ASR: NA Spanish (6UB5)	Basic S51_ASR1_RES, S51_ASR2_RES: "CD_1_SPNA" to "CD_16_SPNA", "CD_UNKNOWN_SPNA", "CD_PHONE_SPNA", "CD_PHONE+_SPNA" "IW_SINO_SPNA"
asrspca	ASR-CD-SPCA-6UB5	ASR: Castillian Spanish (6UB5)	Basic S51_ASR1_RES, S51_ASR2_RES: "CD_1_SPCA" to "CD_16_SPCA", "CD_UNKNOWN_SPCA"

a. The abbreviation "NA" stands for "North American".
b. The old version of TTS does not support, among other things, TTS Names (it only handles NA English).

Note that the prepackaged board types provide only the connected digit (S51_CD_ASRTYP) ASR type. To use other ASR types, such as grammar-based ASR (S51_GC_ASRTYP), you must create your own board type using the *mkboardtype* command. Refer to the **TTS and ASR User's Guide** for a more detailed step-by-step instruction about how to do this.

5.1.2 Creating Custom Board Types

If none of the standard board types provided with the LSPS software suit the application needs, it is possible to create a custom board type(s) using the *mkboardtype* command. You can use the *mkboardtype* utility to create a custom board type which provides any combination of available resources. Note that the symbol *$NEW-TYPE* should be replaced with the actual name of the new board type.

```
mkboardtype -t $NEW-TYPE
```

This is an interactive command which asks a series of questions and then creates a file */usr/lsps/cfg/customer/$NEW-TYPE.cfg*. This file then contains the definition of the new board type. This definition is then processed automatically by the *genini* program, which generates the actual files needed by the new board type and places them in the appropriate directories.

When allocating an ASR resource, the application must not only specify the resource type (S51_ASR1_RES and/or S51_ASR2_RES), but also the *ASR Type* and *ASR Name*. The current *board type* also determines which *ASR Types* and *ASR Names* can be used by the application (are available) when allocating ASR Resources on that type of board. This is described in more detail in a separate manual **TTS and ASR User's Guide**.

Note that reinstalling or updating LSPS software does not remove the previously created custom board types created by *mkboardtype*. When the LSPS software is reinstalled or updated, the only LSPS files that are guaranteed not to change are in */usr/lsps/cfg/customer*. During LSPS reinstallation or update, the *genini* program is run again to regenerate the actual configuration files needed by the standard and custom board types, but the basic definitions in */var/cfg/customer* do not change.

5.1.3 Loading (Booting) Speech Boards

A speech board must be rebooted to change it's board type. One way to reboot a speech board is:

```
s51boot -t $TYPE -b $BDNUM
```

Where $BDNUM is the speech board number (0-N) and $TYPE is the desired board type.

Another similar command is *reboot51*. Yet another way is to use the *bdcmd* command. This command interactively queries for board number, action (reboot) and the board type.

This kind of board type change lasts only until the next system reboot. If the system is rebooted, all board types change back to their *system startup types*.

The board **system startup types** are initially determined by the contents of the file */usr/lsps/cfg/lsps/startup.ini*. This file also contains extensive comments about its contents and how to change it. But do not change the contents of this file, because this file is overwritten during LSPS software (re)installation. To customize the system startup board types you must hand-edit the file */usr/lsps/cfg/customer/startup.ini*. This file overrides any assignments in the */usr/lsps/cfg/lsps/startup.ini* file. If this file does not exist, copy it first from */usr/lsps/cfg/lsps* directory. Here is an example of *customer/startup.ini*:

```
# BoardDefaults specifies a board model to board boot type.
[BoardDefaults]
6UB5=Basic-6UB5
#
# BoardAssignments specifies specific board boot types for specific
# board numbers. This section should only be used when the same
# model of board needs to be booted into different types.
[BoardAssignments]
Board0=ASR-CD
Board1=TTS
```

The [BoardDefaults] section defines the board type for any board which is not in the [BoardAssignments] section. The above example sets the default boot board type to "ASR-CD" for board 0, "TTS" to board 1 and to "Basic-6UB5" for all other boards.

5.1.4 Generating the H.110 Bus Clock

If after booting the speech board, the status51 shows a "**No Clock**" state, you are missing a **clock signal on the H.100 bus**. Normally the H.100 bus clock is generated by another board, usually by a telephony interface board.

If the 6UB5 speech board is the only board in your system, it is possible to have it generate the H.100 bus clock. To get the **speech board to generate H.110 bus clock**, edit the custom startup file */usr/lsps/cfg/customer/startup.ini*. More details about this file are in "Loading (Booting) Speech Boards" on page 42. The entry for the board which should be generating the clock should look something like:

```
[BoardAssignments]
Board0=ASR-CD
Board0Arg=SCBusMaster=1
```

In general, the directive for clock generation using a specific board must follow the board type assignment for this board and must be in the following format:

```
BoardNArg=SCBusMaster=Flag
```

Where:

N ..... is the board number 0-31

Flag ..... is one of: 0 (no clock generation), 1 (generate clock A), 2 (generate clock B).

After changing the startup.ini file, you must either kill the *lspsspstartup* process for the board N, or reboot the host.

5.2 Speech API Interface Summary

The API interface to the speech board capabilities is defined in the header file */usr/include/lsps/s51.h*. This file contains not only the data structures, #defines and function prototypes, but also some significant comments. Therefore it is a good idea for a programmer who is starting to use the speech board API to get familiar with this file.

Each Speech Processing Board provides a number of channels and each channel can be configured to provide one or more resources (ASR, Play, Record, etc.). The absolute maximum number of channels handled by one board 'N' is given by the variable *S51_EQP.BdStat[N].MaxChan*,[4] but in reality it may be less depending on the board model, available board memory and on the mixture of the resources used on each channel. Each of these channels is represented by a Channel ID. This ID is constructed from the board number and a channel number by using the **S51_MakeCid() macro**:

```
ChanId = S51_MakeCid(SpBoard, SpChan);
```

The arguments to the *S51_MakeCid()* are as follows:

SpBoard ..... the physical board (0 to S51_BD_MAX)

SpChan..... the channel number (1 to S51_CHAN_MAX).

The resulting Channel ID "handle" then contains (bit-packed) information about the board and the channel.

The **Speech Processing Board Interface** consists of the following 6 functions:

S51_Init() ....initializes the interface and returns the board information.

S51_Term() ....shuts down the interface.

S51_CfgChan()..... .....allocates (on the first call) and sets up one channel (configuration parameters).

S51_CmdChan()... .....requests an action (command) on one channel.

4. Currently this is 64, which is given by the overall limit of 64 different H.100 bus time slots available at any one time.

S51_CloseChan()... .....closes and releases one channel.

S51_EventChan().. .....gets event(s) from any channel.

5.3 Speech Processing Channel Resources

Each speech processing channel is allocated and (re)configured by calling the *S51_CfgChan()* function. Each channel can be configured to have any combination of the following resources:

Play (S51_PLAY_RES)..... ..... Plays from a set of files or from a memory buffer.

Record (S51_REC_RES)..... ..... Records into a file or a memory buffer.

DTMF Detector (S51_DTMF_RES) ..... Detects Dual Tone Multiple Frequency (DTMF) digits.

TDD Detector (S51_TDD_RES). ..... Detects Telecommunication Device for the Deaf (TDD) digits.

CPT Detector (S51_TDET_RES. ..... Detects Call Progress Tones (CPT) (dial tone, busy tone, etc.).

Tone Generator (S51_TGEN_RES)..... Generates tones (DTMF, MF), dial tone, etc.).

Text to Speech (S51_TTS_RES). ..... Converts Text To Speech (TTS).

Timeslot Connect #1 (S51_CON1_RES)..... Connects two H.100 bus time slots by receiving from one and transmitting on another. First configured Connect resource on this channel.

Timeslot Connect #2 (S51_CON2_RES)..... Second configured Connect resource on this channel.

Coder (S51_CODER_RES)..... ..... also known as "transoder", converts between different audio codings,

ASR #1 (S51_ASR1_RES)..... ..... First configured speech recognition (ASR) resource on this channel.

ASR #2 (S51_ASR2_RES)..... ..... Second configured speech recognition (ASR) resource on this channel.

A group of channel resources can be allocated and/or completely (re)configured at once by a single call to *S51_CfgChan()*. Also, a single call to *S51_CmdChan()* can control any or all of the configured resources. In *synchronous (blocking) mode* these function calls do not complete until all the resources complete the requested operations. If the application is getting events (via callback functions or *S51_EventChan()*), the interface generates the following events:

- in-progress events (for example, a single DTMF digit detected),
- resource allocation and/or configuration completed events.

5.4 Initialization of the Speech Processing Interface

The **S51_EQP structure** is returned by the *S51_Init()* function. The structure describes the number of boards, the characteristics of each board, and the state of each board.

```
typedef struct
{
   u_short      Fd;                 /*file descriptor to use for poll()*/
   u_short      NumBoards;          /*how many boards configured in this host*/
   u_short      NumUse;             /*how many are available for use*/
   S51_BD_STAT  BdStat[S51_BD_MAX+1];/*board characteristics and state*/
}
S51_EQP;
```

The variables in these structures are:

Fd ....file descriptor to be used on the UNIX host by *poll(2)* to obtain data from any Speech Processing Board,

NumBoards . .....number of boards configured in the host files,

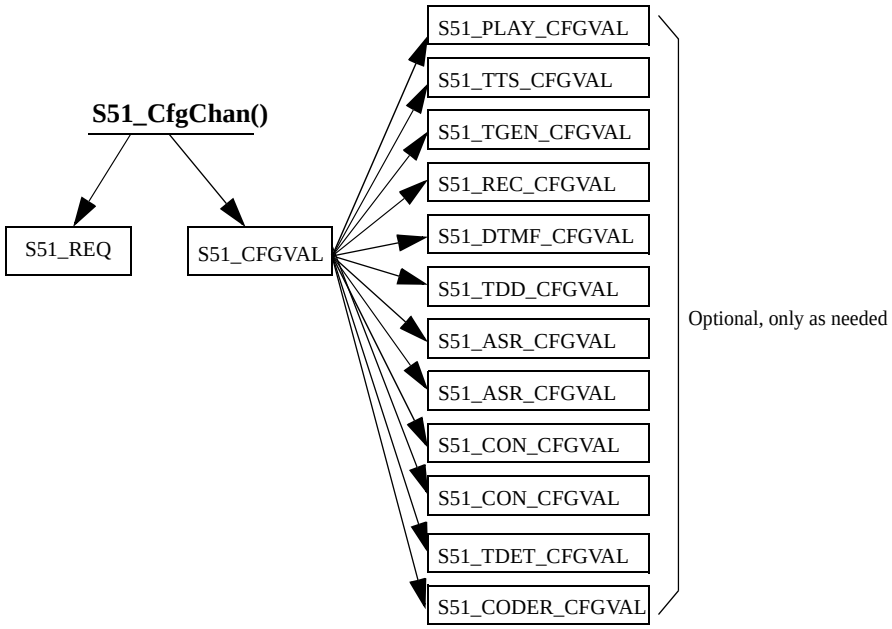
NumUse ....number of usable boards as detected by the driver,

BdStat..... .....current state of the board hardware, firmware, and the board control channel. Refer to the “BCC Channel State” on page 64 for the description of the fields in this structure.

5.5 Configuration Requests

This section defines the high-level data structures and constants used in sending configuration request to the speech processing channel. The configuration requests are issued via the function *S51_CfgChan()*. A single function call is sufficient to set up all configuration parameters for all channel resources. To do this, the function requires a number of related data structures to provide configuration *requests* and configuration *values*.

S51_CfgChan() requires a Channel ID, a Request structure, a Configuration Value structure, and an error string pointer. See “LSPS API Function Reference” on page 147 for an overview.



This **first argument**, the channel ID should be created with the *S51_MakeCid()* macro.

The **second argument** to *S51_CfgChan()* is a pointer to an **S51_REQ data structure**[5] to define which configuration values are being changed by the S51_CFGVAL structure. It is highly recommended that this structure is cleared using the *A5_Zerout()* macro before it is used to insure that all unused fields are clear.

```
typedef struct
{
    u_long       BdReq;       /*board control channel cfg./cmd. request*/
    u_long       PlayReq;     /*play configuration/command request*/
    u_long       TtsReq;      /*text to speech configuration/command request*/
    u_long       TgenReq;     /*tone generator configuration/command request*/
    u_long       TdetReq;     /*call progress tone detector cfg/cmd request*/
    u_long       RecReq;      /*record configuration/command request*/
    u_long       DtmfReq;     /*DTMF configuration/command request*/
    u_long       TddReq;      /*TDD configuration/command request*/
    u_long       Asr1Req;     /*ASR #1 configuration/command request*/
    u_long       Asr2Req;     /*ASR #2 configuration/command request*/
    u_long       Con1Req;     /*Connect #1 configuration/command request*/
    u_long       Con2Req;     /*Connect #2 configuration/command request*/
    u_long       CoderReq;    /*speech trans-coder*/
}
S51_REQ;
```

5. The same structure is used to define commands for the *S51_CmdReq()* function.

Each of the long variables contains a bit-masked value. Some of the more common constants which can be OR-ed into each of these variables are listed below. These common constants are used by different resources (Play, ASR, etc.) and their meaning is described in more detail in each resource section.

S51_FREE_CFGREQ..... ..... deallocate resource

S51_ALLOC_CFGREQ.... ..... allocate resource (usually implicit)[6]

S51_CONNTS_CFGREQ. ..... set the Tid field value (connect to timeslot)

S51_ENCODING_CFGREQ..... set the Encoding field value (type of speech encoding),

S51_MAXDIGITS_CFGREQ .... set the MaxDigits field value (maximum number of digits collected by API),

S51_INITEFLOOR_CFGREQ ... set the InitEfloor field value (initial energy threshold),

S51_ENDEFLOOR_CFGREQ ... set the EndEfloor field value (termination energy threshold),

S51_INITSILT_CFGREQ. ..... set the InitSilTim timer value (initial silence timer),

S51_ENDSILT_CFGREQ. ..... set the EndSilTim timer value (end timer value),

S51_MAXLISTENT_CFGREQ . set the MaxListenTim timer value (maximum listen timer),

S51_EACHDIGIT_CFGREQ..... generate in-progress digit events (when a digit is detected)

S51_STARTDLYT_CFGREQ .... set the StartDelay field value (insert delay before starting the resource),

S51_TONESET_CFGREQ..... set the ToneSet field value (define set of tones to use or detect),

S51_CUSTFORMAT_CFGREQ set the CustFormat field value (activate customized behavior for a resource),

S51_RESMODE_CFGREQ..... set the ResMode field value (set the resource to the given mode of operation),

S51_CBFUN_CFGREQ.... ..... set the CbFun field value for a resource (address of a callback function),

S51_ABORT_CFGREQ.... ..... abort the current configuration request,

There are many other additional configuration requests which are specific to only a single resource. These are described in more detail in the individual resource sections.

The **third argument** to *S51_CfgChan()* is a pointer to the **S51_CFGVAL data structure** which contains the values of the channel configuration parameters. These structures are also described in more detail in the individual resource sections.

```
typedef struct
{
const     int                (*CbFun)();  /*per-channel callback function*/
const     S51_PLAY_CFGVAL*   PlayCfgVal;  /*Play resource configuration values*/
const     S51_TTS_CFGVAL*    TtsCfgVal;   /*TTS resource configuration values*/
const     S51_TGEN_CFGVAL*   TgenCfgVal;  /*Tone gen resource config. values*/
const     S51_TDET_CFGVAL*   TdetCfgVal;  /*CPT resource configuration values*/
const     S51_REC_CFGVAL*    RecCfgVal;   /*Record resource configuration values*/
const     S51_DTMF_CFGVAL*   DtmfCfgVal;  /*DTMF resource configuration values*/
const     S51_TDD_CFGVAL*    TddCfgVal;   /*TDD resource configuration values*/
const     S51_ASR_CFGVAL*    Asr1CfgVal;  /*ASR #1 resource configuration values*/
const     S51_ASR_CFGVAL*    Asr2CfgVal;  /*ASR #2 resource configuration values*/
const     S51_CON_CFGVAL*    Con1CfgVal;  /*Connect #1 resource config values*/
const     S51_CON_CFGVAL*    Con2CfgVal;  /*Connect #2 resource config values*/
const     S51_CODER_CFGVAL*  CoderCfgVal; /*speech transcoder values*/
          u_long             UsrData1;    /*appl data: saved in channel struct*/
          u_long             UsrData2;    /*appl data: saved in channel struct*/
}
S51_CFGVAL;
```

6. The “allocate” request is implicit in any configuration request which is referring to an unallocated resource.

The *UsrData1* and *UsrData2* fields are simply copied to the channel status structure S51_CH and are reserved for the application to be used in any way to store data associated with the channel.

The **last argument** is a pointer to a string pointer. It is set to reference an error string if an error has occurred. Otherwise it points to an empty string. If not interested in an error messages, the application can set this argument to NULL.

Programmers Note: Per-Channel Callback Function in Configuration Request

You may notice that the per-channel callback function *S51_CFGVAL.CbFun* does not really have a place for the corresponding S51_CBFUN_CFGREQ. To set the per-channel callback function *S51_CFGVAL.CbFun*, set this field to the function address. If there is no per-channel callback function, **you must set this field to NULL to disable it!!!**

Programmers Note: Handling of Incomplete *S51_CfgChan()* Function Call:

In blocking mode, if the application gets control while a blocked *S51_CfgChan()* function call is in progress[7] (completion code is *A5_INPROG*), the application must do one of two things:

- repeat the *S51_CfgChan()* call with the same arguments as the initial call, or
- call *S51_CfgChan()* with the *S51_ABORT_CFGREQ* request.

Otherwise any other call to *S51_CfgReq()* or *S51_CmdReq()* for this channel is rejected with "operation already in progress" error code (*A5_INPROG*).

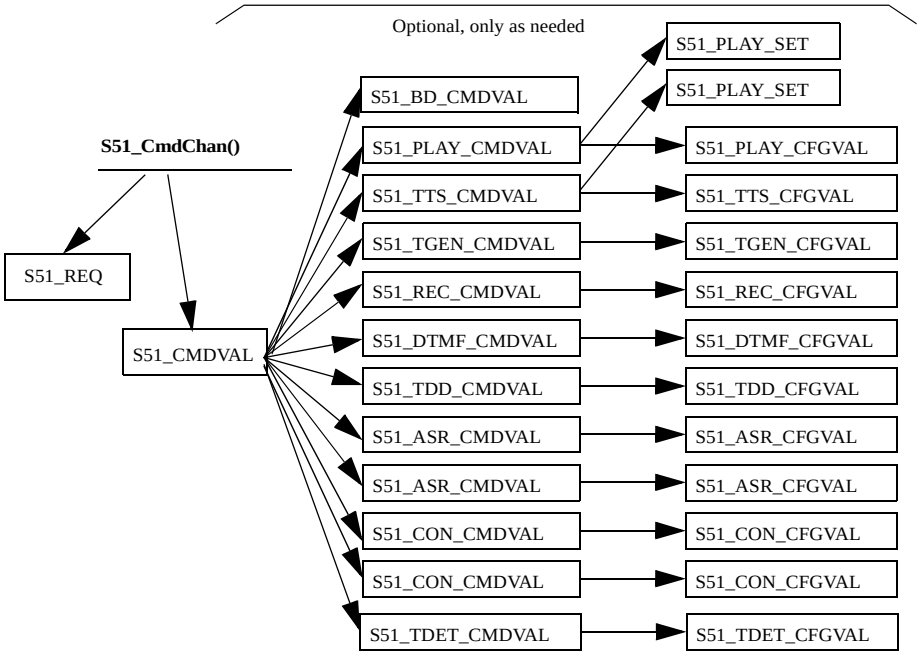
In asynchronous mode, while the *S51_CfgChan()* is in progress (completion code is *A5_INPROG*), the application must do one of two things:

- wait for the *S51_CfgChan()* call to complete, or
- call *S51_CfgChan()* with the *S51_ABORT_CFGREQ* request.

5.6 Command Requests

This section describes the high-level data structures and constants used in sending commands to the speech processing channel. The speech processing commands are issued using *S51_CmdChan()*. A single function call is sufficient to issue commands to all channel resources. To do this, the function requires a number of related data structures to provide command *requests* and command *values*. *S51_CmdChan()* requires a Channel ID, a Request structure, a Command Value structure, and an error string pointer.

7. This happens when the function call is interrupted by a signal.



This **first argument**, the channel ID should be made with the *S51_MakeCid()* macro.

The **second argument** to *S51_CmdChan()* is a pointer to an **S51_REQ data structure** which defines the commands that are being issued. Note that this is the same structure that is used by the *S51_CfgChan() and described previously.* In this case, though, the field values are different. Only one command per resource can be issued at one time. These commands *cannot* be OR-ed together in the same "*Req" variable.

The following commands/actions can be issued to the resources by setting the appropriate ***Req* fields in the S51_REQ data structure described before. Not every action is applicable to each type of resource.

S51_START_CMDREQ.... ..... Start a resource. As soon as the resource is fully started and running, this request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT.

S51_READY_CMDREQ.. ..... Set up to start a resource later, either automatically when triggered by another resource (via RTC) or by a Start command issued later. This does not start the resource, but only makes it ready to go. This request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT.

S51_CANCEL_CMDREQ ..... Gracefully stop a resource discarding all on-board buffered data. The Play or TTS resources stop generating speech immediately, discarding all speech which may have been sent to the board but not yet spoken. The Record resource discards all on-board buffered speech, even some that may have been recorded before the command was issued. As soon as the resource is fully stopped, this request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT. If the resource is not running when this request is issued, the *S51_CmdChan()* completes immediately upon return and no event is generated.

S51_STOP_CMDREQ..... ..... Gracefully stop a resource without discarding on-board buffered data. For Play, this is equivalent to a cancel command. TTS resource may play text which has already been converted to speech coding when the command was issued. Record resource writes out (saves) all the speech that was detected up to the point when the command was issued. As soon as the resource is fully stopped, this request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT. If the resource is not running when this request is issued, the *S51_CmdChan()* completes immediately upon return and no event is generated.

S51_ABORT_CMDREQ.. .....Emergency stop an ongoing command and return immediately.[8] This command always completes immediately on return from the *S51_CmdChan()* function (synchronously), even when the API operates in asynchronous mode[9] and no event is generated. It should be used only to immediately terminate the operation with a disregard for the effects on the resource. Although it guarantees that the API state gets immediately reset, the behavior of the affected resource is undefined, except that the resource will eventually stop within about a second.

S51_PAUSE_CMDREQ... .....Pause play. This is currently used only by the multilingual TTS resource. As soon as the resource is paused, this request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT.

S51_RESUME_CMDREQ.....Resume play after pause. This is currently used only by the multilingual TTS resource. As soon as the resource is resumed, this request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT

S51_SKIPFWD_CMDREQ .....Skip forward and start playing from a new position. The position is given in the command value data structure. This is currently used only by the multilingual TTS resource. As soon as the skip is completed, this request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT.

S51_SETVOLUME_CMDREQ..Set the play volume. This is currently used only by the multilingual TTS resource. As soon as the volume is changed, this request is acknowledged by an event of type S51_INPROG_EVT + S51_CMDREQ_EVT

The **third argument** to the *S51_CmdChan()* function is a pointer to an **S51_CMDVAL data structure**. It defines values that are needed by some of the commands.

```
typedef struct
{
const     S51_BD_CMDVAL*      BdVal;       /*Board Control Channel command values*/
const     S51_PLAY_CMDVAL*    PlayVal;     /*Play command values*/
const     S51_TTS_CMDVAL*     TtsVal;      /*TTS command values*/
const     S51_TGEN_CMDVAL*    TgenVal;     /*Tone Generator command values*/
const     S51_TDET_CMDVAL*    TdetVal;     /*CPT Detector command values*/
const     S51_REC_CMDVAL*     RecVal;      /*Record command values*/
const     S51_DTMF_CMDVAL*    DtmfVal;     /*DTMF command values*/
const     S51_TDD_CMDVAL*     TddVal;      /*TDD command values*/
const     S51_ASR_CMDVAL*     Asr1Val;     /*ASR #1 command values*/
const     S51_ASR_CMDVAL*     Asr2Val;     /*ASR #2 command values*/
const     S51_CON_CMDVAL*     Con1Val;     /*Connect #1 command values*/
const     S51_CON_CMDVAL*     Con2Val;     /*Connect #2 command values*/
const     S51_CODER_CMDVAL*   CoderVal     /*values for transcoder command*/
          u_long              UsrData1;    /*appl data: saved in channel struct*/
          u_long              UsrData2;    /*appl data: saved in channel struct*/
}
S51_CMDVAL;
```

The *UsrData1* and *UsrData2* fields are simply copied to the channel status structure S51_CH and are reserved for the application to be used in any way to store data associated with the channel.

Details of the individual command value structures (*PlayVal*, *RecVal*, etc.) are described later in the corresponding Resource Data Sections.

The **last argument** is a pointer to a string pointer. It will reference an error string if an error has occurred. Otherwise it points to an empty string. If not interested in an error message, the application can set this argument to NULL.

8. This is akin to a pilot (API) ejecting from a plane (the board), not really caring what happens to the plane.
9. This means that there are no completion events delivered to the application.

Programmers Note: Handling of Incomplete *S51_CmdChan()* Function Call:

In blocking mode, if the application gets control while a blocked *S51_CmdChan()* function call is in progress[10] (completion code is A5_INPROG), the application must do one of two things:

- repeat the *S51_CmdChan()* call with the same arguments as the initial call, or
- call *S51_CfgChan()* with *S51_STOP_CMDREQ*, or *S51_CANCEL_CMDREQ* or *S51_ABORT_CMDREQ* request.

Otherwise any other call to *S51_CfgReq()* or *S51_CmdReq()* for this channel is rejected with "operation already in progress" error code (*A5_INPROG*).

In asynchronous mode, while the *S51_CmdChan()* is in progress (completion code is *A5_INPROG*), the application must do one of two things:

- wait for the *S51_CmdChan()* call to complete, or
- call *S51_CmdChan()* with *S51_STOP_CMDREQ*, or *S51_CANCEL_CMDREQ* or *S51_ABORT_CMDREQ* request.

5.7 Run-Time Control (RTC)

In some cases the commands issued by application from the host, and the channel events issued from the board, will not be transmitted and processed fast enough through the host-board interface to implement certain real-time capabilities. One example is the *barge-in* capability, which in general means the ability to immediately stop playing a prompt when the user starts speaking or dialing. Run-Time Control (RTC) provides almost instantaneous control of the Speech Processing Board because processing of the command takes place only on the board and no time-sensitive actions are required from the host or from the application. The RTC is specified by setting the RTC control values in the command value structure. They define the trigger events and the automatic commands that will be issued when the trigger events occur.

Each RTC control is set up to be triggered by a particular event from a single resource and then to send a single command to another resource, after which it is deleted. Each RTC control is *owned* (configured with) by the resource which triggers it. For example an RTC control to start TTS when Play completes is configured with the Play configuration request because it is triggered by a Play event but it controls a TTS resource. The RTC can be used only for control of resources which are within the same channel. When a resource completes on its own, is stopped or is deallocated, all of the RTC controls which are either triggered by it or which control it are automatically deleted.

The API prevents the application from setting up RTC in which the resource would control itself. That is, the RTC where the triggered and controlled resource is the same is rejected. One exception is the RTCs controlling the three resource timers. A resource which has these timers can enable them using it’s own event.

The most common RTC control fields are:

- **AtComplStartRes** field specifies which resources should be started when the current resource completes on its own. It does not apply if the resource is stopped, cancelled, or aborted.
  For example: start DTMF and Record when this resource completes:
  ```
  AtComplStartRes = S51_DTMF_RES|S51_REC_RES;
  ```
- **AtComplStopRes** field specifies which resources should be stopped when the current resource completes on its own. It does not apply if the resource is stopped, cancelled, or aborted. The stopped resource’s termination reason for stopping will contain the name of the resource that stopped it via RTC. For example if a resource was stopped by a DTMF via RTC, its termination reason will be S51_DTMF_TRM.

10. This happens when the function call is interrupted by a signal.

- **AtEnergyStopRes** field specifies which resources should be stopped when the current resource detects input signal energy. The stopped resource's termination reason for stopping will contain the name of the resource that stopped it via RTC.
- **AtDigitStopRes** field specifies which resources should be stopped when the current resource detects the first digit. The stopped resource's termination reason for stopping will contain the name of the resource that stopped it via RTC.
- **AtComplEnableResTimers** field specifies the resources whose three timers (Initial Silence Timer, End Silence Timer, and Maximum Listen Timer (see "Timer Control of Speech Processing Resources" on page 53) should be enabled when the current resource completes on its own. It does not apply if the resource is stopped, cancelled, or aborted.
  For example: enable DTMF and Record timers when this resource completes:
  ```
  AtComplEnableResTimers = S51_DTMF_RES|S51_REC_RES;
  ```
- **AtStopEnableResTimers** field specifies the resources whose three timers (see "Timer Control of Speech Processing Resources" on page 53) should be enabled when the current resource is stopped by an application request or via RTC. It does not apply if the resource completes and stops on its own.

In synchronous (blocking) mode th*e S51_CmdChan()* does not return until all of the resources have completed or have stopped. The application could rely on a callback function and explicitly stop all of the channel resources when, for example, the DTMF Resource detects all of the digits and completes. But, it is much cleaner and easier to use the RTC in the DTMF command value structure:

```
AtComplStopRes = S51_REC_RES|S51_ASR1_RES;
```

In general, when using more than one detection-type of resource (DTMF, ASR, etc.), it is common (and recommended) to use the RTC to stop all the other detection resources on the channel once one of the detection resources completes.

The RTC is also useful for stopping other resources on the channel when it becomes obvious that they are not needed anymore. For example, if the application is detecting DTMF, TDD, and ASR, detection of the first DTMF digit would justify stopping the TDD and ASR resources. In this case the DTMF command value structure would contain:

```
AtDigitStopRes = S51_TDD_RES|S51_ASR1_RES;
```

5.8 Completion of a Request on a Speech Processing Channel

This section describes and discusses the method by which an application program should monitor the progress and the completion of a configuration or a command request on a speech processing channel. There are several variables that the programmer needs to consider: **ComplCode** (completion code), **ErrMsg** (text error message), **ComplRes** (completion resource), **TermReason** (reason for completion or termination), and **ResErrCode** (resource-specific error code).

5.8.1 Relevant Variables

ComplCode is the completion code and it indicates the state of the last issued request for the given channel:

A5_UNDEF ..... indicates no request was ever issued

A5_INPROG ..... indicates a request has started but not yet completed

A5_OK..... ..... indicates a request completed successfully

A5_*_ERR.. ..... indicates a request completed with a failure

The **ComplCode** variable can be found in all of the following structures:

1. In the channel structure S51_CH.ComplCode, where it reflects the overall current channel completion state according to the following rules:

- if at least one resource's ComplCode is currently A5_INPROG, the channel's ComplCode is also A5_INPROG,
- if all resources in the channel completed successfully (A5_OK), the channel's ComplCode is also A5_OK,
- if all resources in the channel completed, but at least one resource completed with a failure, the channel's ComplCode is the same as the first failed resource (A5_*_ERR).

2. In each of the resource state structures S51_*_STAT.ComplCode, where it reflects the current completion state of the single resource.
3. In each event structure *S51_EVT.ComplCode*, where it reflects the completion state of the resource associated with the event at the time when the event occurred. The implications are, for example, that this completion code may be older (and thus different) than the current/latest resource completion state in *S51_*_STAT.ComplCode*.

ErrMsg is a pointer to a text message associated with the completion code, and it describes details of a failure completion of a resource. It is meant for human consumption to help locate problems in the code or in hardware. In a case of a non-failure completion code, this points to a null string. It is never a null pointer. Because there is a close one-to-one relation between the ErrMsg and ComplCode variables (they always come as a pair), the rules for ErrMsg are the same as for the ComplCode.

ComplRes is a variable which indicates the last resource which stopped "naturally", that is, completed on its own without being stopped by another resource or by an expiring timer or by an explicit stop command. Usually it is used to indicate which of the "recognition resources" detected a complete valid input. For example, if DTMF, TDD, and ASR Resources are started on the channel and each is configured to stop the others at completion via RTC, on completion this variable will show the resource which recognized the input and not the other ones that were stopped via RTC. The ComplRes can be found only in the channel structure *S51_CH.ComplRes*.

TermReason is variable indicates the reason why a resource decided to complete or stop the requested operation:

S51_UNDEF_TRM .....Request is in progress, or no request was ever issued.

51_ERROR_TRM.. .....Request completed with an error, consult ComplCode and ErrMsg for detailed error explanations.

51_DONE_TRM.... .....The request completed "naturally", that is, it was not stopped by a timer or some other exception.

51_*_TRM ....One of a multitude special conditions that could cause the request to complete. These include timer expiration, stop request (by the application or by another resource), reaching some boundary condition and many others.

The TermReason variable is closely associated with the ComplCode and can be found in the following structures:

- In each of the resource state structures S51_*_STAT.TermReason, where it reflects the current termination reason of the single resource.
- In each completion event structure S51_EVT.TermReason, where it reflects the termination reason of the resource associated with the event at the time when the termination event occurred.

ResErrCode is a variable which contains an internal error code provided by the resource firmware for certain types of error completions. It is not set by all resources and currently these code values are not documented and are used mostly for debugging the resource firmware. Some of the resources use the header file *s51err.h* to create these error codes.

When an **error** occurs, the application can try to **recover** by first stopping all the resources on the channel and then by deallocating and reallocating the resources on the channel. This in effect restarts the software (program) on the board which implements the failed resource. Although in many cases simply reissuing the request (without reallocating the resource) may work, it is not recommended as a reliable way of error recovery.

5.8.2 API Operation

When a request is issued via *S51_CfgReq()* or *S51_CmdReq()* function and it immediately fails, the function returns a NULL pointer instead of the pointer to the channel structure S51_CH. In this case the application should normally check the *S51_Errno* global variable and the *ErrMsg* pointer set by the function call. If the application saved the channel pointer from a previous successful request to this channel, the application can also refer to the *S51_CH.ComplCode* and *S51_CH.ErrMsg* variables to get the same completion information. In this case the completion codes for individual resources are not changed.

When in **non-blocking asynchronous** mode (A5_ASYNC_MOD), the various completion codes are continuously being updated to reflect the current situation:

1. For each individual resource, the *ComplCode*, *ErrMsg*, *TermReason* and *ResErrCode* show the current completion state of the resource.
2. For the entire channel, the *S51_CH.ComplCode* and *S51_CH.ErrMsg* contain copies of the values for one of the resources on the channel:
   - If there is a resource which have not yet completed operation (in-progress), the channel completion code is also “in-progress”.
   - If all the resources completed but at least one completed with a failure, the channel completion code will reflect the completion code of the first failed resource.
   - If all resources completed successfully, the channel completion code will also indicate a successful completion.

When in **blocking synchronous** operation (A5_BLOCK), the application will normally get to see the completion codes only until all the resources complete and at that time the channel completion code also indicates completion (successful or failed). But even in this mode the application can use callback functions to “peek” at the channel state while the request operation is in progress.

In **summary**, in most cases the application should only need to check the completion indicators in the channel state structure *S51_CH* and can ignore all others. Only in special cases (application dependent) the application may need to refer to the completion indicators in the resource state structures *S51_*_STAT* or in the event structure *S51_EVT*. In other words, your application should normally be checking only the completion indicators in the channel structure and should refer to the other indicator only if it is absolutely necessary for your implementation.

For **example**, the simplest way to wait for completion in non-blocking mode would be:

```
while (Ch->ComplCode == A5_INPROG)
   (void)S51_EventChan(-1, NULL, &ErrMsg);        /*ignore all events*/
```

There seems to be a tendency among programmers to check every available code and event, possibly turning the above one line of code into a page (or more) of code with equivalent functionality. This adds unnecessary complexity to the code and it is usually a waste of programmer’s time and a waste of system processing resources.

5.9 Timer Control of Speech Processing Resources

There are three general timers used by almost every Speech Processing Resource.[11] Each allocated resource has it’s own set of timers. They only run after the resource has started and are stopped automatically when the resource stops. They have reasonable default values. Their values may also be changed by the application because they are part of the configuration.

11. Some resources (such as ASR) use additional timers. Except for their time-expiration values, these additional timers cannot be controlled by the application. They are described in the appropriate resource sections.

- **Initial Silence Timer** (IST) starts automatically when the resource starts and stops automatically when the resource detects input signal energy. Its expiration typically indicates that there was no signal detected at all for too long of an interval. The expiration automatically stops the resource with the termination reason of S51_INITIM_TRM.
- **End Silence Timer** (EST) re-starts automatically every time the resource stops detecting signal energy. Its expiration typically indicates that the input signal has ceased for too long of an interval. The expiration automatically stops the resource with the termination reason of S51_ENDTIM_TRM.
- **Maximum Listen Timer** (MLT) starts automatically when the resource detects a signal or energy and stops automatically only when the resource stops. Its expiration typically indicates that the input signal was being detected for too long. The expiration automatically stops the resource with the termination reason of S51_LISTENTIM_TRM.

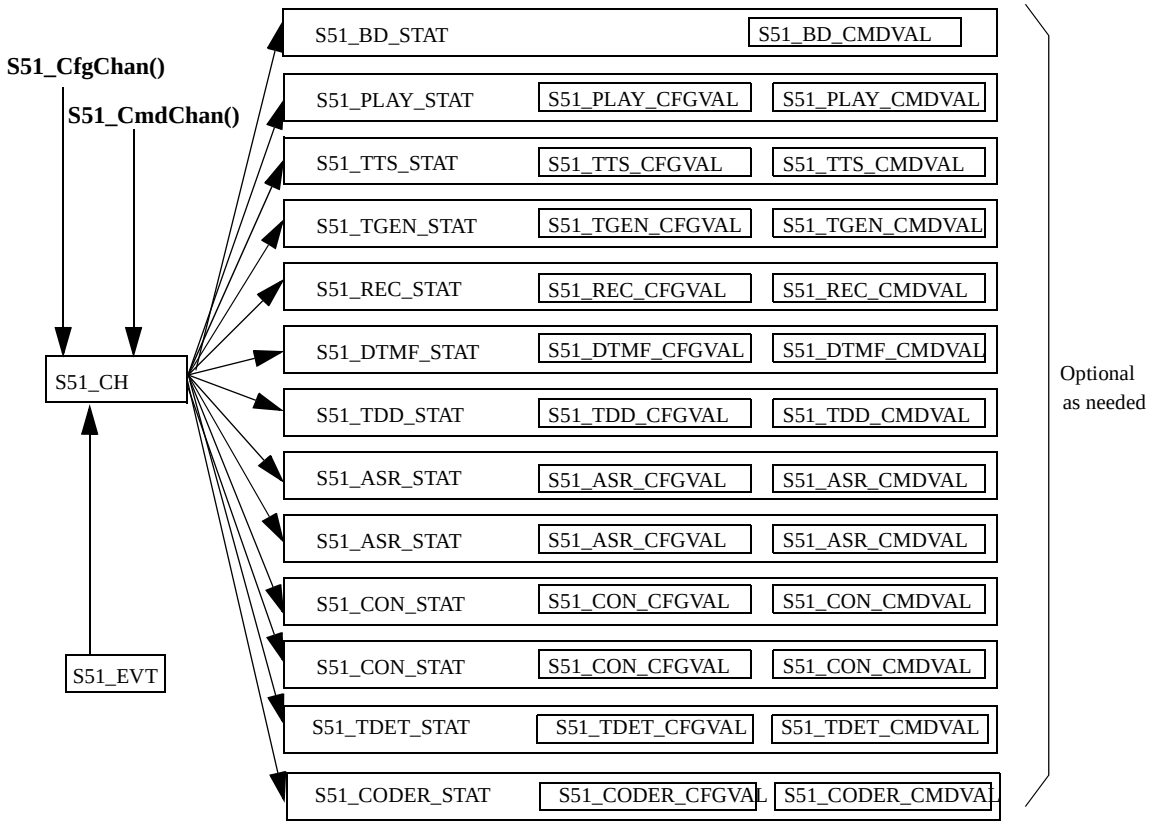
These three timers can be disabled during the current command by setting the *InhTimers* field in the command value structure to a non-zero value before calling *S51_CmdChan()*. Once they have been disabled, they can be enabled either via Run-Time Control (RTC, described later) or by reissuing the command by calling *S51_CmdChan()* function again with the same arguments except with the *InhTimers* field set to 0.

While the timers are inhibited, they will not start. Once the timers are enabled they will function normally. If a timer would have started except for the fact that it was inhibited, it will start as soon as the timers are enabled.

For **example,** if the application first plays a prompt and then listens for DTMF digits, it may not want the Initial Silence Timer to start until after the prompt has completed. This is accomplished by inhibiting the three timers in the initial command and then enabling them via RTC when the Play Resource stops. So, when the DTMF Resource starts, its Initial Silence Timer will be inhibited (normally it would have already started). The Initial Silence timer will then start as soon as it is enabled at the completion of the Play command.

5.10 Speech Processing Channel Data

The **S51_CH data structure** contains all of the information that is kept about each speech processing channel. It contains the *current state* and *configuration* of the channel. It does not provide information about the events (state changes) on the channel. Among other things it contains the cumulative set of configuration values set by the application and the last command issued by the application.



The pointer to this structure is returned from each *S51_CfgChan()* and *S51_CmdChan()* function call. The S51_CH* channel pointer for a particular channel does not change between the API calls until the channel is closed, so the application has the option of saving the pointer. The pointers to other structures stored inside of the channel structure may change between API function calls, so the application should not be saving them.

```
typedef struct
{
          S51_CID             Cid;         /*Channel ID*/
          u_char              Bd;          /*decomposed Cid*/
          u_char              Port;        /*decomposed Cid*/
          u_long              ComplCode;   /*completion code for last request*/
          u_short             ComplRes;    /*resource triggered the completion*/
const     char*               ErrMsg;      /*error message for the last request*/
const     S51_BD_STAT*        BdStat;      /*board control channel status*/
const     S51_PLAY_STAT*      PlayStat;    /*play resource status*/
const     S51_TTS_STAT*       TtsStat;     /*TTS resource status*/
const     S51_TGEN_STAT*      TgenStat;    /*tone generator status*/
const     S51_TDET_STAT*      TdetStat;    /*tone detector status*/
const     S51_REC_STAT*       RecStat;     /*record resource status*/
const     S51_DTMF_STAT*      DtmfStat;    /*DTMF resource status*/
const     S51_TDD_STAT*       TddStat;     /*TDD resource status*/
const     S51_ASR_STAT*       Asr1Stat;    /*ASR #1 resource status*/
const     S51_ASR_STAT*       Asr2Stat;    /*ASR #2 resource status*/
const     S51_CON_STAT*       Con1Stat;    /*Connect #1 resource status*/
const     S51_CON_STAT*       Con2Stat;    /*Connect #2 resource status*/
const     S51_CODER_STAT*     CoderStat;   /*speech transcoder status*/
const     u_long              UsrData1;    /*appl data: saved from request*/
const     u_long              UsrData2;    /*appl data: saved from request*/
}
S51_CH;
```

The internal pointers to the status structures for resources not allocated on this channel are set to NULL. The descriptions of the individual resource status structures (*PlayStat*, *RecStat*, etc.) are in the corresponding Resource Data Sections later. The other fields are:

Cid ....Channel ID,

Bd, Port ....Cid decomposed into a board (1 to S51_BD_MAX) and channel number (1 to MaxChan),

ComplCode . .....Completion code for the current or last request (A5_OK or A5_*_ERR, see file *a5api.h*),

ComplRes.... .....The resource that triggered the completion of the last command by first completing on its own and then stopping all the other resources.

ErrMsg ....Text message for the last request (if failed).

UsrData1 and UsrData2 ....contain values copied from the last configuration or command request issued on this channel. This can be used by the application to store arbitrary data associated with the channel.

5.11 Speech Processing Event Data

The **S51_EVT data structure** contains a description of a single speech processing event. It is either returned from the *S51_EventChan()* function or passed to the application's callback function. The structure does not contain data describing the current channel status, for this purpose it contains a pointer to the corresponding channel structure. Note that the channel status pointer points to the channel status structure which describes the current (latest and most up-to-date) channel status and not the channel status as it was when the event occurred. This is a subtle but sometimes an important difference.

```
typedef struct
{
          S51_CID      Cid;         /*event on which channel*/
const     S51_CH*      Ch;          /*channel and its current state*/
          u_long       EventType;   /*see s51.h*/
          u_short      ResType;     /*event from which resource*/
          u_long       ComplCode;   /*generic error code or A5_OK*/
          u_long       TermReason;  /*success completion code or S51_ERROR_TRM*
          u_long       ResErrCode;  /*detail error code, resource dependent*/
const     char*        ErrMsg;      /*completion err msg*/

                                    /*S51_DIGIT_EVT only*/
          float        ThisDigitTime;/*relative timestamp (sec since start)*/
          char         ThisDigit[S51_INDIGS_SIZ+1];/*last digit(s) received*/

                                    /*S51_BOUNDARY_EVT only (ML-TTS)*/
          u_char       BoundaryType;/*S51_WORD_BND, S51_MINOR_BND, S51_MAJOR_BND*
          u_char       BoundaryListSize;/*actual size*/
          u_long       BoundaryList[S51_BOUNDLIST_MAX];

}
S51_EVT;
```

The fields are:

Cid ....Channel ID

Ch ....pointer to the channel structure.

Event Type .. ..constants *S51_*_EVT* (in *s51.h*) in bitmask format,

ResType..... .....resource that generated the event (*S51_*_RES* in *s51.h*),

ComplCode . .....completion code for this event (A5_OK or A5_*_ERR in *a5api.h*),

TermReason. .....reason for termination (*S51_*_TRM* in *file s51.h*),

ResErrCode . .....more detailed error code provided by the resource, usually resource-specific (in s51err.h),

ErrMsg ....text message for error completion,

ThisDigit..... ..... this is used only in the DTMF/MF/TDD digit detection when the application uses an API mode that delivers digits as they are detected (streaming mode, burst mode, each-digit mode). It contains a group of digits (zero, one or more) that were detected at one time.

ThisDigitTime..... contains a timestamp associated with the *ThisDigit* field, marking the precise time when the last of the digits in the group was detected. The time is expressed in seconds (with millisecond precision) since the resource started. It is used only for DTMF/MF/TDD digit detection in streaming mode.

BoundaryType..... This is used only for multi-lingual TTS to report when the TTS reached a certain boundary (or group of boundaries) in the input text. This field contains the type of boundary being reported (word, minor phrase, major phrase).

BoundaryListSize .. This is used only for multi-lingual TTS to inform the application about the number of boundaries being reported in this event. It is the size of the BoundaryList array.

BoundaryList ..... This is used only for multi-lingual TTS to inform the application about the boundary (or last in a group of boundaries) which were just reached by the TTS. The boundary events are reported in units of bytes measured from the beginning of the complete text given to the TTS to speak in this command request.

The **Event Types** (S51_EVT.EventType) are made up of Event Class and Event Types, bitwise OR-ed together. That means that the application must test this variable by individually testing the bits using the bitwise AND operator.

The **Event Class** characterizes the event using one of three broad categories:

S51_COMPL_EVT ..... this is a final completion event,

S51_UNSOL_EVT. ..... unsolicited event,

S51_INPROG_EVT ..... in-progress event,

The **Event Type** describes the reason why the event was generated:

S51_CFGREQ_EVT ..... response to a configuration request,

S51_CMDREQ_EVT ..... response to a command request,

S51_BDSTATE_EVT ..... board state changed,

S51_INIT_EVT..... ..... initialization completed,

S51_RESLOST_EVT ..... resource spontaneously deallocated,

S51_CHRESET_EVT ..... channel spontaneously reset,

S51_CMDSTART_EVT ... command started ok,

S51_RECG1ST_EVT ..... first utterance/digit recognized,

S51_ENERGY_EVT ..... energy detected,

S51_DIGIT_EVT... ..... digit detected,

S51_TONE_EVT.... ..... tone detected,

S51_DPULSE_EVT ..... dial pulse detected,

S51_INDEX_EVT.. ..... TTS index marker spoken,

S51_BDTEMP_EVT ..... board temperature changed,

S51_BOUNDARY_EVT .. speaking boundary reached (multi-lingual TTS only),

S51_LED_EVT..... ..... LED lights on the board changed (6UB5 board only),

S51_HCLK_EVT .. .. source of H.100 bus clock (A/B) has changed.

The **Termination Reason** (S51_EVT.TermReason) is associated with a completion event (a requested operation completed). It gives a more detailed information about how and/or why the operation completed:

S51_DONE_TRM.. ..... completed as expected, no errors,

S51_ERROR_TRM ..... error termination (see ComplCode),

S51_STOP_TRM... ..... explicit stop was issued,

S51_ABORT_TRM ..... explicit/implicit abort was issued,

S51_CANCEL_TRM.....explicit cancel issued,
S51_INITIM_TRM. .....Initial Silence Timer expired,
S51_ENDTIM_TRM .....End Silence Timer expired,
S51_LISTENTIM_TRM...Maximum Listen Timer expired,
S51_UTTERTIM_TRM....Maximum Utterance Timer expired,
S51_MAXDIG_TRM .....MaxDigits reached,
S51_TERMDIG_TRM.....TermDigit recognized,
S51_TERMTONE_TRM ..Termination tone recognized,
S51_COLLISION_TRM...subword training collision,
S51_IDENTICAL_TRM ..subword training collision, identical,
S51_DIALPULSE_TRM ..DTMF dial pulse detected,
S51_OVERFLOW_TRM..application data buffer overflow,
S51_2SHORT_TRM.....ASR utterance is too short,
S51_2LONG_TRM .....ASR utterance is too long,
S51_PLAY_TRM... .....stopped by Play,
S51_TTS_TRM..... .....stopped by Text to Speech,
S51_TGEN_TRM... .....stopped by Tone Generator,
S51_TDET_TRM... .....stopped by Tone Detector,
S51_REC_TRM..... .....stopped by Record,
S51_DTMF_TRM.. .....stopped by DTMF Detector,
S51_TDD_TRM..... .....stopped by TDD Detector,
S51_ASR1_TRM.... .....stopped by ASR #1,
S51_ASR2_TRM.... .....stopped by ASR #2.

5.12 Speech Processing Board Control Channel (BCC)

The Speech Processing Board Control Channel (BCC) is not a standard speech processing channel. It is a special virtual channel which is used to get detailed board status information and to send commands to the board which affect the entire board. For example a command to reboot the board. It cannot allocate any resources and all configuration and command requests are applied to the board hardware/firmware which are represented by this BCC.

To control the speech boards via the BCC, the API for most part does not interact with the boards directly, but interacts with the speech board boot process (daemon) *lspsspboot*. There is one *lspsspboot* running for each equipped speech board. These programs are started automatically by *lspsspstartup*, which is started automatically by the system when the host machine is booted. The lspsspboot programs are the ones which directly interact with the speech boards. In other words, the BCC does not work unless the *lspsspstartup* is running.

To create a channel identifier (CID) for such a channel use channel 0 (zero) in the standard *S51_MakeCid()* macro. For example, the CID for the BCC of board 3 is:[12]

```
S51_MakeCid(3, 0);
```

Any number of application processes may have the same BCC allocated at the same time. By allocating the appropriate BCC, the application can:

- Get the **firmware state** of the board - whether the firmware on the board is ready to handle calls.
- Gets **other board information**, (for example, the board temperature, which firmware is loaded, etc.).
- Gets **board state-change events** S51_BDSTATE_EVT and S51_BDTEMP_EVT notifying the application about the board and firmware state changes.
- Gets spontaneous **channel reset events** S51_CHRESET_EVT, these usually occur when the board goes out of service or is rebooted and they may be used to notify the application that a particular channel has lost all previously allocated resources and all of its configuration values were reset to the default firmware values. Such spontaneously reset channel stays allocated with no resources. The application must reallocate all resources after the board is again in service.
- Set the **H.110 clock source** and clock failure fallback strategy,
- Set the **board LEDs** (6UB5 only),
- Set the board into **detached** state to get it ready for **hot swap** (6UB5 only),
- Send commands to the board to **reboot** it or to take it in or out of service.

5.12.1 Control of 6UB5 board LEDs

At any time that the application has the BCC channel open, it can control three of the four LEDs on the board. The speech board boot daemon (*lspsspboot*) also has a capability to automatically maintain/change the LEDs based on the current Board State. Whether the automatic setting/change of LEDs by the *lspsspboot* boot process is in effect is determined in two ways:

1. The initial setting of the LEDs in one of the initialization files.
2. Whether the he API enabled or disabled the automatic setting of the LEDs by the boot process.

When automatically controlled by the boot process, the meaning of the individual LEDs is “hard-coded” in the boot process as follows:

- Red LED: out of service, no firmware loaded,

12. Note that the API numbers the ports starting from 1, therefore port 0 does not correspond to a real port.

- Yellow LED: board is booting,
- Green LED: board is in-service.

If the automatic control is initially disabled, the board's LEDs are set with the Red LED on and other LEDs off. As long as the automatic control is disabled, it is the application's responsibility to maintain the states of the LEDs. Even if the automatic control of the LEDs is enabled, the application still has a full control of the LEDs via the API. If the LEDs change either automatically or because of application request, the BCC receives a an LED change event.

The above paragraph applies to the three LEDs: Red, Green and Yellow. The application can not directly control the Blue LED. This LED is always controlled by the boot Process and it reflects the current "Hot Swap" state of the board. So when the application changes the board state to "detached", the blue LED is set to on and all other LEDs to off.

5.12.2 Board Hot Swap

The 6UB5 board provides a limited form of a hot swap capability. Hot swapping refers to the capability of removing and reinserting the board while under power. The following sequence must be followed when hot-swapping the 6UB5 board:

- change the board state to "detached"; the blue LED is illuminated in this state,
- unplug the board,
- plug in the 6UB5 board (in the same slot),
- change the board state to "in-service"; this attaches the board and reboots it.

The board state can be changed to **detached** and later to **in-service** in one of two ways:

- via the API by sending the command S51_DETACH_CMDREQ or S51_INS_CMDREQ respectively to the BCC,
- via the SDK command **bdcmd** or a user-written command.

Note that the new board must be plugged in the same PCI slot from which the previous board was removed. The hot swap capability does not support plugging in a board on a previously unused PCI slot.

5.12.3 Control of the H.110 Bus Clock

The board control channel (BCC) keeps track the current H.100 bus clock source being used, so the application can determine which clock is being used by simply examining the BCC status structure. The BCC configuration parameters enable the application to control some properties of the H.110 clock sourcing:

1. Select primary H.110 clock source A or B (default is A),
2. Control automatic clock fallback (on or off, default is on).

If the clock source changes either automatically via fallback or because of application request, the BCC receives a clock source change event.

While the board (firmware) state is not in service (A5_BD_INS), the clock state is undefined (S51_UNK_HCLK).

5.12.4 6UB5 Board CPU Performance Measurements

An application can obtain performance measurements for the CPUs on 6UB5 boards by issuing a S51_GETPERF_CMDREQ command to the BCC. Such command immediately (on return from *S51_CmdReq()*) fills in the array of performance measurement structures in the BCC state data structure S51_BD_STAT.

Each S51_BDCPU_PERF data structure in the S51_BD_STAT.CpuPerf array contains a snapshot of measurements from one CPU as follows:

Index	CPU
0	IO processor
1	CPU A
2	CPU B
3	CPU C

The S51_BDCPU_PERF data structure contains the following fields:

```
typedef struct
{
        u_long          TotalCpuTicks;
        u_long          IdleCpuTicks;
        u_long          TotalMemKb;
        u_long          IdleMemKb;
}
S51_BDCPU_PERF;
```

The fields contain the following values:

TotalCpuTicks..... this field is incremented with each "CPU tick". The duration of the CPU tick is not officially defined (and may change with the board hardware version), so only ratio calculation (percent busy) using *TotalCpuTicks* and *IdleCpuTicks* make sense. Also, this counter is never reset, so it occasionally overflows (every few minutes), so the application must be able to deal correctly with the overflow situation.

IdleCpuTicks..... this field is incremented with each "CPU tick" when the CPU is found to be "idle". It has the same characteristics as the *TotalCpuTicks* counter (undefined tick period and periodic overflow).

TotalMemBk ..... total memory usable by the CPU.

IdleMemKb. ..... unused memory at the time the snapshot was taken.

5.12.5 BCC Configuration

The first argument to the *S51_CfgChan()* function is a pointer to a S51_REQ structure. To change the default configuration parameters for the BCC, the following constants can be OR-ed into the *S51_REQ.BdReq* field to indicate which configuration parameters are to be changed

Request Bit Fields	S51_BD_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate BCC
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_H110CLK_CFGREQ	H110ClkPrimary H110ClkFallbkEnable	designates primary H.110 clock enables/disable clock failure fallback

The **S51_BD_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan()* and it defines the configuration values for the BCC.

```
typedef struct
{
   u_char       H110ClkPrimary;     /*designate primary: S51_A_HCLK, S51_B_HCLK*/
   u_char       H110ClkFallbkEnable;/*enable clock fallback: 0==disabled*/
}
S51_BD_CFGVAL;
```

The fields are:

H110ClkPrimary..... .....designates and sets the source of the primary H.110 clock (6UB5 board model only):

- S51_A_HCLK use A-clock as the primary H.110 clock,
- S51_B_HCLK use B-clock as the primary H.110 clock

H110ClkFallbkEnable.....if set to 0, it disables automatic switching from primary to secondary clock when the current clock fails. Otherwise it enables the automatic clock source switching.

5.12.6 BCC Commands

A board command is issued by calling *S51_CmdChan()* function and is specified by setting S51_REQ.BdReq to one of the following values. All of these requests (with the exception of S51_PING_CMDREQ) are sent to the board boot process which is running on the host (they are not sent directly to the board). This process is responsible for down-loading to the board, changing the board states and for monitoring its state and sanity. Generally these commands “complete” as soon as an acknowledgment is received from the boot process. So, for example, even though a boot request “completes” right away (within a second), the actual board booting of the board completes much later. At that later time the application receives an unsolicited board-state-change event. This occurs in both asynchronous and synchronous modes.

Once issued, a command cannot be stopped, only aborted. Such S51_ABORT_CMDREQ does not affect the completion of the requested (ongoing) operation (the operation will complete), but only forces a reset of the API state. This is meant to get the API out of any "hanging" condition.

S51_BOOT_CMDREQ..... ..... Sends a request to the boot process to reboot the board. The command value must include the desired board type. The completion of this request (completion event and ComplCode set to A5_OK) means that the board reboot requests has been received and started. The board firmware state (BdFwState) will first change to S51_BD_BOOT and after the boot succeeded the state will change to S51_BD_INS. If the boot fails, the board will either stay in S51_BD_BOOT state or will change state to S51_BD_OOS. The application must monitor the unsolicited board state events to determine when the boot completes and the board is in service.

S51_OOS_CMDREQ..... ..... Sends a request to the boot process to stop the firmware on the board and change the board firmware state to S51_BD_OOS. Once the state is changed, the application receives an unsolicited board state event.

S51_INS_CMDREQ..... ..... Sends a request to the boot process to reboot the firmware on the board. If the board is currently in the S51_BD_INS state, the request completes normally, but the board state does not change. If the board is in any other state, the request has the effect of S51_BOOT_CMDREQ. Once the state is changed, the application receives an unsolicited board state event.

S51_DETACH_CMDREQ ..... Send a request to change the board state to "detached" (A5_BD_DETACH) to make it ready for hot-swap (6UB5 board only). This works similarly to the S51_OOS_CMDREQ in that once the state is changed, the application receives an unsolicited board state event. To return the board back to S51_BD_INS state, issue a S51_BOOT_CMDREQ or S51_INS_CMDREQ.

S51_LED_CMDREQ..... ..... Send a request to change the on/off state of the LEDs on the board. Once the LEDs are changed, the application receives an unsolicited LED state change event (S51_LED_EVT). The S51_LED_CMDREQ always requires (reads) the following fields: LedAutoControl, LedRed, LedYellow, LedGreen. All other commands ignore all of the above-mentioned fields. Note that you cannot control the blue LED, it is always an indicator of a "detached" state.

S51_PING_CMDREQ..... ..... Send an “alive inquiry” message to the board firmware. This request in not meant to be used as a standard method of getting the board state. For such a purpose, the application automatically gets an event for every board state change. The board state in the channel structure is also continuously updated to reflect the board state. The S51_PING_CMDREQ command provides an application with a way to send a quick message to the board to make sure that the board is not dead. This command can succeed only if the board's firmware is running and is responding to requests. During other states (for example during “booting” of the board), this command fails with a timeout event

(A5_TIMEOUT_ERR). On success, it generates S51_BDSTATE_EVT + S51_CMDREQ_EVT event. If you want a reliable current board state, look in the board control channel structure, or use the standard board state events. If you do not trust the board's state provided by the interface and want to "ping" the board yourself, use S51_PING_CMDREQ and be ready to handle a possible request timeout (A5_TIMEOUT_ERR) error.

S51_GETPERF_CMDREQ.....This command is applicable only to the 6UB5 and 6UB6 speech boards. It causes the API to fill the board CPU performance data structures S51_BD_STAT.CpuPerf. This command always completes immediately on return from the *S51_CmdChan()* function.

The **S51_BD_CMDVAL data structure** is part of an argument to *S51_CmdChan()* and is used to send commands to the board.

```
typedef struct
{
   u_char       Ucl;               /*if !=0 override any user locks*/
   u_char       LedAutoControl;    /*Autocontrol by ayc51boot process: 1==on*/
   u_char       LedRed;            /*Red LED state: 1==on*/
   u_char       LedGreen;          /*Green LED state: 1==on*/
   u_char       LedYellow;         /*Yellow LED state: 1==on*/
   char         Reason[S51_REASON_SIZ+1];/*why (optional)*/
   char         BdFwType[S51_BDFWTYPE_SIZ+1];/*bd boot config (optional)*/
}
S51_BD_CMDVAL;
```

The fields are described below.

Ucl..... ..... A board can be "locked" by a user[13] to prevent other users from rebooting it or changing its state. If this field is not zero, a board boot or state change command will override any current board locks by any user. This field is applicable to S51_BOOT_CMDREQ, S51_OOS_CMDREQ, S51_DETACH_CMDREQ.

LedAutoControl..... This switch turns on (if set to 1) or off (if set to 0) the automatic control of the board LEDs by the boot process. It is required for the S51_LED_CMDREQ. Applicable only to 6UB5 board.

LedRed..... ..... This switch turns on (if set to 1) or off (if set to 0) the red LED on the board. It is required for the S51_LED_CMDREQ command, otherwise it is ignored. Applicable only to 6UB5 board.

LedGreen .... This switch turns on (if set to 1) or off (if set to 0) the green LED on the board. It is required for the S51_LED_CMDREQ command, otherwise it is ignored. Applicable only to 6UB5 board.

LedYellow... ..... This switch turns on (if set to 1) or off (if set to 0) the yellow LED on the board. It is required for the S51_LED_CMDREQ command, otherwise it is ignored. Applicable only to 6UB5 board.

Reason..... ..... An optional arbitrary text string that identifies the reason for issuing a command to reboot the board or to change the board state. If not specified, it defaults to the string "LSPS-API". This field is applicable to S51_BOOT_CMDREQ, S51_OOS_CMDREQ, S51_DETACH_CMDREQ.

BdFwType... ..... This value is required for board reboot and in-service commands. This field is applicable only to S51_BOOT_CMDREQ, S51_OOS_CMDREQ, S51_DETACH_CMDREQ. It identifies which firmware package to load into the Speech Processing Board and it must match one of the firmware packages installed on the host. The new type stays in effect until explicitly changed or until the host is rebooted, at which time the board type reverts back to the default type. The current board type reported by the board is in the field *S51_BD_STAT.BdFwType*.

13. When a user logs into a board using the *debug51* command, the board is "locked" using the host OS user ID. The board can also be locked by using the *lock51* command.

5.12.7 BCC Channel State

The **S51_BD_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of one Speech Processing Board. Note that there are two groups of data fields. The first group is updated only by a call to *S51_Init()* function and it includes fields from *BdHwModel* through *BdHwModelDescr*. The fields in this group do not change after the host is rebooted (or after the first *S51_Init()* call). The second group contains all the other fields, these are updated continuously as long as the application has the BCC open. If the BCC is not open, the fields in the second group are not set nor updated. Also note that fields starting with *LedAutoControl* are applicable to 6UB5 board model only.

```
typedef struct                /*the following fields are set by S51_Init() call*/
{
                              /*board state info*/
   u_short      BdHwModel;    /*A5_6UB5_BD, etc. (see a5api.h)*/
   u_short      BdHwState;    /*board hardware state, A5_BD_* in a5api.h*/
   u_short      BdHwVersion;  /*board hardware model, internal*/
   u_short      BdHwRev;      /*board hardware revision, numeric*/
   u_short      MaxChan;      /*max number of channels*/
                              /*CPU bus info*/
   u_short      CpuBusType;   /*A5_PCI_BUS, etc. (above)*/
   u_short      PciBusNumber; /*on which PCI bus*/
   u_short      PciSlotNumber;/*PCI slot number*/
   char         BdHwModelDescr[S51_BDMODEL_SIZ+1];/*model description*/
   char         BdHwSerialNum[S51_SERNUM_SIZ+1]; /*serial number*/

                              /*the following fields are only set & kept up*/
                              /*after the board control channel is open*/
   u_long       CfgReq;       /*only S51_ALLOC_CFGREQ*/
   S51_BD_CFGVAL CfgVal;      /*current config values*/
   u_long       CmdReq;       /*last command completed ok*/
   S51_BD_CMDVAL CmdVal;      /*values for the last command*/
   u_long       ComplCode;    /*result of last FW state query*/
   u_long       TermReason;   /*S51_ERROR_TRM or S51_DONE_TRM*/
   u_long       OosReason;    /*why last OOS: S51_REASONOOS_* in s51err.h*/
   char         ReasonMsg[S51_REASON_SIZ+1];/*why state changed: string*/
const char*     ErrMsg;       /*error message for last request*/
                              /*board hardware/firmware info*/
   u_short      BdFwState;    /*board firmware/software state A5_BD_* in a5api.h*/
   u_short      BdTemp;       /*current temperature state, S51_*_TEMP above*/
   u_long       BdFwVersion;  /*RTE version (internal)*/
   char         BdFwType[S51_BDFWTYPE_SIZ+1];/*board boot type (which software)*/
                              /*LED info*/
   u_char       LedAutoControl;/*Autocontrolled by ayc51boot process: 0==off*/
   u_char       LedRed;       /*Red LED state: 0==off*/
   u_char       LedGreen;     /*Green LED state: 0==off*/
   u_char       LedYellow;    /*Yellow LED state: 0==off*/
   u_char       LedBlue;      /*Blue LED state: 0==off*/
                              /*H.100 bus clock source info*/
   u_char       H110ClkPrimary;/*designated: S51_*_HCLK*/
   u_char       H110ClkUsed;  /*actual: S51_*_HCLK*/
   u_char       H110ClkFallbkEnable;/*clock fallback: 0==disabled*/
   u_char       H110ClkFallbkStatus;/*clock fallback: 0==using primary clock*/
   S51_BDCPU_PERF      CpuPerf[S51_BDCPU_MAX]; /*board processor perf msmts*/
}
S51_BD_STAT;
```

The fields set by *S51_Init()*:

BdHwModel .....board hardware model: A5_6UB5_BD (a5api.h)

BdHwState.. ..... board hardware state as reported by the driver (A5_BD_* in file *a5api.h*),

BdHwVersion..... board hardware model as reported by the driver (unpublished values),

BdHwRev ... . board hardware revision as reported by the driver (unpublished values),

MaxChan..... ..... absolute maximum number of channels that this board can handle,

CpuBusType ..... type of speech bus supported by the board: A5_PCI_BUS (*a5api.h*),

PciBusNumber..... connected to which PCI bus (always 0),

PciSlotNumber..... board plugged into which PCI slot,

BdHwModelDescr.. ..... textual description of the BdHwModel field ("6UB5"),

BdHwSerialNum.... ..... board serial number.

The fields set and updated by the BCC:

CfgReq..... ..... OR-ed value of all board configuration requests completed on this channel,

CfgVal..... ..... configuration values set so far,

CmdReq .... the last completed board command on this channel,

CmdVal .... values used for the last completed board command,

ComplCode . ..... completion code for the last completed board request (*a5api.h*),

TermReason ..... reason for termination of the last completed request on this channel (S51_*_TRM in s51.h),

OosReason .. .. reason why board is out of service (S51_REASONOOS_* in s51err.h),

ReasonMsg . ..... text description of why board is out of service,

ErrMsg..... ..... text error message associated with the completion code,

BdFwState .. .. board firmware state reported by the board boot process (A5_BD_* in *a5api.h*, also below),

BdTemp..... ..... current board temperature range (s51_*_TEMP in *s51.h*, also see below),

BdFwVersion ..... the version of the RTE operating system running in the board (unpublished values),

BdFwType... ..... name of the firmware package or version which is currently loaded on the board.

The fields set and updated by the BCC but are applicable only to 6UB5 board model:

LedAutoControl..... whether LEDs are auto-controlled by the boot process (1) or not (0),

LedRed..... ..... state of the red LED: on (1) or off (0),

LedGreen .... state of the green LED: on (1) or off (0),

LedYellow... ..... state of the yellow LED: on (1) or off (0),

LedBlue..... ..... state of the blue LED: on (1) or off (0), cannot be directly changed by the application.

The fields set and updated by the BCC but are applicable only to 6UB5 board model. If the board is not in service, the actual values are set to "unknown" (S51_UNKN_HCLK):

H110ClkPrimary..... ..... designated primary H.110 clock (S51_*_HCLK in *s51.h*),

H110ClkUsed..... ..... actual H.110 clock being used (S51_*_HCLK in *s51.h*),

H110ClkFallbkEnable ..... H.110 clock fallback is enabled (1) or not (0),

H110ClkFallbkStatus..... using designated primary H.110 clock (0) or not (1).

The fields updated by the BCC only on explicit command request:

CpuPerf..... ..... performance statistics for board's CPUs. Refer to "6UB5 Board CPU Performance Measurements" on page 60 describing these measurements in more detail.

The Board Temperature constants are defined in s51.h:

S51_UNKN_TEMP..... unknown,

S51_NORM_TEMP ..... normal range,

S51_WARM_TEMP .....warm range,
S51_HOT_TEMP.... .....temperature limit exceeded.

The Board States are redefinition of standard API codes, all are in a5api.h:

A5_BD_UNKN..... .....not known,
A5_BD_INS..... .....in service,
A5_BD_OOS..... .....out of service, *BdFwState* only,
A5_BD_NOCLK.... .....out of service, no H.110 bus clock,
A5_BD_BOOT..... .....booting,
A5_BD_DETACH.. .....detached state, ready for hot-swap,
A5_BD_UNEQP..... .....not equipped, absent,
A5_BD_BAD..... .....unusable,
A5_BD_ERR..... .....other (unknown) board error.

The Board Out Of Service reasons are defined in s51err.h:

S51_REASONOOS_INSERVICE..... board is in service
S51_REASONOOS_OVERTEMP ..... temperature too high
S51_REASONOOS_MANUAL.. ..... because of manual or application software request
S51_REASONOOS_MEMERR.. ..... board memory error
S51_REASONOOS_INTEG..... ..... board integrity failure (whatever that means)
S51_REASONOOS_NOCLOCK ..... no H.110 bus clock,
S51_REASONOOS_IOFAIL..... ..... I/O DB failure
S51_REASONOOS_CPUFAIL... ..... CPU Failure
S51_REASONOOS_CONFIG.... ..... Invalid load configuration
S51_REASONOOS_HOSTERR. ..... Host boot-software error
S51_REASONOOS_OTHER..... ..... Whatever
S51_REASONOOS_BOOTFAIL ..... Boot failure
S51_REASONOOS_EXIT..... ..... ayc51boot (host boot-software) is not running
S51_REASONOOS_ABSENT.... ..... Board absent (not there)
S51_REASONOOS_REBOOT... ..... Manual or application reboot request
S51_REASONOOS_DIAGNOSE ..... Manual or application diagnose request

5.12.8 BCC Code Example

The following example is taken out of the command line utility *s51boot*. It requests to reboot a speech board into a given type and waits for the board boot to complete.

```
          int          BdNum = -1;
          char         BdType[80];
const     S51_EQP*     Eqp;
          S51_CID      BccCid;
const     S51_CH*      Bcc;
extern    void         Quit(const char* ErrMsg);

#define   USAGE \
```

```
    "-bBdNum -tBdType\n" \
    "\t-b bdnum   Board to reboot\n" \
    "\t-t type    board type\n"

#define   MAXWAIT      120    /*seconds*/

void main(int Argc, char* Argv[])
{
          int          Opt;
          S51_REQ      Req;
          S51_CMDVAL   CmdVal;
          S51_BD_CMDVAL BccCmdVal;
          u_long       StartTime;
const     char*        ErrMsg;

    /*parse command line options*/
    opterr = 0;
    while ((Opt = getopt(Argc, Argv, "b:t:?")) != EOF)
    {
          switch (Opt)
          {
          case 'b':
                BdNum = atoi(optarg);
                break;
          case 't':
                strcpy(BdType, optarg);
                break;
          default:
                printf("usage: %s %s\n", Argv[0], USAGE);
                exit(1);
          }
    }

    /*check command parameters*/
    if (BdNum < 0)
          Quit("-b BoardNumber must be specified");
    if (*BdType == 0)
          Quit("-t BoardType must be specified");

    /*init API*/
    Eqp = S51_Init(A5_BLOCK_MOD, NULL, &ErrMsg);
    if (!Eqp)
          Quit(ErrMsg);


    /*allocate BCC*/
    A5_Zerout(Req);
    BccCid = S51_MakeCid(BdNum, 0);
    Req.BdReq = S51_ALLOC_CFGREQ;
    Bcc = S51_CfgChan(BccCid, &Req, NULL, &ErrMsg);
    if ((!Bcc) || (Bcc->ComplCode != A5_OK))
          Quit(ErrMsg);

    /*request to reboot the board*/
    A5_Zerout(Req);
    A5_Zerout(CmdVal);
    A5_Zerout(BccCmdVal);
    Req.BdReq = S51_BOOT_CMDREQ;
    CmdVal.BdVal = &BccCmdVal;
```

```
    strcpy(BccCmdVal.Reason, "manual command");
    strcpy(BccCmdVal.BdFwType, BdType);
    Bcc = S51_CmdChan(BccCid, &Req, &CmdVal, &ErrMsg);
    if ((!Bcc) || (Bcc->ComplCode != A5_OK))
           Quit(ErrMsg);

    printf("Booting of board %d as %s: started, waiting up to %d sec...\n",
           BdNum, BdType, MAXWAIT);

    /*wait up to MAXWAIT seconds max for boot to complete*/
    StartTime = time(NULL);
    while (((time(NULL) - StartTime) < MAXWAIT)
    && (Bcc->BdStat->BdFwState != A5_BD_INS))
    {
           /*make illusion of something happening*/
           printf("."); fflush(stdout);

           /*wait for any events - but no need to look at them*/
           (void) S51_EventChan(3000, NULL, NULL);
    }
    printf("\n");

    /*report failure*/
    if (Bcc->BdStat->BdFwState != A5_BD_INS)
    {
           if (Bcc->BdStat->ErrMsg[0])
                  Quit(Bcc->BdStat->ErrMsg);

           if ((time(NULL) - StartTime) >= MAXWAIT)
                  Quit("timeout");

           Quit("no detail info available");
    }

    /*success*/
    Quit(NULL);
    exit(0);
}

/*report, cleanup and exit*/
void Quit(const char* ErrMsg)
{
    /*clean up*/
    if (Bcc)
           (void) S51_CloseChan(BccCid, NULL);
    if (Eqp)
           S51_Term();

    /*report error*/
    if ((ErrMsg) && (*ErrMsg))
    {
           printf("Board %d failed to reboot: %s\n", BdNum, ErrMsg);
           exit(1);
    }

    /*report success*/
    printf("Board %d rebooted OK as %s\n", BdNum, BdType);
    exit(0);
}
```

5.13 Play Resource

This section describes the Play Resource and the data structures and constants needed to configure and use it. The Play Resource is used to play a set of files and/or memory buffers, separated by optional silence periods. Files and buffers may be audio coded in various ways (see S51_*_CODE constants in file *s51.h*). The Play resource requires an output (transmit) time slot and at least one file or buffer. Once started, the resource plays all files and buffers in the play list in order. When all files and buffers in the list have been played and all the speech has been sent to the time slot, a play completion event will be generated. If the application issues a Stop or Cancel command, play will stop immediately. There is no difference between Stop and Cancel for the Play Resource.

Several features have be built into the Play Resource to provide uninterrupted, click-free output even under heavy host load. The Play Resource will not begin speech output until it has internally buffered four seconds of data or until an end-of-file. This is intended to prevent data underflow. Heavily used announcements such as the service introduction or menu prompts can be manually cached on the board to reduce disk access. To eliminate clicks and other connection noise at the beginning of an announcement, the Play Resource opens the output time slot immediately upon allocation, and begins generating silence. Time slots on the timeslot bus that are not connected to a resource are not guaranteed to be silent. Thus, it is highly desirable to include the time slot parameter in the Play allocation request. The output of a silent Play Resource will not conflict with another output (such as the Tone Generator Resource or the TTS Resource) *on the same board.* The built-in resource conflict checks in the API should be used to prevent *other* boards from driving this timeslot.

It is not possible to "disconnect" the timeslot assigned to the play resource. The only way to truly disconnect the timeslot is to deallocate (S51_FREE_CFGREQ) the play resource. A bit more information is in “Programmers Note: Disconnecting the H.110 Timeslots (HTS):” on page 27.

5.13.1 Play Resource Configuration

The second argument to the S51_CfgChan() function is a pointer to a S51_REQ structure. To change the default configuration parameters for the Play resource, the following constants can be OR-ed into the *S51_REQ.PlayReq* field to indicate which configuration parameters are to be changed

Request Bit Fields	S51_PLAY_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the callback function
S51_CONNTS_CFGREQ	Tid	set the timeslot
S51_ENCODING_CFGREQ	Encoding	set the speech data encoding type
S51_PREPLAY_CFGREQ	PreplayFile	the preplay file for the echo canceller training

The **S51_PLAY_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan()* and it defines the configuration values for the Play Resource.

```
typedef struct
{
   int          (*CbFun)();  /*play resource callback function*/
   A5-TID       Tid;         /*connect to this timeslot*/
   u_short      Encoding;    /*play data format (see file s51.h)*/
   char         PreplayFile[S51_FILENAME_SIZ+1];/*Echo training file name*/
}
S51_PLAY_CFGVAL;
```

The fields are:

Tid ....timeslot to connect to,

Encoding[14] .. ..type of speech coding,

- S51_MULAW_CODE is Mu-Law 64kbps coding,
- S51_ALAW_CODE is A-law 64kbps coding,
- S51_LINEAR_CODE is linear 128kbps coding,
- S51_LINEARLE_CODE is linear little-endian, 128kbps coding,
- S51_G711_MU_CODE is G.711 Mu-law 64kbps, bit-reversed coding,
- S51_G711_A_CODE is G.711 A-law 64kbps, bit reversed coding,
- S51_OKIADPCM32_COD is OKI ADPCM 32kbps encoding coding,
- S51_WAVE_CODE is Microsoft Wave file with the encoding specified in the header.

CbFun.....callback function for play on this channel, **must be set to NULL** if not used

PreplayFile .. ..the file name for Echo Canceller training.

If the encoding is S51_WAVE_CODE, the speech data (whether in the file or in a memory buffer) must be preceded by a valid WAVE header. The contents of the header then determines the actual speech encoding in the file. Currently LSPS Play resource can play the following encodings of the WAVE files:

Symbolic Name	Hex Value	Bits/Sample	Samples/Sec
WAVE_FORMAT_PCM	0x001	16	8000
WAVE_FORMAT_PCM	0x002	8	8000
WAVE_FORMAT_ALAW	0x006	8	8000
WAVE_FORMAT_MULAW	0x007	8	8000
IBM_FORMAT_MULAW	0x101	8	8000
IBM_FORMAT_ALAW	0x102	8	8000

5.13.2 Play Resource Commands

The **S51_PLAY_CMDVAL data structure** is part of an argument to *S51_CmdChan()*, and is used to provide values for the play command. One command can request to play several files and/or buffers. These are defined by the array of **S51_PLAY_SET** structures.

```
typedef struct
{
          char          PlayFile[S51_FILENAME_SIZ+1];/*file name*/
          char*         PlayBuf;            /*or buffer address*/
          u_long        PlayBufSize;        /*buffer size*/
          float         StartDelayTim;      /*wait before play, in seconds*/
}
S51_PLAY_SET;
```

There are several important consideration that the programmer must keep in mind when setting up and using the S51_PLAY_SET structure:

- If the *S51_PLAY_SET.PlayFile* contains a relative pathname, the API converts it into an absolute pathname and overwrites this field. In another words, upon return from *S51_CmdChan()*, the *S51_PLAY_SET* structure owned by the application may have been modified by the API by replacing all relative file pathnames with absolute pathnames.

14. This is the encoding of the input of the Play resource (file or memory). The output (H.110 bus) is usually Mu-Law (by default), but it can be changed system wide to A-Law by changing a parameter in the system configuration file.

- If the *S51_PLAY_SET.PlayBuf* is specified, the contents of this buffer must remain unchanged (and valid) until the Play Resource reports completion, that is, until the Play Resource completion code is no longer A5_INPROG
- The *S51_PLAY_SET* data structure passed to *S51_CmdChan()* can be destroyed upon return from *S51_CmdChan()*, because the API make a copy of it and uses this copy while the play is in progress.

```
typedef struct
{
          u_long              PlayCfgReq;   /*additional (optional) configuration*/
const     S51_PLAY_CFGVAL*    PlayCfgVal;
const     S51_PLAY_SET*       PlaySet;      /*set of files or buffers to play*/
          u_short             SetSize;      /*size of PlaySet*/
          long          StartPosition;      /*within the entire play data, bytes*/
          u_short       AtComplStartRes;    /*resource to start on play complete*/
          u_short       AtComplStopRes;     /*resource to cancel on play completes*/
          u_short       AtComplEnableResTimers;/*enable the three timers on compl*/
          u_short       AtStopEnableResTimers;/*enable the three timers upon stop*/
}
S51_PLAY_CMDVAL;
```

The fields are:

PlayFile ....name of the play file,

PlayBuf ....address of memory buffer[15] to play from instead of a file,[16]

PlayBufSize ....size of PlayBuf or zero (integer 0) for PlayFile,

StartDelayTim..... .....delay before starting to play this file or buffer (milliseconds), 0 to disable the timer,

PlayCfgReq..... .....optional “last-second” configuration request,

PlayCfgVal..... .....values associated with PlayCfgReq,

PlaySet ....set (array) of files and/or buffers to play,

StartPosition..... .....the byte offset that the play data starts,

SetSize ....size of PlaySet,

AtComplStartRes... .....set of resources to start when play completes (OR-ed together),

AtComplStopRes... .....set of resources to stop when play completes (OR-ed together),

AtComplEnableResTimers .....set of resources whose timers should be enabled when play completes,

AtStopEnableResTimers... .....set of resources whose timers should be enabled when play is stopped.

The parameter *StartPosition* can be used to start playing from a position other than the beginning of the file list. For the purpose of positioning, all items sent to play (files and buffers) are concatenated into one play list. *StartPosition* then applies to this play list. The *PreplayFile* (in the configuration request) is not part of this list. When StartPosition is > 0, the *PreplayFile* is not played and play will resume where it was stopped, assuming that the *StartPosition* was obtained from a previously stopped play from the same play list. Thus to resume a play from the position at which it was stopped, save the *BytesPlayed* value in the Play State structure S51_PLAY_STAT when the play is stopped, and set the *StartPosition* to this value while using the same play list as before. If StartPosition is larger than the size of all play items, the Play will not complete (it will "hang" generating silence) and the only way out of this state is to send Stop or Cancel command to the Play resource.

Note that while playing from a memory buffer, the API is using the application’s buffer and the contents of this buffer must remain unchanged.

15. To use a memory buffer instead of a play file, the PlayFile must be set to zeroes (NULL string) and both, the PlayBuf and PlayBuf-Size must be set. The memory must be left unchanged while the play is in progress. In other words, the API plays the speech data directly from the buffer provided by the application.
16. This is an address of host memory, which was obtained either by static allocation, by dynamic allocation (malloc) or by memory mapping a file.

5.13.3 Play Resource State

The **S51_PLAY_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the Play Resource.

```
typedef struct
{
   u_long              FwVersion;   /*version of the play firmware on the board*/
   u_long              CfgReq;      /*the fields in CfgVal that are set*/
   S51_PLAY_CFGVAL     CfgVal;      /*current configuration*/
   u_long              CmdReq;      /*last command completed ok (request)*/
   S51_PLAY_CMDVAL     CmdVal;      /*last command completed ok (values)*/
   u_long              ComplCode;   /*configuration/command completion code*/
   u_long              TermReason;  /*why completed*/
   u_long              ResErrorCode;/*detailed error code, see s51err.h*/
const char*            ErrMsg;      /*error message for last request*/
   u_long              BytesPlayed; /*number of bytes played at completion*/
   u_long              SamplesPlayed;/*number of samples played so far*/
   u_char              PlayListDone;/*non-0 if all data played out*/
}
S51_PLAY_STAT;
```

The variables are:

FwVersion ... .version of the play firmware running on the board,

CfgReq ....OR-ed value of all play configuration requests completed on this channel,

CfgVal ....all values used to configure the play resource on this channel so far,

CmdReq ....the last completed play command on this channel,

CmdVal..... .....values used for the last completed play command,

ComplCode . .....completion code for the last completed play request on this channel,

TermReason. .....reason for termination for the last completed play request on this channel,

ResErrorCode.....low-level error code provided by the Play resource, see s51err.h,

ErrMsg ....text error message associated with the completion code,

SamplesPlayed .....number of samples played so far (see note below),

PlayListDone.....non-0 if all data played,

BytesPlayed. .....number of bytes played during the last completed play command. This number may be slightly higher (up to 16 bytes) than the size of data in the original play request. Also see note below.

PlayListDone is set to non-0 only if all the data in the last Play Set was completely played. It can be used to determine whether or not the play was stopped in the middle (by RTC, Stop, Cancel or Abort), or completed on its own.

The **BytesPlayed** includes the number of bytes (samples) that were played from the given play list. It does not include any internal generated silence (for example due to *StartDelayTim*). On the other hand, **SamplesPlayed** reflects the actual number of samples (bytes) played into the H.110 bus, and is related to the real time between the start command and when the Play stopped. Note that the terms *bytes* and *samples* can be used interchangeably because the Play resource always generates speech data of 8-bit (1 byte) samples at the rate of 8000 samples per second using either Mu-law (default) or A-law (optional) encoding. This is true whether it plays into a real timeslot (H.110 bus) or into a virtual timeslot (VTS).

5.13.4 Play Example

The following code fragment allocates and configures a Play resource and plays a file in blocking (synch) mode.

```
Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
...
S51_PLAY_SET        PlaySet[1];
S51_PLAY_CFGVAL     PlayCfgVal;
S51_CFGVAL          CfgVal;
S51_PLAY_CMDVAL     PlayCmdVal;
S51_CMDVAL          CmdVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

/*clear data structures*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(PlayCfgVal);

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*allocate and configure play to connect to a time slot*/
Req.PlayReq = S51_CONNTS_CFGREQ;
PlayCfgVal.Tid = A5_MakeHtsTid(20);
CfgVal.PlayCfgVal = &PlayCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

/*clear data structures*/
A5_Zerout(Req);
A5_Zerout(PlayCmdVal);
A5_Zerout(CmdVal);
A5_Zerout(PlaySet);

/*play one file*/
Req.PlayReq = S51_START_CMDREQ;
strcpy(PlaySet[0].PlayFile, “announce.mu8”);
PlayCmdVal.PlaySet = PlaySet;
PlayCmdVal.SetSize = 1;
CmdVal.PlayVal = &PlayCmdVal;
Ch = S51_CmdChan(Cid, &Req, &CmdVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);
```

5.14 Record Resource

This section describes the Record Resource and the low-level data structures and constants used in configuring and using the Record Resource on a channel. Record is a resource which records into a file or into a memory buffer from an input time slot. The input time slot is always coded in 8-bit/sample 8000 samples/sec Mu-law or A-law.[17] The written file or the buffer may be audio coded in various ways (see S51_*_CODE constants in file *s51.h*). The Record Resource requires an input time slot and an output file or a buffer.

After the Record is started, the input is scanned until the signal energy exceeds a specified energy threshold (*InitEfloor* configuration parameter). When this threshold is exceeded, the recording actually starts to record the input signal data[18] and the application receives an S51_ENERGY_EVT event. The recorded data will also include input signal received *PrefixTim* seconds before the initial energy threshold was exceeded.

Once started, the recording continues until one of the following happens:

- the IST timer expires, that is, energy level never exceeds *InitEfloor* during the first *InitSilTim* seconds,[19] or
- the EST timer expires, that is, energy level stays below the *EndEfloor* level for *EndSilTim* seconds,[20] or
- the MLT timer expires, that is, the recording has lasted for more than the *MaxListenTim* seconds,[21] or
- the recording buffer overflows, or
- the Record Resource is Stopped or Cancelled by the application explicitly or via the RTC.

If the Record Resource was **Cancelled**, all input data not yet written to the file or the buffer are discarded and the application gets an immediate Record completion event.

If the Record completed on its own (end-recording timeout) or was **Stopped**, the Record Resource determines the end of recorded input data as follows.

- Take the Record Resource stop time (stopped explicitly or by RTC or on its own by a timer) and go back *StopClipTim* seconds.
- From this point, search back to the last time *EndEfloor* energy was detected.
- Add *PostFixTim* to the last energy detection time. This is the end point of the recorded input data.

5.14.1 Record Resource Configuration

The second argument to the S51_CfgChan() function is a pointer to a S51_REQ structure. To change the default configuration parameters for the Record resource, the following constants can be OR-ed into the *S51_REQ.RecReq* field to indicate which configuration parameters are to be changed.

17. This is true even if the input is from a virtual timeslot. That means that if the application is recording from a virtual timeslot, it must assure that the writer of the virtual timeslot uses this standard encoding. This is the speech coding which the LSPS uses as the standard for coding on the timeslots.
18. All data before this event is discarded.
19. Only after the Initial Silence Timer is enabled.
20. Only after the End Silence Timer is enabled.
21. Only after the Maximum Listen Timer is enabled.

Request Bit Fields	S51_REC_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the call back function
S51_CONNTS_CFGREQ	Tid	set the timeslot
S51_ENCODING_CFGREQ	Encoding	set the speech data encoding type
S51_RESMODE_CFGREQ	ResMode	set the recording resource mode
S51_INITEFLOOR_CFGREQ	InitEfloor	set the initial energy detect floor in dBm
S51_ENDEFLOOR_CFGREQ	EndEfloor	set the ending energy detect floor in dBm
S51_INITSILT_CFGREQ	InitSilTim	set the initial silence time in seconds
S51_ENDSILT_CFGREQ	EndSilTim	set the ending silence timer in seconds
S51_MAXLISTENT_CFGREQ	MaxListenTim	set the maximum listen time in seconds
S51_PREFIXT_CFGREQ	PrefixTim	set the silence time to add before recording in seconds
S51_POSTFIXT_CFGREQ	PostfixTim	set the silence time to add after recording in seconds
S51_STOPCLIPT_CFGREQ	StopClipTim	set the time to clip the end of recording in seconds
S51_STARTDLYT_CFGREQ	StartDelayTim	set the time to delay the recording in seconds
S51_CONNTS_2_CFGREQ	Tid2	set the secondary timeslot
S51_DISCTS_2_CFGREQ	Tid2	disconnect secondary timeslot
S51_INTBUFSIZE_CFGREQ	IntBufSize	set the internal on board buffer size

The **S51_REC_CFGVAL data structure** is a part of *S51_CFGVAL* argument to *S51_CfgChan()* and it defines the configuration values for the Record Resource. Note that any timer set to the value 0.0 will never fire and any energy threshold set to minus 99.0 or lower will always be exceeded. All thresholds are in dBm and they must be negative.

```
typedef struct
{
   int          (*CbFun)();   /*record resource callback function*/
   A5_TID       Tid;          /*connect to this timeslot*/
   A5_TID       Tid2;         /*secondary timeslot*/
   u_short      Encoding;     /*record data format*/
   u_long       ResMode;      /*record resource mode*
   long         IntBufSize;   /*internal on board buffer size*
   float        InitEfloor;   /*initial energy detection floor in dBm*/
   float        EndEfloor;    /*ending energy detection floor in dBm*/
   float        InitSilTim;   /*initial silence time out*/
   float        EndSilTim;    /*ending silence time out*/
   float        MaxListenTim; /*maximum record time*/
   float        PrefixTim;    /*silence to add before recording*/
   float        PostfixTim;   /*silence to add after recording*/
```

```
    float        StopClipTim;  /*clip end of recording*/
    float        StartDelayTim;/*start delay (seconds)*/
}
S51_REC_CFGVAL;
```

The fields are:

Tid ....timeslot to connect to,

Tid2 ....secondary timeslot to connect to,

Encoding[22] .. ..type of speech coding (see file *s51.h*),

- S51_MULAW_CODE is Mu-Law 64kbps coding,
- S51_ALAW_CODE is A-law 64kbps coding,
- S51_LINEAR_CODE is linear 128kbps coding,
- S51_LINEARLE_CODE is linear little-endian, 128kbps coding,
- S51_G711_MU_CODE is G.711 Mu-law 64kbps, bit-reversed coding,
- S51_G711_A_CODE is G.711 A-law 64kbps, bit reversed coding,
- S51_OKIADPCM32_CODE is OKI ADPCM 32 encoding coding,
- S51_WAVEMULAW_CODE is WAVE 8-bits/sample, 8000 samples/sec, Mu-law (0x7)[23]
- S51_WAVEALAW_CODE is WAVE 8-bits/sample, 8000 samples/sec, A-law (0x6)
- S51_WAVELINEAR_CODE is WAVE 16-bits/sample, 8000 samples/sec, Linear(0x1)

ResMode ....record resource mode,

- S51_RECSILENCE_RMD is used to record both speech and the leading and trailing silence signal between the start and stop time. The default is not to record leading or trailing silence. Note that when using this mode, the application receives the "energy detect" event (S51_ENERGY_EVT) immediately when the Record module is started, even if there is no speech. This is because this mode is implemented by making the speech detect energy threshold equal to silence.

IntBufSize ... .internal on board buffer size,

InitEfloor..... .....energy level (dBm) which defines “silence vs. signal” before recording starts,

EndEfloor.... .....energy level (dBm) which defines “silence vs. signal” once the recording starts,

InitSilTim ....maximum length of initial silence, if exceeded no speech data is recorded (seconds). Allowed range is >0.2sec.

EndSilTim ... .after recording starts, length of silence which stops the recording (seconds). Allowed range is >0.1 sec to < 8.0 sec.

MaxListenTim.....maximum overall recording time (seconds), includes any initial silence plus the duration of the recording, allowed range is > 0.05 sec,

PrefixTim ....used to set the starting point of the recorded signal to PrefixTim before the record was started (in seconds). The intent is to add some short silence time before the recording to prevent any unintentional clipping. This is applied after StartDelayTim. Allowed range is >=0.0 sec.

PostfixTim... .....used to set the end point of the recorded signal to PostfixTim after the recording has stopped (in seconds). The intent is to add some short silence time after the recording to prevent any unintentional clipping. This is applied after StopClipTim. Allowed range is >=0.0 sec and < 2.0 sec.

StopClipTim .....used to clip the end of recording by StopClipTim seconds when the recording was stopped. The intent is to clip out any signals which were used to stop the recording via RTC (such as a DTMF digit). The StopClipTim is applied before PostfixTim as described before. Allowed range is >= 0.0 sec.

22. This is the encoding of the output of the Record resource (file or memory). The input (H.110 bus or VTS) must be 8 bits/sample 80000 samples/sec, usually Mu-Law (by default), but it can be changed system wide to A-Law by changing a parameter in the system configuration file.

23. Also see notes in “Recording WAVE Files” on page 78.

StartDelayTim..... used to delay the beginning recording by StartDelayTim seconds when the recording was started. The StartDelayTim is applied before PrefixTim as described before. Allowed range is >= 0.0 sec.

CbFun .... callback function for record on this channel.

Note that all timer values are in seconds. Any timer that is set to 0.0 sec will expire immediately (when enabled).

5.14.2 Record Resource Commands

The **S51_REC_CMDVAL structure** is part of an argument to the *S51_CmdChan()* function and is used to provide data for the record commands.

```
typedef struct
{
   u_long       RecCfgReq;          /*additional configuration (optional)*/
const S51_REC_CFGVAL* RecCfgVal;
   long         MaxRecordSize;      /*size of RecBuf or Recfile*/
const char*     RecBuf;             /*buffer to record*/
   char         RecFile[S51_FILENAME_SIZ+1];/*or file to record*/
   u_char       InhTimers;          /*inhibit IST, EST, and MLT*/
   u_char       AppendFile;         /*if not 0, append to Recfile*/
   u_short      AtEnergyStopRes;    /*which resources to stop on energy*/
   u_short      AtComplStopRes;     /*which resources to cancel on completion*/
   u_short      AtComplEnableResTimers;/*enable the three timers on completion*/
   u_short      AtStopEnableResTimers;/*enable the three timers on stop*/
}
S51_REC_CMDVAL;
```

The fields are:

RecCfgReq.. ..... “last-second” configuration request (optional),

RecCfgVal... ..... values associated with RecCfgReq,

MaxRecordSize..... maximum size (in bytes) of the recording buffer (RecBuf) or a file,

RecBuf .... memory address of the recording buffer[24] or NULL if recording to a file,

RecFile..... ..... name of the recording file or NULL string if recording to a buffer,

InhTimers.... ..... if not 0, automatically inhibit the IST, EST, and MLT timers at start,

AppendFile . ..... if not 0, append to recording file,

AtEnergyStopRes .. which resources to stop when initial energy is detected,

AtComplStopRes ... which resources to stop when recording completes on its own,

AtComplEnableResTimers ..... set of resources whose timers should be enabled when the Record resource completes,

AtStopEnableResTimers... ..... set of resources whose timers should be enabled when the Record resource is stopped.

Programmers Note: Take Care When Stopping Recording

Stopping Record via RTC or explicitly at the end of channel operation (when all other resources have completed) could truncate the end of recording because not all of the data may have yet been received by the Record resource. Therefore we recommend to let the record complete on its own (timeout of EST timer).

24. This is an address of host memory, which was obtained either by static allocation, by dynamic allocation (malloc) or by memory mapping a file.

5.14.3 Record Resource State

The **S51_REC_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the Record Resource.

```
typedef struct
{
   u_long              FwVersion;    /*version of the record firmware on the board*/
   u_long              CfgReq;       /*valid fields in CfgVal*/
   S51_REC_CFGVAL      CfgVal;       /*current configuration*/
   u_long              CmdReq;       /*last command completed (request)*/
   S51_REC_CMDVAL      CmdVal;       /*last command completed (values)*/
   u_long              ComplCode;    /*configuration or command completion code*/
   u_long              TermReason;   /*why completed*/
   u_long              ResErrorCode;/*detailed error code, see s51err.h*/
const char*            ErrMsg;       /*error message for last request*/
   u_long              BytesRecd;    /*number of bytes recorded at completion*/
   float               EnergyDetectTime;/*seconds since start*/
   float               EnergyMax;    /*maximum detected energy dBm*/
}
S51_REC_STAT;
```

The fields are:

FwVersion ... .version of the record firmware running on the board,

CfgReq ....OR-ed value of all record configuration requests completed on this channel so far,

CfgVal ....all values used to configure the record resource on this channel so far.

CmdReq ....the last completed record command on this channel,

CmdVal..... .....values used for the last completed record command,

ComplCode . .....completion code for the last completed record request on this channel,

TermReason., ...termination reason for the last completed record request on this channel,

ResErrorCode.....low-level error code provided by the Record Resource, see s51err.h,

ErrMsg ....text error message associated with the completion code,

BytesRecd ... .number of bytes recorded during the last completed record command, for WAVE files this includes the WAVE header,

EnergyDetectTime.. .....relative time stamp (in seconds since last start) when initial energy was detected,

EnergyMax.. .....maximum energy (dBm) detected during the last completed recording.

5.14.4 Recording WAVE Files

The recorded WAVE data always starts with a WAVE header. This is always true, whether recording into a file or into memory. The Record Resource can currently record in one of three WAVE formats:

API Encoding Type	Bits/Sample	Samples/Sec	Coding	WAVE Format Tag
S51_WAVELINEAR_CODE	16	8000	Linear	0x1
S51_WAVEMULAW_CODE	8	8000	Mu-law	0x7
S51_WAVEALAW_CODE	8	8000	A-Law	0x6

Also keep the following in mind when recording WAVE files:

- Any Wave recording which is in progress or was terminated by a fatal error has a Wave header with an infinite Wave chunk size and invalid values in its format chunk, thus preventing it from being played.
- Any Wave recording which was terminated by a Cancel has a Wave header which properly reflects the amount data which was actually written to the file, thus making the file playable as far as possible.
- Any Wave recording which was terminated by a Stop or a timer has a Wave header which properly reflects the amount data recorded and written to the file, thus making the file playable in its entirety.

5.14.5 Record Example

The following code fragment allocates and configures a Record resource and records in blocking (synch) mode.

```
Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
...
S51_REC_CFGVAL      RecCfgVal;
S51_CFGVAL          CfgVal;
S51_REC_CMDVAL      RecCmdVal;
S51_CMDVAL          CmdVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

/*clear data structures*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(RecCfgVal);

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*allocate and configure record resource*/
Req.RecReq = S51_CONNTS_CFGREQ|S51_ENCODING_CFGREQ|S52_MAXLISTENT_CFGREQ;
RecCfgVal.Tid = A5_MakeHtsTid(20);
RecCfgVal.Encoding = S51_ADPCM32_CODE;
RecCfgVal.MaxListenTim = 20;
CfgVal.RecCfgVal = &RecCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

/*clear data structures*/
A5_Zerout(Req);
A5_Zerout(RecCmdVal);
A5_Zerout(CmdVal);

/*record to a file*/
Req.RecReq = S51_START_CMDREQ;
strcpy(RecCmdVal.File, “record.pcm”);
CmdVal.PlayVal = &PlayCmdVal;
Ch = S51_CmdChan(Cid, &Req, &CmdVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);
```

5.15 DTMF Digit Detector Resource

This section describes the DTMF (Dual Tone Multiple Frequencies) Resource and the associated DTMF data structures. The DTMF Resource can detect DTMF or MF (Multiple Frequencies) Digits[25] on a timeslot. The DTMF detector uses an advanced algorithm which is resistant to talk-off (false recognition of DTMF in the presence of speech). The MF detector uses the Goertzel algorithm. The DTMF Resource also has the capability to detect dial pulses[26] while listening for digits. The resource can be configured to stop if dial pulses are detected.

The DTMF Resource can deliver detected digits in one of three ways:

1. In the **default mode,** the resource detects and collects digits and delivers the collected digits to the application in one string when the resource stops. The digits are placed in the resource state structure (S51_DTMF_STAT). Even in this mode, the application can specify that an event (S51_DIGIT_EVT) be generated for each detected digit by setting S51_EACHDIG_CFGREQ. There is a maximum number of digits that can be detected at one time (S51_INDIGS_SIZ[27]).
2. In **stream mode,** the resource detects but does not collect digits. The digits are delivered to the application one at a time as they are detected (via S51_DIGIT_EVT). After the resource stops, the resource state structure contains only the last detected digit. In this mode, the application is responsible for collecting the digits.
3. In **burst mode,** the resource detects and partially collects the digits but delivers them to the application as they are detected (via S51_DIGIT_EVT). In this mode, the resource may deliver more than one digit at a time. After the resource stops, the resource state structure contains only the last delivered digit(s). In this mode it is the application's responsibility to collect all of the digits.

The DTMF Resource also has additional modes which can be used in combination:

- the S51_PULSEDETECT_RMD mode can be specified to enable the detection of dial pulses,
- the S51_STOPONPULSE_RMD may be used with S51_PULSEDETECT_RMD to stop on detection of pulses,
- the S51_IGNORETONEATSTART_RMD can be used with other modes to ignore any tone which is present when the resource is started.

Once started, the DTMF Resource runs until one of the following happens:

- the IST timer expires, that is, no digits were detected during initial *InitSilTim* seconds,
- the EST timer expires, that is, no digits were detected for *EndSilTim* seconds,
- the MLT timer expires, that is, the detection has lasted for more than the *MaxListenTim* seconds,
- maximum number of digits was detected (as defined by S51_MAXDIGS_CFGREQ/MaxDigits parameter),
- one of the terminating digits (in *TermDigits*) was detected,
- dial pulse was detected while the pulse detection was enabled and the stop-on-pulse mode (S51_STOPONPULSE_RMD) was on,
- the DTMF Resource is Stopped or Cancelled by the application explicitly or via the RTC.

5.15.1 DTMF Detector Resource Configuration

The second argument to the S51_CfgChan() function is a pointer to a S51_REQ structure. To change the default configuration parameters for the DTMF resource, the following constants can be OR-ed into the *S51_REQ.DtmfReq* field to indicate which configuration parameters are to be changed.

25. Only one or the other at one time. The set of detected MF digits is known as CCITT Q.320 System R1 Register Signalling.
26. Even though this is only detection and not recognition, it is not 100% reliable.
27. Currently this is 63.

Request Bit Fields	S51_DTMF_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the call back function
S51_CONNTS_CFGREQ	Tid	set the timeslot value
S51_RESMODE_CFGREQ	ResMode BurstCount (only need to be set if it is in S51_BURST_RMD) BurstInterval (only need to be set if it is in S51_BURST_RMD)	set the DTMF resource mode
S51_INITSILT_CFGREQ	InitSilTim	set the initial silence time in seconds. The resource will stop if first digit is not detected in this period.
S51_ENDSILT_CFGREQ	EndSilTim	set the ending silence time in seconds. The resource will stop if no more digits are detected in this period.
S51_EACHDIG_CFGREQ	<none>	set the mode to send the event on each detected digit
S51_MAXLISTENT_CFGREQ	MaxListenTim	set the maximum listen time in seconds
S51_MAXDIGS_CFGREQ	MaxDigits	set the maximum number of digits to collect
S51_TONESET_CFGREQ	ToneSet	set to either S51_DTMF_TSET or S51_R1MF_TSET
S51_STARTDLYT_CFGREQ	StartDelayTim	Wait before starting to look for energy which would initiate the recording
S51_TERMDIG_CFGREQ	TermDigits	set the terminate digits that the DTMF will stop when these digits are detected
S51_MINDIGWIDTH_CFGREQ	MinDigWidth	set the tone duration time in seconds for detection
S51_IGNORELEADDIG_CFGREQ	IgnoreLeadDig	specify a set of digits to be ignored at the start of detection
S51_IGNOREDIG_CFGREQ	IgnoreDig	specify a set of digits to be ignored

The **S51_DTMF_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan()* and it defines the configuration parameters for the DTMF Resource.

```
typedef struct
{
   int          (*CbFun)();  /*DTMF Resource callback function*/
   A5_TID       Tid;         /*connect to this timeslot*/
   u_short      ToneSet;     /*S51_DTMF_TSET or S51_R1MF_TSET*/
```

```
    u_short      MaxDigits;   /*maximum number of digits collected*/
    u_long       ResMode;     /*DTMF Resource mode*/
    u_short      BurstCount;  /*used for S51_BURST_RMD*/
    float        BurstInterval;/*used for S51_BURST_RMD*/
    float        InitSilTim;  /*initial silence time out*/
    float        EndSilTim;   /*end silence time out*/
    float        MaxListenTim;/*maximum listen time out*/
    float        StartDelayTim;/*start delay in seconds*/
    float        MinDigWidth; /*minimum tone duration for detection*/
    char         IgnoreLeadDig[S51_IGNOREDIG_SIZ+1]/*ignore lead digits*/
    char         IgnoreDig[S51_IGNOREDIG_SIZ+1]/*always ignore digits*/
    char         TermDigits[S51_TERMDIG_SIZ+1]/*stop on any of these*/
}
S51_DTMF_CFGVAL;
```

The fields are:

Tid ....timeslot to connect to,

ToneSet..... .....which tones to detect, can be one of:

- S51_DTMF_TSET which contains 16 DTMF digits: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘*’, ‘#’, ‘A’, ‘B’, ‘C’, and ‘D’, or
- S51_R1MF_TSET which contains 12 MF digits: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘K’ and ‘S’,

MaxDigits.... .....stop when this many digits are detected (default is *S51_INDIGS_SIZ*),

ResMode ....resource mode, can be any combination of the following:

- S51_STREAM_RMD sets the resource to stream mode,
- S51_BURST_RMD sets the resource to burst mode,
- S51_PULSEDETECT_RMD turns on dial pulse detection,
- S51_STOPONPULSE_RMD stops the resource when a dial pulse is detected,
- S51_IGNORETONEATSTART_RMD ignores tone at start,

BurstCount .. ..used in S51_BURST_RMD to set the maximum number of digits delivered at one time,

BurstInterval .....used in S51_BURST_RMD to set the burst mode interval in seconds,

InitSilTim ....stop when the first digit is not detected within this time (seconds), must be >0.0 sec,

EndSilTim, ...stop if no more digits are detected within this time (seconds), must be >0.0 sec

MaxListenTim.....stop if the detection has started this long ago (seconds), must be >0.0 sec,

StartDelayTim.....start delay time (seconds), value of 0 disables the timer,

MinDigWidth .....minimum tone duration for detection (seconds),

IgnoreLeadDig .....ignore the leading digits if the digit tone was already present when the resource was started,

IgnoreDig ....always ignore the specified digits,

TermDigits... .....stop on the digits specified,

CbFun..... .....callback function for DTMF on this channel.

5.15.2 DTMF Detector Resource Commands

The **S51_DTMF_CMDVAL data structure** is part of *S51_CMDVAL* argument to the *S51_CmdChan()* function, it used to provide data for the DTMF command.

```
typedef struct
{
   u_long              DtmfCfgReq;         /*additional (optional) configuration*/
const S51_DTMF_CFGVAL* DtmfCfgVal;
```

```
    u_char              InhTimers;          /*inhibit IST, EST, and MLT*/
    u_short             AtDigitStopRes;     /*resources to stop on first digit*/
    u_short             AtComplStopRes; /*resources to stop when DTMF completes*/
    u_short             AtComplEnableResTimers;/*enable timers when DTMF completes*/
    u_short             AtStopEnableResTimers;/*enable timers when DTMF stops*/
}
S51_DTMF_CMDVAL;
```

The fields are:

DtmfCfgReq..... .....optional “last-second” configuration request,

DtmfCfgVal..... .....values associated with DtmfCfgReq,

InhTimers..... .....if not 0, automatically inhibit the IST, EST, and MLT timers at start,

AtDigitStopRes..... .....resources to stop when the first digit is detected,

AtComplStopRes..... .....resources to stop when DTMF completes on its own,

AtComplEnableResTimers .....resources whose timers should be enabled when the DTMF completes,

AtStopEnableResTimers... .....resources whose timers should be enabled when the DTMF is stopped.

5.15.3 DTMF Detector Resource State

The **S51_DTMF_STAT data structure** is part of the channel structure (*S51_CH*) and it defines the current state of the DTMF Resource.

```
typedef struct
{
    u_long              FwVersion;
    u_long              CfgReq;       /*valid fields in CfgVal*/
    S51_DTMF_CFGVAL     CfgVal;       /*current configuration*/
    u_long              CmdReq;       /*last command completed (request)*/
    S51_DTMF_CMDVAL     CmdVal;       /*last command completed (values)*/
    u_long              ComplCode;    /*DTMF configuration/command completion code*/
    u_long              TermReason;   /*why stopped*/
const char*             ErrMsg;       /*error message for last request*/
    u_long              ResErrCode;   /*detail error code, s51err.h*/
    float               LastDigitTime;/*relative time stamp (seconds since start)*/
    char                LastDigit[S51_INDIGS_SIZ+1];/*last digit(s) received*/
    u_short             DigitCnt;     /*#of digits detected (size of Digits[])*/
    char                Digits[S51_INDIGS_SIZ+1];/*detected digits*/
}
S51_DTMF_STAT;
```

The fields are:

FwVersion... .....version of the DTMF firmware running on the board,

CfgReq..... .....OR-ed value of all DTMF configuration requests completed on this channel,

CfgVal..... .....all values used to configure the DTMF resource on this channel so far.

CmdReq..... .....the last completed DTMF command on this channel,

CmdVal..... .....values used for the last completed DTMF command,

ComplCode. .....completion code for the last completed DTMF request on this channel,

TermReason .....termination reason for the last completed DTMF request on this channel,

ResErrCode. .....low-level error code provided by the DTMF Resource, see s51err.h,

ErrMsg..... .....text error message associated with the completion code,

LastDigitTime.....time (seconds) from the last start until the last detected digit; applicable only if the *LastDigit* field is set (which means only if detecting in streaming mode or in burst mode),

LastDigit..... .....last digit(s)[28] received in the “event on each digit” and the streaming or burst mode is in effect,

DigitCnt..... .....this field is updated in real-time as digits are detected showing the number of digits collected so far (in default more) or detected at this event (in streaming and burst mode); this is always the length of string in the *Digits* field,

Digits..... .....in the default mode these are all the detected and collected digits, in the streaming and burst mode this is the last digit(s) that was detected and delivered to the application.

5.15.4 DTMF Example

The following code fragment allocates and configures a DTMF resource and detects digits in blocking (synch) mode.

```
S51_DTMF_CFGVAL     DtmfCfgVal;
S51_CFGVAL          CfgVal;
S51_DTMF_CMDVAL     DtmfCmdVal;
S51_CMDVAL          CmdVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(DtmfCfgVal);

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*allocate and configure the DTMF detector resource*/
Req.DtmfReq = S51_CONNTS_CFGREQ|S51_MAXLISTENT_CFGREQ|S51_TERMDIG_CFGREQ;
DtmfCfgVal.Tid = A5_MakeHtsTid(20);
DtmfCfgVal.MaxListenTim = 20;
DtmfCfgVal.TermDigits[0] ='#';
CfgVal.DtmfCfgVal = &DtmfCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
      return(ErrMsg);

/*clear data structures*/
A5_Zerout(Req);
A5_Zerout(DtmfCmdVal);
A5_Zerout(CmdVal);

/*detect DTMF digits*/
DtmfCmdVal.AtDigitStopRes = S51_PLAY_RES;
CmdVal.DtmfVal = &DtmfCmdVal;
Req.DtmfReq = S51_START_CMDREQ;
Ch = S51_CmdChan(Cid, &Req, &CmdVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
      return(ErrMsg);
```

28. In streaming mode, there is always only a single digit in the LastDigit field. In burst mode, the LastDigit field contains a group of 1 to N digits.

5.16 TDD Digit Detector Resource

This section describes the TDD (Telecommunications Device for the Deaf) Digit Detector Resource including the low-level data structures and constants used to configure and use this resource on a channel. This resource detects digits using the Baudot-45 signalling. This is the most common code used in the USA for communication between deaf parties using TDD devices. The TDD Resource provided here detects a subset of the TDD characters: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’. ‘9’, space, and ‘c’ (to indicate “continue”). In many ways this resource is very similar to the DTMF detector.

The TDD Resource can deliver the detected digits in one of two ways:

1. In the **default mode,** the resource detects and collects the digits and delivers these digits to the application in one string when it stops. They are placed in the resource state structure (S51_TDD_STAT). Even in this mode, the application has the option to get an event (S51_DIGIT_EVT) for each detected digit. In this mode there is a maximum number of digits that can be detected at one time (S51_INDIGS_SIZ[29]).
2. In the **stream mode,** the resource detects but does not collect the digits. Each digit is delivered to the application one at a time as it is detected (via S51_DIGIT_EVT). After the resource stops, the resource state structure contains only the last digit detected. In this mode, it is the application’s responsibility to collect the digits.

Note that unlike the DTMF Detector resource, the TDD Resource does not provide burst mode.

Once started, the TDD Resource runs until one of the following happens:

- the IST timer expires, that is, no digits were detected during initial *InitSilTim* seconds,
- the EST timer expires, that is, no digits were detected for *EndSilTim* seconds,
- the MLT timer expires, that is, the detection has lasted for more than the *MaxListenTim* seconds,
- the TDD resource is Stopped or Cancelled by the application explicitly or via the RTC,
- maximum number of digits was detected (as defined by S51_MAXDIGS_CFGREQ/MaxDigits parameter).

5.16.1 TDD Detector Resource Configuration

The second argument to the S51_CfgChan() function is a pointer to a S51_REQ structure. To change the default configuration parameters for the TDD resource, the following constants can be OR-ed into the *S51_REQ.TddReq* field to indicate which configuration parameters are to be changed.

Request Bit Fields	S51_TDD_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource.
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request.
S51_CBFUN_CFGREQ	CbFun	set the call back function.
S51_CONNTS_CFGREQ	Tid	set the timeslot value.
S51_RESMODE_CFGREQ	ResMode	set the TDD Resource mode.
S51_INITSILT_CFGREQ	InitSilTim	set the initial silence time in seconds. The resource will stop if first digit is not detected in this period.

29. Currently this is 63 digits.

Request Bit Fields	S51_TDD_CFGVAL Members	Description
S51_ENDSILT_CFGREQ	EndSilTim	set the ending silence time in seconds. The resource will stop if additional digits are not detected in this period.
S51_MAXLISTENT_CFGREQ	MaxListenTim	set the maximum listen time in seconds.
S51_STARTDLYT_CFGREQ	StartDelayTim	set the time to delay the detection in seconds.
S51_EACHDIG_CFGREQ	<none>	set the mode so an event is sent for each detected digit.
S51_MAXDIGS_CFGREQ	MaxDigits	set the maximum number of digits to collect.

The **S51_TDD_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan()* and it defines the configuration parameters for the TDD Resource.

```
typedef struct
{
   int          (*CbFun)();  /*DTMF resource callback function*/
   A5_TID       Tid;         /*connect to this timeslot*/
   u_short      MaxDigits;   /*maximum number of digits collected*/
   u_long       ResMode;     /*TDD resource mode*/
   float        InitSilTim;  /*initial silence time out*/
   float        EndSilTim;   /*end silence time out*/
   float        MaxListenTim;/*maximum listen time out*/
   float        StartDelayTim;/*start delay in seconds*/
}
S51_TDD_CFGVAL;
```

The fields are:

Tid ....timeslot to connect to,

MaxDigits.... .....stop when this many digits are detected (default is *S51_INDIGS_SIZ*),

ResMode ....resource mode, if the default mode is not desired, the following mode may be set:

- S51_STREAM_RMD sets the resource to stream mode (described before),

InitSilTim ....stop when the first digit is not detected within this time interval (seconds), must be >0.0 sec

EndSilTim, ...stop if additional digits are not detected within this time interval (seconds), must be >0.0 sec

MaxListenTim.....stop if the detection has exceeded this time interval (seconds), must be >0.0 sec,

StartDelayTim.....start delay time (seconds), value of 0 disables the timer,

CbFun..... .....callback function for the TDD Resource on this channel.

5.16.2 TDD Resource Commands

The **S51_TDD_CMDVAL data structure** is part of *S51_CMDVAL* argument to the *S51_CmdChan()* function, it used to provide data for the TDD command.

```
typedef struct
{
   u_long              TddCfgReq;          /*additional (optional) configuration*/
const S51_TDD_CFGVAL*  TddCfgVal;
   u_char              InhTimers;          /*inhibit IST, EST, and MLT*/
   u_short             AtDigitStopRes;     /*resources to stop on the first digit*/
   u_short             AtComplStopRes; /*resources to stop when TDD completes*/
   u_short             AtComplEnableResTimers;/*enable timers when TDD completes*/
   u_short             AtStopEnableResTimers;/*enable timers when TDD stops*/
}
S51_TDD_CMDVAL;
```

The fields are:

TddCfgReq ....optional “last-second” configuration request,

TddCfgVal..... .....values associated with TddCfgReq,

InhTimers..... .....if not 0, automatically inhibit the IST, EST, and MLT timers at start,

AtDigitStopRes..... .....resource(s) to stop when the first digit is detected,

AtComplStopRes..... .....which resources to stop when TDD completes on its own,

AtComplEnableResTimers.....set of resources whose timers should be enabled when the TDD completes,

AtStopEnableResTimers... .....set of resources whose timers should be enabled when the TDD is stopped.

5.16.3 TDD Resource State

The **S51_TDD_STAT data structure** is part of the channel structure (*S51_CH*) and it defines the current state of the TDD Resource.

```
typedef struct
{
   u_long              FwVersion;
   u_long              CfgReq;      /*valid fields in CfgVal*/
   S51_TDD_CFGVAL      CfgVal;      /*current configuration*/
   u_long              CmdReq;      /*last command completed (request)*/
   S51_TDD_CMDVAL      CmdVal;      /*last command completed (values)*/
   u_long              ComplCode;   /*configuration/command TDD completion code*/
   u_long              TermReason;  /*reason for stopping*/
const char*            ErrMsg;      /*error message for the last request*/
   u_long              ResErrCode;  /*detailed error code, s51err.h*/
   float               LastDigitTime;/*relative time stamp (seconds since start)*/
   char                LastDigit[2];/*last digit received*/
   u_short             DigitCnt;    /*digits detected (size of Digits[])*/
   char                Digits[S51_INDIGS_SIZ+1];/*detected digits*/
}
S51_TDD_STAT;
```

The fields are:

FwVersion... .....version of the TDD firmware running on the board,

CfgReq..... .....OR-ed value of all TDD configuration requests completed on this channel,

CfgVal..... .....all values used to configure the TDD Resource on this channel so far,

CmdReq..... .....the last completed TDD command on this channel,

CmdVal..... .....values used for the last completed TDD command,

ComplCode. .....completion code for the last completed TDD request on this channel,

TermReason .....termination reason for the last completed TDD request on this channel,

ResErrCode. .....low-level error code provided by the TDD Resource, see s51err.h,

ErrMsg..... .....text error message associated with the completion code,

LastDigitTime.....time (seconds) from the last start until the last detected digit; applicable only if the *LastDigit* field is set (which means only if detecting in streaming mode),

LastDigit..... .....last digit received in the streaming mode (as a 1-digit string),

DigitCnt..... .....this field is updated in real-time as digits are detected showing the number of digits collected so far (in default mode) or is 0 or 1 in streaming mode (only one digit is delivered at a time); this is always the length of string in the *Digits* field,

Digits ....in the default mode these are all the detected and collected digits so far, in the streaming mode this is the last digit that was detected and delivered to the application.

5.16.4 TDD Detector Example

The following code fragment allocates and configures a TDD resource and detects digits TDD in blocking (synch) mode.

```
Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
...
S51_TDD_CFGVAL      TddCfgVal;
S51_CFGVAL          CfgVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

/*clear data structures*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(TddCfgVal);

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*configure (listen on) the time slot 20 and specify maximum listen time*/
Req.TddReq = S51_CONNTS_CFGREQ|S51_MAXLISTENT_CFGREQ;
TddCfgVal.Tid = A5_MakeHtsTid(20);
TddCfgVal.MaxListenTim = 15.0;
CfgVal.TddCfgVal = &TddCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

/*start TDD digit detection command - note that command structure is not needed*/
A5_Zerout(Req);
Req.TddReq = S51_START_CMDREQ;
Ch = S51_CmdChan(Cid, &Req, NULL, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);
```

5.17 Tone Generator Resource

This section describes the Tone Generator Resource and the low-level data structures and constants used for its configuration and use. The Tone Generator Resource can play tones into a timeslot. To start, it requires an output (transmit) time slot, a tone set, and the tone name(s) to generate.

It is not possible to "disconnect" the timeslot assigned to the Tone Generator resource. The only way to truly disconnect the timeslot is to deallocate (S51_FREE_CFGREQ) the ToneGen resource. A bit more information is in “Programmers Note: Disconnecting the H.110 Timeslots (HTS):” on page 27.

A **Tone Set** defines a fixed group of tones. The Tone Generator resource can handle only one Tone Set at any one time. The built-in Tone Sets are:

DTMF Tone Set..... ..... the valid tones are: ‘,’, ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘*’, ‘#’, ‘A’, ‘B’, ‘C’, and ‘D’. The comma represents 100 msec of silence. Each digit is 120 msec in duration followed by 80 msec of silence. Except for adding commas (‘,’), this timing cannot be changed. Note that this is slower than the standard (common) DTMF outpulsing speed (60msec/40msec). The reason is to improve reliability when communicating with certain telephone/PBX switches which cannot handle the fast outpulsing speed.

MF Tone Set..... .....the valid tones are: ‘,’, ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘K’, and ‘S’, The comma represents a 100 msec of silence. The letters ‘K’ and ‘S’ represent the digits “KP” and “ST”, respectively. The MF tone set implements CCITT Q.320 Signalling System R1 Register Signalling.

Call Progress Tone Set.....also known as the *CP* Tone Set for short. Only one of these tones may be specified in each command request. The standard supported tones are:

- Busy tone,
- Dial tone,
- Ringback tone,
- Reorder (fast busy) tone,
- Offhook (receiver off hook) tone,
- Confirmation tone,
- Message Waiting tone,
- a number of other additional custom-defined tones (see header file *s51.h*).

Music Tone Set..... .....the valid tones are ‘,’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, and ‘g’. A comma represents 20 msec of silence, A-G represents the octave starting at 440 Hz, a-g represents the octave starting at 880 Hz.

Test Bit Pattern..... .....the "tone" is a 1-byte (8-bit) arbitrary pattern which is then transmitted on the output timeslot. This is handled in the same way as any "infinite" tone - it is generated indefinitely until either the MaxDuration timer expires, or the resource is stopped by an explicit Stop/Cancel/Abort command.

For **finite** tones (DTMF, etc.), once the resource starts, the resource plays the requested tones until:

- all the tones are played out,
- the resource receives a Stop, Cancel, or Abort either directly from the application or via RTC.

For **infinite** tones (dial tone, etc.) the resource continues to play the tone until:

- the MaxDuration timer expires (if set),
- the resource receives a Stop, Cancel, or Abort either directly from the application or via RTC.

To permit easy addition of new tones, the complete definition of the available Tone Sets and the tones themselves is implemented by text definitions in a **Tone Generator Initialization File**. The Tone Generator resource has an **internal** (built-in) initialization file. An **external** tone definition file, where application-dependent custom tone definitions could be specified, may be defined in the *CfgFile* field. **This ability to use an externally defined Tone Generator Initialization File is not yet supported by the Tone Generator Resource.**

5.17.1 Tone Generator Configuration

The second argument to the *S51_CfgChan()* function is a pointer to a S51_REQ structure. To change the default configuration parameters for the Tone Generator resource, the following constants can be OR-ed into the *S51_REQ.TgenReq* field to indicate which configuration parameters are to be changed.

Request Bit Fields	S51_TGEN_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the call back function
S51_CONNTS_CFGREQ	Tid	set the timeslot value
S51_TONESET_CFGREQ	ToneSet	set the group of tones that can be generated
S51_MAXDURATION_CFGREQ	MaxDuration	set the time duration to generate the tone
S51_CFGFILE_CFGREQ	CfgFile	set the name of external tone definition file

The **S51_TGEN_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan* and it defines the configuration values for the Tone Generator Resource.

```
typedef struct
{
   int          (*CbFun)();         /*callback function*/
   A5_TID       Tid;                /*timeslot to connect to*/
   float        MaxDuration;        /*maximum duration for unlimited tones*/
   char         CfgFile[S51_FILENAME_SIZ+1];/*tone definition file*/
   char         ToneSet[S51_TONESET_SIZ+1];/*one of the tone-set*/
}
S51_TGEN_CFGVAL;
```

The fields are:

Tid ....output timeslot to connect to,

CbFun..... .....callback function for tone generation on this channel,

MaxDuration .....applicable only to the infinite tones, when set it limits them to this value (seconds),

CfgFile ....the file name for a user defined tone,[30]

ToneSet..... .....specifies a name for a group of tones that can be generated, that is, which can be in the command's ToneName field. This field is required. Changing the ToneSet value automatically clears the MaxDuration to default (no limit). The ToneSet field can contain one of two types of data:

30. Not yet supported.

- An ASCII string naming the ToneSet. The names are: "CPT" (Call Progress Tones), "DTMF", "R1MF", or "Music". For **example**:
  ```
  strcpy(TgenCfgVal.ToneSet, “CPT”);
  ```

- Or a one-byte value in ToneSet[0] as defined above (S51_*_TSET). The following **example** does the same thing as the previous example, that is, configure the call progress tone set:
  ```
  TgenCfgVal.ToneSet[0] = S51_CPT_TSET;
  ```

5.17.2 Tone Generator Commands

The **S51_TGEN_CMDVAL data structure** is part of S51_CFGVAL argument to *S51_CmdChan()* function, and it provides values for the Tone Generator Resource (Tgen) command.

```
typedef struct
{
   u_long              TgenCfgReq;         /*additional (optional) configuration*/
const S51_TGEN_CFGVAL* TgenCfgVal;
   char                ToneName[S51_TONEOUT_SIZ+1];/*which tone*/
   u_short             AtComplStartRes; /*resources to start when Tgen completes*/
   u_short             AtComplStopRes; /*resources to stop when Tgen completes*/
   u_short             AtComplEnableResTimers;/*enable timers when Tgen completes*/
   u_short             AtStopEnableResTimers;/*enable timers when Tgen stops*/
}
S51_TGEN_CMDVAL;
```

The fields are:

TgenCfgReq..... .....optional “last-second” configuration request,

TgenCfgVal..... .....values associated with TgenCfgReq,

ToneName..... .....defines which tones to play from the defined ToneSet. This can be one of:

- a sequence of "digit" tones with one ASCII character per tone, for example: ",01234567890#*ABCD" for DTMF and ",01234567890KS" for R1MF.
- a single tone name as an ASCII string. For example the "CPT" Tone Set (a.k.a. S51_CPT_TSET) can be one of: "busy", "dialtone", "confirm", "msgwait", "reorder", or "off-hook".
- a one-byte value in ToneName[0] as a standard #define constant (S51_*_TONE). They are:
  S51_BUSY_TONE, S51_OFFHOOK_TONE, S51_REORDER_TONE,
  S51_DIAL_TONE, S51_CONFIRM_TONE, S51_INTERCEPT_TONE
  S51_MSGWAIT_TONE, S51_RINGBACK_TONE,
- a one-byte value to be used as a bit test pattern by the S51_BITPATTERN_TSET tone set.

AtComplStartRes..... .....resources to start when tone generator completes on its own,

AtComplStopRes..... .....resources to stop when tone generator completes on its own,

AtComplEnableResTimers.....resources whose timers should be enabled when the tone generator completes,

AtStopEnableResTimers... .....resources whose timers should be enabled when the tone generator is stopped.

The application has a choice of using either *#defined* constants or text string names for certain Tone Names. It is recommend to use *#defined* constants when possible. If misspelled, the #define will be detected during compilation, while a misspelled string will be detected only during run-time.

5.17.3 Tone Generator State

The **S51_TGEN_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the Tone Generation Resource for this channel.

```
typedef struct
{
   u_long              FwVersion;
   u_long              CfgReq;      /*valid fields in CfgVal*/
   S51_TGEN_CFGVAL     CfgVal;      /*current configuration*/
   u_long              CmdReq;      /*last command completed (request)*/
   S51_TGEN_CMDVAL     CmdVal;      /*last command completed (values)*/
   u_long              ComplCode;   /*configuration/command completion code*/
   u_long              TermReason;  /*why stopped*/
   u_long              ResErrCode;  /*detailed error code, s51err.h*/
const char*            ErrMsg;      /*error message for last request*/
}
S51_TGEN_STAT;
```

The variables are:

FwVersion ... .version of the tone generator firmware running on the board,

CfgReq ....OR-ed value of all tone generator configuration requests completed on this channel,

CfgVal ....all values used to configure the tone generator resource on this channel,

CmdReq ....the last completed tone generator command on this channel,

CmdVal..... .....values used for the last completed tone generator command,

ComplCode . .....completion code for the last completed tone generator request on this channel,

TermReason. .....termination reason for the last completed tone generator request,

ResErrCode . .....low-level error code provided by the tone generator, see s51err.h,

ErrMsg ....text error message associated with the completion code.

5.17.4 Tone Generator Example

The following code fragment allocates and configures a Tone Generator resource and plays a busy tone in blocking (synch) mode.

```
S51_TGEN_CFGVAL     TgenCfgVal;
S51_CFGVAL          CfgVal;
S51_TGEN_CMDVAL     TgenCmdVal;
S51_CMDVAL          CmdVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*clear data structures*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(TgenCfgVal);

/*allocate and configure the Tone Generator Resource for CPT tones*/
Req.TgenReq = S51_CONNTS_CFGREQ|S51_MAXDURATION_CFGREQ|S51_TONESET_CFGREQ;
TgenCfgVal.Tid = A5_MakeHtsTid(20);
TgenCfgVal.MaxDuration = 20;
strcpy(TgenCfgVal.ToneSet, “CPT”);
CfgVal.TgenCfgVal = &TgenCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

A5_Zerout(Req);
A5_Zerout(TgenCmdVal);
A5_Zerout(CmdVal);

/*play busy tone forever or until stopped by another resource via RTC*/
strcpy(TgenCmdVal.ToneName, “busy”);
CmdVal.TgenVal = &TgenCmdVal;
Req.TgenReq = S51_START_CMDREQ;
Ch = S51_CmdChan(Cid, &Req, &CmdVal, &ErrMsg);
```

5.18 Call Progress Tone Detector Resource

This section describes the Call Progress Tone (CPT) Detector Resource and the low-level data structures and constants used for configuration and use. The CPT Detector Resource can detect tones from a timeslot. To start, it requires an input (receive) time slot, and the tone name(s) to detect. The standard set of detectable tones are:

- Busy tone,
- Dial tone,
- Ringback tone,
- Reorder (fast busy) tone,
- Offhook (receiver off hook) tone.

The resource continues to detect the tone until one of the following happens:

- the MaxListen timer expires (if set),
- the InitSilence timer expires (if set),
- the EndSilence timer expires (if set),
- the TermSubset tone is detected,
- the resource receives a Stop, Cancel, or Abort either directly from the application or via RTC.

While it is running, a S51_TONE_EVT is generated every time a tone is detected. Once it stops, the last detected tone is in the S51_TDET_STAT.LastToneName (as a string) and in S51_TDET_STAT.LastTone (as a constant).

To permit the easy addition of new tones, the complete definition of the available tones is implemented by text definitions in a **Tone Detection Initialization File**. The Tone Detector by default uses the *cpt.cfg* initialization file[31] (defines dial tone, busy tone, etc.). An alternate (different) tone definition file can also be specified using the *CfgFile* variable. This file can be used to specify application-dependent custom tone definitions to be used by the CPT Detector Resource to detect user-defined tones. The syntax of the tone description can be deduced by examining the default *cpt.cfg* initialization file.

5.18.1 Tone Detector Resource Configuration

The second argument to the *S51_CfgChan()* function is a pointer to a S51_REQ structure. To change the default configuration parameters for the Tone Detector resource, the following constants can be OR-ed into the *S51_REQ.TdetReq* field to indicate which configuration parameters are to be changed.

Request Bit Fields	S51_TDET_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current configuration request
S51_CBFUN_CFGREQ	CbFun	set the call back function
S51_CONNTS_CFGREQ	Tid	set the timeslot
S51_INITSILT_CFGREQ	InitSilTim	set the initial silence time (in seconds), the resource will stop if the tone is not detected in this period

31. Part of the lspsrte package.

Request Bit Fields	S51_TDET_CFGVAL Members	Description
S51_ENDSILT_CFGREQ	EndSilTim	set the ending silence time in seconds resource will stop if no more tone is detected in this period
S51_MAXLISTENT_CFGREQ	MaxListenTim	set the maximum listen time in seconds
S51_CFGFILE_CFGREQ	CfgFile	set the file name of user defined tone
S51_DETECTSUBSET_CFGREQ	DetectSubset	specify a set of tone names that can be detected
S51_TERMSUBSET_CFGREQ	TermSubset	specify a set of tone names that the resource will stop when these are detected

The **S51_TDET_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan* and it defines the configuration values for the Call Progress Tone Detector Resource:

```
typedef struct
{
    int          (*CbFun)();         /*callback function*/
    A5_TID       Tid;                /*timeslot to connect to*/
    float        InitSilTim;         /*initial silence time out in seconds*/
    float        EndSilTim;          /*end silence time out in seconds*/
    float        MaxListenTim;       /*maximum listen time out in seconds*/
    char         CfgFile[S51_FILENAME_SIZ+1];/*tone definition file*/
    S51_TONE_NAME DetectSubset[S51_TDETSUBSET_MAX];/*tone detect subset*/
    S51_TONE_NAME TermSubset[S51_TDETSUBSET_MAX];/*terminate tone subset*/
}
S51_TDET_CFGVAL;
```

The fields are:

Tid..... ..... timeslot to connect to,

CbFun .... callback function for tone detector on this channel,

InitSilTim.... ..... time to wait for the first tone (seconds), must be >0.0 sec,

EndSilTim... ..... time to wait after first tone is detected (seconds), must be >0.0 sec

MaxListenTim ..... maximum listen time (seconds), must be >0.0 sec,

CfgFile .... the file name for the user defined tone(s)

DetectSubset ..... this field specifies a subset of tone names from the configuration file which will be detected. If it is not specified, all tones defined in the configuration file can be detected. The *DetectSubset.ToneName* field can contain one of two types of data:

- an ASCII string naming the tone. For example:
  `strcpy(TdetCfgVal.DetectSubset[0].ToneName,“busy”);`

  The list of tone names defined in the standard configuration file follows:

  ```
  busy              ringing
  dialtone          reorder           _intercept
  ```

- Or one-byte value in ToneName[0] as defined above (S51_*_TSET). The following example does the same thing as the previous one, that is, configures the detector to detect only the busy tone:
  `TdetCfgVal.DetectSubset[0].ToneName[0] = S51_BUSY_TONE;`

The list of tones #defines supported in the standard configuration file follows:

S51_BUSY_TONE,	S51_DIAL_TONE,	S51_CONFIRM_TONE,
S51_MSGWAIT_TONE,	S51_RINGBACK_TONE,	
S51_REORDER_TONE,	S51_INTERCEPT_TONE,	S51_OFFHOOK_TONE,

TermSubset.. .....this field specifies a set of tone names which will terminate the CPT detector action when detected. They are defined using the same way as the tones in the *DetectSubset* field.

5.18.2 Call Progress Tone Detector Commands

The **S51_TDET_CMDVAL data structure** is part of S51_CFGVAL argument to *S51_CmdChan()* function, and it provides values for the Call Progress Tone Detector Resource command.

```
typedef struct
{
   u_long              TdetCfgReq;         /*additional (optional) configuration*/
const S51_TGEN_CFGVAL* TdetCfgVal;
   u_char              InhTimers;          /*flag:!=0 inhibit the timers at start*/
   u_short             AtComplStartRes; /*resources to start when Tdet completes*/
   u_short             AtComplStopRes; /*resources to stop when Tdet completes*/
   u_short             AtComplEnableResTimers;/*enable timers when Tdet completes*/
   u_short             AtStopEnableResTimers;/*enable timers when Tdet stops*/
}
S51_TDET_CMDVAL;
```

The fields are:

TdetCfgReq..... ..... optional “last-second” configuration request,

TdetCfgVal..... ..... values associated with TdetCfgReq,

InhTimers .... if not 0, automatically inhibit the IST, EST, and MLT timers at start,

AtComplStartRes..... ..... resources to start when tone detector completes on its own,

AtComplStopRes..... ..... resources to stop when tone detector completes on its own,

AtComplEnableResTimers ..... resources whose timers should be enabled when the tdet resource completes,

AtStopEnableResTimers.... ..... resources whose timers should be enabled when the tdet resource is stopped.

5.18.3 Call Progress Tone Detector Resource State

The **S51_TDET_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the Call Progress Tone Detector Resource for this channel.

```
typedef struct
{
   u_long              FwVersion;
   u_long              CfgReq;      /*valid fields in CfgVal*/
   S51_TDET_CFGVAL     CfgVal;      /*current configuration */
   u_long              CmdReq;      /*last command completed (request)*/
   S51_TDET_CMDVAL     CmdVal;      /*last command completed (values)*/
   u_long              ComplCode;   /*configuration/command completion code*/
   u_long              TermReason;  /*why stopped*/
   u_long              ResErrCode;  /*detail error code, s51err.h*/
const char*            ErrMsg;      /*error message for last request*/
   char                LastToneName[S51_TONENAME_SIZ+1];/*last tone detected*/
   char                LastTone;    /*last tone as binary constant*/
}
S51_TDET_STAT;
```

The variables are:

FwVersion ..... version of the Call Progress Tone Detector firmware running on the board,
CfgReq ..... OR-ed value of all tone detector configuration requests completed on this channel,
CfgVal ..... all values used to configure the tone detector resource on this channel so far.
CmdReq ..... the last completed tone detector command on this channel,
CmdVal ..... values used for the last completed tone detector command,
ComplCode ..... completion code for the last completed tone detector request on this channel,
TermReason ..... termination reason for the last completed tone detector request,
ResErrCode ..... low-level error code provided by the tone detector resource, see s51err.h,
ErrMsg ..... error message for last command,
LastToneName ..... tone name detected in string,
LastTone ..... tone detected in binary.

5.18.4 Call Progress Tone Detector Example

The following code fragment allocates and configures a Call Progress Tone Detector resource and detects either a busy tone or a ringback tone in blocking (synch) mode.

```
S51_TDET_CFGVAL     TdetCfgVal;
S51_CFGVAL          CfgVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*allocate and configure Call Progress Tone Generator Resource*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(TdetCfgVal);
Req.TdetReq = S51_CONNTS_CFGREQ|S51_MAXLISTENT_CFGREQ|S51_DETECTSUBSET_CFGREQ;
TdetCfgVal.Tid = A5_MakeHtsTid(10);
TdetCfgVal.MaxListenTim = 20;
TdetCfgVal.DetectSubset[0].ToneName[0] = S51_BUSY_TONE;
TdetCfgVal.DetectSubset[1].ToneName[0] = S51_RINGBACK_TONE;
CfgVal.TdetCfgVal = &TdetCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

/*start & complete detection, command values are not needed and not provided*/
A5_Zerout(Req);
Req.TdetReq = S51_START_CMDREQ;
Ch = S51_CmdChan(Cid, &Req, NULL, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);
```

5.19 Connect Resource

This section describes the Connect Resource and the low-level data structures and constants used in configuring and using it. The Connect Resource connects two time slots, that is, it copies the signal from one time slot to another in real time. Note that the Connect Resource never stops on its own, but must be stopped explicitly by issuing a Stop or a Cancel request, or via Run Time Control (RTC).

It is not possible to "disconnect" the timeslot assigned to the Connect resource. The only way to truly disconnect the timeslot is to deallocate (S51_FREE_CFGREQ) the Connect resource. A bit more information is in “Programmers Note: Disconnecting the H.110 Timeslots (HTS):” on page 27.

5.19.1 Connect Resource Configuration

The second argument to the S51_CfgChan() function is a pointer to a S51_REQ structure. To allocate and configure the Connect Resource, the following constants can be OR-ed into the *S51_REQ.ConReq* field to indicate which configuration parameters are to be changed.

Request Bit Fields	S51_CON_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the callback function
S51_COPYDLYT_CFGREQ	CopyDelayTim	set the Copy Delay timer
S51_CONNTS_CFGREQ	RecvTid XmitTid	set the receive and transmit timeslots

The **S51_CON_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan* and it defines the configuration values for the Connect Resource:

```
typedef struct
{
   int          (*CbFun)();         /*callback function*/
   A5_TID       RecvTid;            /*receive time slot*/
   A5_TID       XmitTid;            /*transmit time slot*/
   u_long       CopyDelayTim;       /*S51_COPYDLYT_CFGREQ*/
}
S51_CON_CFGVAL;
```

The fields are:

CbFun..... .....callback function for Connect Resource on this channel,

RecvTid..... .....receive timeslot to connect to,

XmitTid..... .....transmit timeslot to connect to.

CopyDelayTim.....value of the Copy Delay Timer. This is the maximum amount of data that is discarded right after the module receives the start command. The default is 0.05 sec (50 msec). Setting it too low on a real (physical) timeslot can cause underrun errors. The value must be >= 0.0 sec. If the input timeslot RecvTid is a virtual timeslot, the API always forces this to be zero automatically.

5.19.2 Connect Resource Commands

The **S51_CON_CMDVAL data structure** is part of S51_CMDVAL argument to *S51_CmdChan()* function, and it provides values for the Connect Resource command. Because there are no values needed by the command to the connect resource, the only data that are in the command structure are the optional attached configuration data.

```
typedef struct
{
          u_long              ConCfgReq;    /*additional (optional) configuration*/
const     S51_CON_CFGVAL*     ConCfgVal;
}
S51_CON_CMDVAL;
```

The fields are:

ConfgReq.... optional “last-second” configuration request,

ConCfgVal .. values associated with ConCfgReq.

5.19.3 Connect Resource State

The **S51_CON_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the Connect Resource for this channel.

```
typedef struct
{
   u_long              FwVersion;
   u_long              CfgReq;       /*valid fields in CfgVal*/
   S51_CON_CFGVAL      CfgVal;       /*current configuration*/
   u_long              CmdReq;       /*last command completed (request)*/
   S51_CON_CMDVAL      CmdVal;       /*last command completed (values)*/
   u_long              ComplCode;    /*configuration/command completion code*/
   u_long              TermReason;   /*why stopped*/
   u_long              ResErrCode;   /*detail error code, s51err.h*/
   char*               ErrMsg;       /*error message from last request*/
}
S51_CON_STAT;
```

The variables are:

FwVersion... ..... version of the Connect Resource firmware running on the board,

CfgReq..... ..... OR-ed value of all Connect configuration requests completed on this channel,

CfgVal..... ..... all values used to configure the Connect resource on this channel so far,

CmdReq..... ..... the last completed Connect command on this channel,

CmdVal..... ..... values used for the last completed Connect command,

ComplCode. ..... completion code for the last completed Connect request on this channel,

TermReason ..... termination reason for the last completed Connect request,

ResErrCode. ..... low-level error code provided by the Connect Resource, see s51.h,

ErrMsg..... ..... error message for last command.

5.19.4 Connect Example

The following code fragment allocates and configures a Connect resource and connects virtual timeslot 10 on board 0 to a H.110 bus timeslot.

```
Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
...
S51_CON_CFGVAL      ConCfgVal;
S51_CFGVAL          CfgVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*allocate and configure Connect Resource*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(ConCfgVal);
Req.Con1Req = S51_CONNTS_CFGREQ;
ConCfgVal.XmitTid = A5_MakeHtsTid(20);
ConCfgVal.RecvTid = A5_MakeVtsTid(0, 10, V5_SRES_READ|V5_HOST_WRITE);
CfgVal.Con1CfgVal = &ConCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
      return(ErrMsg);

/*temporarily connect timeslots - no command values are needed*/
A5_Zerout(Req);
Req.Con1Req = S51_START_CMDREQ;

/*runs and blocks until stopped by RTC or by STOP/CANCEL command*/
Ch = S51_CmdChan(Cid, &Req, NULL, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
      return(ErrMsg);
```

5.20 Transcoder Resource

This section describes the Transcoder Resource and the low-level data structures and constants used in configuring and using it. The Transcoder Resource reads coded speech in a specific coding type from one timeslot, converts the coding to another type and writes the converted coding to another timeslot. Any data which is either read or written in a format other than the LSPS default speech data format[32] is usually read or written from/to a virtual timeslot (VTS). This is because the hardware timeslots (HTSs) are fixed in which speed and which format they can handle, so sending some incompatible data format to an HTS simply does not work. That is why the transcoder is usually used to translate the standard LSPS data format to a format required by certain applications on the host, using the VTS as a medium of getting the transcoded speech data from the board to the application.

Note that the Transcoder Resource works similar to the Connect Resource in that it never stops on its own, but must be stopped explicitly by a Stop or a Cancel request (explicitly or via RTC). Also note that depending on the type of translation being done, the transcoder may use a significant percentage of CPU time on the board and thus may severely limit the number of channels that can be handled by that board.

5.20.1 Transcoder Resource Configuration

The second argument to the S51_CfgChan() function is a pointer to a S51_REQ structure. To allocate and configure the Transcoder Resource, the following constants can be OR-ed into the *S51_REQ.CoderReq* field to indicate which configuration parameters are to be changed.

Request Bit Fields	S51_CODER_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the callback function
S51_CONNTS_CFGREQ	RecvTid XmitTid	set the receive and transmit timeslots
S51_ENCODING_CFGREQ	RecvEncoding XmitEncoding	set the receive and transmit speech encoding formats
S51_CODERBUFSIZE_CFGREQ	RecvBufSize XmitBufSize	set the receive and transmit buffers in the Transcoder module (in bytes, optional)
S51_COPYDLYT_CFGREQ	CopyDelayTim	set the Copy Delay timer (in msec, optional)

The **S51_CODER_CFGVAL data structure** is part of *S51_CFGVAL* argument to *S51_CfgChan* and it defines the configuration values for the Transcoder Resource:

```
typedef struct
{
   int          (*CbFun)();         /*callback function*/
   A5_TID       RecvTid;            /*receive time slot*/
   A5_TID       XmitTid;            /*transmit time slot*/
```

32. The default is 8-bit Mu-Law. This can be changed (system-wide) to 8-bit A-Law by changing a system configuration file /usr/lsps/cfg/customer/startup.ini.

```
    u_short      RecvEncoding;       /*input data format, S51_ENCODING_CFGREQ*
    u_short      XmitEncoding;       /*output data format*/
    u_long       CopyDelayTim;       /*S51_COPYDLYT_CFGREQ*/
    u_long       RecvBufSize;        /*in bytes, S51_CODERBUFSIZE_CFGREQ*
    u_long       XmitBufSize;        /*in bytes*/
}
S51_CODER_CFGVAL;
```

The fields are:

CbFun..... .....callback function for Transcoder Resource on this channel,

RecvTid..... .....receive timeslot to connect to,

XmitTid..... .....transmit timeslot to connect to.

RecvEncoding.....type of speech encoding used on the input timeslot, refer to the table below.

XmitEncoding.....type of speech encoding used on the output timeslot, refer to the table below.

CopyDelayTim.....value of the Copy Delay Timer in msec, it only applies if reading from an HTS. This is the maximum amount of data that is discarded right after the module receives the start command. The default is 0.05 sec (50 msec). Setting it too low on a real (physical) timeslot can cause underrun errors. The value must be >= 0.0 sec. If the input timeslot RecvTid is a virtual timeslot, the API always forces this to be zero automatically. Under normal situations he application should not be setting this value.

RecvBufSize .....size of the input data buffer in bytes. The default value is 4096 bytes and he allowed range is 1024 to 131072 bytes. Under normal situations the application should not need to be setting this value. Be aware that there is a limit on the total memory allocated to all of the VTS buffers and that using large buffers or many VTS could result in running out of this memory.

XmitBufSize .....size of the output data buffer in bytes. The default value is 4096 bytes and he allowed range is 1024 to 131072 bytes. Under normal situations the application should not need to be setting this value. Be aware that there is a limit on the total memory allocated to all of the VTS buffers and that using large buffers or many VTS could result in running out of this memory.

5.20.2 Supported Speech Coding Types

The following are the supported speech coding types for both, input and/or output, but only when reading and/or writing from virtual timeslots (VTSs):

S51_MULAW_CODE .... Mu-Law 64kBps

S51_ALAW_CODE..... ..... A-law 64kBps

S51_LINEAR_CODE..... ..... linear 128kBps

S51_LINEARLE_CODE..... ..... linear little-endian, 128kBps

S51_LINEAR8_CODE..... ..... linear 8-bit

S51_G711_MU_CODE..... ..... G.711 Mu-law 64kBps, bit-reversed

S51_G711_A_CODE..... ..... G.711 A-law 64kBps, bit reversed

S51_G726_ADPCM16_CODE... ..... G.726 ADPCM 16kBps encoding

S51_G726_ADPCM24_CODE... ..... G.726 ADPCM 24kBps encoding

S51_G726_ADPCM32_CODE... ..... G.726 ADPCM 32kBps encoding

S51_G726_ADPCM40_CODE... ..... G.726 ADPCM 40kBps encoding

S51_RHET_ADPCM32_CODE.. ..... Rhetorex ADPCM 32kBps encoding

S51_OMD_SUBBAND_CODE.. ..... OMD SubBandCode encoding

S51_LCCELP_CODE..... ..... low complexity CELP code

Note that the OKI ADPCM 32kBps (S51_OKI_ADPCM32_CODE) is not yet supported.

5.20.3 Transcoder Resource Commands

The **S51_CODER_CMDVAL data structure** is part of S51_CFGVAL argument to *S51_CmdChan()* function, and it provides values for the Transcoder Resource command.

```
typedef struct
{
   u_long              CoderCfgReq;        /*additional (optional) configuration*/
   S51_CODER_CFGVAL*   CoderCfgVal;
}
S51_CODER_CMDVAL;
```

The fields are:

CoderCfgReq ..... optional “last-second” configuration request,

CoderCfgVal ..... values associated with CoderCfgReq.

5.20.4 Transcoder Resource State

The **S51_CODER_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the Transcoder Resource for this channel.

```
typedef struct
{
   u_long              FwVersion;
   u_long              CfgReq;      /*valid fields in CfgVal*/
   S51_CODER_CFGVAL    CfgVal;      /*current configuration*/
   u_long              CmdReq;      /*last command completed (request)*/
   S51_CODER_CMDVAL    CmdVal;      /*last command completed (values)*/
   u_long              ComplCode;   /*configuration/command completion code*/
   u_long              TermReason;  /*why stopped*/
   u_long              ResErrCode;  /*detail error code, s51err.h*/
const char*            ErrMsg;      /*error message from last request*/
}
S51_CODER_STAT;
```

The variables are:

FwVersion... ..... version of the Transcoder Resource firmware running on the board,

CfgReq..... ..... OR-ed value of all Transcoder configuration requests completed on this channel,

CfgVal..... ..... all values used to configure the Transcoder resource on this channel so far,

CmdReq .... the last completed Transcoder command on this channel,

CmdVal .... values used for the last completed Transcoder command,

ComplCode. ..... completion code for the last completed Transcoder request on this channel,

TermReason ..... termination reason for the last completed Transcoder request,

ResErrCode. ..... low-level error code provided by the Transcoder Resource, see s51.h,

ErrMsg..... ..... error message for last command.

5.20.5 Transcoder Example

The following code fragment allocates a Transcoder resource and connects it to two virtual timeslots on the board. This host program then reads a file, feeding it’s contents into the transcoder, and then reads the output of the transcoder and writes it to another file. This code is modeled on the example *vts_loop.c* which is a part of the SDK.

```
FdRead = open(ReadFile, O_RDONLY);
FdWrite = open(WriteFile, O_RDWR|O_CREAT|O_TRUNC, 0644);
```

```
/*init API in block mode with no callback*/
S51Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
if (!S51Eqp)
       Cleanup(ErrMsg);
V5Eqp = V5_InitVts(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
if (!V5Eqp)
       Cleanup(ErrMsg);

/*create VTSs*/
TidIn = A5_MakeVtsTid(BdNum, VtsNumIn, V5_HOST_READ|V5_SRES_WRITE);
VtsIn = V5_CreateVts(TidIn, 0, 0, &ErrMsg);
if ((!VtsIn) || (VtsIn->ComplCode & A5_ANY_ERR))
       Cleanup(ErrMsg);

TidOut = A5_MakeVtsTid(BdNum, VtsNumOut, V5_HOST_WRITE|V5_SRES_READ);
VtsOut = V5_CreateVts(TidOut, 0, 0, &ErrMsg);
if ((!VtsOut) || (VtsOut->ComplCode & A5_ANY_ERR))
       Cleanup(ErrMsg);

/*allocate and configure transcoder channel*/
A5_Zerout(Req);
A5_Zerout(CfgVal);
A5_Zerout(CoderCfgVal);
Req.CoderReq = S51_CONNTS_CFGREQ|S51_ENCODING_CFGREQ;
CoderCfgVal.RecvTid = TidOut;
CoderCfgVal.XmitTid = TidIn;
CoderCfgVal.RecvEncoding = RecvEncoding;
CoderCfgVal.XmitEncoding = XmitEncoding;
CidS51 = S51_MakeCid(BdNum, PortNum);
ChS51 = S51_CfgChan(CidS51, &Req, &CfgVal, &ErrMsg);
if ((!ChS51) || (ChS51->ComplCode != A5_OK))
       Cleanup(ErrMsg);

/*in synch mode the transcoder would block on S51_CmdChan() until stopped*/
/*so run it in async mode*/
if (!S51_Init(A5_ASYNC_MOD, CbS51, &ErrMsg))
       Cleanup(ErrMsg);

/*start Transcoder*/
A5_Zerout(Req);
Req.CoderReq = S51_START_CMDREQ;
ChS51 = S51_CmdChan(CidS51, &Req, NULL, &ErrMsg);
if ((!ChS51) || (ChS51->ComplCode != A5_INPROG))
       Cleanup(ErrMsg);

/*wait for transcoder-running (connection up) event*/
ConnUp = 0;
while (!ConnUp)
       (void) S51_EventChan(-1, &EventCnt, &ErrMsg);

/*open VTSs*/
VtsIn = V5_OpenVts(TidIn, NULL, &ErrMsg);
VtsOut = V5_OpenVts(TidOut, NULL, &ErrMsg);

if ((!VtsIn) || (VtsIn->ComplCode & A5_ANY_ERR))
|| (!VtsOut) || (VtsOut->ComplCode & A5_ANY_ERR))
       Cleanup(ErrMsg);
```

```
/*switch VTS to asynch mode*/
V5Eqp = V5_InitVts(A5_ASYNC_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
if (!V5Eqp)
       Cleanup(ErrMsg);

/*enter read/write loop*/
V5Fd = V5Eqp->Fd;
Eof = 0;
XferBuf = malloc(XBufSize);
while (!VtsIn->DrainSpeech)
{
       /*first responsibility is to always handle any pending events*/
       /*also, if there is nothing I can do in the loop below, wait a while*/
       if ((VtsOut->AvailToWrite == 0) || (VtsIn->AvailToRead == 0))
              WaitTime = 3000;
       else
              WaitTime = 0;

       (void) V5_EventVts(WaitTime, &EventCnt, &ErrMsg);

       /*read transcoder VTS, write file*/
       XReadSize = XBufSize;
       if (XReadSize > VtsIn->AvailToRead)
              XReadSize = VtsIn->AvailToRead;

       if (XReadSize > 0)
       {
              VtsIn = V5_ReadVts(TidIn, XferBuf, XReadSize, &ErrMsg);
              if ((!VtsIn) || (VtsIn->ComplCode & A5_ANY_ERR))
                     Cleanup(ErrMsg);

              Ret = write(FdWrite, XferBuf, VtsIn->BytesRead);
              if (Ret != VtsIn->BytesRead)
                     Cleanup("file write error");
       }

       /*if EOF on input file detected previously, all data has been written*/
       /*skip reading file/writing VTS*/
       if (Eof)
              continue;

       /*read input file, write VTS*/
       WriteSize = XBufSize;
       if (WriteSize > VtsOut->AvailToWrite)
              WriteSize = VtsOut->AvailToWrite;

       if ((WriteSize > 0) && (!Eof))
       {
              /*read input file*/
              Ret = read(FdRead, XferBuf, WriteSize);

              /*end of input file*/
              if (Ret == 0)
              {
                     Eof = 1;
                     WriteSize = 0;
                     if (V5_DataEndVts(TidOut, V5_DRAIN, &ErrMsg) != A5_OK)
```

```
                        Cleanup(ErrMsg);
                }

                /*file read error*/
                else if (Ret < 0)
                        Cleanup("file read error");

                /*have some more data to write to VTS*/
                else
                        WriteSize = Ret;

                /*now write VTS*/
                if (WriteSize > 0)
                {
                        VtsOut = V5_WriteVts(TidOut, XferBuf, WriteSize, &ErrMsg);
                        if ((!VtsOut) || (VtsOut->ComplCode & A5_ANY_ERR))
                                Cleanup(ErrMsg);
                }
        }
    }

    Cleanup("");
    exit(0);
}

/*transcoder callback function - detects when transcoder has started*/
static int
CbS51(const S51_EVT* Evt)
{
    if (Evt->EventType & S51_INPROG_EVT)
        ConnUp = 1;

    return(1);
}

/*function to cleanup and exit the program*/
static void
Cleanup(const char* ComplMsg)
{
const     char*        ErrMsg;
          S51_REQ      Req;

    if (ComplMsg && *ComplMsg)
        fprintf(stderr, "Program terminating on error: %s\n", ComplMsg);
    if (FdRead >= 0)
        (void) close(FdRead);
    if (FdWrite >= 0)
        (void) close(FdWrite);
    if (XferBuf)
        free(XferBuf);

    if ((ChS51) && (S51_Init(A5_BLOCK_MOD, NULL, &ErrMsg)))
    {
        A5_Zerout(Req);
        Req.CoderReq = S51_STOP_CMDREQ;
        ChS51 = S51_CmdChan(CidS51, &Req, NULL, &ErrMsg);
        if ((!ChS51) || (ChS51->ComplCode != A5_OK))
            fprintf(stderr, "S51_CmdChan(STOP) failed, %s\n", ErrMsg);

        if (!S51_CloseChan(CidS51, &ErrMsg))
```

```
                fprintf(stderr, "S51_CloseChan() failed, %s\n", ErrMsg);
    }

    if (S51Eqp)
          S51_Term();
    if (!V5Eqp)
          return;

    (void) V5_InitVts(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
    if (VtsIn)
          (void) V5_CloseVts(TidIn, &ErrMsg);
    if (VtsOut)
          (void) V5_CloseVts(TidOut, &ErrMsg);
    if (VtsIn)
          (void) V5_DestroyVts(TidIn, &ErrMsg);
    if (VtsOut)
          (void) V5_DestroyVts(TidOut, &ErrMsg));

    V5_TermVts();
    return;
}
```

5.21 Text to Speech (TTS) Resource

This section describes the Text To Speech (TTS) Resource, the low-level data structures and constants used in configuring and using it. The Text To Speech Resource can play speech into a timeslot from input buffers or files. Among other things, the application can specify the voicing gender and the speaking rate of the TTS output.

The application can choose to use one of the two different TTS modules:

North-American English Only.. ..... This is also known as the "old TTS module." In this document it is abbreviated as **EN-TTS**. This is the TTS module which was the only one available up to Release 1.2. At this time its only advantage is that it provides higher per-board channel capacity of 90 channels of TTS on 6UB5 board. The disadvantages are: it provides NA English only, it has a slightly lower voice quality (as compared to the multilingual TTS) and it has a limited set of features. This TTS provides the following features:

- voice gender,
- speaking rate,
- index markers[33].

Multilingual TTS..... ..... This is also known as the "new" TTS module which was not available in Release 1.2 and before. In this document it is abbreviated as **ML-TTS**. It's limitation as compared to the old TTS software is a smaller channel capacity: 30 channels for the 6UB5 board. It provides the following features:

- Multiple Languages (North American English, Italian, German, Canadian French, European French, North American Spanish, Castilian Spanish),
- pause/resume,
- language and/or speaker,
- voice gender,
- speaking rate,
- index markers,
- volume control (initial and during play),
- speech boundary reporting and control (events/reports and skip forward during play),
- speech voicing parameters control.

These two TTS modules are mutually exclusive, that is, only one of them can be loaded on the board at one time. Which TTS module is loaded on the board is determined by which *Board Type* was used to boot the specific board. The LSPS software comes with board types which include either one of these TTS modules. The application developer also has an option to create a custom board type with the desired TTS module using the tool *mkboardtype*.

Note that although the application has no choice about which TTS module it allocates (that is determined by the board type), it is nevertheless informed about which type of TTS it allocated. This enables the application to be coded in such way that it can handle either type of TTS module. An attempt to set parameters applicable only to ML-TTS when EN-TTS was allocated results in an error.

It is not possible to "disconnect" the timeslot assigned to the TTS resource. The only way to truly disconnect the timeslot is to deallocate (S51_FREE_CFGREQ) the TTS resource. A bit more information is in “Programmers Note: Disconnecting the H.110 Timeslots (HTS):” on page 27.

33. Index markers are sometimes also called **bookmarks**.

Programmers Note: Mixing Play and TTS

The TTS Resource is completely independent of the Play Resource. This means that in order to mix playing of recordings and playing TTS, the application must either start and stop them consecutively one at a time, or it can set up a run-time control (RTC) condition which starts Play or TTS when the other type of play resource completes.

There are two other aspect of the TTS which are described in **TTS and ASR User's Guide** (a separate manual):

- The **Embedded TTS Speech Controls** are inserted in the text which is being spoken by the application and they control how or whether the text is translated and the characteristics of the generated speech.[34]
- The **Rule-Based Text Preprocessing** is done on the text before it is spoken. Some input text is converted or expanded so the result is more natural and more understandable.

Both of these two aspects are implemented in a completely different way for the North-American English (both in EN-TTS and ML-TTS) and for other languages (ML-TTS). Therefore they are described in their own separate chapters in the **TTS and ASR Users Guide**:

- Embedded Controls and Text Preprocessing for EN-TTS, and
- Embedded Controls and Text Preprocessing for ML-TTS

5.21.1 TTS Resource Configuration

The seconds argument to the *S51_CfgChan()* function is a pointer to a S51_REQ structure. To change the default configuration parameters for the TTS resource, the following constants can be OR-ed into the *S51_REQ.TtsReq* field to indicate which configuration parameters are to be changed. Note that it is not possible to "disconnect" the timeslot assigned to the TTS resource. The only way to truly disconnect the timeslot is to deallocate (S51_FREE_CFGREQ) the TTS resource. A bit more information is in “Programmers Note: Disconnecting the H.110 Timeslots (HTS):” on page 27.

Request Bit Fields	S51_TTS_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate TTS resource
S51_ALLOC_CFGREQ	<none>	allocate TTS resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the callback function
S51_CONNTS_CFGREQ	Tid	set the speech output timeslot
S51_TTSGENDER_CFGREQ	Gender	set to S51_MALE_TTS or S51_FEMALE_TTS
S51_TTSRATE_CFGREQ	Rate	set the initial rate of the speech
S51_TTSNAME_CFGREQ	TtsName	selects the language and/or speaker (ML-TTS only)
S51_VOLUME_CFGREQ	Volume	sets the initial volume (ML-TTS only)
S51_BOUNDLISTSIZ_CFGREQ	WordBoundaryEvtListSize MinorBoundaryEvtListSize MajorBoundaryEvtListSize	defines how many boundary markers are delivered in each boundary event; all of these must always be defined together (ML-TTS only)

34. Controlling the characteristics of the generated speech has nothing to do with text preprocessing

Request Bit Fields	S51_TTS_CFGVAL Members	Description
S51_SPEECHPARAMS_CFGREQ	SpeechAspiration SpeechFVTScale SpeechBVTScale SpeechTopLine SpeechRefLine SpeechBottomLine	defines the initial speech parameters; all of these must always be defined together (ML-TTS only)
S51_STARTDLYT_CFGREQ	StartDelayTim	defines a pause (in seconds) before the given text is spoken (ML-TTS only)

The *S51_CFGVAL* structure used by *S51_CfgChan*() contains a pointer to an S51_TTS_CFGVAL structure which defines configuration values for the TTS Resource.

```
typedef struct
{
   int          (*CbFun)(const void*);     /*TTS resource callback function*/
   A5_TID       Tid;          /*connect to this time slot*/
   u_short      Gender;       /*S51_MALE_TTS, S51_FEMALE_TTS*/
   u_short      Rate;         /*speaking rate*/

                              /*all following params require multi-lingual TTS*/
   float        StartDelayTim;/*start delay (sec)*/
   float        Volume;       /*linear: 0.0 to 2.0, default 1.0*/
   char         TtsName[S51_TTSNAME_SIZ+1];/*for S51_MULTILING_TTSTYP*/

                              /*S51_BNDLISTSIZE_CFGREQ: group of 3*/
   u_char       WordBoundaryEvtListSize;   /*0 to S51_BOUNDLIST_MAX*/
   u_char       MinorBoundaryEvtListSize;  /*0 to S51_BOUNDLIST_MAX*/
   u_char       MajorBoundaryEvtListSize;  /*0 to S51_BOUNDLIST_MAX*/

                              /*S51_SPEECHPARAMS_CFGREQ: group of 6*/
   float        SpeechAspiration;
   float        SpeechFVTScale;
   float        SpeechBVTScale;
   float        SpeechTopLine;
   float        SpeechRefLine;
   float        SpeechBottomLine;
}
S51_TTS_CFGVAL;
```

The following fields apply to both, EN-TTS and ML-TTS:

CbFun.....callback function for text to speech on this channel,

Tid .....transmit timeslot to connect to,

Gender.....flag to indicate use of male or female speaking voice[35], the settings are:

- S51_MALE_TTS
- S51_FEMALE_TTS

Rate .....the speech rate, The settings are:

- S51_MINRATE_TTS

35. Although this is generally true, for ML-TTS this can be used in a general way to select one of the two speakers/voices available in the selected language. At this time this is not done and the gender always selects a speaker/voice of the appropriate gender. Also if the gender is not specified, the default is the first voice found in the initialization file - usually male.

- S51_SLOWRATE_TTS
- S51_NORMRATE_TTS (default)
- S51_FASTRATE_TTS
- S51_MAXRATE_TTS

The following fields apply only to ML-TTS:

StartDelayTim..... silence (in seconds) which will be inserted before the ML-TTS starts speaking the text.

Volume..... ..... the initial speech volume, set on a linear scale from 0.0 to 2.0. The default volume is 1.0.

TtsName..... ..... this is used to select the language. The selected language must be part of the *Board Type*, that is, in the configuration of the software loaded on the board. A single *Board Type* could be set up to contain more than one language[36]. This field must be either undefined (a default language will be selected), or must contain one of the following:

"ENNA" ..... English - North American

"SPNA" ..... Spanish - North American

"SPCA"..... Spanish - Castilian

"FR"..... French - European

"FRCA" ..... French - Canadian

"IT"..... Italian

"GE" ..... German

WordBoundaryEvtListSize ..... Number of word boundary reports which should be grouped together and delivered in one S51_BOUNDARY_EVT event. The word boundary events are reported in the units of number of bytes from the beginning of the text being spoken. If set to 0, no word boundary reports will be generated.

MinorBoundaryEvtListSize..... Number of minor phrase boundary reports which should be grouped together and delivered in one S51_BOUNDARY_EVT event. The minor phrase boundary events are reported in the units of number of bytes from the beginning of the text being spoken. If set to 0, no minor phrase boundary reports will be generated.

MajorBoundaryEvtListSize..... Number of major phrase (sentence) boundary reports which should be grouped together and delivered in one S51_BOUNDARY_EVT event. The major phrase boundary events are reported in the units of number of bytes from the beginning of the text being spoken. If set to 0, no major phrase boundary reports will be generated.

SpeechAspiration... ..... Sets the breathiness of the voice. The value of 0.0 is considered normal (it is also the default). The valid range is 0.0 to 60.0.

SpeechFVTScale.... ..... Sets the front vocal tract size. Increasing values create a larger resonant chamber, while decreasing values create a smaller one. The value is 1.0 considered the normal size (it is also the default value). Specify a value within the range 0.04 to 1.5.

SpeechBVTScale.... ..... Sets the back vocal tract size. Increasing values create a larger resonant chamber, while decreasing values create a smaller one. The value 1.0 is considered the normal size (it is also the default value). Specify a value from within the range 0.04 to 1.5.

SpeechTopLine..... ..... Sets the maximum pitch for the LTTS audio output (in Hz). The value 160 is considered normal (it is also the default value). Specify a value within the range 1 to 500 Hz.

SpeechRefLine..... ..... Sets the average pitch value for the LTTS audio output (in Hz). The value 90 is considered normal (it is also the default value). Specify a value within the range 1 to 500 Hz.

SpeechBottomLine. ..... Sets the minimum pitch for the LTTS audio output (in Hz). The value 65 is considered normal (it is also the default value). Specify a value within the range 1 to 500 Hz.

36. By creating a custom board type with *mkboardtype* tool. The prepackaged board types contain only a single language per board type.

5.21.2 TTS Resource Commands

Beside the standard commands of S51_START_CMDREQ, S51_READY_CMDREQ, S51_CANCEL_CMDREQ, S51_STOP_CMDREQ and S51_ABORT_CMDREQ, the **ML-TTS** resource allows certain other commands to be issued during play. They can be issued either while operating in an asynchronous mode, or from a callback function in any mode.

S51_PAUSE_CMDREQ..... ..... pause TTS,

S51_RESUME_CMDREQ..... ..... resume paused TTS,

S51_SKIPFWD_CMDREQ..... ..... skip forward to the specified (N-th) boundary marker,

S51_SETVOLUME_CMDREQ.. ..... change the volume to the specified value.

The **S51_TTS_CMDVAL data structure** is part of S51_CFGVAL argument to *S51_CmdChan()* function and it provides values for the Text To Speech resource command. Note that all the fields must be set to valid values for every call to S51_CmdChan(). Fields which are not applicable to a particular function call must be set to 0 (zeroes).

```
typedef struct
{
   char               PlayFile[S51_FILENAME_SIZ+1];/*file name*/
const char*           PlayBuf;     /*or buffer address*/
   u_long             PlayBufSize;  /*buffer size*/
   float              StartDelayTim;/*wait before play, in sec*/
}
S51_PLAY_SET;
```

There are several important consideration that the programmer must keep in mind when setting up and using the S51_PLAY_SET structure:

- If the *S51_PLAY_SET.PlayFile* contains a relative pathname, the API converts it into an absolute pathname and overwrites this field. In another words, upon return from *S51_CmdChan()*, the *S51_PLAY_SET* structure owned by the application may have been modified by the API by replacing all relative file pathnames with absolute pathnames.
- If the *S51_PLAY_SET.PlayBuf* is specified, the contents of this buffer must remain unchanged (and valid) until the TTS Resource reports completion, that is, until the TTS Resource completion code is no longer A5_INPROG
- The *S51_PLAY_SET* data structure passed to *S51_CmdChan()* can be destroyed upon return from *S51_CmdChan()*, because the API make a copy of it and uses this copy while the play is in progress.

```
typedef struct
{
   u_long             TtsCfgReq;   /*additional (optional) config*/
   S51_TTS_CFGVAL*    TtsCfgVal;
   S51_PLAY_SET*      PlaySet;     /*set of files or buffers to use*/
   u_short            SetSize;     /*size of PlaySet*/\

                                   /*all the following require multilingual TTS*/
   char               OutputFile[S51_FILENAME_SIZ+1];/*supersedes timeslot (Tid)*/

                                   /*for S51_SETVOLUME_CMDREQ only*/
   float              Volume;      /*adjust volume 0.0 to 2.0*/

                                   /*for S51_SKIPFWD_CMDREQ only*/
                                   /*S51_WORD_BND, S51_MINOR_BND, S51_MAJOR_BND*/
   u_short            SkipFwdBoundaryType;
   u_long             SkipFwdBoundaryCount;
```

```
                        /*Run-time control: OR-ed resource names*/
    u_short             AtComplStartRes;/*resources to start when TTS completes*/
    u_short             AtComplStopRes;/*resources to stop when TTS completes*/
    u_short             AtComplEnableResTimers;/*enable timers when TTS completes*/
    u_short             AtStopEnableResTimers;/*enable timers when TTS stops*/
}
S51_TTS_CMDVAL;
```

The fields are:

PlayFile .... name of the play file,

PlayBuf .... address of memory buffer[37] to play from instead of a file,[38]

PlayBufSize .... size of PlayBuf or zero for PlayFile,

StartDelayTim..... ..... delay before starting to play this file or buffer (milliseconds), 0 to disable the timer,

TtsCfgReq..... ..... Optional “last-second” configuration request.

TtsCfgVal..... ..... Values associated with *TtsCfgReq*.

PlaySet .... Set of files or buffers to be played by TTS.

SetSize .... Specifies the size of the *PlaySet*.

OutputFile..... ..... If specified, it supersedes the timeslot and all generated speech is stored in this file. This field is referenced only by the S51_START_CMDREQ and S51_READY_CMDREQ commands.

Volume..... ..... Change the volume during play to this new value (linear 0.0 to 2.0). This field is referenced only by the S51_SETVOLUME_CMDREQ command when using multi-lingual TTS only (ML-TTS).

SkipFwdBoundaryType .... Boundary type referenced in the *SkipFwdBoundaryCount* field (below). Applicable to MLTTS only.

- S51_WORD_BND word boundary,
- S51_MINOR_BND minor phrase boundary,
- S51_MAJOR_BND major phrase (sentence) boundary.

SkipFwdBoundaryCount .. Skip forward (during play) this number of boundaries of type *SkipFwdBoundaryType*. This is a relative count from the next boundary of this type to be spoken. This field is referenced only by the S51_SKIPFWD_CMDREQ command and is applicable to MLTTS only.

AtComplStartRes..... ..... Resources to start when TTS completes on its own.

AtComplStopRes..... ..... Resources to stop when TTS completes on its own.

AtComplEnableResTimers ..... Resources whose timers should be enabled when the TTS Resource completes.

AtStopEnableResTimers... ..... Resources whose timers should be enabled when the TTS Resource is stopped.

So for **example**, if the ML-TTS was given the text "The quick brown fox jumped over a lazy dog." and is speaking the word "jumped" when the application requests to skip 3 word boundaries, the ML-TTS will cut off the word "jumped" in the middle and immediately start speaking the word "dog". If, on the other hand, in the same situation the application requested to skip 100 word boundaries, the ML-TTS would cut off the currently spoken word "jumped" and would immediately complete the application’s play request.

37. To use a memory buffer instead of a play file, the PlayFile must be set to zeroes (NULL string) and both, the PlayBuf and PlayBufSize must be set.

38. This is an address of host memory, which was obtained either by static allocation, by dynamic allocation (malloc) or by memory mapping a file. The application must not change the contents of the buffer while the TTS Resource is playing it.

5.21.3 TTS Resource State

The **S51_TTS_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the Text To Speech Resource for this channel.

```
typedef struct
{
   u_long              FwVersion;
   u_long              CfgReq;      /*valid fields in CfgVal*/
   S51_TTS_CFGVAL      CfgVal;      /*current configuration*/
   u_long              CmdReq;      /*last command completed (request)*/
   S51_TTS_CMDVAL      CmdVal;      /*last command completed (values)*/
   u_long              ComplCode;   /*configuration/command completion code*/
   u_long              TermReason;  /*why stopped*/
   u_long              ResErrCode;  /*detail error code, s51err.h*/
   long                IndexMarker; /*last index marker received*/
   u_long              TtsType;     /*S51_*_TTSTYP, ML-TTS only*/
   u_long              LastWordBoundary;   /*bytes offset, ML-TTS only*/
   u_long              LastMinorBoundary;  /*bytes offset, ML-TTS only*/
   u_long              LastMajorBoundary;  /*bytes offset, ML-TTS only*/
const char*            ErrMsg;             /*error message for last request*/
}
S51_TTS_STAT;
```

The variables are:

FwVersion ... .version of the TTS Resource firmware running on the board,

CfgReq ....OR-ed value of all TTS configuration requests completed on this channel,

CfgVal ....all values used to configure the TTS Resource on this channel so far,

CmdReq ....the last completed TTS command on this channel,

CmdVal..... .....values used for the last completed TTS command,

ComplCode . .....completion code for the last completed TTS request on this channel,

TermReason. .....termination reason for the last completed Tts request,

ResErrCode . .....low-level error code provided by the TTS Resource, see s51err.h,

IndexMarker .....the index marker for the last played index[39],

TtsType..... .....type of TTS module allocated, determined automatically by the "board type" (firmware loaded) of the speech board being used.

- S51_ENNA_TTSTYP
- S51_MULTILING_TTSTYP

LastWordBoundary. .....last **word** boundary spoken (in bytes from the beginning of text); set only if word boundary reports were requested,

LastMinorBoundary .....last **minor phrase** boundary spoken (in bytes from the beginning of text); set only if minor phrase boundary reports were requested,

LastWordBoundary. .....last **major phrase** boundary spoken (in bytes from the beginning of text); set only if major phrase boundary reports were requested,

ErrMsg ....text error message associated with the completion code.

39. Refer to the **TTS and ASR User's Guide** on how to insert index markers into the text.

5.21.4 TTS Events

Besides the standard events, the TTS resource can get the S51_BOUNDARY_EVT event (ML-TTS only). This event is generated only if the application requested boundary reports using the S51_BOUNDLISTSIZ_CFGREQ TTS configuration request. This type of event contains the following:

Cid .... channel ID of the channel which generated the event

Ch..... ..... pointer to the channel structure

EventType ... . S51_INPROG_EVT | S51_BOUNDARY_EVT

ComplCode . ..... A5_UNDEF

TermReason ..... S51_UNDEF_TRM

ResErrCode 0

ErrMsg .... ""

BoundaryType..... Boundary type being reported:

- S51_WORD_BND word boundary
- S51_MINOR_BND minor phrase boundary
- S51_MAJOR_BND major phrase (sentence) boundary

BoundaryListSize .. size of the *BoundaryList* array,

BoundaryList ..... array of boundary reports in increasing order, the last boundary in the list is the last one spoken; each reported boundary is an absolute offset (in bytes) from the beginning of the text.

5.21.5 TTS Example

The following code fragment allocates and configures a TTS resource, and starts speech. Because it does not specify the *S51_TTSNAME_CFGREQ/TtsName*, it uses the default (or only) language.

```
Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
...
S51_TTS_CFGVAL      TtsCfgVal;
S51_CFGVAL          CfgVal;
S51_TTS_CMDVAL      TtsCmdVal;
S51_CMDVAL          CmdVal;
S51_REQ             Req;
S51_CID             Cid;
S51_PLAY_SET        TtsSet[1];
char*               ErrMsg;
S51_CH*             Ch;

/*use board 0, channel 1*/
Cid = S51_MakeCid(0, 1);

/*allocate and configure TTS Resource*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(TtsCfgVal);
Req.TtsReq = S51_CONNTS_CFGREQ|S51_TTSGENDER_CFGREQ;
TtsCfgVal.Tid = A5_MakeHtsTid(20);
TtsCfgVal.Gender = S51_FEMALE_TTS;
CfgVal.TtsCfgVal = &TtsCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

/*play one file*/
A5_Zerout(Req);
A5_Zerout(TtsCmdVal);
A5_Zerout(CmdVal);
A5_Zerout(TtsSet);
CmdVal.TtsVal = &TtsCmdVal;
strcpy(TtsSet[0].PlayFile, “foo.txt”);
TtsCmdVal.PlaySet = TtsSet;
TtsCmdVal.SetSize = 1;
CmdVal.TtsVal = &TtsCmdVal;
Req.TtsReq = S51_START_CMDREQ;
Ch = S51_CmdChan(Cid, &Req, &CmdVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);
```

5.22 The Automatic Speech Recognition (ASR) Resource

The Automatic Speech Recognition Resource (ASR) listens to speech input on a timeslot and returns information indicating the content of the speech input. It also contains a speech energy detector which is used to discriminate silence from speech, that is, to detect the beginning and end of speech. This section describes the low-level data structures for configuring and using ASR Resources[40] on a channel. There are different flavors of the ASR resource. This section covers only the programming interfaces to the ASR resource. How to configure the different ASR types, what they do and how to use them is covered in **TTS and ASR User's Guide**.

5.22.1 ASR Resource Configuration

The second argument to the *S51_CfgChan()* function is a pointer to a *S51_REQ* structure. To change the default configuration parameters for the ASR resource, the following constants can be OR-ed into the *S51_REQ.AsrReqN* field to indicate which configuration parameters are to be changed, and the corresponding values need to be provided in the *S51_ASR_CFGVAL* structure. Note that some of the parameters are marked by '*'. These parameters require that the ASRType and ASRName parameters (S51_ASRTYPE_CFGREQ) be configured at the time when this parameter is being configured.

Request Bit Fields	S51_ASR_CFGVAL Members	Description
S51_FREE_CFGREQ	<none>	deallocate resource
S51_ALLOC_CFGREQ	<none>	allocate resource (this is implicit when referring to an unallocated resource)
S51_ABORT_CFGREQ	<none>	abort current request
S51_CBFUN_CFGREQ	CbFun	set the call back function
S51_CONNTS_CFGREQ	Tid	set the timeslot
S51_RESMODE_CFGREQ	ResMode	set the ASR resource mode
S51_INITSILT_CFGREQ	InitSilTim	set the initial silence time in seconds
S51_ENDSILT_CFGREQ	EndSilTim	set the ending silence timer in seconds
S51_MAXLISTENT_CFGREQ	MaxListenTim	set the maximum listen time in seconds
S51_GAPT_CFGREQ	GapTim	set the time out between words in seconds
S51_MINUTTERT_CFGREQ	MinUtterTim	set the minimum utterance time in seconds
S51_MAXUTTERT_CFGREQ	MaxUtterTim	set the maximum utterance time in seconds
S51_STARTDLYT_CFGREQ	StartDelayTim	set the time to delay the ASR detection in seconds
S51_REPTOPN_CFGREQ*	ReportTopN	report the ASR top 1, 2 or 3 result candidates
S51_REP1ST_CFGREQ*	Report1stFlg	only in the Connect Digit mode, if set to 1, report the first recognition
S51_EC_CFGREQ*	RefTid EchoCancelFlg	turn on/off built-in software echo cancellation

40. Up to two ASR Resources may be used on a channel at any time.

Request Bit Fields	S51_ASR_CFGVAL Members	Description
S51_EFLOOR_CFGREQ	Efloor	set the energy detect floor in dBm
S51_REJECT_CFGREQ*	RejectThr	set the rejection threshold parameter
S51_CONFLEVEL_CFGREQ	ConfidenceLevel	set the confidence level parameter (alternate way of specifying the rejection threshold)
S51_SENSITIVITY_CFGREQ	Sensitivity	set the sensitivity level (alternate way of specifying the energy floor)
S51_SPEEDVSACC_CFGREQ	SpeedVsAccuracy	set the trade-off between speed and accuracy
S51_JSGFPATH_CFGREQ	JsgfPath	set the view path for finding the JSGF grammar files
S51_STRPARAM_CFGREQ	StringParam	set the (undocumented) special feature or debugging parameter
S51_ASRTYPE_CFGREQ	AsrType, AsrName, GramFile: if S51_SW_ASRTYP or S51_NAMERECOG_ASRTYP, for collision detection in S51_SWTRN_ASRTYP or S51_NAMETRAIN_ASRTYP GramInfo: only if using the S51_GC_ASRTYP. ExtGramInfo (optional): only if using the S51_GC_ASRTYPE.	set the ASR type, ASR Name, and if necessary, grammar files.
S51_TAGGEDCNTN_CFGREQ	TaggedContentFlg	turn on (1) or off (0, default) the filtering of delivered result based on whether the recognized word has a tag associated with it.
S51_CNTNLESSWORD_CFGREQ	ContentLessWords	specify a file name which contain a list of words which should be filtered out from the final result of ASR recognition.

The **S51_ASR_CFGVAL data structure** is a part of *S51_CFGVAL* argument to *S51_CfgChan()* and it defines the configuration parameters for the ASR Resource.

```
typedef struct
{
   int          (*CbFun)();         /*ASR Resource callback function*/
   A5_TID       Tid;                /*connect to this timeslot*/
   A5_TID       RefTid;             /*for EchoCancelFlg==1 only*/
   u_short      AsrType;            /*ASR Type: IW, CD, GC, etc.*/
   char         AsrName[S51_ASRNAME_SIZ+1];/*label in the asr.ini file*/
   char         GramFile[S51_FILENAME_SIZ+1];/*grammar file (subword tables)*/
   S51_GRAM_SET GramInfo[S51_GRAMINFO_MAX];/*grammar based grammar files */
   u_short      GramSetSize;        /*number of grammar files in graminfo*/
   short        ExtGramSetSize;     /*size of ExtGramInfo array*/
   S51_GRAM_SET*ExtGramInfo;        /*additional grammar info array*/
   u_char       ReportTopN;         /*CD, GC, Name: report top 1 to 3 candidates*/
   u_char       Report1stFlg;       /*report first recognition*/
```

```
    u_char       EchoCancelFlg;      /*turn on/off software echo cancellation*/
    u_char       TaggedContentFlg;   /*only return tagged results*/
    u_char       ResMode;            /*ASR resource mode*/
    float        InitSilTim;         /*initial silence time*/
    float        EndSilTim;          /*ending silence time*/
    float        GapTim;             /*inter-attempt time*/
    float        MinUtterTim;        /*minimum utterance time*/
    float        MaxUtterTim;        /*maximum utterance time*/
    float        MaxListenTim;       /*maximum listen timeout*/
    float        StartDelayTim;      /*start delay time*/
    float        Efloor;             /*energy detect floor in dBm*/
    float        RejectThr;          /*rejection threshold parameter*/
    float        ConfidenceLevel;/*normalized rejection threshold 0.0-1.0*/
    float        Sensitivity;/*normalized energy floor 0.0-1.0*/
    float        SpeedVsAccuracy;/*normalized trade-off parameter 0.0-1.0*/
    char         JsgfPath[S51_JSGFPATH_SIZ+1];/*JSGF view path*/
    char         StringParam[S51_STRPARAM_SIZ+1];/*reserved & undocumented*/
    char         ContentLessWords[S51_FILENAME_SIZ+1];/*do not return these words*/
}
S51_ASR_CFGVAL;
```

The variables are:

CbFun .... Callback function for ASR on this channel.

Tid..... ..... Timeslot on which to listen (required).

RefTid .... Reference timeslot for software echo cancellation, used only for request S51_EC_CFGREQ and when *EchoCancelFlag* is set to 1.

AsrType..... ..... General type of ASR (connected digit, subword, etc.), required, more description below,

AsrName..... ..... defines the ASR model, grammar, and the “lex” file (recognized text strings), required, more description below.

GramFile..... ..... File containing subword tables (obsolete), or file containing the collision table for the training phase of the Name-ASR type of recognition.

GramInfo..... ..... Used only for grammar-based recognition. This data structure is used to specify the grammars to "plug into" or "drop into" the base grammar. The grammars are identified by the grammar name (*FieldName*), the lexical file name (*LexFile*), the grammar file (*GramFile*), and the grammar type (*GramType*). Each of these names is an ASCII strings that the application must specify when a new grammar is used. See **TTS and ASR User’s Guid**e for a more detailed explanation. The current supported GramTypes are: "jsgf", "*gc*", "*namelist*" and “*null*”.

If the grammar and/or lex file has a relative pathname (does not start with a ’/’), the API converts this file name to an absolute pathname using the current working directory as the base.

For the 'jsgf' type of grammars, the API allows the application to use the JSGF method of specifying the pathnames of grammar and lex files. This format uses the '.' (dot) as a directory separator. The API recognizes this type of pathname and verifies the existence of the specified grammar and lex files which are specified in this format. A "dot-format" pathname which does not start with a dot is converted to a standard pathname by prepending the current working directory to the pathname. To specify an absolute pathname, the "dot-format" pathname must start with a dot.

More information on dynamically loading the ASR grammars is in “Passing the Drop-In Grammar Information to the ASR” on page 122.

GramSetSize ..... Number of GramInfo items are used (the maximum is 8).

ExtGramInfo..... This is used as an extension of the *GramInfo* field (above), and therefore the same information applies here. The difference from the *GramInfo* field is that this is a pointer to an array of S51_GRAM_SET data structure. This array is outside of the API’s memory space, and must be con-

structed in the application's memory space. This memory space can be released after the S51_CfgChan() function returns, that is, the array is copied into an API-internal memory. At this time, the maximum size of this array is limited to 100 members. The drop-in grammars can be specified in either or in both arrays: in the *GramInfo* array and in the *ExtGramInfo* array. This means that the absolute number of drop-in grammars that can be specified is 108.

More information on dynamically loading the ASR grammars is in "Passing the Drop-In Grammar Information to the ASR" on page 122.

ExtGramSetSize .....size of the *ExtGramInfo* array (see above).

ReportTopN. .....Report top 1 to 3 candidates for CD, GC or Name ASR (1 to 3). Requires AsrType and AsrName to be also set.

Report1stFlg .....If !=0, the ASR reports first recognition, that is, the ASR recognized what could be the first part of the recognized final result. An S51_RECG1ST_EVT event will be delivered to the application. Requires AsrType and AsrName to be also set.

EchoCancelFlg.....Turns on (1) or off (0) the built-in software echo cancellation (default is off). Requires AsrType and AsrName to be also set.

TaggedContentFlg.. .....This is a boolean (flag, value 0 or 1) which causes only tagged words to be considered as having content. If this flag is turned on (value of '1'), words which are not explicitly tagged (usually represented by "_0_" or "__" in the returned tag string) are considered to have no content. In this situation, contentless words (and the associated empty tags) are not returned to the application. For example if the default text and tag strings returned from ASR are:

text: "please call my mother at home"

tags: "_{CALL}_ _{MOM}_ _{HOME}"

With the content flag turned on, the returned strings are:

text: "call mother home"

tags: "{CALL}_{MOM}_{HOME}"

Note that this can be dangerous if used incorrectly. If the grammar does not use tags and the *TaggedContentFlg_flag* is turned on, all words are content-less and so no recognized words or tags will be returned.

ResMode ....ASR resource mode:

- S51_PUREDIGIT_RMD sets the CD ASR to deliver only digits '0' through '9'.
- S51_REL2_RMD changes the way that null (empty) tags are reported to the method used in releases previous to 3.0.
- S51_REL3_RMD changes the way that null (empty) tags are reported to the method used in release 3.0 and later.

InitSilTim ....Maximum length of initial silence, when exceeded recognition never starts (seconds), must be >0.2 sec. The default value is 2.0 sec.

EndSilTim ... .After recognition starts, length of silence which stops the recognition (seconds), must be >0.1 sec. The default value is 500 sec.

GapTim ....Length of low energy between utterances which defines the utterance end (seconds), must be >0.05 sec. The default value is 0.75 sec.

MinUtterTim .....Minimum utterance length (seconds), must be >0.05 sec. The default value is 0.12 sec.

MaxUtterTim.....Maximum utterance length(seconds), must be >1.0 sec. The default value is 20 sec.

MaxListenTim.....Maximum overall recognition time (seconds), must be >1.0 sec. The default value is 21 sec.

StartDelayTim.....Start delay time in seconds, value of 0 disables the timer, must be >= 0.0 sec. The default value is 0 sec.

Efloor .... Energy level (dBm) which defines "silence" before recognition starts. The default value for the Efloor parameter is -52 dBm. An alternate way of specifying the energy detector sensitivity is through the *Sensitivity* field.

RejectThr .... Rejection threshold. It is closely related to the *S51_RECG_WORD.Confidence* value delivered with the recognition results. Any recognition result which has the *Confidence* value less than the *RejectThr* value is rejected, it is not considered to be valid and it is not reported to the application. Requires AsrType and AsrName to be also set. An alternate way of specifying which results should be reported and which rejected is via the *ConfidenceLevel* parameter.

ConfidenceLevel.... this is an alternative method of specifying the *RejectThr* parameter. It is a unit-less number normalized to a range of 0.0 to 1.0, with the default value of 0.5. It performs a function similar to the *RejectThr* ASR parameter, that is, it is used by the ASR resource to determine whether a given recognized utterance should be reported as a recognition, or should be "rejected" and the ASR should continue with the recognition. If the application specifies both parameters, the *ConfidenceLevel* and the *RejectThr*, the API reports an error. The ASR resource uses either the value of *ConfidenceLevel* or of *RejectThr*, whichever was received last from the API.

Sensitivity ... . This is an alternative method of specifying the *Efloor* parameter. It is a unit-less number normalized to a range of 0.0 to 1.0, with the default value of 0.5. It performs a function similar to the *Efloor* ASR parameter, that is, it is used by the ASR resource to distinguish the transition from silence to speech. If the application specifies both parameters, the *Sensitivity* and the *Efloor*, the API reports an error. The ASR resource uses either the value of *Sensitivity* or of *Efloor*, whichever was received last from the API.

SpeedVsAccuracy.. This is a unit-less number normalized to a range of 0.0 to 1.0, with the default value of 0.5. It serves as a hint to the ASR resource to make certain trade-offs between the processing speed and accuracy of recognition.

JsgfPath..... ..... This parameter contains a set of directories in which the ASR resource should look for grammar files. This is one way to send the value of the JSGF path to the ASR resource. The other method is via the environmental variable JSGFPATH. More details are in “JSGF Path” on page 129.

StringParam ..... This is arbitrary string which can be passed from the application to the ASR resource. Some of it's uses are to help debugging or to temporarily implement some future special feature in the ASR under normal situations, this parameter should not be used by applications.

ContentLessWords. ..... this is a name of a file which contains a list of words which the ASR should consider as having no content. Any words contained in this file will be left out of the returned text string. Using the example shown in the *TaggedContentFlag* field (above), if the words *"please"*, *"my"* and *"at"* were placed in a file which was passed down via this parameter, the resulting text and tag strings would look the same as if the *TaggedContentFlg* parameter was set.

The maximum length of this file name is the same as for any other file names passed through the speech API: S51_FILENAME_SIZ (127 bytes). As for all other API files, the API shall convert any relative file pathname to an absolute file pathname using the current working directory. The internal format of the file is as follows:

- ASCII text,
- one word per line, lines are delimited by a newline character '\n',
- case is not significant, that is, upper and lower case character have the same meaning.

The type of ASR is specified during the ASR resource configuration by using a combination of two fields in the structure S51_ASR_CFGVAL: *AsrType* and *AsrName*. More details can be found in the **TTS and ASR User’s Guide** (a separate book).

AsrType..... defines the “high level” type of ASR, the following are currently supported:

- S51_CD_ASRTYP connected digit
- S51_IW_ASRTYP isolated word
- S51_SW_ASRTYP subword

- S51_SWTRN_ASRTYP Subword ASR training phase
- S51_NAMETRAIN_ASRTYP Name-recognition training phase
- S51_GC_ASRTYP grammar-based ASR

AsrName .....defines the set of ASR parameters to use. It is a text string and it must match a label in the ASR initialization file. This label defines a group of parameters used by the ASR software.

5.22.2 Passing the Drop-In Grammar Information to the ASR

When the application "drops-in"[41] grammars during run-time, it must pass the information about the grammars to the ASR resource. The grammar information is passed using the S51_GRAM_SET data structures, which are part of the ASR configuration data structure S51_ASR_CFGVAL.

```
typedef   struct
{
   char         FieldName[S51_ASRNAME_SIZ+1];/*gram field name*/
   char         GramFile[S51_FILENAME_SIZ+1];/*gram file name*/
   char         LexFile[S51_FILENAME_SIZ+1];/*lexicon file name*/
   char         GramType[S51_ASRNAME_SIZ+1];/*grammar type */
}
S51_GRAM_SET;
```

There are two arrays of these structures: *GramInfo* and *ExtGramInfo*. The rules for error checking of the contents of both, the basic *GramInfo* structure and the extended *ExtGramInfo* structure are the same. For example, the API automatically converts relative pathnames to absolute pathnames, applies the *JSGF Path* to the pathnames, checks the existence of the specified grammar files, etc. Also note that there is no restriction on which data structure the application may use in any given situation. For example the application which needs to download only 3 grammars may choose to use only the *ExtGramInfo* interface and not to use the basic *GramInfo* interface at all. Or the application may use both interfaces at the same time. The only difference between the *GramInfo* and *ExtGramInfo* is that the information in *GramInfo* is downloaded to the board before the *ExtGramInfo* information.

If the application needs to load 8 or fewer grammars, it would normally use the *GramInfo* array.

```
strcpy(AsrCfg.GramInfo[0].FieldName, "DYNAMIC");
strcpy(AsrCfg.GramInfo[0].GramFile, "Main.gram");
strcpy(AsrCfg.GramInfo[0].LexFile, "Main.lex");
strcpy(AsrCfg.GramInfo[0].GramType, "jsgf");
strcpy(AsrCfg.GramInfo[1].FieldName, "MAINMENU");
strcpy(AsrCfg.GramInfo[1].GramFile, "MainMenu.gram");
strcpy(AsrCfg.GramInfo[1].LexFile, "MainMenu.lex");
strcpy(AsrCfg.GramInfo[1].GramType, "jsgf");
AsrCfg.GramSetSize = 2;
AsrCfg.ExtGramSetSize = 0;
```

Only when the application must load more than 8 grammars, it must use the *ExtGramInfo*.

```
S51_GRAM_SET     GramSet[30];
...
A5_Zerout(GramSet);
...
strcpy(GramSet[28].FieldName, "EMAILMENU");
strcpy(GramSet[28].GramFile, "EmailMenu.gram");
strcpy(GramSet[28].LexFile, "EmailMenu.lex");
strcpy(GramSet[28].GramType, "jsgf");
strcpy(GramSet[29].FieldName, "VMAILMENU");
```

41. In many applications it is sufficient (and more efficient) to preload the grammars during the speech board boot. This is done by creating custom board types using the *mkboardtype* command.

```
strcpy(GramSet[29].GramFile, "VmailMenu.gram");
strcpy(GramSet[29].LexFile, "VmailMenu.lex");
strcpy(GramSet[29].GramType, "jsgf");
AsrCfg.ExtGramInfo = GramSet;
AsrCfg.ExtGramSetSize = 30;
AsrCfg.GramSetSize = 0;
```

Once the application passed the pointer to the *ExtGramInfo* array to the API, the API allocates it's own memory and copies the contents of this array into it's own address space. This means that after the return from the *S51_CfgChan()* function which received the pointer to this array, the API does not use this memory space anymore and the memory can be released by the application.

Another (simpler) way to pass short (less than 128 bytes) grammars to the ASR is by using *In-Line grammars*. This is described next.

5.22.3 In-Line Grammars

The API allows for passing JSGF grammars in-line and also the lex data from API to the ASR resource "in-line". The in-line grammars are placed in the *GramInfo.GramFile* field and the in-line lex files are placed in the *GramInfo.LexFile* field. That means that each in-line grammar or lex file is limited to 127 characters. The other two grammar parameters (*FieldName* and *GramType*) must also be filled in appropriately.[42] API detects the in-line grammar and/or in-line lex information by scanning for a '<' character or for a whitespace character in the name of the grammar file and/or in the lex file. If such characters are detected, the API does not treat in-line grammar/in-line lex information as files, but passes them "as-is" to the ASR resource.

Because the text of an in-line grammar must be placed in the *GRAM_SET.GramFile* and the (optional) in-line lex information must be placed in *GRAM_SET.LexFile*, they are both limited to 127 characters. Below is the example of specifying an in-line drop-in grammar.

```
#define   GRAMMAR_EA13 \
    "#JSGF V1.0;grammar EmailAction13;public <DYNAMIC>=[read|get][next|previous];"

   CfgReq.Asr1Req |= S51_ASRTYPE_CFGREQ;
   AsrCfgVal.AsrType = S51_GC_ASRTYP;
   strcpy(AsrCfgVal.AsrName, "DYNAMIC");
   strcpy(AsrCfgVal.GramInfo[0].FieldName, "DYNAMIC");
   strcpy(AsrCfgVal.GramInfo[0].GramFile, GRAMMAR_EA13);
   strcpy(AsrCfgVal.LexFile[0], "");
   strcpy(AsrCfgVal.GramType, "jsgf");
   AsrCfgVal.GramSetSize = 1;
   CfgVal.Asr1CfgVal = &AsrCfgVal;
   SpCh = S51_CfgChan(SPId, &CfgReq, &CfgVal, &ErrMsg);
   if ((!SpCh) || (SpCh->ComplCode & A5_ANY_ERR))
          ...error...
```

Note that the grammar name (in this case "EmailAction13") would normally match the grammar file name, but in this case it is irrelevant, although still required. The *FieldName* field must contain the matching drop-in point as defined in a higher level grammar into which this grammar will be dropped in. In other words, the same rules apply to the in-line grammars as to grammars which are stored in files.

In summary, the "in-line" means that instead of storing the grammar in a file and providing the file name to the API, the application replaces the grammar file name with the grammar itself. This feature has been designed to be used with very short "trivial" grammars. Any grammar which exceeds 127 characters must still be stored in a file.

42. The *FieldName* must be the name of the public JSGF grammar, and the *GramType* must be "jsgf"

5.22.4 ASR Resource Commands

The **S51_ASR_CMDVAL data structure** is an argument to the *S51_CmdChan()* function and is used to provide command values for the ASR Resource.

```
typedef struct
{
   u_long       AsrCfgReq;          /*additional (optional) configuration*/
   S51_ASR_CFGVAL* AsrCfgVal;       /*additional configuration values*/
   S51_SW_ENT   TrainSwEntry;       /*subword table and entry for training*/
   u_char       VocabSubset[S512_VOCABSUBSET_SIZ+1];/*vocabulary subset*/
   u_char       InhTimers;          /*inhibit IST, EST, and MLT timers*/
   u_short      AtEnergyStopRes;    /*resources to stop on energy*/
   u_short      At1stRecgStopRes;   /*resources to stop on first detected digit*/
   u_short      AtComplStopRes; /*resources to stop when ASR completes*/
   u_short      AtComplEnableResTimers;/*enable timers when ASR completes*/
   u_short      AtStopEnableResTimers;/*enable timers when ASR stops*/
   u_short      At1stRecgEnableResTimers;/*enable the 3 timers on 1st recg*/
}
S51_ASR_CMDVAL;
```

The variables are:

AsrCfgReq ....optional “last-second” configuration request,

AsrCfgVal..... .....values associated with AsrCfgReq,

TrainSwEntry..... .....table and entry that should be checked for a collision during subword training,

VocabSubset ....vocabulary subset; if defined, this is the set of words (comma-separated ASCII words) which narrow down the word set to be recognized, for example “*0,1,2*” for *ASR-CD*,

InhTimers ....if not 0, automatically inhibit the IST, EST, and MLT timers at start,

AtEnergyStopRes.... .....resource(s) to stop when the *Efloor* level exceeded,

At1stRecgStopRes.. .....resource(s) to stop when first digit detected, this is for CD-ASR and GC-ASR only, it also automatically enables the *S51_ASR_CFGVAL.Report1stFlg* feature,

AtComplStopRes.... .....resources to stop when ASR completes.

AtComplEnableResTimers ..... resources whose timers should be enabled when the ASR Resource completes,

AtStopEnableResTimers.... ..... resources whose timers should be enabled when the ASR Resource is stopped,

At1stRecgEnableResTimers ..... resources whose timers should be enabled when ASR reports first recognition.

Programmers Note: Using Virtual Timeslot (VTS) as Input to ASR

When using a VTS as an input to the ASR, the writer of the VTS must keep writing data into the VTS until the ASR module completes the recognition. If all of the actual data has been sent before the ASR command completes, the application must keep sending "silence" data until the ASR completes or is stopped. If this is not done, the ASR could "lock up" and never complete. This is a known bug and will be corrected in the future.

Programmers Note: Memory Usage by Dynamically Loaded Grammars

When loading grammars dynamically via *S51_ASR_CFGVAL.GramInfo* (a.k.a. "drop-in grammars"), each grammar used is loaded in a separate part of memory. This means that even if all the channels use the same drop-in grammar, each channel uses a different "copy" of the grammar in a separate piece of memory. The implications are that a drop-in grammar may end up using a lot more memory than if the same grammar had been preloaded. In some cases (large grammars), this may limit the number of channels that can be run on one board.

5.22.5 ASR Resource State

The **S51_ASR_STAT data structure** is a part of the channel structure (*S51_CH*) and it defines the current state of the ASR Resource allocated to this channel. This is where the results of the recognition are delivered to the application.

```
typedef struct
{
    u_long              FwVersion;
    u_long              CfgReq;      /*valid fields in CfgVal*/
    S51_ASR_CFGVAL      CfgVal;      /*current configuration*/
    u_long              CmdReq;      /*last command completed (request)*/
    S51_ASR_CMDVAL      CmdVal;      /*last command completed (values)*/
    u_long              ComplCode;   /*configuration/command completion code*/
    u_long              TermReason;  /*why stopped*/
    u_long              ResErrCode;  /*detail error code, s51err.h*/
    char*               ErrMsg;      /*error message for last request*/
    float               EnergyDetectTim;/*relative time stamp since start*/
    float               EnergyMax;   /*maximum energy detected in dBm*/
    u_char              RecgTopN;    /*real size of SW Entry or RecgWord*/
    S51_RECG_WORD       RecgWord[S51_TOPN_MAX];/*recognized word(s)*/
    S51_PARSED_WORD     ParsedWord[S51_TOPN_MAX];/*recognized & parsed word(s)*/
    S51_COLLIDE_WORD    CollideWord; /*collided word(s)*/
    S51_SW_ENT          SwEntry[S51_TOPN_MAX;/*recognized/collided subword entry*/
}
S51_ASR_STAT;
```

The variables are:

FwVersion ..... version of the ASR firmware running on the board,

CfgReq ..... OR-ed value of all ASR configuration requests completed on this channel,

CfgVal ..... all values used to configure the ASR Resource on this channel so far.

CmdReq ..... the last completed ASR command on this channel,

CmdVal ..... values used for the last completed ASR command,

ComplCode ..... completion code for the last completed ASR request on this channel,

TermReason ..... reason why ASR resource stopped (see *S51_*_TRM* in file *s51.h*),

ResErrCode ..... low-level error code provided by the ASR Resource, see s51err.h,

ErrMsg ..... text error message associated with the completion code,

EnergyDetectTim ... time stamp (in seconds) since start of the recognition when energy was detected,

EnergyMax ..... maximum energy detected (dBm).

RecgTopN ..... actual size of *RecgWord* and *ParsedWord* arrays.

SwEntry ..... table and entry recognized or collided with, used only for subword ASR,

RecgWord ..... top recognized word(s) or number(s) for CD and GC ASR types, or a transcription for the training phase of the *NameRecog* type of ASR,

ParsedWord ..... contains the same information as RecgWord, but the recognition result string and the tag string are parsed into individual words (see “Results of the Recognition” on page 126),

CollideWord ..... used only for NameTrain, transcriptions of collided word(s).

5.22.6 Results of the Recognition

The results of recognition may be one or more recognition candidates. The number of candidates is in the *RecgTopN* field. This is the actual size of the *RecgWord* and *ParsedWord* arrays. The first entry in these arrays contains the most likely (if any) candidate of the recognition. If the ASR was requested to deliver more than one candidate (top N candidates), these are in the subsequent members of the array in descending order by probability.

The ASR interface reports the ASR results in two formats: unparsed (raw) and parsed. It uses the *RecgWord* array for unparsed results and the *ParsedWord* array for parsed results. Both, the unparsed and the parsed result consists of two types of data: 1) the recognized words and 2) their tags.

Each word can have zero or more tags associated with it. The set of tags associated with one word is called a metatag. In it's unparsed (raw) format, all tags are delivered as a single string and the individual metatags are separated by underscores '_'. Within each metatag, each tag is delimited by a set of curly braces '{}'.

More details are below and also in "Obtaining Grammar-Based Recognition Results from the ASR Resource" on page 29 in the "ASR & TTS User's Guide".

5.22.6.1 Unparsed (Raw) Results

The unparsed results are delivered in the **RecgWord** array of structures of type S51_RECG_WORD:

```
typedef   struct
{
   char         Text[S51_ASRWORD_SIZ+1];/*recognized word*/
   char         Tag[S51_ASRWORD_SIZ+1];/*recognized tags*/
   float        Confidence;/*0 means 'not provided'*/
}
S51_RECG_WORD;
```

The fields are:

Text.....This is the recognized speech. It is a string containing all the recognized words, each word separated by a space. For example: "*collect call*". See the paragraph below for a more detailed description.

Tag.....These are the tags and metatags[43] which correspond to the recognized words. The tags are strings assigned to words in the ASR grammar. This string contains all tags for all recognized word (words in the *Text* field). Words with no assigned tags are represented by NULL string,[44] that is, there are two adjacent metatag separators (_) with no text between them. For example if in the recognized phrase "*collect call*", only the word "*collect*" had an assigned tag of "*COLLECT*", the tag string would then be: "{COLLECT}_". See the next paragraph for a more detailed description.

Confidence .. ..except for the fact that this number increases with increased confidence, it does not really contain any information useful to the application. Therefore applications should not attempt to use this field. Currently the only use of this field is by the ASR module itself and it relates to the rejection threshold *S51_ASR_CFGVAL.RejectThr* value. Any recognition result which has the *Confidence* value less than the *RejectThr* value is rejected, it is not considered to be valid and it is not reported to the application.

The *RecgWord.Text* format is as follows.

- All recognized words are in one string.
- Each recognized word is separated by a single space ' '.

43. These are referred to as "metatags" because each word can have several tags and all tags for one word are concatenated together into a "metatag" and separated by curly braces '{}'. In turn, all metatags are concatenated together and separated by underscores (_).

44. When using the Release 2 compatibility mode (ResMode set to S51_REL2_RMD), each empty (NULL) tag is represented by a digit '0'.

- If a word contains a space, it is quoted. The following is an example of three words:
```
one "foo bar" two
```
- A backslash '\' starts a 2-character escape sequence, that is, a backslash followed by a single character. This escape removes any special meaning of the 2nd character. The following is an example of three words with a middle word which includes a double quote:
```
one foo\"bar two
```

The *RecgWord.Tag* format is as follows.

- All tags for all words are concatenated into a single string.
- A group of tags associated with one word is called a "metatag". Each metatag matches one word in the recognized word string. There is a one to one correspondence between the words and the metatags. On the other hand, each word can have several tags - all of these are grouped into one metatag.
- Metatags within the tag string are separated by a single underscore '_'. The following is an example of a sequence of five metatags: 1) NULL, 2) a single-tag metatag "Atag1", 3) NULL, 4) a multiple-tag metatag "Btag1" "Btag2" and 5) a single-tag metatag "Ctag1":
```
"_{Atag1}_ _{Btag1}{Btag2}_{Ctag1}"
```
- Each metatag can in turn consist of zero or more tags. Each tag is delimited by a pair of curly braces '{' and '}'. The following is an example of a metatag containing three tags:
```
"{Atag1}{Atag2}{Atag3}"
```
- A backslash '\' starts a two-character escape sequence, that is, a backslash followed by a single character. This escape removes any special meaning of the 2nd character. The following is an example of three tags with a middle tag which includes a right curly brace '}':
```
{one}{foo\}bar}{two}
```

5.22.6.2 Parsed Results

The parsed results are delivered in the **ParsedWord** array of structures of type S51_PARSED_WORD:

```
typedef   struct
{
    long         WordCount;    /*#of recognized words (size of arrays)*/
                               /*all arrays are indexed by the word index*/
    char*        Text[S51_ASRWORD_MAX];/*recognized words - one per word*/
    S51_TAGSTR*  TagStr[S51_ASRWORD_MAX];/*word metatags & tags (strings)*/
    u_long       TagNum[S51_ASRWORD_MAX];/*word numeric tags -old interface*/
}
S51_PARSED_WORD;
```

The fields in the **S51_PARSED_WORD** data structure are:

WordCount.. ..... The number of recognized and parsed words. This is the actual sizes of the *Text*, the *TagStr* and *TagNum* arrays.

Text .... These are the individual recognized words. See the paragraph below for a more detailed description.

TagStr..... ..... These are the tags (or set of tags) for each word. Each word has one corresponding element in the *TagStr* array. More detailed description follows below.

TagNum .... If one of the tags for a word contains a numeric string value, this value is converted into a binary value and stored here. This field is mainly intended to maintain a limited compatibility with the application grammars written in LSPS Release 2.x using the GSL specification.

The *ParsedWord.Text* format is as follows:

- Each word 'n' is delivered as a separate string in *ParsedWord.Text[n]*.

- All escape sequences inside of each word are replaced by the corresponding single character.
- All quotes and whitespace surrounding the words are removed.

The parsed tags are delivered in the **TagStr** array of structures of type S51_TAGSTR:

```
typedef   struct
{
   long         TagCount;    /*actual size of Tag array - must be the 1st field!*/
   char*        MetaTag;     /*the entire metatag string (unparsed)*/
   char*        Tag[1];      /*parsed tags - variable size array of size TagCount*/
}
S51_TAGSTR;
```

The fields in the **S51_TAGSTR** data structure are:

TagCount..... .....The number of individual tags parsed out of the word's metatag (*MetaTag* field). This is also the size of the variable array *Tag[]*.

MetaTag..... .....The complete tag for one word. Contains all word's tags concatenated into one string. More detailed description is below.

Tag..... .....Set of tags for one word parsed out of the word's metatag. More detailed description is below.

The format of the **parsed metatags**[45] in *ParsedWord.TagStr* is as follows:

- Each set of strings representing the metatag for word 'm' and the corresponding parsed set of tags is delivered as a *S51_TAGSTR* structure in ParsedWord[m].TagStr.
- The *S51_TAGSTR* metatag structure contains the word's metatag string, the set of parsed-out tag strings for the word and the number of parsed tags in the tag set.
- The metatag string is delivered in *ParsedWord.TagStr.MetaTag* and it is obtained by parsing the unparsed tag string by using the underscores (_) as separators. No other processing is done on the metatag string. For example the backslash '\' escapes and quotes are left intact. In other words, the internal format of a metatag string is the same as the format for the unparsed tags.
- The tag strings for each word are in *ParsedWord.TagStr.Tag* and they are parsed out of each metatag string by using curly braces '{}' as separators. All escape sequences inside of each tag string are replaced by the corresponding single character. All curly braces, quotes and whitespace surrounding the tag string are removed. The *ParsedWord.TagStr.Tag* is a variable size array and its size is in *ParsedWord.TagStr.TagCount*.
- A metatag structure representing a NULL metatag contains a null metatag string, a count of zero tags and a zero-length array of tag strings.

The tags generated by a compiled GSL grammar[46] are significantly different:

- all tags are always numeric,
- there is always only a single tag for each word,
- multiple tags assigned to a single word in the GSL grammar are delivered as an arithmetic sum of all the tags for the word,
- empty tags are delivered either as empty strings: "_ _", or if the *Release 2 Compatibility Mode*[47] is used, they are delivered as zeroes: "0_0_0".

45. Each word can have one or more tags associated with it. A set of tags associated with one word is called a "metatag". The metatag is delivered in two ways: 1) as a single string containing all tags for the word, and 2) as a set of parsed single tags for the word.
46. Obsolete and not supported in 3.x Releases.
47. This is done by setting the S51_ASR_CFGVAL.ResMode to S51_REL2_RMD.

To maintain a limited compatibility with application grammars written using the GSL specification,[48] the binary (converted) value of the tag is delivered in the *ParsedWord.TagNum* field, which was previously the *ParsedWord.Tag* field. For the JSGF grammars, this field is set to the numerical value of the first numeric tag found for each word. If the word does not have a numerical tag, the *TagNum* field is set to zero.

5.22.7 JSGF Path

The API gets the string-value of this parameter from the application in one of two ways: either as a part of the ASR configuration request, or from the environment variable "JSGFPATH". The specification in the configuration request always takes precedence to that of the environmental variable. All relative pathnames in the JSGF path specification are automatically converted to absolute pathnames using the current working directory. If the JSGF path parameter is not provided by the application, the API's JSGF path parameter ends up containing only the current working directory.

The JSGF path parameter is used to search for and to modify path names for the grammar and lex files when the grammar type is "jsgf". This parameter has no effect on handling of grammar and lex files for other types of ASR grammars - such as compiled JSGF grammar files which are of type "gc". In other words, grammars written to the JSGF specification and then compiled are no longer of "jsgf" type. They are compiled into the "gc" type and the JSGF path is not applicable to the "gc" type grammars.

The JSGF path parameter consists of one or more local-host absolute pathname directories separated by semicolons (;). A dot represents the absolute pathname of the current working directory. The API parses the provided JSGF path parameter and extract a set of absolute directory pathnames. The API then uses this set of directory pathnames to find and convert any "jsgf" relative file names provided by the application as follows:

- Create absolute filenames by prepending each pathname from the JSGF path variable to the relative file path name specified by the application and check if such file exists at this location.
- If the file exists at one of the absolute filenames created by the above process, replace the relative file name with this absolute pathname and use this absolute pathname when sending information to the ASR resource.

The file names which are located and converted to absolute file names using the JSGF path list are fields in the S51_ASR_CFGVAL data structure: *GramInfo.GramFile* and *GramInfo.LexFile*.

In addition, the ASR resource itself uses the JSGF path to locate JSGF grammar files which were "included" in the grammar files explicitly specified via the API.

48. Used in Release 2.x of LSPS. No longer officially supported.

5.22.8 ASR-CD Example

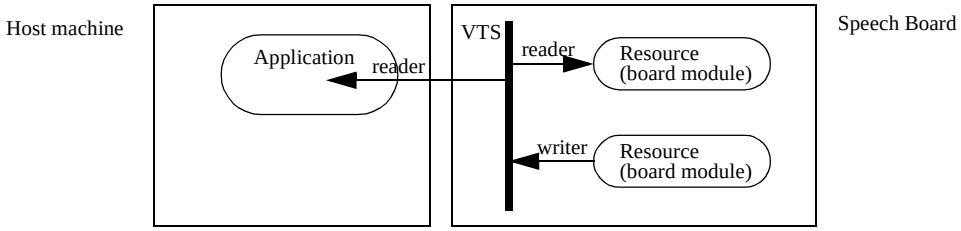
The following code fragment is an example of detecting spoken digits, and it shows configuration, set up, and the start of recognition using a connected digit ASR.

```
Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
...
S51_ASR_CFGVAL      AsrCfgVal;
S51_CFGVAL          CfgVal;
S51_REQ             Req;
S51_CID             Cid;
char*               ErrMsg;
S51_CH*             Ch;

/*use board 0 channel 1*/
Cid = S51_MakeCid(0, 1);

/*clear data structures*/
A5_Zerout(CfgVal);
A5_Zerout(Req);
A5_Zerout(AsrCfgVal);

/*configure ASR Resource*/
Req.Asr1Req = S51_CONNTS_CFGREQ|S51_ASRTYPE_CFGREQ|S51_RESMODE_CFGREQ
             |S51_MAXLISTENT_CFGREQ;
AsrCfgVal.Tid = A5_MakeHtsTid(20);
AsrCfgVal.AsrType = S51_CD_ASRTYP;
strcpy(AsrCfgVal.AsrName, “CD_UNKNOWN”);
AsrCfgVal.ResMode = S51_PUREDIGIT_RMD;
AsrCfgVal.MaxListenTim = 30;

/*allocate and configure ASR resource*/
CfgVal.Asr1CfgVal = &AsrCfgVal;
Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

/*start ASR*/
A5_Zerout(Req);
Req.Asr1Req = S51_START_CMDREQ;
Ch = S51_CmdChan(Cid, &Req, NULL, &ErrMsg);

/*check for error*/
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
       return(ErrMsg);

/*detection completed*/
printf(“Detecet digits: %s\n”, Ch->Asr1Stat->RecgWord[0].Text);
```

6. Virtual Timeslots

6.1 Introduction

This part of the API provides control of and interface to virtual timeslots (VTS). Virtual timeslots are a mechanism for exchanging arbitrary data between the resources on the speech board and also between an application running on the host and the resources on the board. The following diagram shows an example of speech data transfer between the board modules and application program using a VTS.



Note that only one host process can read a given VTS at one time. There is no such limit for resources on the board. Also, the VTSs are not supported by the LSPS Remote API (RAPI).

6.1.1 Description of Virtual Timeslots (VTS)

The virtual timeslots work in a similar way to physical timeslots (HTSs), and the virtual timeslot number can be used in place of physical timeslot number in any and all API “TID” variables and parameters. The physical and the virtual timeslots have many properties in common:

- carry speech data transparently from a writer to a reader[49],
- automatically synchronizes writers and readers with each other,
- can perform “drain speech”, which waits until all already-written data is actually transferred to the receiver, then automatically generates silence until another write,
- can perform “discard speech”, which discards any remaining buffered data, then automatically generates silence until another write,
- for multiple readers, each reader receives all data,
- for multiple writers, the individual writers must cooperate to ensure that only one is actively writing at any instant in time.

But there are also significant differences in how VTS work as compared to the real timeslots:

- VTS does not exist when a board is booted, but must be created explicitly by a host application or by a resource on the board. Conversely, if a VTS is not needed, it should be destroyed to conserve resources. There are no restrictions on who can create or destroy which VTS. So for example, a VTS created by a board resource (program running on the board) can be destroyed by application running on the host. When a VTS is destroyed, all readers and writers are notified. In some sense, the VTSs work in a similar fashion to UNIX files or UNIX shared memory.
- A virtual timeslot (VTS) is local to each speech processing board and it cannot be used to send data from one board to another. That means, for example, that VTS 28 on board 1 and VTS 28 on board 2 are unrelated.
- A VTS is directly accessible from the host (application) for either writing or reading, but not both.
- The data transfer rate through a VTS is determined by the slowest reader or writer.[50]

49. VTS can have multiple writers and multiple readers, providing that the readers/writers are modules on the board.

- Although the VTS timeslot IDs have the same data type as any other timeslot type, they must be created using their own macro *A5_MakeVtsId*. The VTS numbering range is from 0 to A5_VTSNUM_MAX.
- Depending on what direction of data transfer the VTS can perform, it can be one of three types: 1) host writes and board modules read, 2) host reads and board modules write, and 3) board modules both read and write.
- Only a single host process can access each VTS (reading or writing).

6.1.2 Drain Speech and Discard Speech

For physical timeslots, the details of interface to the physical timeslots are hidden from the host application by the board modules and by the API. But for virtual timeslots the host application interfaces to the timeslots directly, and thus must be able to deal with two new situations: drain speech and discard speech.

The **Drain Speech** request may be issued only by the current writer. This pushes all data previously written to the readers. After this it is the responsibility of each reader to internally generate silence whenever data is not available in a timely manner (which must be determined by the reader). In general, the ultimate speech data consuming process of a data-stream pipeline (wherever it is located) must be the one to generate the silence. Intermediate handlers cannot do so properly. Intermediate handlers of the speech stream should merely block waiting for data.

The **Discard Speech** request may be issued by any reader or writer. This discards any buffered data and indicates to each reader and to the current writer that a discard speech was performed. There is one exception: the reader or writer that issued discard speech does not get discard speech indication. When a discard speech indication is received, it is then the responsibility of each reader to internally generate silence until the next write is performed. It is also the responsibility of the writer to stop generating data until instructed to start again.

Note that both, the Drain Speech and Discard Speech are **conditions** of the VTS and **not events**. That means that the application must check for these conditions after every read (even if the read returned 0 bytes), because it will not receive an event (or even any speech data) when the condition changes.

6.1.3 API Interface Summary

The API interface to the VTS is defined in the header file */usr/include/lsps/v5vts.h*. This file contains not only the data structures, #defines and function prototypes, but also some significant comments. Therefore it is a good idea for a programmer who is starting to use the VTS API to get familiar with this file.

Each VTS is represented by an ID. This timeslot ID (TID) is constructed from the board number, timeslot number and a timeslot type by using the **A5_MakeVtsTid() macro**:

```
VtsId = A5_MakeVtsTid(SpBoard, VtsNum, VtsType);
```

The arguments to the *A5_MakeVtsTid()* are as follows:

SpBoard..... the physical board 0 to A5_BD_MAX (see a5api.h).

VtsNum ..... the VTS number 0 to A5_VTSNUM_MAX (see a5api.h).

VtsType ..... one of the three VTS types, constructed by bitwise OR-ing together two of the following constants: V5_HOST_READ, V5_SRES_READ, V5_HOST_WRITE, V5_SRES_WRITE in one of the following valid combinations (v5vts.h):

V5_HOST_READ | V5_SRES_WRITE

V5_HOST_WRITE | V5_SRES_READ

V5_SRES_READ | V5_SRES_WRITE

50. Recall that a VTS can have many readers but only one writer.

For example:

```
S51_PLAY_CFGVAL.Tid = A5_MakeVtsTid(BdNum, VtsNum, V5_HOST_READ|V5_SRES_WRITE);
```

Of course when using the speech API with a VTS, the speech API board number (in the channel ID S51_CID) must match the board number in the VTS TID and the data operation being performed (read vs. write) must match the VTS type in the TID.

The **VTS Interface on the host** consists of the following 11 functions:

V5_Init()..... .....initializes the interface and returns the speech board state information.

V5_Term()..... .....shuts down the interface.

V5_CreateVts()..... .....creates one VTS and returns a pointer to VTS data structure.

V5_DestroyVts()... .....destroys existing VTS.

V5_OpenVts()..... .....opens access to a VTS, data can be now read or written.

V5_CloseVts()..... .....closes access to a VTS, data can no longer be read or written.

V5_AbortVts()..... .....prematurely terminates asynchronous operation in progress (create, destroy, open, close).

V5_EventVts()..... .....get events for any opened VTS.

V5_ReadVts()..... .....read speech data from one VTS.

V5_WriteVts()..... .....write speech data to one VTS.

V5_DataEndVts(). .....send a drain speech or discard speech to one VTS.

The following functions can operate in either blocking mode or in asynchronous (non-blocking) mode: *V5_CreateVts()*, *V5_DestroyVts()*, *V5_OpenVts()* and *V5_CloseVts()*. All the other functions always complete immediately without blocking.

A more detailed description of these functions is in the “LSPS API Function Reference” on page 147.

6.2 VTS Data Structures

There are three data structures used by the VTS API: VTS equipment configuration (V5_EQP), VTS State (V5_VTS) and VTS Event (V5_EVT).

6.2.1 VTS Equipment Configuration V5_EQP

This structure is returned from the VTS API initialization function *V5_Init()* and it contains basic state information about the boards which can be used to create VTSs. It is a subset of information returned by the *S51_Init()* function.

```
typedef   struct
{
   HANDLE       Fd;                /*file descriptor to use for poll()*/
   u_short      NumBoards;         /*how many boards in this host*/
   u_short      NumUse;            /*how many are available for use*/
   V5_BD_STAT   BdStat[A5_BD_MAX + 1];
}
V5_EQP;
```

The fields are:

Fd..... .....file descriptor to be used on the UNIX host by *poll(2)* to obtain data or event from any VTS,

NumBoards. .....number of boards configured in the host files,

NumUse..... .....number of usable boards as detected by the driver,

BdStat..... .....current state of the board hardware, firmware, and the board control channel. Refer to “BCC Channel State” on page 64 covering the **Board Control Channel** state for the description of the fields in this structure.

6.2.2 VTS State Data Structure V5_VTS

This structure contains the current information which the API has about the VTS. It is also used to return results of VTS API function calls, for example the completion code and an error message. Pointer to this structure is returned from the following API functions: *V5_CreateVts()*, *V5_OpenVts()*, *V5_ReadVts()* and *V5_WriteVts()*.

```
typedef struct
{
          A5_TID       Tid;
          u_short      BdNum;       /*board number*/
          u_short      VtsNum;      /*timeslot number*/
          u_short      VtsType;     /*timeslot type*/
          u_long       ComplCode;   /*A5_INPROG, A5_OK, error code*/
const     char*        ErrMsg;      /*error message*/
          char         DrainSpeech; /*drain speech condition is in effect*/
          char         DiscardSpeech;/*discard speech condition is in effect*/
          u_long       AvailToRead; /*how many bytes available to read*/
          u_long       BytesRead;   /*how many bytes read into ReadBuf*/
          char*        ReadBuf;     /*user-provided buffer*/
          u_long       AvailToWrite;/*how many bytes available to write*/
          u_long       BytesWritten;/*how many bytes written from WriteBuf*/
          char*        WriteBuf;    /*user provided buffer*/
}
V5_VTS;
```

The variables are:

Tid ....VTS timeslot ID,

BdNum..... .....board number derived from the Tid (0 to A5_BD_MAX),

VtsNum..... .....timeslot number derived from the Tid (0 to A5_VTSNUM_MAX),

VtsType..... .....VTS type derived from the Tid,

ComplCode . .....completion code associated with the last function call (A5_OK, A5_INPROG, A5_*_ERR),

ErrMsg..... .....text error message associated with the last function call,

DrainSpeech .....set to 1 while there is a drain speech condition in effect, otherwise 0. It may be set or cleared by *V5_ReadVts()* and may be cleared by *V5_EventVts()*. The application must always test this condition after returning from one of these two functions, even if there was no data read or event delivered. This is an indication to the application that there is no more real-time speech coming (at least for a while), and in that sense it is equivalent to an "end-of-file" condition on a UNIX readable device. Because there is no more data coming from the VTS, the application itself may run out of real-time speech data and it may have to generate silence until the speech data stream starts up again and there is data to read. This condition is cleared when there is data again available to be read from the VTS.

DiscardSpeech.....set to 1 while there is a discard speech condition in effect, otherwise 0. It may be set or cleared by *V5_ReadVts()* and may be cleared by *V5_EventVts()*. The application must always test this condition after returning from one of these two functions, even if there was no data read or event delivered. This is an indication to the application that all data (both read and to be written) that can still be discarded by the application should be discarded. Also, if the application is reading from the board, the application will not be receiving any data (for a while) in lieu of silence, this is similar to the DrainSpeech condition, and so it is also equivalent to an "end-of-file" condition on a UNIX readable device. Presumably the application would want to flush/discard all data that it has in it's own buffers or pipe-

lines, and if reading, assume that there is silence being received from the VTS. All other speech data that is under control of the board, the driver and the API is discarded automatically. This condition is cleared when there is data again available to be read from the VTS.

AvailToRead ..... number of bytes that are currently waiting to be read. Note that 0 bytes in this field does not indicate a drain-speech condition. It is set by *V5_ReadVts()* or *V5_EventVts()*.

BytesRead... ..... number of bytes read in the last read *V5_ReadVts()*.

ReadBuf .... address of user buffer into which the last BytesRead were copied by the last *V5_ReadVts()*.

AvailToWrite..... number of bytes available for the host to write. It is set by *V5_WriteVts()* or *V5_EventVts()*.

BytesWritten ..... number of bytes written in the last *V5_Write()*.

WriteBuf .... address of user buffer from which BytesWritten were written out by the last *V5_WriteVts()*.

6.2.3 VTS Event Data Structure V5_EVT

The application receives events via *V5_EventVts()* or via a callback function in a form of a V5_EVT data structure.

```
typedef struct
{
          V5_TID       Tid;
          u_long       EventType;   /*see below*/
          u_long       ComplCode;   /*A5_INPROG, A5_OK, error code*/
const     char*        ErrMsg;      /*same as V5_VTS.ErrMsg*/
const     V5_VTS*      Vts;         /*associated with this VTS*/
}
V5_EVT;
```

The variables are:

Tid..... ..... VTS timeslot ID,

EventType... ..... one of four event types, described below,

ComplCode. ..... completion status for the last function call: A5_INPROG, A5_OK, A5_*_ERR, see *a5api.h*

ErrMsg..... ..... textual explanation of ComplCode (on error),

Vts..... ..... pointer to the VTS data structure associated with this VTS.

There are four types of events:

V5_READMORE_EVT.... .....when the application receives this event, it means that there is more data available to read. The application must call the *V5_ReadVts()* function to get the data. Note that in order to receive this event, *AvailToRead* does not have to reach 0 first, but only to increase.

V5_WRITEMORE_EVT.. .....when the application receives this event, it means that there are internal buffers again available to write. This event is generated only after the API first runs out internal write buffers, that is, after *AvailToWrite* reaches zero. The application can now again call the *V5_WriteVts()* function to write data. The total size of available write buffers is in *AvailToWrite*. The “AvailToWrite==0” condition is set by *V5_WriteVts()* call and cleared by *V5_EventVts()* call.

V5_DESTROY_EVT..... .....this is an indication that another process (host or board) destroyed the VTS. The process receives this indication only if it either created the VTS originally or has it currently open. In the second case (open), this event is accompanied by a DiscardSpeech condition. Except for V5_COMPL_EVT, no events are generated for the process which destroyed the VTS via the *V5_DestroyVts()* call.

V5_COMPL_EVT..... .....marks the completion of an asynchronous operation on VTS. In other words, completion of one of the following requests: *V5_CreateVts()*, *V5_DestroyVts()*, *V5_OpenVts()* or *V5_CloseVts*. The completion code is in *V5_VTS.ComplCode* (could be success or error).

Programmers Note: Calling VTS Functions From an VTS Callback Functions:

The application should read and process all the pending VTS events before issuing any VTS request. If this is not the case, the events could back up and could potentially overflow internal queues and be lost. The implication of this is that is not recommended (and strongly discouraged) to call VTS functions from VTS callback function in asynchronous mode. This is because there is a possibility that the application may not process events in timely manner, causing the pending events to accumulate and back up and eventually some events may be lost. In summary, don't do it.

6.3 VTS Example

This code fragment allocates and starts up a Text to Speech (TTS) module which plays a text file into a VTS. The program then reads TTS speech data from this VTS and writes the coded speech data into a file. In other words, the program converts a test file into a speech file. The complete source for this program can be found in *vts_tts.c*.

```
/*initialize VTS API in blocking mode*/
VtsEqp = V5_InitVts(A5_BLOCK_MOD, NULL, &ErrMsg);
if (!VtsEqp)
   return(ErrMsg);
VtsFd = VtsEqp->Fd;

/*create and open VTS*/
Tid = A5_MakeVtsTid(Board, Timeslot, V5_HOST_READ|V5_SRES_WRITE);
Vts = V5_CreateVts(Tid, 8, 4096,&ErrMsg);
if ((!Vts) || (Vts->ComplCode != A5_OK))
   return(ErrMsg);

Vts = V5_OpenVts(Tid, NULL, &ErrMsg);
if ((!Vts) || (Vts->ComplCode != A5_OK))
   return(ErrMsg);

/*initialize speech API in blocking mode*/
Eqp = S51_Init(A5_BLOCK_MOD|A5_CLEANUP_MOD, NULL, &ErrMsg);
if (!Eqp)
   return(ErrMsg);

/*allocate and configure TTS, TTS events are handled in a callback function*/
Cid = S51_MakeCid(Board, Channel);
A5_Zerout(Req);
CfgVal.CbFun = NULL;
Req.TtsReq = S51_CONNTS_CFGREQ|S51_CBFUN_CFGREQ;
TtsCfgVal.Tid = Tid;
TtsCfgVal.CbFun = TtsCbFun;   /*see the end of this listing*/
CfgVal.TtsCfgVal = &TtsCfgVal;

Ch = S51_CfgChan(Cid, &Req, &CfgVal, &ErrMsg);
if ((Ch == NULL) || (Ch->ComplCode != A5_OK))
   return(ErrMsg);

/*change TTS to async mode so I can read VTS while playing TTS*/
if (!S51_Init(A5_ASYNC_MOD, NULL, &ErrMsg))
   return(ErrMsg);

/*start playing TTS - asynch mode*/
A5_Zerout(Req);
A5_Zerout(CmdVal);
A5_Zerout(TtsCmdVal);
A5_Zerout(TtsSet);
strcpy(TtsSet.PlayFile, PlayFile);
```

```
TtsCmdVal.PlaySet = &TtsSet;
TtsCmdVal.SetSize = 1;
CmdVal.TtsVal = &TtsCmdVal;
Req.TtsReq = S51_START_CMDREQ;
Ch = S51_CmdChan(Cid, &Req, &CmdVal, &ErrMsg);
if ((Ch == NULL) || (Ch->ComplCode & A5_ANY_ERR))
   return(ErrMsg);

/*setup and run polling loop*/
memset(Pfd, 0, sizeof(Pfd));
Pfd[0].fd = Eqp->Fd;    /* S51 (TTS) */
Pfd[0].events = POLLIN;
Pfd[1].fd = VtsFd;      /* V5 (VTS) */
Pfd[1].events = POLLIN;
TtsDone = 0;     /*set to 1 in TTS callback function, TtsCbFun*/

while ((!TtsDone) || (!Vts->DrainSpeech))
{
   if (poll(Pfd, 2, -1) <= 0)
          continue;

   /*TTS events are handled in callback function TtsCbFun()*/
   if (Pfd[0].revents & POLLIN)
          (void) S51_EventChan(0, &EvtCnt, &ErrMsg);

   /*VTS events are needed only to wake us up*/
   if (((Pfd[1].revents & POLLIN) == 0)
   || (V5_EventVts(0, &EvtCnt, &ErrMsg) == NULL))
          continue;

   /*read VTS - must be done at least once to possibly set Vts->DrainSpeech*/
   (void) V5_ReadVts(Tid, Buf, sizeof(Buf), &ErrMsg);

   /*write all incoming data to file*/
   while (Vts->AvailToRead > 0)
   {
          (void) write(FileFdOut, Buf, Vts->BytesRead);
          (void) V5_ReadVts(Tid, Buf, sizeof(Buf), &ErrMsg);
   }
}

/*clean up*/
(void) S51_CloseChan(Cid, &ErrMsg);
S51_Term();
(void) V5_CloseVts(Tid, &ErrMsg);
(void) V5_DestroyVts(Tid, &ErrMsg);
V5_Term();
return 0;

/*
** TTS callback function
*/
int
TtsCbFun(S51_EVT* Evt)
{
   if (Evt->ComplCode != A5_INPROG)
          TtsDone = 1;

   return(0);
}
```

7. The Echo Canceller Interface

7.1 Introduction

This part of the API provides an interface to the Echo Canceller (EC) hardware. The hardware is located on the 6UB5 speech board. In this chapter, the functions, structures, and constants used to initialize and configure the echo canceller channels are described in detail. The header file *e53.h* contains the functions prototypes and constant definitions.

Each Echo Canceller Card provides a number of hardware echo cancellation ports.[1] Each one of these ports is associated with an API EC channel (which is a virtual entity), which is in turn identified by a EC Channel ID. This ID is constructed from the EC board number and the EC port number by using the **E53_MakeCid() macro**:

```
ChanId = E53_MakeCid(Board, Port);
```

The arguments are as follows:

Board..... the physical EC board (0 to E53_BD_MAX),

Port..... the EC port number (1 to *N for real EC channels*) or 0 for the EC Board Control Channel (BCC).[2]

The resulting Channel ID parameter then contains (bit-packed) information about the EC board and the EC port.

The Echo Canceller Card Interface consists of the following five functions:[3]

E53_Init()..... .....initializes the EC API and returns the board information.

E53_Term()..... .....shuts down the EC API.

E53_CfgChan()..... .....sets up EC channel configuration and parameters.

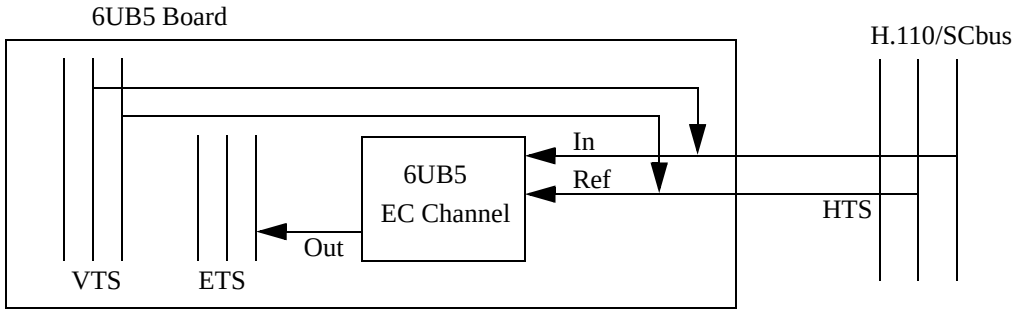
E53_CloseChan().. .....closes and releases one EC channel.

E53_EventChan(). .....gets an event from any EC channel.

7.2 EC Channels on 6UB5 Board

In addition to speech processing capabilities, each 6UB5 board provides 64 EC channels of echo cancellation. The same 6UB5 board shows up twice with the same board number: 1) in the speech processing API as a speech board, and 2) in the echo cancellation API as an EC board. The speech processing and the echo cancellation functionalities on the same board are controlled independently through the two separate API subsystems.

Each EC channel uses three timeslots: input, reference and output. The input and the reference timeslots can be either hardware bus timeslots (HTS) or virtual timeslot on the same board (VTS). The output timeslot must be a specific and particular "Echo Cancellation" timeslot (ETS) on the same 6UB5 board. This means that the output of the EC channel can be read only by resources on the same board.



1. The application should use the value E53_EQP.PortPerBd to get the actual number of ports provided by the board.
2. Note that this is different from the Speech Board Control Channel (also abbreviated BCC in “Speech Processing Board Control Channel (BCC)” on page 59)
3. The CmdChan() function does not exist in this API, simply because it is not necessary.

Since this is essentially a speech board, booting of the 6UB5 board is described in “Speech Processing Boards - Boot and Initialization” on page 39.

7.3 Initialization of the EC Interface

The EC API interface is initialized by calling *E53_Init()*. This function sets the API operating mode (blocking, asynchronous event, asynchronous poll), sets the global EC callback function (optional) and returns information about the state of the EC hardware in this host in the **E53_EQP** data structure. The different API operating modes work in a similar way to the API operating modes used by the Speech Processing API. The returned *E53_EQP* data **structure** describes the number of EC boards, the model of the boards and the state of each board.

```
typedef struct
{
   u_short      BdHwModel;    /*A5_6UB5_BD, etc. (see a5api.h)*/
   u_short      SpeechBusType;/*A5_H110_BUS*/
   HANDLE       Fd;           /*file descriptor to use for poll()*/
   u_short      NumBoards;    /*how many boards in this host*/
   u_short      NumUse;       /*how many are available for use*/
   E53_BD_STAT  BdStat[E53_BD_MAX];/*board info, indexed by board number*/
}
E53_EQP;
```

The fields are:

BdHwModel .....the hardware model of the EC boards equipped in this host:

- A5_6UB5_BD (6UB5)

SpeechBusType.....describes the type of speech bus used by the echo canceller boards:

- A5_H110_BUS

Fd ....file descriptor to be used on the UNIX host by *poll(2)* to obtain data from any EC Board,

NumBoards . .....number of EC boards configured in the host files,

NumUse ....number of usable EC boards as detected by the driver,

BdStat..... .....current state of the EC board hardware, firmware, and the board control channel, indexed by board number. The description of the *E53_BD_STAT* data structure follows.

The *E53_EQP* data structure contains an array of **E53_BD_STAT** structures which is indexed by board number and each member contains information about one EC board. For the 6UB5 board, most of the fields are set by a call to *E53_Init()*, but some fields are only set by opening an *EC Board Control Channel* (BCC).‘

```
typedef   struct
{                                   /*The following fields are set by the*/
                                    /*E53_Init() call*/

   u_short      BdHwModel;          /*A5_6UB5_BD, etc. (see a5api.h)*/
   u_short      BdHwState;          /*board hardware state*/
   u_short      BdHwVersion;        /*hdwre version of each board*/
   u_short      BdHwRev;            /*board hardware revision, numeric*/
   u_char       Enabled;            /*0/1, duplicates info in BdFwState redundant*/
   u_short      CpuBusType;         /*A5_PCI_BUS, etc. (above)*/
   u_short      PciBusNumber;       /*PCI bus number*/
   u_short      PciSlotNumber;      /*PCI slot number*/
   char         BdHwModelDescr[E53_BDMODEL_SIZ+1];/*model description*/
   char         BdHwSerialNum[E53_SERNUM_SIZ+1];/*serial number*/

                                    /*The following fields are set*/
                                    /*by board control channel events*/
   u_short      BdFwState;          /*board firmware state*/
   u_char       PortPerBd;          /*ports per board*/
}
E53_BD_STAT;
```

The following fields are always updated only by *E53_Init()*:

BdHwModel ..... Board hardware model: A5_6UB5_BD (a5api.h).

BdHwState.. ..... Board hardware state as reported by the driver (A5_BD_INS, A5_BD_DETACH or A5_BD_UNEQP as defined in *a5api.h*). Note that this field has only three possible values: unequipped, in-service (that is, hardware is available) and detached (for hot-swap). This state does not contain any information about the state of the firmware or the set/reset state of the hardware.

BdHwVersion..... Board hardware version (unpublished values, do not use).

BdHwRev ... . Board hardware revision. In some cases API handles different revisions slightly differently.

Enabled .... If not 0 (zero), the board firmware is running and the board's *BdFwState* is in service. This information is derived directly from the *BdFwState* (see below), and is here only for backward compatibility.

CpuBusType ..... Type of speech bus supported by the board: A5_PCI_BUS (*a5api.h*).

PciBusNumber..... Connected to which PCI bus (always 0).

PciSlotNumber..... The PCI slot into which the board is plugged into. The meaning varies with the operating system. For SCO Unixware, this is the physical slot in the chassis as reported by the BIOS. For Solaris, this is the logical PCI device number.

BdHwModelDescr.. ..... Textual description of the BdHwModel field ("6UB5").

BdHwSerialNum.... Board serial number.

The following fields are initially set by *E53_Init()* and are updated by opening an *EC Board Control Channel*:

BdFwState .. .. Board firmware state (A5_BD_* in *a5api.h)*. This field reflect the state of firmware on the EC board, which means that in addition to the two states of A5_BD_OOS and A5_BD_INS, the board can be in the same firmware states as a 6UB5 speech processing board.[4]

PortPerBd.... ..... Maximum number of EC ports on the board, regardless of whether the board is in service or out of service. For unequipped boards this field is set to 0.

7.4 EC Channel Configuration Requests

This section describes the data structures and constants used in sending configuration requests to the channel. The configuration requests are issued via the *E53_CfgChan()* function.

The first argument to the *E53_CfgChan()* function is the channel ID, or *Cid*. The term 'channel' can apply to either an EC Board Control Channel or to an individual echo cancellation channel. If the *E53_MakeCid()* port argument is zero, it creates a channel ID for an *EC Board Control Channel*, which applies the request to the entire board. If the port is greater than zero, an individual channel is selected.

The second argument is composed of multiple requests OR-ed together. Which requests are allowed (or make sense) depends on whether it is BCC or EC channel, and also depends on the hardware model of the board.

The third argument is a pointer to a data structure *E53_CFGVAL*, which contains data values associated with the requests specified in the second argument.

The forth argument is a pointer to text string. If the E53_CfgChan() reports an error, it points this pointer to a textual error message.

Note that the EC API does not explicitly defines a **channel allocation request**. The first configuration request for a given channel performs an implicit allocation of that channel. To **deallocate** a previously allocated EC channel, use the function *E53_CloseChan()*.

4. Refer to "Speech Processing Board Control Channel (BCC)" on page 59.

Programmers Note: Handling of Incomplete *E53_CfgChan()* Function Call:

In blocking mode, if the application gets control while a blocked *E53_CfgChan()* function call is in progress[5] (completion code is *A5_INPROG*), the application must do one of two things:

- repeat the *E53_CfgChan()* call with the same arguments as the initial call, or
- call *E53_CfgChan*() with the *E53_ABORT_CFGREQ* request.

Otherwise any other call to *E53_CfgReq()* or *E53_CmdReq()* for this channel is rejected with "operation already in progress" error code (*A5_INPROG*).

In asynchronous mode, while the *E53_CfgChan()* is in progress (completion code is *A5_INPROG*), the application must do one of two things:

- wait for the *E53_CfgChan()* call to complete, or
- call *E53_CfgChan*() with the *E53_ABORT_CFGREQ* request.

7.4.1 Configuration Requests for EC Board Control Channel (BCC)

The BCC for the **6UB5** board does not accept any type of configuration requests, except for the implicit allocation request. Therefore for the 6UB5 board model, the only valid BCC request is a BCC allocation request:

```
BccChan = E53_CfgChan(BccCid, 0, NULL, &ErrMsg);
```

7.4.2 Configuration Requests for Echo Cancellation Channel

The following table describes each request and the corresponding values in the *E53_CFGVAL* structure which must be set. Some requests do not require any value structure entries. Structure entries which are not needed for a particular combination of requests will be ignored.

Note the peculiar behavior of **E53_CONN_XMITTS_CFGREQ** and **E53_DISC_XMITTS_CFGREQ** on **6UB5** board model. Because on **6UB5** the transmit timeslot is hard-wired into a specific ETS type of timeslot, the connect and disconnect request for this timeslot do not really have any real effect, but they are still accepted and handled as if they actually did something. The other consequence of the transmit timeslot being hard-wired is that the application must assign only a specific ETS timeslot to a specific EC channel on 6UB5. To select the correct ETS timeslot, the application has two choices:

- Set the *XmitTid* to 0. When the connect request completes, the XmitTid will contain the correct ETS timeslot for the EC port. The application can then use this value to set, for example, the input timeslot of an ASR resource:

```
/*connect EC timeslots*/
EcCid = E53_MakeCid(EcBdNum, EcPortNum);
EcCfgVal.XmitTid = 0;
EcCfgVal.RcvTid = A5_MakeHtsTid(500);
EcCfgVal.RefTid = A5_MakeHtdTid(501);
EcChan = E53_CfgChan(EcCid, E53_CONNTS_CFGREQ, &EcCfgVal, &ErrMsg);
...
/*connect ASR input timeslot to EC transmit timeslot*/
SpCid = S51_MakeCid(SpBdNum, SpPortNum);
A5_Zerout(SpCfgReq);
A5_Zerout(SpCfgVal);
AsrCfgVal.Tid = EcChan->CfgVal.XmitTid;
SpCfgReq.Asr1Req = S51_CONNTS_CFGREQ;
SpCfgVal.Asr1CfgVal = &AsrCfgVal;
SpChan = S51_CfgChan(SpCid, &SpCfgReq, &SpCfgVal, &ErrMsg);
```

5. This happens when the function call is interrupted by a signal.

- Use the macro **E53_Cid2XmtEtsTid()** to translate a given EC channel ID to the proper **6UB5** transmit TID:

```
/*connect EC timeslots*/
EcCid = E53_MakeCid(EcBdNum, EcPortNum);
EcXmitTid = E53_Cid2XmtEtsTid(EcCid);
EcCfgVal.XmitTid = EcXmitTid;
EcCfgVal.RcvTid = A5_MakeHtsTid(500);
EcCfgVal.RefTid = A5_MakeHtdTid(501);
EcChan = E53_CfgChan(EcCid, E53_CONNTS_CFGREQ, &EcCfgVal, &ErrMsg);
...
/*connect ASR input timeslot to EC transmit timeslot*/
SpCid = S51_MakeCid(SpBdNum, SpPortNum);
A5_Zerout(SpCfgReq);
A5_Zerout(SpCfgVal);
AsrCfgVal.Tid = EcXmitTid;
SpCfgReq.Asr1Req = S51_CONNTS_CFGREQ;
SpCfgVal.Asr1CfgVal = &AsrCfgVal;
SpChan = S51_CfgChan(SpCid, &SpCfgReq, &SpCfgVal, &ErrMsg);
```

Programmers Note: 6UB5 EC Input Timeslot Cannot be Shared:

There is currently a serious limitation on the input or reference timeslots used by a 6UB5 echo canceller, in that they cannot be used (for reading) by any resource on the same 6UB5 board. The API does not issues any error messages and does not prevent the application from reading the same timeslot as is read by the 6UB5 echo canceller. But the result is that there is no speech transmitted on this timeslot.

Request Bit Fields	E53_CFGVAL Members	Description
E53_CBFUN_CFGREQ	CbFun	Set the callback function for this channel.
E53_CONNTS_CFGREQ	XmitTid RcvTid RefTis	Connect the transmit, receive, and reference time slots.
E53_DISCTS_CFGREQ	<none>	Disconnect the transmit, receive, and reference time slots. Note that 6UB5 cannot disconnect XmitTid (although it pretends to).
E53_CONN_XMTTS_CFGREQ	XmitTid	Connect transmit time slot.
E53_CONN_RCVTS_CFREQ	RcvTid	Connect receive time slot.
E53_CONN_REFTS_CFGREQ	RefTid	Connect reference (prompt) time slot.
E53_DISC_XMTTS_CFGREQ	<none>	Disconnect transmit time slot. Note that 6UB5 cannot disconnect XmitTid (although it pretends to).
E53_DISC_RCVTS_CFREQ	<none>	Disconnect receive time slot.
E53_DISC_REFTS_CFGREQ	<none>	Disconnect reference (prompt) time slot.
E53_EC_CFGREQ	EcOn	Enable (non-zero) or disable (0) echo cancellation. Also see the related request E53_EC_INITCH.
E53_INITCH_CFGREQ	<none>	Reinitialize the channel. This restarts echo canceller adaptation and is recommended for a new phone call. This request will enable the port regardless of EcOn and E53_EC_CFGREQ.
E53_EXTFREEZE_CFGREQ	ExtFreeze	Enable (non-zero) or disable (0) external freeze. Halt adaptation but continue cancellation with current characteristics.
E53_ABORT_CFGREQ		Immediately terminate an incomplete (in-progress) E53_CfgChan() function call.

7.4.3 Configuration Values Structure E53_CFGVAL

The third arguments to *E53_CfgChan()* is the **E53_CFGVAL structure**. It contains the values for the EC channel or BCC configuration request.[6]

```
typedef struct
{
   A5_TID       RecvTid;           /*in timeslot ID (per channel only)*/
   A5_TID       XmitTid;           /*out timeslot (per channel only)*/
   A5_TID       RefTid;            /*reference timeslot (per channel only)*/
   u_char       EcOn;              /*echo cancellation switch (channel or board)*/
   u_char       ToneDetectOn;      /*tone detect switch (BCC only)*/
   u_char       ExtFreeze;         /*external freeze on or off flag*/
   u_char       CenterClip;        /*center clip filter on or off*/
   int          (*CbFun)(const void*);/*channel callback function*/
   u_long       UsrData1;          /*appl data: saved in channel struct*/
   u_long       UsrData2;          /*appl data: saved in channel struct*/
}
E53_CFGVAL;
```

The values are:

RcvTid..... .....timeslot to receive data for the echo canceller channel,

XmitTid..... .....timeslot to transmit data from the echo canceller channel,

RefTid ....timeslot to use as reference input by the echo canceller,

EcOn..... .....flag to turn echo cancellation on (non-zero, default) or off (0),

ToneDetectOn .....flag to turn modem tone detection on (non-zero, default) or off (0),

ExtFreeze ....flag to turn on external freeze (non-zero) or off (0, default).

CenterClip ... .flag to turn center clip on (non-zero) or off (0, default).

CbFun..... .....pointer to the callback function for this channel,

UsrData1 and UsrData2 ....contain application-defined which values which are copied to the channel structure by the API. This can be used by the application to store arbitrary data associated with the channel.

Example showing the setting up of the Echo Canceller board. This code works for either AYC53 or 6UB5 board model:

```
int                EcBdNum;
int                EcPortNum;
E53_EQP            EcEqp;
E53_CFGVAL         EcCfgVal;
u_long             EcReq;
E53_CID            EcCid;
E53_CH*            EcCh;
const char*        ErrMsg;

EcEqp = E53_Init(A5_BLOCK_MOD, NULL, &ErrMsg)
if (!EcEqp)
       return (ErrMsg);
if (EcEqp->BdStat[EcBdNum].BdHwState != A5_BD_INS)
       return("Board unusable");

/*configure the E53 time slot and turn on the echo canceller for that port*/
EcReq = E53_CONNTS_CFGREQ|E53_EC_CFGREQ;
A5_Zerout(EcCfgVal);
EcCfgVal.RcvTid = A5_MakeHtsTid(20);
```

6. You may notice that Release 2.1 contains some additional fields in this structure. They are non-functional and should not be used.

```
E53CfgVal.XmitTid = 0;
E53CfgVal.EcOn = 1;

/*EcCh->CfgVal.XmitTid is now set to EC transmit TID for both board models*/
EcCid = E53_MakeCid(EcBdNum, EcPortNum);
EcCh = E53_CfgChan(EcCid, EcReq, &EcCfgVal, &ErrMsg);
if ((!Ch) || (Ch->ComplCode != A5_OK))
      return(ErrMsg);
```

7.5 Channel Data

The **E53_CH data structure** contains the current state of the channel.

```
typedef struct{
   E53_CID      Cid;         /*EC channel ID*/
   u_char       Bd;          /*decomposed ChanId*/
   u_char       Port;        /*ditto*/
   u_long       ComplCode;   /*completion code from last request*/
   E53_BD_STAT* BdStat;      /*the board state*/
const char*     ErrMsg;      /*error message for last request*/
   u_long       CfgReq;      /*valid fields in CfgVal*/
   E53_CFGVAL   CfgVal;      /*current channel configuration*/
   u_long       UsrData1;    /*appl data: saved from request*/
   u_long       UsrData2;    /*appl data: saved from request*/
}
E53_CH;
```

The variables are:

Cid .... EC Channel ID,

Bd, Port..... ..... Cid decomposed into a board and port number,

ComplCode. ..... Completion code for the current or last request (see *a5api.h*),

ErrMsg..... ..... text message for the last request (if failed),

BdStat .... pointer to the current EC board state,

CfgReq..... ..... OR-ed value of all configuration requests completed on this channel,

CfgVal..... ..... values used to configure this channel so far.

UsrData1 and UsrData2.... contain values copied from the last configuration request issued on this channel. This can be used by the application to store arbitrary data associated with the channel.

7.6 Event Data

The E53 API can work in either synchronous (blocking) or asynchronous mode. In both modes the function behaves very much like the corresponding speech interface functions (*S51_CfgChan()*). In **synchronous mode** the function *E53_CfgChan()* may block for a very short period of time (usually less than 100 msec) and the requested operation completes before the function returns. In **asynchronous mode**, the function returns immediately, the E53_CH->ComplCode is set to A5_INPROG and the application must receive events from the *E53_EventChan()* function to determine when the requested operation is completed.

For more detailed information about the two modes of operation, refer to descriptions in “Asynchronous Signal Mode” on page 23 and “Synchronous Mode” on page 24, and also to the manual page for the *E53_EventChan()* function in “LSPS API Function Reference” on page 147.

The **E53_EVT structure** contains a description event received from an EC channel. The events are delivered either via a callback function (if defined), or by calling *E53_EventChan()* function (in non-blocking modes only).

```
typedef struct
```

```
{
   E53_CID      Cid;         /*event on which channel*/
   E53_CH*      Ch;          /*ptr to channel state*/
   u_long       EventType;   /*see below*/
   u_long       ComplCode;   /*completion code*/
   char*        ErrMsg;      /*error message*/
}
E53_EVT;
```

The variables are:

Cid..... .....EC Channel ID

Ch..... .....pointer to the EC channel data structure (channel state),

Event Type .. ..is a bit-masked value consisting of two parts: *Event Class* and *Event Type*.

Event Class is one of the following bits:

- E53_COMPL_EVT: completion event (usually completion of a configuration request),
- E53_UNSOL_EVT: unsolicited event (usually BCC channel reporting that a 6UB5 board changed firmware state),
- E53_INPROG_EVT: in-progress event (not used at this time).

Event Type is one of the following bits:

- E53_CFGREQ_EVT: configuration request event (usually completion),
- E53_BDSTATE_EVT: board firmware changed state (supported for 6UB5 only at this time).

ComplCode . .....completion code for this event (see *a5api.h*),

ErrMsg ....text message for error completion,

8. LSPS API Function Reference

This chapter contains the function reference pages for most of the functions and macros available in the API. They are sorted in alphabetical order. These reference pages deal mainly with the syntax of the function calls and macros and with their direct arguments. They do not describe the contents, usage or meaning of hundreds of indirect parameters, such as data structures and fields within those data structures. Also these pages do not describe the detailed capabilities of the LSPS hardware and firmware supported by the API. For all that information, the programmer should refer to the previous chapters.

NAME

A5_DestrThrInfo

SYNOPSIS

```
#include <lsps/a5api.h>
void      A5_DestrThrInfo(void);
```

DESCRIPTION

Remove the current thread from being tracked by the API.

There is a limit of 200 threads that the API can handle at one time. In some instances this may not be enough - for example if a new thread is created for every call instead of the threads being reused. In such cases, the application should remove unused threads. This should be done only in the free (default) threading mode, and at the time when the current thread does not expect to call the LSPS API anymore.

There is no harm done if the application removes the API thread information (using this call) and then uses this thread again to call the API. The API simply creates a new thread information for this thread.

Note that when the API runs out of the internal thread information structures, the API automatically reuses thread information structure for the least recently used (LRU) thread. Therefore the application does not have to call this function. On the other hand, if the application is terminating a thread, it is recommended that it calls this function to prevent automatical deletion of information for a LRU thread that may still exist.

RETURNS

N/A

NAME

A5_MakeHtsTid, A5_MakeEtsTid, A5_MakeVtsTid, A5_GetTid*

SYNOPSIS

```
#include <lsps/a5api.h>
A5_TID     A5_MakeHtsTid(int TsNum);
A5_TID     A5_MakeEtsTid(int BdNum, int TsNum);

#include <lsps/v5vts.h>
A5_TID     A5_MakeVtsTid(int BdNum, int TsNum, int VtsType);

int        A5_GetTidBdNum(A5_TID Tid);
int        A5_GetTidTsNum(A5_TID Tid);
int        A5_GetTidType(A5_TID Tid);
```

DESCRIPTION

The first three macros are used to create Timeslot IDs (TIDs) of different types: hardware bus timeslot (HTS), 6UB5 echo canceller output timeslot (ETS) and virtual timeslot (VTS). The last three macros are used to pull out items of information from an existing TID. The individual arguments to these macros are described in the next paragraph.

RETURNS

The **A5_MakeHtsTid** macro returns a timeslot ID for a hardware bus timeslot (H.110). The **TsNum** argument defines the timeslot on the bus and generally it must be in the range of 0 to A5_HTSNUM_MAX. The specific valid range depends on the hardware bus:

H.110 .....0 to 4095

The **A5_MakeEtsTid** macro returns a timeslot ID for a echo canceller channel output timeslot on a 6UB5 board. The **BdNum** argument defines the 6UB5 board and must be in the range of 0 to A5_BD_MAX. The **TsNum** argument defines the EC timeslot on the 6UB5 board and must be in the range of 0 to A5_ETSNUM_MAX.

The **A5_MakeVtsTid** macro returns a timeslot ID for a virtual timeslot (VTS) on a 6UB5 speech board. The **BdNum** argument defines the speech board and must be in the range of 0 to A5_BD_MAX. The **TsNum** argument defines the individual VTS on the speech board and must be in the range of 0 to A5_VTSNUM_MAX. The **VtsType** argument defines the type of the VTS and must be one of the following constants (defined in the *v5vts.h* header file):

- V5_HOST_READ|V5_SRES_WRITE
- V5_HOST_WRITE|V5_SRES_READ

The **A5_GetTidBdNum** macro returns the board number associated with the given TID. It of course is not applicable to the HTS type of TID.

The **A5_GetTidType** macro returns the TID type associated with the given TID. Note that this is **NOT** related in any way to the *VtsType* argument passed to the *A5_MakeVtsTid* macro. The returned value is one of:

- A5_HTS_TYP hardware bus (H.110) timeslot
- A5_ETS_TYP 6UB5 echo canceller channel timeslot
- A5_VTS_TYP virtual timeslot

The **A5_GetTidTsNum** macro returns the timeslot number associated with the given TID.

NAME

A5_SignalHandler, A5_SignalCheckpoint

SYNOPSIS

```
#include <lsps/a5api.h>
void      A5_SignalHandler(int SigNum);
void      A5_SignalCheckpoint(void);
```

DESCRIPTION

UNIX signal management functions.

The **A5_SignalHandler** function should be called from the application signal handler function to force a return from a blocked API call on this signal. It only works in the blocking (synchronous) mode. It has no effect in non-blocking modes. This function can also be used as a direct UNIX signal handler function, in which case the blocked API calls will always be interrupted for the specific signal. For example:

```
sigset(SIGALRM, A5_SignalHandler);
```

will cause the alarm signal to always unblock a blocked API call.

The **A5_SignalCheckpoint** establishes a point in application code after which any signal is considered to have occurred within the API. If a signal occurs after this function call, it will be under the control of the *A5_SignalHandler()* and could be used to unblock an upcoming blocked API call. If this function is not called, the signal is considered to have occurred inside of the API only while the program is executing inside of the API function call.

More details on signal handling can be found in “Signal Handling (Unix Only)” on page 31.

RETURNS

None.

NAME

A5_StaThread

SYNOPSIS

```
#include <lsps/a5api.h>
int       A5_StaThread(const char** ErrMsg);
```

DESCRIPTION

Switches the API to single-threaded apartment mode.

The arguments are:

ErrMsg..... returned error message or NULL.

The **A5_StaThread** function switched the API threading mode from free-threaded to single-threaded apartment mode (STA). It must be called from the main thread before any other API functions are called by the application. For description of threading modes, see “Using API in a Multi-Threaded Program Environment” on page 32.

RETURNS

1 (success) or 0 (failed, also see ErrMsg).

NAME

A5_StrBdState, A5_BdState2Str

SYNOPSIS

```
#include <lsps/a5api.h>
char*      A5_StrBdState(u_long BdState);
char*      A5_BdState2Str(u_long BdState);
```

DESCRIPTION

Converts API board state code to a descriptive text message.

The **A5_StrBdState** function returns a free text description of the provided API board (card) state. For example if the error code is A5_BD_INS, this function returns “in-service”. If the provided board state code is not recognized, the function returns a string containing “???”.

The **A5_BdState2Str** function returns the constants #define in text format which represents the value of the board state code. For example if the error code is A5_BD_INS, this function returns a string “A5_BD_INS”. If the board state code is not recognized, the function returns a pointer to an empty string.

RETURNS

Pointer to an internally stored text string. Overwriting the returned string is a very bad idea.

NAME

A5_Strerror, A5_Errno2Str

SYNOPSIS

```
#include <lsps/a5api.h>
char*      A5_Strerror(unsigned long ErrCode);
char*      A5_Errno2Str(unsigned long ErrCode);
```

DESCRIPTION

Converts API error number to a text error message.

The **A5_Strerror** function returns a free text description of the provided API error code. For example if the error code is A5_UNDEF, this function returns “not set or not equipped”. If the provided error code is not recognized, the function returns a string containing “???”.

The **A5_Errno2Str** function returns the constants #define in text format which represents the value of the error code. For example if the error code is A5_UNDEF, this function returns a string “A5_UNDEF”. If the error code is not recognized, the function returns a pointer to an empty string.

RETURNS

Pointer to an internally stored text string. Overwriting the returned string is a very bad idea.

NAME

A5_Zerout

SYNOPSIS

```
#include <lsps/a5api.h>
A5_Zerout(VarName);
```

DESCRIPTION

A macro to initialize structures, arrays and variables of all kinds to all zeroes.

Because in some cases the API requires that all unused data fields in a structure, array or any other variable type are set to zeroes, this macro frequently comes handy. Note that the **VarName** argument must be the actual variable name and not a pointer to it.

NAME

E53_CfgChan

SYNOPSIS

```
#include <lsps/e53.h>
const E53_CH*    E53_CfgChan(E53_CID Cid, u_long CfgReq, E53_CFG* CfgVal,
                                                        const char** ErrMsg);
```

DESCRIPTION

Allocates and/or configures one echo cancellation channel.

Cid ..... EC channel ID created using *E53_MakeCid()* macro.

CfgReq .... selects which parameters to configure (bit mask),

CfgVal..... new values for the EC configuration parameters, NULL rests the selected parameters to hardware defaults.

ErrMsg..... returned error message or NULL.

This function blocks in A5_BLOCK_MOD mode and returns immediately with A5_INPROG ComplCode in A5_ASYNC_MODE.

When an EC channel is allocated, it is automatically reset and disconnected from any timeslots that it may have been connected to at the time.

RETURNS

Pointer to the *E53_CH* structure or NULL on error.

Returns a pointer to the E53_CH channel structure on success, or NULL on invalid function call. To determine the success or failure of the request, the application must examine the *E53_CH->ComplCode* variable. The returned pointer remains valid (does not change between function calls) until the channel is closed. On return, the values in the channel structure show the current configuration and the current status of the EC channel or the EC board control channel (BCC). In the asynchronous modes these structures do not necessarily reflect the requested changes. This is because in asynchronous mode the requested parameters are not set immediately on return, but the application must wait for reception of a completion event.

ALSO SETS

E53_CH->ComplCode.. .....completion code for the request

E53_Errno ....external errno for a failure (see file *a5api.h*)

*ErrMsg..... .....text description of the failure (for NULL return)

NAME

E53_CloseChan

SYNOPSIS

```
#include <lsps/e53.h>
int       E53_CloseChan(E53_CID Cid, const char** ErrMsg);
```

DESCRIPTION

Closes down a channel previously opened by *E53_CfgChan()* and disconnects the channel from the H.110 bus time slot(s).

Cid.....channel ID, it must be created using the *E53_MakeCid()* macro.

ErrMsg .....returned error message or NULL.

This function always completes on return, so on return the values in the channel structure show the new configuration. In other words, it is always synchronous non-blocking.

RETURNS

1 (success) or 0 (failed).

ALSO SETS

E53_Errno..... external errno for a failure (see file *a5api.h*)

*ErrMsg ..... text description of the failure (for NULL return)

NAME

E53_EventChan

SYNOPSIS

```
#include <lsps/e53.h>
const E53_EVT**  E53_EventChan(int WaitTime, int* EventCnt,
                                                const char** ErrMsg);
```

DESCRIPTION

Echo canceller channel event retrieval function.

When the application is using *asynchronous event mode*, this function must be called by the application when there is input data pending on the file descriptor associated with the Echo Canceller interface (*E53_EQP.Fd*). One way to do it in UNIX is to use the *poll(2)* system call to detect that there is data to be read from the device driver associated with the Echo Canceller API and then the application calls this function to retrieve the event(s).

When the application is using *synchronous mode*, this function must be called only to retrieve any **unsolicited events** in the same way as described above for the asynchronous event mode.

This function must be called to retrieve events even if the application is using **callback functions**. This is because the callback functions are called from this function. The one exception is the UNIX A5_SIGPOLL_MOD mode, where the callback functions are called from a UNIX signal handler.

The **arguments** are:

WaitTime . ..... maximum time (in msec) to wait for an event, -1 for unlimited time.

EventCnt.. ..... returned number of events (structures), that is, size of the returned array.

ErrMsg..... ..... returned error message or NULL.

Normally this function may block because it **returns** only when at least one of the following is true:

- the WaitTime expired,
- there was a signal (UNIX),
- there is at least one event,
- any one of the API callback functions was called and it returned 0.

RETURNS

Returns a pointer to an array of event structures *E53_EVT*, or NULL on no event. The size of the array is returned in the *EventCnt* argument. This pointer (and therefore the previous event structure data) becomes invalid on the next call to *E53_EventChan()*.

If there are **no events** and the function returned because of one of the four conditions listed above, the function returns NULL.

If there was an **error** (invalid argument, for example), the function also returns NULL. The only way that the application can distinguish a normal NULL return from an error return is by examining error code in the external variable *E53_Errno* which contains one of the standard LSPS success or error codes (see a5api.h).

ALSO SETS

E53_Errno .....external errno for a failure (see file *a5api.h*)

*EventCnt.....size of the returned array

*ErrMsg.....text description of the failure

NAME

E53_Init

SYNOPSIS

```
#include <lsps/e53.h>
const E53_EQP*  E53_Init(int Mode, int (*CbFun)(const E53_EVT*),
                                                  const char** ErrMsg);
```

DESCRIPTION

Initializes the echo canceller API interface.

Initializes the interface between the application and the echo canceller interface, opens the driver file descriptor, and provides board-equipage information. Must be called before any other E53 functions. May be called repeatedly to change the current mode or to define a callback function.

Mode .....determines the overall behavior of the interface. One of the following mode parameters should be used:

A5_BLOCK_MOD selects the synchronous mode in which the function calls block until completion or error. This is the default mode, in which the function calls returns only after the request completes. Note that this default is different from the default mode in the other interfaces (speech S51).

A5_ASYNC_MOD selects the asynchronous event mode, in which the interface function never blocks and the application gets events when the application calls the *E53_EventChan()* function. The events are delivered either to the callback function or returned from the *E53_EventChan()* function.

A5_SIGPOLL_MOD selects the asynchronous poll mode, in which the interface function never blocks and the application gets events when the callback function is called from a SIGPOLL handler. In this mode, the *CbFun* argument is required.

Any one of the following can be also "OR-ed" into the Mode parameter.

A5_CLEANUP_MOD causes the interface to automatically deallocate and clean up orphaned resources and HTS timeslot transmitters which the current application process is trying to use.

A5_NORESCHK_MOD causes the interface to skip resource allocation conflict checking. Resource allocation tracking tables are not updated in shared memory.

CbFun .....the default callback function which the interface will invoke when an event happens on any echo canceller channel that was opened by the application. The argument is a pointer to the event structure E53_EVT* for the corresponding channel. This is the default callback function for all events on the channels opened by the application. This callback function can be superseded by the per-channel callback functions set up via the *E53_CfgChan()* call. Or it can be set to NULL if no callback is desired. The value returned by the callback function is significant only in synchronous mode as follows: zero (integer 0) means to immediately return from any currently blocked API function, otherwise, stay blocked in the current function.

ErrMsg .....returned pointer to an error message, or NULL.

This function always completes on return, so on return the values in the channel structure show the new configuration. In other words, it is always synchronous non-blocking.

RETURNS

Pointer to the *E53_EQP* structure or NULL for a failure.

ALSO SETS

E53_Errno..... external errno for a failure (see file *a5api.h*)

*ErrMsg ..... text description of the failure

NAME

E53_MakeCid, E53_GetBd, E53_GetPort, E53_GetType

SYNOPSIS

```
#include <lsps/e53.h>
E53_CID   E53_MakeCid(int Board, int Port);
int       E53_GetBd(E53_CID Cid);
int       E53_GetPort(E53_CID Cid);
int       E53_GetType(E53_CID Cid);
```

DESCRIPTION

Macros to create and break up an echo canceller channel identifier.

The **E53_MakeCid** creates a Channel ID which is used for the E53_CfgChan() and E53_CloseChan() functions. The arguments are the physical board number (numbering starts at zero) and the board's physical echo canceller port number.

The **E53_GetBd** retrieves the board number information from an existing channel identifier.

The **E53_GetPort** retrieves the port number information from an existing channel identifier.

The **E53_GetType** retrieves a board type information (A5_6UB5_BDTYP) from an existing channel identifier.

Note that port 0 applies to the entire board and is useful for configuration requests which involve the entire board such as reset, enable, etc. Ports 1 through *E53_Eqp.Bd[N].PortsPerBd* (64 for the current Echo Canceller Card) apply to individual echo canceller ports.

NAME

E53_Term

SYNOPSIS

```
#include <lsps/e53.h>
void      E53_Term(void)
```

DESCRIPTION

Shuts down the echo canceller API interface

Closes the file descriptor associated with the Echo Canceller driver and cleans up. Does not change the current board port configuration or port SCbus connections which have been set up on the board.

This function always completes on return, so on return the values in the channel structure show the new configuration. In other words, it is always synchronous non-blocking.

NAME

S51_CfgChan

SYNOPSIS

```
#include <lsps/s51.h>
const     S51_CH*     S51_CfgChan(S51_CID Cid, S51_REQ* CfgReq,
                            S51_CFGVAL* CfgVal, const char** ErrMsg);
```

DESCRIPTION

This function allocates and configures resources on a speech processing channel.

Cid ..... channel ID, it must be created using the *S51_MakeCid()* macro.

CfgReq .... which configuration parameters are being provided in CfgVal.

CfgVal..... values of resource configuration parameters.

ErrMsg..... returned error message or NULL.

This function blocks in A5_BLOCK_MOD mode and returns immediately with A5_INPROG ComplCode in A5_ASYNC_MODE. One exception is if the CfgReq is all zeroes (no request): in such case the function always returns and completes immediately.

RETURNS

Returns a pointer to the S51_CH channel structure on success, or NULL on invalid function call. To determine the success or failure of the request, the application must examine the *S51_CH->ComplCode* variable. The returned pointer remains valid (does not change between function calls) until the channel is closed. On return, the values in the channel structure show the current configuration and the current status of all the resources on the channel. In the asynchronous modes these structures do not necessarily reflect the requested changes. This is because in asynchronous mode the requested parameters are not set immediately on return, but the application must wait for reception of a completion event.

ALSO SETS

S51_CH->ComplCode ..completion code for the current request

S51_Errno.....external error number indicating a failure (see file *a5api.h*)

ErrMsg.....text description of the failure.

NAME

S51_CloseChan

SYNOPSIS

```
#include <lsps/s51.h>
int       S51_CloseChan(S51_CID Cid, const char** ErrMsg);
```

DESCRIPTION

Closes a speech processing channel.

Deallocates a speech processing channel and all its resources which were previously allocated by *S51_CfgChan()* and disconnects the channel from all H.110 time slot(s).

Cid.....channel ID, created using the *S51_MakeCid()* macro.

ErrMsg.....returned error message or NULL.

This function always completes on return, so on return the values in the channel structure show the new configuration. In other words, it is always synchronous non-blocking.

Note that this function call instantaneously terminates all resources on the channel and flushes any data in internal buffers which were not yet written to disk.

RETURNS

1 (success) or 0 (failed)

ALSO SETS

S51_Errno..... external error number indicating a failure (see file *a5api.h*)

ErrMsg..... text description of the failure.

NAME

S51_CmdChan

SYNOPSIS

```
#include <lsps/s51.h>
const     S51_CH*     S51_CmdChan(S51_CID Cid, S51_REQ* CmdReq,
                                  S51_CMDVAL* CmdVal, const char** ErrMsg);
```

DESCRIPTION

This is a general purpose function which issues a request to the speech processing channel.

Cid ..... channel ID, created using the *S51_MakeCid()* macro.

CmdReq... which command(s) to execute.

CmdVal.... values to use by the requested command(s).

ErrMsg..... returned error message or NULL

This function blocks in A5_BLOCK_MOD mode and returns immediately with A5_INPROG ComplCode in A5_ASYNC_MODE. One exception is if the CmdReq is all zeroes (no request): in such case the function always returns and completes immediately.

RETURNS

Pointer to the S51_CH structure on success, or NULL on invalid function call. To determine the success or failure of the request, the application must examine the *S51_CH->ComplCode* variable. The returned pointer remains valid (does not change between calls) until the channel is closed. On return, the values in the channel structure show the current configuration and the current status. In the asynchronous mode these do not necessarily reflect the requested command(s). This is because the requested command(s) are not executed immediately on return, but the application must wait for reception of a completion event.

ALSO SETS

S51_CH->ComplCode .. ..completion code for the current request

S51_Errno..... .....external error number indicating a failure (see file *a5api.h*)

ErrMsg..... .....text description of the failure.

NAME

S51_EventChan

SYNOPSIS

```
#include <lsps/s51.h>
const S51_EVT**  S51_EventChan(int WaitTime, int* EventCnt,
                                                const char** ErrMsg);
```

DESCRIPTION

Speech processing event retrieval function.

When the application is using *asynchronous event mode*, this function must be called by the application when there is input data pending on the file descriptor associated with the Speech Processing interface (*S51_EQP.Fd*). One way to do it in UNIX is to use the *poll(2)* system call to detect that there is data to be read from the device driver associated with the Speech Processing API and then the application calls this function to retrieve the event(s).

When the application is using *synchronous mode*, this function must be called only to retrieve any **unsolicited events** in the same way as described above for the asynchronous event mode.

This function must be also called to retrieve events even if the application is using **callback functions**. This is because the callback functions are called from this function. The one exception is the UNIX A5_SIGPOLL_MOD mode, where the callback functions are called from a UNIX signal handler.

The **arguments** are:

WaitTime.. .....maximum time (in msec) to wait for an event, -1 for unlimited time.

EventCnt .. ..returned number of events (structures), that is, size of the returned array.

ErrMsg..... .....returned error message or NULL.

Normally this function may block and it **returns** only when at least one of the following is true:

- the WaitTime expired,
- there was a signal (UNIX),
- there is at least one event,
- any one of the API callback functions was called and it returned 0.

RETURNS

Returns a pointer to an array of event structures *S51_EVT*, or NULL on no event. The size of the array is returned in the *EventCnt* argument. This pointer (and therefore the previous event structure data) becomes invalid on the next call to *S51_EventChan()*.

If there are **no events** and the function returned because of one of the four conditions listed above, the function returns NULL.

If there was an **error** (invalid argument, for example), the function also returns NULL. The only way that the application can distinguish a normal NULL return from an error return is by examining error code in the external variable *S51_Errno* which contains one of the standard LSPS success or error codes (see a5api.h).

ALSO SETS

S51_Errno..... external error number indicating a success or a failure (see file *a5api.h*)

EventCnt ..... size of the returned array

ErrMsg..... text description of the failure

NAME

S51_Init

SYNOPSIS

```
#include <lsps/s51.h>
const S51_EQP*   S51_Init(int Mode, int (*CbFun)(const S51_EVT*),
                                                  const char** ErrMsg);
```

DESCRIPTION

Initializes the speech processing interface.

Initializes the interface between the application and the speech processing API, opens the driver file descriptor, and provides board-equipage information. Must be called before any other S51 functions. May be called repeatedly to change the current mode or to (re)define a callback function.

Mode..... determines the overall behavior of the interface. One of the following mode parameters should be used:

A5_BLOCK_MOD selects the synchronous mode in which the function calls block until completion or error. The default is asynchronous event mode in which the function calls return immediately and the application receives an event (via *S51_EventChan()*) when the request completes.

A5_ASYNC_MOD selects the asynchronous event mode, in which the interface function never blocks and the application gets events when the application calls the *S51_EventChan()* function. The events are delivered either to the callback function or returned from the *S51_EventChan()* function.

A5_SIGPOLL_MOD selects the asynchronous poll mode, in which the interface function never blocks and the application gets events when the callback function is called from a SIGPOLL handler. In this mode, the *CbFun* argument is required.

Any one of the following can be also "OR-ed" into the Mode parameter.

A5_CLEANUP_MOD causes the interface to automatically deallocate and clean up orphaned resources and SCbus transmitters which the current application process is trying to use.

A5_NORESCHK_MOD causes the interface to skip resource allocation conflict checking. Resource allocation tracking tables are not updated in shared memory and thus the automatic cleanup cannot be done by the API.

CbFun..... the default callback function which the interface will invoke when an event happens on any speech processing channel that was opened by the application. The argument will be a pointer to the event structure S51_EVT* for the corresponding channel. This will be the default callback function for all events on the channels opened by the application. This callback function can be superseded by the per-channel callback functions set up via the *S51_CfgChan()* call. Or it can be set to NULL if no callback is desired. The value returned by the callback function has the following meaning: zero (integer 0) means to immediately return from any currently blocked API function, otherwise, stay blocked in the current function.

ErrMsg..... returned pointer to an error message, or NULL.

This function always completes on return, so on return the values in the channel structure show the new configuration. In other words, it is always synchronous non-blocking.

RETURNS

Pointer to the S51_EQP structure or NULL to indicate a failure.

ALSO SETS

S51_Errno.....external error number indicating a failure (see file *a5api.h*)

*ErrMsg.....text description of the failure

NAME

S51_MakeCid, S51_GetBd, S51_GetPort, S51_GetType

SYNOPSIS

```
#include <lsps/s51.h>
S51_Cid    S51_MakeCid(Board, Port);
int        S51_GetBd(S51_CID Cid);
int        S51_GetPort(S51_CID Cid);
int        S51_GetType(S51_CID Cid);
```

DESCRIPTION

Macros to create and break up speech processing channel identifier.

The **S51_MakeCid** macro creates a Channel ID which is used for the S51_CfgChan() and S51_CloseChan() functions. The arguments are the physical board number (numbering starts at zero) and the board's virtual port number.

The **S51_GetBd** retrieves the board number information from an existing channel identifier.

The **S51_GetPort** retrieves the port number information from an existing channel identifier.

The **S51_GetType** retrieves a board type information (A5_6UB5_BDTYP) from an existing channel identifier.

Note that port 0 applies to the entire board and is useful for configuration requests which involve the entire board such as reboot, etc. Ports 1 through *S51_CHAN_MAX* are the individual virtual ports.

NAME

S51_Term

SYNOPSIS

```
#include <lsps/s51.h>
void      S51_Term(void);
```

DESCRIPTION

Shuts down the speech processing interface.

Closes the file descriptor associated with the Speech Processing Board driver and deallocates all speech channels allocated to this program.

This function always completes on return. In other words, it is always synchronous non-blocking.

WARNING

Deallocates all the resources allocated by this process, in other words anything that this process was doing with the Speech Processing boards terminates immediately. On the other hand, if a process terminates without calling this function, no cleanup is done whatsoever and some orphaned resources may be left behind.

Even though this function deallocates all speech channels, it is still recommended that the application deallocates the channels explicitly via *S51_CloseChan()* before calling this function.

NAME

USR_Trace, USR_PrintTrace, USR_TraceSw

SYNOPSIS

```
#include <lsps/a5api.h>
void      USR_Trace(TraceFlg, (char* Format, .../* args */));
void      (*USR_PrintTrace)(char* Text);
u_long    USR_TraceSw;
```

DESCRIPTION

Application debugging macro.

The **USR_Trace** macro can be used by the application to insert debugging statements into the application code. The **TraceFlg** argument is logically AND-ed with a global external variable **USR_TraceSw** variable, and if the result is not zero, the text message defined by the second argument is passed to a default trace output function A5_PrintTrace. The second argument is a print-like specification enclosed in parentheses, and it defines the text of the trace message. Example:

```
USR_Trace(0x0004, ("the value of data->var is %d", data->var));
```

Note that there is a 8kB limit on length of the generated message, anything longer will be truncated.

By default the output is handled by an internal trace print function, which normally sends the text to the standard output. If the standard output is closed, the output text is appended to a default log file:

/tmp/lsps_api.log..... for UNIX

The standard trace output function *A5_PrintTrace* can be replaced by an application customized function by setting the **USR_PrintTrace** external function pointer to point to a custom trace output function. Note that the text passed to the trace output function does NOT contain a trailing newline, this must be supplied by the trace output function.

This application debugging functionality is built using the API internal debugging functionality which includes the macros A5_Trace, S51_Trace, E53_Trace and V5_Trace. These macros should not be used for application debugging.

More details on debugging and traces can be found in “Debugging” on page 36.

NAME

V5_AbortVts

SYNOPSIS

```
#include <lsps/v5vts.h>
void      V5_AbortVts(A5_TID Tid);
```

DESCRIPTION

Terminates the current asynchronous operation on a virtual timeslot (VTS) and cleans up. This should be used to prematurely stop an asynchronous in-progress request (function call) to create, destroy, open or close a VTS. Although this function resets the internal API state, whether the aborted call changed the state of the VTS (for example created a VTS) is undefined.

This function returns immediately, therefore it is always a non-blocking synchronous call.

Tid ..... virtual timeslot (VTS) ID, created with *A5_MakeVtsTid()* macro

NAME

V5_CloseVts

SYNOPSIS

```
#include <lsps/v5vts.h>
long      V5_CloseVts(A5_TID Tid, const char** ErrMsg);
```

DESCRIPTION

Closes one virtual timeslot (VTS data stream) to or from host application. The VTS must have been previously opened via *V5_OpenVts()*. This function can be executed in either blocking or in asynchronous mode.

Tid.....VTS ID created using the *A5_MakeVtsTid()* macro.

ErrMsg.....returned pointer to an error message, or NULL.

This function blocks in A5_BLOCK_MOD mode and returns immediately with A5_INPROG ComplCode in A5_ASYNC_MODE.

RETURNS

A5_OK..... success,

A5_INPROG..... asynchronous operation is in progress,

A5_*_ERR..... failed (also see *ErrMsg for descriptive message).

ALSO SETS

V5_VTS->ComplCode. ..... completion code for the current request,

*ErrMsg .... on error it contains a text error message.

NOTES

It is important that the host application closes the VTSs that it no longer needs. This is because even if an open VTS is destroyed by another program, it will linger in a “zombie” state until all readers and writers close their access to it. In other words, it works similar to the UNIX files. Therefore leaving a VTS opened unnecessarily may result in a “lost” VTS and will waste resources.

NAME

V5_CreateVts

SYNOPSIS

```
#include <lsps/v5vts.h>
const V5_VTS*    V5_CreateVts(A5_TID Tid, long BufCnt, long BufSize,
                                                    const char** ErrMsg);
```

DESCRIPTION

Creates one virtual timeslot (VTS) on the board. This function can be executed in either blocking or in asynchronous mode.

Tid ..... VTS ID created using *A5_MakeVtsTid()*.

BufCnt ..... number of internal buffers to allocate to this VTS. If set to 0, the system allocates a "reasonable" default number of buffers. There is a maximum limit of 16*128 (2048) buffers for all VTSs on one board.

BufSize .... Size of each internal buffer (in bytes). If 0, the system allocates buffers of "reasonable" default size.

ErrMsg..... returned pointer to an error message, or NULL.

This function blocks in A5_BLOCK_MOD mode and returns immediately with A5_INPROG ComplCode in A5_ASYNC_MOD.

RETURNS

V5_VTS* ....for success. The return (completion) code is in *V5_VTS->ComplCode* and also in *V5_Errno* (in both blocking and asynchronous modes).

NULL ....for failure, the return code is in *V5_Errno* and the *ErrMsg* argument is set to point to a text error message.

ALSO SETS

V5_VTS->ComplCode.. .....completion code for the current request.

*ErrMsg..... .....on error it contains a text error message.

NOTES

It is possible for an application to create VTS to which it has no read or write access, for example of type: V5_SRES_READ|V5_SRES_WRITE.

An attempt to create a VTS which already exists succeeds only if the BufCnt and BufSize parameters match the already existing VTS, in which case the function returns a reference to the already existing VTS. Otherwise it will fail.

The number of VTSs that can be created on each speech board is limited by available memory and by how much memory buffer space is allocated to each VTS (the *BufCnt* and *BufSize* parameters). So even if the application does not exceed the per-VTS buffer limit, it could possibly exhaust the available memory on the board if a large number of VTSs is created by the application(s).

NAME

V5_DataEndVts

SYNOPSIS

```
#include <lsps/v5vts.h>
long      V5_DataEndVts(A5_TID Tid, long Action, const char** ErrMsg);
```

DESCRIPTION

Marks an end of the virtual timeslot (VTS) data stream from host application to board. This function must be called when the application has no more data to transmit but it does not want to close the VTS at this time. This is always a non-blocking synchronous call.

Tid.....VTS ID created using *A5_MakeVtsTid()*.

Action .....one of the following two constants

V5_DRAIN.....end of data stream, silence follows.

V5_DISCARD.....end of data stream, discard all data in internal buffers and pipelines, silence follows.

ErrMsg.....returned pointer to an error message, or NULL.

RETURNS

A5_OK..... success.

A5_TRYAGAIN_ERR . out of transfer buffers, try again after short delay,

A5_SNDMSG_ERR..... could not send message, try again after short delay.

A5_*_ERR..... other error.

ALSO SETS

*ErrMsg ..... on error it contains a text error message.

NAME

V5_DestroyVts

SYNOPSIS

```
#include <lsps/v5vts.h>
long      V5_DestroyVts(A5_TID Tid, const char** ErrMsg);
```

DESCRIPTION

Destroys one virtual timeslot (VTS) on the board. This function can be executed in either blocking or in asynchronous mode.

Tid ..... VTS ID created using the *A5_MakeVtsTid()* macro.

ErrMsg..... returned pointer to an error message, or NULL.

This function blocks in A5_BLOCK_MOD mode and returns immediately with A5_INPROG ComplCode in A5_ASYNC_MODE.

RETURNS

A5_OK .....success,

A5_INPROG .....asynchronous operation is in progress,

A5_*_ERR .....failed (also see *ErrMsg for descriptive message).

ALSO SETS

V5_VTS->ComplCode.. .....completion code for the current request,

ErrMsg..... .....on error it contains a text error message.

NOTES

The VTS being destroyed could have been created by any process (host or board process).

If the application has the VTS open, it is recommended that the application closes the VTS before destroying it. If the application does not close it, an implicit close is done on the VTS before it is destroyed.

Any existing V5_VTS pointer to this VTS becomes invalid upon completion of this request.

It is possible to destroy a VTS which is currently opened by a host or a board process. In such case the VTS is not immediately destroyed, but becomes "invisible" to all processes (except as noted later). Indeed, immediately after a V5_DestroyVts() completes, a new VTS can be created with the same VTS ID. The exception are the process(es) which still have the now "destroyed" VTS open. These processes do not notice anything different about this VTS and can still read or write to it, but this is still the "old" VTS. This VTS is finally destroyed after all processes close the access to this VTS (via V5_CloseVts). This is similar to how file deletion is handled in the UNIX operating system.

An attempt to destroy a nonexisting VTS results in an error.

NAME

V5_EventVts

SYNOPSIS

```
#include <lsps/v5vts.h>
const V5_EVT**  V5_EventVts(int WaitTime, int* EventCnt,
                                              const char** ErrMsg);
```

DESCRIPTION

The application must call the *V5_EventVts()* function when there is data pending to be read on the file descriptor returned by *V5_InitVts()*. Alternately, the application could instead call *V5_EventVts()* function at any time when it has nothing to do (with appropriate non-zero WaitTime). The second method works only if the application must wait on data only from file descriptors which are part of the LSPS API (S51, E53, V5).

WaitTime.. .....Maximum time (in milliseconds) to block. When this is 0, this function does not block. When -1, it blocks indefinitely, subject to the “premature return” conditions. This function will return "prematurely", that is, before the WaitTime expires if one of the following takes place:

1. one or more events are to be delivered to the application,

2. a caught interrupt (signal) occurred.

EventCnt .. ..pointer to number of events delivered, that is, the size of returned V5_VTS array.

ErrMsg ....returned pointer to an error message, or NULL.

RETURNS

Returns an array (of size *EventCnt) of pointers to V5_EVT data structures, one for each virtual timeslot (VTS) that has a new event to report. If there are no events, the function returns NULL and sets *V5_Errno* to A5_OK. If there was an error, the function returns NULL and sets *V5_Errno* appropriately to one of the A5_*_ERR error codes.

ALSO SETS

*EventCnt ..... size of the returned array of pointers.

*ErrMsg ..... on error it contains a text error message.

NOTES

This is the only function that actually reads the data from the board. Also, all callback functions are called from here. That means that if the application does not call this function when there is data on the file descriptor, the API will not work correctly or even not at all.

The application must monitor the file handle returned by *V5_InitVts()* and keep calling this function when necessary as long as it has at least one virtual timeslot (VTS) opened.

The application receives each event only once, either via the *V5_EventVts()* or via a callback function. If a callback function is defined, it takes precedence.

It is acceptable to call *V5_ReadVts()*, *V5_WriteVts()* or *V5_DataEndVts()* from a callback function.

In some cases, even though the *poll()* indicated that there is data on the file descriptor, the *V5_EventVts()* may return no events. This is a normal and expected behavior.

This function handles callbacks on behalf of all parts of the LSPS API (V5, S51, E53).

Each V5_EVT structure contains only one event notification, for example V5_READMORE_EVT. In other words, the different types of events are not OR-ed together.

NAME

V5_InitVts

SYNOPSIS

```
#include <lsps/v5vts.h>
const V5_EQP*    V5_InitVts(int Mode, int (*CbFun)(const V5_EVT* Event),
                                                    const char** ErrMsg);
```

DESCRIPTION

Initializes the virtual timeslot (VTS) API and can set the common callback function.

Mode..... determines the overall behavior of the interface. The mode parameter applies to (is used by) the following calls: *V5_CreateVts()*, *V5_DestroyVts()*, *V5_OpenVts()*, *V5_CloseVts()* and *V5_DataEndVts()*. All other functions are non-blocking and synchronous. One of the following mode parameters should be used:

A5_BLOCK_MOD selects the synchronous mode in which the function calls block until completion or error. The default is asynchronous event mode in which the function calls return immediately and the application receives an event (via *V5_EventVts()*) when the request completes.

A5_ASYNC_MOD selects the asynchronous event mode, in which the interface function never blocks and the application gets events when the application calls the *V5_EventVts()* function. This is the default mode. The events are delivered either to the callback function or returned from the *V5_EventVts()* function.

Any one of the following can be also "OR-ed" into the Mode parameter.

A5_CLEANUP_MOD causes the interface to automatically close a VTS which is currently opened by a dead host process. This is done only on one VTS at a time, and only at the time when the current process issues a create or a open request on the specific VTS.

A5_NORESCHK_MOD causes the interface to skip checking for read/write conflicts caused by another process having the VTS already opened when this process tries to open the VTS. There is a restriction of only one host process being able to open a specific VTS at one time (either for reading or writing).

CbFun..... the default callback function which the interface will invoke when an event happens on any VTS that was opened by the application. The argument will be a pointer to the event structure V5_EVT* for the corresponding VTS. This will be the default callback function for all events on the channels opened by the application. Or it can be set to NULL if no callback is desired. The value returned by the callback function is significant only in synchronous mode as follows: zero (integer 0) means to immediately return from any currently blocked API function, otherwise, stay blocked in the current function.

ErrMsg..... returned pointer to an error message, or NULL.

This function always completes on return, so on return the values in the VTS structure show the current parameters associated with the VTS. In other words, it is always synchronous non-blocking.

RETURNS

Pointer to the V5_EQP structure or NULL to indicate a failure.

ALSO SETS

*V5_EQP .....structure which contains board states and board interface parameters,

*ErrMsg.....on error it contains a text error message.

NAME

V5_OpenVts

SYNOPSIS

```
#include <lsps/v5vts.h>
const V5_VTS*    V5_OpenVts(A5_TID Tid, int (*CbFun)(const V5_EVT* Event),
                                                    const char** ErrMsg);
```

DESCRIPTION

Opens one virtual timeslot (VTS) data stream to/from host application program. The VTS must have been created with type of V5_HOST_READ or V5_HOST_WRITE (see *V5_MakeTid*). Once opened, the application must be ready to immediately start receiving VTS events and start reading or writing data from/to this time slot via *V5_EventVts()* and *V5_ReadVts()* or *V5_WriteVts()*. This function can be executed in either blocking or in asynchronous mode.

Tid..... .....VTS ID created using *A5_MakeVtsTid()*.

CbFun ....callback function specific for this VTS or NULL.

ErrMsg..... .....returned pointer to an error message, or NULL.

This function blocks in A5_BLOCK_MOD mode and returns immediately with A5_INPROG ComplCode in A5_ASYNC_MODE.

RETURNS

V5_VTS*..... for success. The return (completion) code is in *V5_VTS->ComplCode* and also in *V5_Errno* (in both blocking and asynchronous modes).

NULL..... for failure, the return code is in *V5_Errno* and *ErrMsg* is set to point to a text error message.

ALSO SETS

*ErrMsg ..... on error it contains a text error message.

NOTES

The application can open VTSs created by other processes on the host or on the board, as long as their types allow writing or reading by the host and as long no other process on the host has it already opened.

NAME

V5_ReadVts

SYNOPSIS

```
#include <lsps/v5vts.h>
const V5_VTS*    V5_ReadVts(A5_TID Tid, char* Buf, u_long BufSize,
                                                    const char** ErrMsg);
```

DESCRIPTION

Read data from the virtual timeslot (VTS). The function fills the user-supplied buffer with available data and returns immediately. The host application can determine the size of available data from the *V5_VTS.AvailToRead* field, and the number of bytes actually read (after the function returns) from the *V5_VTS.BytesRead field*. This is always a non-blocking synchronous call.

Tid ..... VTS ID created using *A5_MakeVtsTid()*.

Buf..... pointer to user's buffer.

BufSize .... size of Buf in bytes (maximum read size).

ErrMsg..... returned pointer to an error message, or NULL.

RETURNS

VTS data structure.....for success. If data was read, the number of bytes transferred to the user's buffer is in *V5_VTS.BytesRead*.

NULL .....for failure, see *V5_Errno* and *ErrMsg*.

ALSO SETS

Buf..... .....copies speech data into the user’s buffer,

V5_VTS->BytesRead.... .....number of bytes copied,

V5_VTS->AvailToRead .....number of bytes left unread,

ErrMsg..... .....on error, set to point to a text error message.

NOTES

The application should keep reading (repeatedly calling this function) until *V5_VTS.AvailToRead* becomes 0.

Once *V5_VTS.AvailToRead* becomes 0, the application must call *V5_EventVts()* to get more data. In other words, repeatedly calling *V5_ReadVts()* once *AvailToRead* becomes 0 is not sufficient to read data from the board.

This function changes the following variables in the V5_VTS data structure:

- AvailToRead
- ReadBuf
- BytesRead
- DrainSpeech
- DiscardSpeech

NAME

V5_TermVts

SYNOPSIS

```
#include <lsps/v5vts.h>
void      V5_TermVts(void);
```

DESCRIPTION

Shuts down the virtual timeslot (VTS) API. The file descriptor/handle returned from *V5_InitVts()* becomes invalid. This is always a non-blocking synchronous call.

NOTES

Although this function does its best to stop any reads/writes and to close any opened VTSs, it is recommended that the application does such cleanup explicitly via *V5_CloseVts()* for each opened channel.

This function call does not destroy any existing VTSs.

NAME

V5_WriteVts

SYNOPSIS

```
#include <lsps/v5vts.h>
const V5_VTS*    V5_WriteVts(A5_TID Tid, char* Buf, u_long BufSize,
                                                    const char** ErrMsg);
```

DESCRIPTION

Write data to the virtual timeslot (VTS). The function transfers user's data to internal buffers and sends them to the VTS on the board. If there are insufficient internal buffers to hold all of the user's data, the function may write only some of the user's data. The application can determine the number of bytes actually written from *V5_VTS.BytesWritten*, and the number of available buffer space from *V5_VTS.AvailToWrite*. This is always a non-blocking synchronous call.

Tid ..... VTS ID created using *A5_MakeVtsTid()*.

Buf..... pointer to user's buffer.

BufSize .... size of Buf in bytes (bytes to write).

ErrMsg..... ..... returned pointer to an error message, or NULL.

RETURNS

VTS data structure.....for success. The number of bytes written from the user's buffer is in *V5_VTS.BytesWritten*.

NULL .....for failure, see *V5_Errno* and *ErrMsg*.

ALSO SETS

V5_VTS->BytesWritten .....number of bytes written from the user’s buffer,

V5_VTS->AvailToWrite .....number of bytes in internal buffers that is available for data transfer,

*ErrMsg..... .....on error it contains a text error message.

NOTES

If all of the data could not be transmitted because of lack of internal buffers, the function returns a good V5_VTS pointer, the *V5_VTS.BytesWritten* is set to the actual bytes written and the *V5_VTS.ComplCode* is set to A5_TRYAGAIN_ERR or A5_SNDMSG_ERR. Application should try again after receiving a V5_WRITEMORE_EVT event.

This function changes the following variables in the V5_VTS data structure:

- AvailToWrite
- BytesWritten
- WriteBuf

9. LSPS SDK Commands and Tools Reference

This chapter contains the reference pages for most useful and most often used LSPS commands and tools. They are sorted in alphabetical order.

NAME

a5dsm - display LSPS API timeslot and channel allocation tables

SYNOPSIS

```
a5dsm
```

DESCRIPTION

Display the current contents of the two shared memory segments used by the API. These segments are used to 1) keep track of SCBus transmitting resources, and 2) keep track of allocated API channels. The display includes:

- Current timeslot transmitter allocation for HTS timeslots (H.110 bus). This list includes speech board channels and EC board channels which are currently transmitting data to the H.110 bus timeslots (HTS).
- Current channel allocation for the LSPS boards.
- Current allocation of virtual timeslots (VTSs).

NOTES

Rebooting a board does not clear it's entries in shared memory, although it should. To forcibly clean up the shared memory, remove the two shared memory segments[1] with keys of 0x7374656d and 0x73746530 (on Unixware 386), or 0x6d657473 and 0x30657473 (on Solaris 386).

If a program initializes the API in the A5_NORESCHK_MOD (no resource checking mode), the resources allocated by such program do not show in the output of *a5dsm* command.

1. Use the Unix commands *ipcs* and *ipcrm* to get shared memory status and remove the shared segments respectively.

NAME

asr_capacity - estimate maximum number of ASR channels on a speech board

SYNOPSIS

```
asr_capacity -t BoardType -a AsrName [-g Gram-4-Tuple1 [-g Gram-4-Tuple2]...]
```

DESCRIPTION

asr_capacity estimates the number of channels that can be run on a 6UB5 board for the given speech board type and the given ASR Capability.

The *ASR Capability* is defined as a combination of an *ASR Type* and *ASR Name*, which corresponds to a section in an ASR inifile.[2] The section name corresponds to the *AsrType* and *AsrName* parameter that an application passes to the API. The inifile can be found in the *BoardType* file that corresponds to the BoardType as reported by status51. For example, BoardType "ASR-CD" has a corresponding BoardType file

```
/usr/lsps/rte/BoardType/ASR-CD
```

which refers to the ASR inifile

```
/usr/lsps/asr/ini/ASR-CD.ini
```

The drop-in (dynamic) grammars may be specified using the Grammar-4-Tuple command line option(s). The term *Grammar-4-Tuple* refers to a group of four ASR parameters which uniquely defines a grammar. These parameters are: *Field Name, Lex File, Grammar File,* and *Grammar Type* and they are described in much detail in the *ASR & TTS User's Guide*. When they are specified together as a 4-tuple (as is the case here), they must be in a single string with no extra spaces and with each parameter separated by a semicolon. For example:

"DYNAMIC;mylexfile.lex;mygramfile.gc;gc"

asr_capacity works by building the grammars specified by the capability and by any additional Grammar-4-Tuples that would have been passed to the API as dynamic grammars potentially used by the application. The complexity of the given ASR Capability is computed using the size and connectivity of the combined grammar graph.

The output of *asr_capacity* is written to stdout. Here is an example of a generated output:

```
estimated number of channels per 6UB5 board for CD_UNKNOWN: 36.3
```

This estimate does not take into account the quality of the spoken speech presented to the application. Noisy conditions, poor quality phones, and poor articulation are among many factors that can reduce the actual channel capacity. A better estimate of channel capacity can usually be found by trying a service in the expected environment and measuring the performance.

Command line options:

-t BoardType... .....Corresponds to a file in /usr/lsps/rte/BoardType directory.

-a AsrName..... .....The type of ASR chosen for the application.

-g Gram-4-Tuple.....Grammar specification for one drop-in grammar. The format is:
drop_in_name;lex_file;grammar_file;grammar_type

EXAMPLE

```
asr_capacity -t ASR-CD -a CD_UNKNOWN
```

SEE ALSO

status51, reboot51, bdstate, bdcmd

2. The ASR initialization files (inifiles) are not directly accessible or changeable by the application. Each combination of ASR Type and ASR Name has it's own section in the ASR inifile.

NAME

bdcmd - send a command to an LSPS speech board

SYNOPSIS

```
bdcmd [-B BdNum][-C A|B][-F 0|1][-E]
```

DESCRIPTION

This is an interactive program which can be used to send a wide variety of commands to the LSPS speech boards. It prompts for the board number, for the command and for any additional necessary parameters and then uses the LSPS API to send the desired command to an LSPS speech board:

boot ..... Reboots the board. The user is first prompted for the new board type.

ins ..... Change the board state to "in-service". If the board is currently in-service (see *bdstate* command), this command has no effect. Otherwise it reboots the given speech board so the board can get to "in-service" state. The user is prompted for the new board type.

oos ..... Change the board state to "out-of-service". While in out-of-service state, the board is dead and it does not respond to any direct commands or pings. The only thing that can be done is to reboot it either directly or via the "in-service" command.

meas ..... Repeatedly acquire board CPU statistics and display them. The command runs for a specific number of seconds.

ping ..... Send a ping message to the board. If the board does not respond, the command reports a time-out.

detach ..... Change the specified board to the detached state, in which the board can be removed and replaced. To return the board back to service, it must be rebooted using the boot command.

led ..... change the illumination state of the board LEDs.

clock ..... change the bus clock configuration: set the primary clock (A or B) and enable or disable the automatic fallback (6UB5 only).

Optional arguments:

-B BdNum ..... determines the board number *BdNum* to which apply the immediately following -C and -F options.

-C ClkId ... . sets the primary clock source to A or B on the board determined by the preceding -B option.

-F 1|0 .... enables (1) or disables (0) the automatic fallback from primary to secondary clock source on the board determined by the preceding -B option.

-E..... ..... enables reporting of all API events received during the command execution.

NOTES

Another way to reboot a board is via the *s51boot* command or via the *reboot51* command.

SEE ALSO

s51boot, bdstate, status51, lspsspmon

NAME

bdstate - show & monitor the state of the LSPS speech and echo canceller boards

SYNOPSIS

```
bdstate [-Q][-S][-E][-C][-V][-X][-I]
```

DESCRIPTION

Displays the current states of the speech boards and echo canceller boards using the LSPS API. Optionally it can block and monitor and report any change in the speech and EC board states. In this mode it runs until interrupted from the keyboard.

-Q..... Display initial board state and quit. Do not continue to monitor board states.

-S ..... Monitor only the speech boards.

-E ..... Monitor only the echo canceller boards.

-C ..... Use a callback function to get and display all events received on the board control channels.

-V ..... Use EventChan() functions to get and display all events on BCC channels.

-X ..... Do not close the board control channels on exit, leave them allocated.

-I ..... Do not ignore errors when opening BCCs, quit on any error.

NOTES

Another way to display speech board state summary is via the *status51* command.

SEE ALSO

status51, bdcmd

NAME

debug51- interactive debugging shell

SYNOPSIS

```
debug51 [-l LogFile][-n][-q][-s] BdNum
```

DESCRIPTION

This starts an interactive debugging shell on the specified speech board. Once started, you can get a summary of available commands by typing "help". To exit, type Ctrl-C or Delete.

Options:

-l LogFile .Save all output in the given logfile.

-n .....Ignore any existing locks on the board. A lock for a given board is automatically set for a user who runs the *debug51* command on that board. The locks prevent two users from running this command on the same board.

-q .....Reduce time-outs.

-s .....Do not clear screen.

BdNum.....Board to debug: 0-32.

NOTES

Currently it resides in */usr/lsps/bin*.

NAME

ear - listen to local or remote timeslot using local speech device

SYNOPSIS

```
ear -s HtsTimeSlot | -S VtsTimeSlot [-h RemoteName][-r RecFile][-a]
          [-e Encoding][-d AudDevice][-b SpeechBd][-p SpeechPt][-v VtsNum]
```

DESCRIPTION

Listens to a given timeslot on the local or on a remote machine, and plays the sound on the local sound device. Also see the manual page for **rear** command.

-s HtsTimeSlot ..... defines the HTS (H.110 bus) timeslot to listen to. Must be < 4096.

-S VtsTimeslot ..... defines the VTS timeslot to listen to, Must be <128, requires -b option.

-h RemoteName..... if specified, the program attempts to connect to this host and will transfer the speech data from the remote host to the local host where they are played. If the -h argument is not specified, the program attempts to use the local LSPS hardware in order to listen to the local timeslot. The remote host must be running */usr/lib/lsps/eardmn*.

-r RecFile..... ..... the speech signal is also recorded to the RecFile.

-a..... ..... record all silence. Normally silence is not recorded. Used together with the -r option.

-e Encoding..... ..... a hexadecimal number defining the encoding on the timeslot. Applies only when listening to the Virtual Timeslots (-S VtsTs option). The default is bit-reversed 8-bit mu-law (LSPS standard). The encoding values are (see s51.h):

0x0001..... Mu-Law 64 kbps (LSPS, default)
0x0002..... A-Law 64 kbps (LSPS)
0x0004..... Linear 128 kbps
0x0008..... Linear little-endian 128 kbps
0x0010..... Linear 8-bit 64 kbps
0x0020..... Mu-Law 64 kbps (G.711)
0x0040..... A-Law 64 kbps (G.711)
0x0080..... OKI ADPCM 32-bit
0x0100..... ADPCM 16 kbps (G.726)
0x0200..... ADPCM 24 kbps (G.726)
0x0400..... ADPCM 32 kbps (G.726)
0x0800..... ADPCM 40 kbps (G.726)
0x1000..... ADPCM 32 kbps (Rhetorex)
0x2000..... OMD SubBand
0x4000..... LC-CELP code
0x8000..... generic wave file to play.

-d AudDevice.. ..... audio device to use, machine and OS dependent. If not specified, the program uses first suitable audio device found.

-b SpeechBd.... ..... speech board to use to listen in on a timeslot. If not specified, the program finds and uses first suitable and usable speech board.

-p SpeechPt..... ..... speech port to use to listen in on a timeslot. If not specified, the program finds and uses first suitable and usable speech port (usually the highest one possible).

-v VtsNum..... ..... virtual timeslot to use to listen in on an HTS timeslot. If not specified, the program finds and uses first suitable and usable VTS (usually the highest one possible).

NOTES

Because this program provides access to both, the local and remote timeslots, it works only on systems which have installed the LSPS API. These include Unixware 386 and Solaris 386. Because these systems do not support local audio hardware, on these systems the program can only record.

For remote listening use the "remote ear" (**rear**) program. It does not need LSPS API on the local host, so it can run on more OSs, such as WinNT and Solaris on SPARC. The remote system must be running the ear daemon (**eardmn**), this is part of the LSPS SDK package *lspssdk*.

NAME

ftcomp - compiler for EN-TTS text preprocessor rules

SYNOPSIS

`ftcomp [-o` *outfile*`] [-g] [-P] [-D` *def*`] [-U` *def*`] [-Y` *dir*`] [-I` *dir*`]` *infile*

DESCRIPTION

This compiler (*ftcomp*) compiles a specification written in the language described in the chapter about the EN-TTS text preprocessor in the *ASR & TTS User's Guide*. It generates a binary file from a source file. This binary file is then read by TTS, which uses the file to determine the matching word sequences and any expansions or changes that need to be done on the word sequence. For details refer to the chapter "TTS Text Preprocessing Customization"

infile.....Name of the input file (required).

outfile.....Name of the output file. If not specified, defaults to input file name less any extensions and extended with *".b"*. For example: *foo -> foo.b*, or *foo.s -> foo.b*.

-P.....Run the *cpp* preprocessor only. The output is in file named the same as the input file, less any dot suffix and suffixed with *".i"*. For ex.: *foo -> foo.i*.

-g.....Include the symbol tables in the outfile. Useful when *ftcdump* is run on the *outfile* later.

The remaining options are passed to the *cpp* preprocessor.

NOTES

Because it requires a big-endian machine, this program currently works only on Solaris on SPARC.

Install the Solaris SPARC version of the program as follows:

- install the lspssdk package on an Intel 386 machine (SCO/Unixware 386 or Solaris 386),
- copy the file */usr/lsps/solsparc/ftcomp* from the 386 machine to a SPARC machine,
- run *ftcomp* on the SPARC machine.

NAME

ftcdump - dump the contents of compiled EN-TTS text proprocessor rules

SYNOPSIS

`ftcdump` *`file`*

DESCRIPTION

Displays the contents of a TTS Preprocessing Rules file. The *file* argument must be a file previously generated by *ftcomp*, preferably with the **-g** option. Output of *ftcdump* goes to s*tdout*.

NOTES

Because it requires a big-endian machine, this program currently works only on Solaris on SPARC.

Install the Solaris SPARC version of this program as follows:

- install the lspssdk package on an Intel 386 machine (SCO/Unixware 386 or Solaris 386),
- copy the file */usr/lsps/solsparc/ftcdump* from the 386 machine to a SPARC machine,
- run *ftcdump* on the SPARC machine.

.

NAME

gc_make - ASR grammar compiler

SYNOPSIS

```
gc_make [-o outfile][-f opt][-g opt][-p opt][-l][-v][-d][-h]
                                          gramfile [lexfile1]...[lexfileN]
```

DESCRIPTION

This script which converts a JSGF ASR grammar file into a syntactic grammar graph that can be used by the Lucent Speech Processing Solutions ASR Resource. The script also creates a lexicon from the input JSGF file. For more details refer to the LSPS *ASR&TTS User's Guide* book.

-o outfile... .....Basename of the output files. The default is *"gramfile"*. The compiled grammar is stored in binary file *outfile.mv*. The output lexicon will be stored in *outfile.lex*.

-f opt..... .....Pass the flag to "*fsg_compile*", the grammar compiler.

-g opt..... .....Pass the flag to "*gc_opt*", the grammar optimizer.

-p opt..... .....Pass the flag to "*pron*", the pronunciation tool.

-l..... .....Do not look up pronunciations in the lexicon.

-v..... .....Verbose mode.

-d..... .....Debug mode.

-h..... .....Print usage summary.

gramfile.... .....Grammar specification file. On **Unix**, a '.gram' extension is assumed if not present in this argument. When running on **WinNT** system, the '.gram' extension *must be omitted*.

lexfileN ....user-supplied lexicon. Any pronunciation in the automatically generated lexicon will be overridden with a corresponding entry in this user lexicon.

If the grammar being compiled refers to **built-in rules**, the JSGFPATH environmental variable must include a directory which contains the built-in rules grammar file *lssbuiltin.gram* (LSPS built-in rules) or *builtin.gram* (VoiceXML built-in rules). Since the *lssbuiltin.gram* contains all the rules that are in *builtin.gram* (and more), using the *lssbuiltin.gram* is sufficient in all cases. These built-in grammar files are installed as a part of the package (*asrenna* or *asrenuk*) in the directory */usr/lsps/asr/asrenna* (for NA English) or in */usr/lsps/asr/asrenuk* (for UK English). If you are compiling on a non-native OS (for example Solaris SPARC or WinNt), you must copy one of these built-in grammar files onto the compiling host and point the JSGFPATH to this directory, so gc_make can find the built-in grammar file.

NOTES

Versions for the following OSs are available: WinNt, SCO/Unixware 386, Solaris 386 and Solaris SPARC. This program is installed as a part of the *lspssdk* package. On a machine which does not have *lspssdk* package installed, install it as follows:

- install the *lspssdk* package on an Intel 386 machine (SCO/Unixware 386 or Solaris 386),
- copy the files from the appropriate OS directory on the Inter 386 machine */usr/lsps/OS/** to the destination machine. The "OS" can be one of the following

  nt386 ..... run on WinNT or Windows98

  solsparc ..... run on Solaris on SPARC

To run this script, two environmental variables must be set correctly: *PATH* and *ASRGC_DIR*.

On **UNIX**, the script sets these variables automatically, although it is harmless to set them manually (correctly). The path "*/usr/lsps/bin*" represents the directory where the ASR GC programs and files are usually installed by default.

```
export ASRGC_DIR=C:/usr/lsps/bin
PATH=$PATH:/$ASRGC_DIR
```

On **UNIX** the '.g' extension is optional:

```
gc_make gramfile.gram [lexfile.lex]
```

On **WinNT** system, the user must set these variables explicitly (manually). Below the path "*C:\lsps\asrgc*" represents the directory where the ASR GC programs and files are usually installed.

```
set ASRGC_DIR=C:\lsps\asrgc
set PATH=%PATH%;%ASRGC_DIR%
```

On **WinNT** system the '.gram' extension must be omitted:

```
gc_make gramfile [lexfile.lex]
```

SEE ALSO

pron, mpron

NAME

genini - regenerate speech board initialization files

SYNOPSIS

```
genini {-A | -t BoardType}[-O][-S][-v][-h|-?][-r RootDir]
```

DESCRIPTION

This command (re)generates the speech board initialization files from the intermediate files created by the *mkboardtype* command. It is run automatically by the *mkboardtype* command, and also when a new version of LSPS software is installed (*lspsrte* package). This allows to use the same (previously created) intermediate speech board configuration files, even if the format of the initialization files used by the speech boards have changed between different releases of LSPS.

This command should be run after any of the intermediate files used to define custom board types (in */usr/lsps/cfg/customer*) were changed manually.

-t BoardType ... Board Type

-A .... Generate speech board initialization files for ALL config files

-O .... Dump complete Options List

-S .... Dump Selected options stack

-v .... Verbose

-?|h .... Help

-r RootDir. ..... Lsps Root directory, default */usr/lsps*.

NOTES

Currently resides in */usr/lsps/bin*.

SEE ALSO

mkboardtype

NAME

lspsinst - LSPS software installation tool for UNIX

SYNOPSIS

```
lspsinst -u [-d pkgdir] [-m]
lspsinst -b MODEL [-d pkgdir] [-m]
lspsinst -a|-t [-d pkgdir]
lspsinst -r
```

DESCRIPTION

This command is designed to simplify the installation of LSPS software packages on UNIX machines. It can be used in most circumstances, although in some custom system configurations the user may have to install the packages explicitly via the *pkgadd* command.

-u..... ..... Updates all previously installed LSPS packages. The scripts reinstalls only the packages which are currently already installed on the system. The packages are reinstalled with exactly the same options as the already installed packages.

-b MODEL ..... This should be used to install new software which is not installed on the system. The MODEL designates the board model for which the software should be installed. This tool installs all of the software which is needed to operate the specified board model. The -m option can be used to inhibit installation of ASR and TTS software. The MODEL can be one of the following:

6UB5 ..... cPCI speech board

6UB6 ..... cPCI speech board

-a..... Installs only the ASR language packages. All ASR languages are installed.

-t ..... Installs only the TTS language packages. All TTS languages are installed.

NOTES

Currently resides in the top directory of the software package on CD-ROM. So for example if the CD-ROM is mounted as */cdrom*, this tool can be found in */cdrom* and would be executed as `/cdrom/lspsinst ...`

SEE ALSO

NAME

lspsspmon - monitor speech board performance and load

SYNOPSIS

```
lspsspmon [-m] BoardNum
```

DESCRIPTION

This command continuously reads measurements from the given speech board and displays them in real time. Some of the measurements are: CPU load (in percent of idle), available/unused memory (in kB) and message queue statistics (delivered/returned/discarded messages).

The arguments are:

-m..... .....show message statistics only (no CPU performance).

-BoardNum .....board number to monitor (0 to 32).

NOTES

On the screen the CPU measurements are continuously overwritten by new measurements, giving an impression of a real-time display. Typing a "return/enter" saves the last displayed line from being overwritten.

Currently resides in */usr/lsps/bin*.

SEE ALSO

bdcmd (command *meas*)

NAME

mkboardtype - create a new type of speech board

SYNOPSIS

```
mkboardtype -t BoardType [-f][-i][-m Release][-s][-r][-v][-r RootDir][-?|-h]
                                                          [-d][-c][-V]
```

DESCRIPTION

This is an interactive program to create or update customized speech board types.

-t BoardType... Board Type.

-f .... Force overwrite of existing files.

-i .... Ignore missing files.

-m Release ..... Release compatibility mode (2.0 - 3.0). >). If specified, this option will result in the section [COMMON] being added to the ASR ini file. The [COMMON] section will contain "ReleaseMode=value" where value is either 2, or 3. For release 2.3 and before the value is 2. For release 3.0 and beyond the value is 3.

-s .... Skip generation of speech board initialization files.

-v..... ..... Verbose.

-V..... ..... Validate, write to stdout only.

-r RootDir ..... LSPS Root directory, default */usr/lsps*.

-?|h..... ..... Help.

Developer debugging options:

-d..... ..... Enable ASR GC Debugging. When this option is specified it will cause the program to collect an debug flag option during custom grammar entry. Note that this debug flag will only be collected for custom grammars. The debug flag collected will be added to the ini file in the form
"debug=+ value"
where value is the value collected. NOTE: A space is required between =+ and the value.

-c..... ..... Check & display definition tree

-V..... ..... Validate, and dump option tree (write to stdout only)

NOTES

Currently resides in */usr/lsps/bin*.

SEE ALSO

genini

NAME

mpron - create pronunciation for a list of words - multiple languages

SYNOPSIS

```
mpron [-f path][-s sil][-w word_sep][-p phone_sep][-x][-l language][-o]
                              [-t file_name][-d][InFile1]...[InFileN]
```

DESCRIPTION

This command creates a pronunciation for a set of words. It reads the given input file(s) (or a standard input if no files given) to get the words. It then generates a list of pronunciations for these words on the standard output. The pronunciations use the phonetic alphabet appropriate to the language being used. Each language has it's own phonetic alphabets, these are listed in the *LSPS TTS and ASR User's Guide*.

This tool is functionally identical to *pron*, except it can obtain phonemic pronunciations for (lists of) words in number of languages. As such, *mpron* has been informally described as "brother of pron." It supports the same list of languages as provided with multi-lingual TTS. As of Release 2.3 the list includes: North American English (enna), Continental French (fr), German (ge), Italian (it), North American Spanish (spna), and Continental Spanish (spca).

-f path..... .....Look for data in *path* directory. Overrides the default of */usr/lsps/mltts*.

-s sil..... .....Set between-word silence phone string to *'sil'*. Overrides the default of *'#'*.

-w word_sep.....Set word separator string to '*word_sep*'. Overrides the default of "" (empty string).

-p phone_sep...Set phone separator string to '*phone_sep*'. Overrides the default of "" (empty string).

-x..... .....Don't output lines that have no pronunciation.

-l language .....Specify language to be used (see the language keywords below). Overrides the default of *'enna'*.

-o..... .....Print phonemes in CHC representation. Overrides the default of "KWC" representation.

-t file_name.....Use alternate channel setup file '*file_name*'. This option should not be used together with '-l' option.

-d..... .....Print debugging information.

-?|h ....Help.

When selecting a language, it can be specified with either a language tag or a language name string. The following tags and strings are currently recognized as valid language specifies:

Language Tag	Language String	Alphabet
enna	North-American English	A-1
it	Italian	I-1
ge	German	G-1
spna	NorthAmericanSpanish	M-1
spca	CastilianSpanish	S-1
fr	French	
frca	CanadianFrench	C-1
enuk	UK English	U-1

mpron is delivered as part of the *lspssdk* package. After installation it is located in */usr/lsps/bin*. There is also a Solaris Sparc executable placed in */usr/lsps/bin/solsparc*. At this time there is no WinNT executable to be placed in */usr/lsps/bin/nt386*.

In order to be able to function properly, the executable needs language specific data files, which are delivered as part of mltts language packages (mlttsenna, mlttsge, mlttsit, etc.) and installed into */usr/lsps/mltts* directory. Data files must be visible to the tool at the time of execution. However, only data for the language currently being used has to be available. In other words, if only the *mlttsenna* (American English) package is installed, *mpron* can be used to obtain pronunciations of words in North American English, even though it cannot be used for obtaining pronunciation of French words (an error message will be issued if such operation is attempted).

For proper operation, *mpron* also requires a "channel setup file", which is used for initializing mltts modules within *mpron*. Language specific channel setup files are located in */usr/lsps/mltts/mltts*{language_code}*/setpdata* directory. The file name is of the form "{language_code}*pron.chn*". *mpron* is set up to look for all the files it needs in the default directory */usr/lsps/mltts*. If for whatever reason the data files are not in their default locations, command line options can be used to provide that information to the tool.

In addition to OS specific executables (*uw386* and *sol386*), the *lspssdk* package provides additional executables intended for running on other operating systems and machines, namely Solaris running of Sparc workstations and Windows NT. These files are located in */usr/lsps/bin/solsparc* and */usr/lsps/bin/nt386*, respectively. Currently *mpron* supports Solaris Sparc but at this time it is unavailable for Windows NT environment.

If Solaris Sparc copy of *mpron* is to be used, it must first be copied to a Solaris Sparc workstation. Data files located in */usr/lsps/mltts* directory must be made visible to the tool for proper operation. If the default data directory cannot be seen from where the executable was copied to, the entire contents of */usr/lsps/mltts* directory (or only a language specific subdirectory therein) must also be copied to a location that can be visible at run time. The location should be conveyed to the executable using the '-f' option.

NOTES

Currently resides in */usr/lsps/bin*.

SEE ALSO

pron, gc_make

NAME

pron - create pronunciation for a list of words - North American English

SYNOPSIS

```
pron [-h|-?][-f InDir][-s Sil][-w WordSep][-p PhonSep][-r RemMod]
                        [-c][-1][-x][InFile1][InFile2]...[InFileN]
```

DESCRIPTION.

This command creates a pronunciation for a set of words. It reads the given input file(s) (or a standard input if no files given) to get the words. It then generates a list of pronunciations for these words on the standard output. The pronunciations use the Phonetic **Alphabet B** For alphabets, see the *ASR & TTS User's Guide*.

The input consists of words (or word groups), each separated by a newline. For example:

```
phone
number
phone-number
```

The output consists of the input word followed by it's pronunciation. For example:

```
phone           fon
number          n^mbR
phone-number    fon#n^mbR
```

The format of the input can be modified by using command line options as described below.

Command line options:

-h or -?..... .....Display command usage summary.

-f InDir..... .....Look for input files in directory *InDir* (default '.').

-s Sil..... .....Set "between-the-word-silence" phone string (in output) to *Sil*. The default is "#". So for example for input word "phone-number" and option `-s ""` (null) would result in:
`phone-number  fon^mbR`

-w WordSep.... .....Set the word separator string (in output) to *WordSep* character. The default is "" (null). If *WordSep* is "" (null, also the default), the output word is the same as the input word string. So for example for input word "phone-number" and option `-w "_"` (underscore), the output would be:
`phone_number  fon#n^mbR`

-p PhonSep..... .....Set the phone separator string (in output) to *PhonSep*. The default is "" (null). So for example using `-p " "` (space) would result in:
`number  n ^ m b R`

-r RemMod..... .....A pronunciation can come from any of the following modules, which are applied in the order given below. **Use this option to remove one or more modules** by adding the corresponding numerical values and passing the total as the *UseMod* value. The default value is 16, which means that the NAME module is normally not applied.

1	ROMAN	16	NAME
2	NUMBER	32	RULES
4	SPELL_CAPS	64	SPELL
8	DICT	128	MULTIPLE_WORDS
		256	JOINED_WORDS

Here is an example of MULTIPLE_WORDS:

A B C

If bit 256 is set with -r, then the string 'A B C' would be pronounced as one string rather than concatenating the three individual pronunciations.

Here is an example of JOINED_WORDS:

BatHelmet

If bit 128 is set with -r, then the string 'BatHelmet' would be pronounced as one string (bethelmet) rather than concatenating the pronunciation of 'Bat' with the pronunciation of 'Helmet'.

-c..... .....Treat the 1st letter of each line as if it was capitalized.

-1..... .....Ignore all but the 1st word of each line.

-x..... .....Do not output lines that have no pronunciation

InFile1 - InFileN Input file(s) containing words.

NOTES

Versions for the following OSs are available: WinNt, SCO/Unixware 386, Solaris 386 and Solaris SPARC. This program is installed as a part of the *lspssdk* package. On a machine which does not have *lspssdk* package installed, install it as follows:

- install the *lspssdk* package on an Intel 386 machine (SCO/Unixware 386 or Solaris 386),
- copy the files from the appropriate OS directory on the Inter 386 machine */usr/lsps/OS/** to the destination machine. The "OS" can be one of the following

  nt386.....run on WinNT or Windows98

  solsparc.....run on Solaris on SPARC

SEE ALSO

mpron, gc_make

NAME

rear - listen to remote timeslot using local speech device

SYNOPSIS

```
rear [-s HtsTimeSlot | -S VtsTimeSlot] -h RemoteName [-r RecFile][-a]
          [-e Encoding][-d AudDevice][-b SpeechBd][-p SpeechPt][-v VtsNum]
```

DESCRIPTION

Listens to a given timeslot on a remote machine, and plays the sound on the local sound device. In reality it is a version of the "*ear*" program with it's local LSPS interface not included.

-s Timeslot..... .....defines the remote HTS (H.110 bus) timeslot to listen to. Must be < 4096.

-S VtsTimeslot. .....defines the remote VTS timeslot to listen to, Must be <128, requires -b option.

-h RemoteName .....the program attempts to connect to this host and the given timeslot on that host and to transfer the speech data from the remote host to the local host where they are played.

-r RecFile ....the speech signal is also recorded to the RecFile.

-a ....record all silence. Normally silence is not recorded. Used together with the -r option.

-e Encoding..... .....a hexadecimal number defining the encoding on the timeslot. Applies only when listening to the Virtual Timeslots (-S VtsTs option). The default is bit-reversed 8-bit mu-law (LSPS standard). For the encoding values see the manual page for *ear*.

-d AudDevice.. .....audio device to use, machine and OS dependent. If not specified, the program uses first suitable audio device found.

-b SpeechBd.... .....speech board to use to listen in on a timeslot. If not specified, the program finds and uses first suitable and usable speech board.

-p SpeechPt..... .....speech port to use to listen in on a timeslot. If not specified, the program finds and uses first suitable and usable speech port (usually the highest one possible).

-v VtsNum..... .....virtual timeslot to use to listen in on a HTS timeslot. If not specified, the program finds and uses first suitable and usable VTS (usually the highest one possible).

NOTES

Version of this program is available for WinNT/Windows98 and for Solaris on SPARC. For other OSs, use the *ear* version of this program. The remote system must be running the ear daemon (*eardmn*) which is a part of the LSPS SDK package *lspssdk* and is installed in */usr/lib/lsps/eardmn*.

This program is installed as a part of the *lspssdk* package. On a machine for which there is no *lspssdk* package available (WinNT, Solaris on SPARC), install it as follows:

- install the *lspssdk* package on an Intel 386 machine (SCO/Unixware 386 or Solaris 386),
- copy the files from the appropriate OS directory on the Intel 386 machine */usr/lsps/OS/** to the destination machine. The "OS" can be one of the following

  nt386 ..... run on WinNT or Windows98

  solsparc ..... run on Solaris on SPARC

SEE ALSO

ear

NAME

reboot51 - reboot LSPS speech board

SYNOPSIS

```
reboot51 -t BoardType BdNum
```

DESCRIPTION

Reboots the given speech board.

-t BoardType... defines the BoardType, that is, the software to load into the board. If not specified, the board will hang in boot state forever without any error messages.

BdNum .... board number to boot: 0-32.

NOTES

Currently resides in */usr/lsps/bin*.

A better way to do the same thing is by using the *s51boot* program.

SEE ALSO

s51boot, bdcmd

NAME

s51boot - reboot LSPS speech board

SYNOPSIS

```
s51boot -b BdNum -t BoardType
```

DESCRIPTION

Reboots the given speech board, waits for the boot sequence to complete and reports results. Checks for and reports errors.

-b BdNum .....board number to boot: 0-32.

-t BoardType ...defines the BoardType, that is, the software to load into the board. If not specified, the board will hang in boot state forever without any error messages.

NOTES

Another way to reboot a speech board is via reboot51 command. Currently this program resides in */usr/lsps/bin*.

SEE ALSO

reboot51

NAME

status51 - display state of all LSPS speech boards in the host

SYNOPSIS

```
status51
```

DESCRIPTION

Shows the current summary state of the speech boards in the local host.

NOTES

Another way to display speech and echo canceller board states is via the *bdstate* command.

Currently this program resides in */usr/lsps/bin*.

SEE ALSO

bdstate

NAME

xplot - display and manipulate speech data

SYNOPSIS

```
xplot
```

DESCRIPTION

This is an interactive Windows program (WinNT/Win98) which can perform several functions:

- open a number of speech files,
- display the contents of these files in several modes,
- play selected portions of the speech data on the local sound device,
- edit the speech data,
- save the modified speech data.

NOTES

Currently this program is available only in the WinNT/Win98 version. It must be installed as follows:

- install the *lspssdk* package on an Intel 386 machine (SCO/Unixware 386 or Solaris 386),
- copy the files */usr/lsps/nt386/xplot.exe* and */usr/lsps/nt386/xplot.ini* from the 386 machine to a WinNT machine,
- edit the *xplot.ini* file, so the *[Audio]* section contains a correct play device name for the *PlayDevice* parameter. To determine the play device name for WinNT, check the Preferred device in the Multimedia Properties dialog box. For example for a SoundBlaster it may be:

  ```
  [Audio]
  PlayDevice=Sound Blaster Playback
  ```

- run *xplot.exe* on the WinNT machine.

Index

Numerics

6UB5
CPU performance measurements 60, 63
EC channels 139
echo canceller description 139
echo canceller transmit timeslot 142
H.110 bus clock control 60
hot swap 60
LED control 59

A

A5_AddCust
manual page 148
A5_ASYNC_MOD
asynchronous event mode 21
A5_BdState2Str
manual page 152
A5_BLOCK_MOD
synchronous mode 24
A5_EINTR
synchronous mode 24
A5_EINTR_ERR
signals 31
A5_Errno2Str
manual page 153
A5_GetTid
manual page 149
A5_INPROG 30
signals 31
synchronous mode 24
A5_MakeEtsTid
manual page 149
A5_MakeHtsTid
manual page 149
A5_MakeVtsTid 132
manual page 149
A5_SignalCheckpoint
manual page 150
A5_SignalHandler
manual page 150
synchronous mode 24
A5_SIGPOLL_MOD 23
A5_StaThread
manual page 151
A5_StrBdState
manual page 152
A5_Strerror
manual page 153
A5_Zerout
manual page 154
a5dsm
manual page 182
tools summary 10
A-Law coding on H.110 bus 26
alloc
tools summary 10
allocation
ASR resource 117
connect resource 98
CPT detector resource 94
DTMF detector resource 80
EC channel 141
play resource 69
record resource 74
resource allocation conflicts 34
speech channel 19
speech resource 19
speech resources 45
TDD detector resource 85
tone generator resource 90
transcoder resource 101
TTS resource 109
API
capabilities and benefits 3
function summary 19
header file in Unix package 14
modes of operation 21
parts of 3
speech API summary 43
system architecture 3
application design
call handling architecture 4
application related errors 28
ASR resource 117
allocation and configuration 117
commands 124
example 130
S51_ASR_CFGVAL 119
S51_ASR_CMDVAL 124
S51_ASR_STAT 125
state 125
asr_capacity
manual page 183
asrenna
software packages - Unix 13
speech board types 41
asrfrca
software packages - Unix 13
speech board types 41
asrge
software packages - Unix 13
speech board types 41
asrit
software packages - Unix 13
speech board types 41

asrspca
software packages - Unix 13
speech board types 41
asrspna
software packages - Unix 13
speech board types 41
asy_playd
examples summary 9
asy_tones
examples summary 9
asynchronous event mode
events 30
signals 31
asynchronous mode
asynchronous event mode 21
asynchronous signal mode 23
speech API operation 53
asynchronous signal mode
events 30
signals 31
AtComplEnableResTimers 51
AtComplStartRes 50
AtComplStopRes 50
AtDigitStopRes 51
AtEnergyStopRes 51
AtStopEnableResTimers 51
automatic speech recognition
See ASR
AYC53
error counters 141

B

basic resources
speech board types 40
BCC 59
allocation and configuration 61
and VTS 134
board state structure 64
channel state 64
cleanup 35
code example 66
commands 62
getting 6UB5 CPU performance measurements 60
H.110 bus clock control 60
hot swap 60
LED control 59
BCC commands
S51_BOOT_CMDREQ 62
S51_DETACH_CMDREQ 62
S51_GETPERF_CMDREQ 63
S51_INS_CMDREQ 62
S51_LED_CMDREQ 62
S51_OOS_CMDREQ 62
S51_PING_CMDREQ 62
bdcmd
hot swap 60
manual page 184
tools summary 10
bdstate
manual page 185
speech board types 39
tools summary 10
blocking mode
See synchronous mode
board control channel 59
See BCC
board state 28
S51_BD_STAT 64
board types
creating custom types 42
prepackaged board types 40
boot
6UB5 echo canceller board 140
bdcmd tool 10
channel reset events 59
cleaning up orphaned resources 36
example code 66
hot swap 60
installing Unix packages 16
lspsspboot daemon 59
lspsspstartup 59
S51_BOOT_CMDREQ 62
s51boot command 42
speech board types 40
speech boards 39, 42
TTS board type 108
user lock 63
VTS 131
boot process
lspsspboot 59
lspsspstartup 59
buftest
tools summary 10

C

call progress tone detector
See CPT detector
call progress tones
See CPT
callback function
glossary 5
CfgChan
asynchronous event mode 21
asynchronous signal mode 23
function summary 19
synchronous mode 24
channel
glossary 5
relation to resources and timeslots 20
channel handoff 36
channel ID
see CID
channel structure
error message 29
CID
E53_CfgChan 155

E53_CfgReq 141
E53_CH data structure 145
E53_CloseChan 156
E53_EVT data structure 146
echo canceller BCC 142
echo canceller channel 139
S51_CfgChan 161
S51_CH data structure 55
S51_CloseChan 162
S51_CmdChan 163
S51_EVT data structure 56
S51CfgChan 45
S51CmdChan 48
speech BCC 59
speech channel 43
cleanup after process crash 35
clock
see H.110 clock
CloseChan
function summary 19
CmdChan
asynchronous event mode 21
asynchronous signal mode 23
function summary 19
synchronous mode 25
codes
application related errors 28
completion 28
error 28
fatal errors 29
non-errors 28
non-fatal errors 29
port errors 29
commands 20
ASR resource 124
connect resource 99
CPT detector resource 96
DTMF detector resource 82
play resource 70
record resource 77
TDD detector resource 86
tone generator resource 91
transcoder resource 103
TTS resource 112
compiler
ASR grammar 190
PARS (ftcomp) 188
compiling applications
on SCO Unixware Unix 17
on Solaris 386 Unix 18
on Unix 17
completion code 28
glossary 5
see codes
speech channel request completion 51
completion event
summary 30
ComplRes
speech channel request completion 52
configuration 20
connect resource 98
allocation and configuration 98
commands 99
example 100
S51_CON_CFGVAL 98
S51_CON_CMDVAL 99
S51_CON_STAT 99
state 99
CPT detector resource 94
allocation and configuration 94
commands 96
example 97
S51_TDET_CFGVAL 95
S51_TDET_CMDVAL 96
S51_TDET_STAT 96
state 96
creating custom board types 42
cust
tools summary 10

D

debug51
manual page 186
debugging
A5_Trace 38
application trace macro USR_Trace 37
debugging library liba5api 17
E53_Trace 38
internal API trace 37
S51_Trace 38
summary 36
user trace example 37
using AP debugging libraries 36
USR_Trace 37
V5_Trace 38
detach
BCC command 62
discard speech
description 132
field in V5_VTS data structure 134
drain speech
description 132
in V5_VTS data structure 134
drivers
system architecture 3
drivers - Unix
lspssp 13
DTMF detector resource 80
allocation and configuration 80
commands 82
example 84
S51_DTMF_CFGVAL 82
S51_DTMF_CMDVALVAL 83
S51_DTMF_STAT 83
state 83
dtmf_str
tools summary 10

dtmf_tst
tools summary 10

E

E53_BD_STAT
structure description 140
E53_CfgChan
EC API configuration requests 141
EC API summary 139
echo canceller BCC 142
manual page 155
E53_CFGVAL
data structure description 144
E53_CH
data structure description 145
E53_Cid2XmtEtsTid
echo canceller transmit timeslot 143
E53_CloseChan
EC API summary 139
manual page 156
E53_EQP
structure description 140
E53_EventChan
EC API summary 139
manual page 157
summary 30
E53_EVT
data structure description 146
summary 30
E53_GetBd
manual page 159
E53_GetPort
manual page 159
E53_GetType
manual page 159
E53_Init
data fields set by 141
EC API summary 139
manual page 158
E53_MakeCid
EC API summary 139
manual page 159
E53_Term
EC API summary 139
manual page 160
ear
manual page 187
tools summary 10
eardmn
summary 14
earsrv
summary 14
EC API
AYC53 error counters 141
BCC configuration request 142
booting of 6UB5 board 140
channel allocation 141
channel configuration request 142
configuration requests 141
E53_BD_STAT data structure 140
E53_CFGVAL data structure 144
E53_CH channel data structure 145
E53_EQP data structure 140
E53_EVT data structure 146
events 145
initialization 140
setting xmit timeslot on 6UB5 142
EC BCC
configuration request 142
fields set by 141
EC board control channel
see EC BCC
echo canceller
6UB5 board 139
API description 139
API initialization 140
BCC 142
cannot share input HTS timeslots 34
echo canceller API
see EC API
echo canceller interface
system architecture 4
echo canceller timeslot
See ETS
EC-Out timeslot
See ETS
End Silence Timer
See EST
ErrMsg
speech channel request completion 52
error codes
see codes
error messages
function argument 30
summary 29
EST
end silence timner - description 54
ETS
description 27
how to set for 6UB5 142
summary 25
event
completion 30
glossary 5
in progress 30
summary 30
unsolicited 30
event structure
error message 29
EventChan 30
asynchronous event mode 22
asynchronous signal mode 23
function summary 19
synchronous mode 25
example1
examples summary 9

example2
examples summary 9
example3
examples summary 9
example4
examples summary 9
example6
examples summary 9
examples
apt_thr multi-threaded program 33
ASR resource 130
BCC code 66
configuring EC channel 144
connect resource 100
CPT detector resource 97
DTMF detector resource 84
free_thr multi-threaded program 32
play resource 73
record resource 79
setting echo canceller ETS timeslot on 6UB5 142
summary list 9
TDD detector resource 88
tone generator resource 93
transcoder resource 103
TTS resource 116
VTS 136

F

ftcdump
manual page 189
ftcomp
manual page 188
function error codes 28

G

gc_make
manual page 190
genini
creating custom board types 42
manual page 192
glossary 5

H

H.110 bus clock
control 60
generating from a speech board 43
H.110 timeslot
See HTS
hot swap
6UB5 60
HTS
access from several boards 26
A-Law coding 26
description 26
disconnecting timeslots 27
hardware limitations on 6UB5 27
silence generators 26
summary 25

I

InhTimers
description 54
Init
asynchronous event mode 21
asynchronous signal mode 23
function summary 19
synchronous mode 24
initial silence timer
See IST
in-progress event
summary 30
installation
SPARC tools 17
Unix 15
WinNT tools 17
installation - UNIX
lspsinst 193
installation - Unix
lspsadm 16
package order 16
pkgadd 15
pkgrm 16
SCO Unixware 2.1.3 17
Solaris 386 16
IST
initial silence timer - description 54

L

LED
BCC command 62
control 59
liba5api
how to use 36
liba5api.a
compiling on Unix 17
in Unix package 14
liba5api.so
compiling on Unix 17
in Unix package 14
liblsps.a
in Unix package 14
liblsps.so
in Unix package 14
lspsadm
installation Solaris 386 16
lspsapi
software packages - Unix 13, 14
lspsapi.a
compiling on Unix 17
lspsapi.so
compiling on Unix 17
lspsinst
manual page 193

lspsrte
software packages - Unix 13
speech board types 40
lspssdk
installation 15
software packages - Unix 13, 14
lspssp
software packages - Unix 13
lspsspboot
BCC 59
lspsspmon
manual page 194
lspsspstartup
BCC 59

M

maximum listen timer
See MLT
metatag 128
mkboardtype
creating custom board types 42
manual page 195
MLT
max listen timer - description 54
mlttsenna
software packages - Unix 13
speech board types 40
mlttsfr
software packages - Unix 13
speech board types 40
mlttsfrca
software packages - Unix 13
speech board types 40
mlttsge
software packages - Unix 13
speech board types 40
mlttsit
software packages - Unix 13
speech board types 40
mlttsspca
software packages - Unix 13
speech board types 41
mlttsspna
software packages - Unix 13
speech board types 41
mpron
manual page 196
multasr
tools summary 10
multi-threading
A5_StaThread manual page 151
apt_thr example 33
free threading mode 32
free_thr example 32
introduction 32
single threaded apartment mode (STA) 33

N

naming conventions 6
ntn
tools summary 10
ntnplus
tools summary 10

O

operating systems
installing LSPS on SCO Unixware 2.1.3 17
installing LSPS on Solaris 386 16
supported UNIX systems 13

P

pkgadd
package installation - Unix 15
pkgrm
package removal - Unix 16
play resource 69
allocation and configuration 69
commands 70
example 73
S51_PLAY_CFGVAL 69
S51_PLAY_CMDVAL 71
S51_PLAY_SET 70
S51_PLAY_STAT 72
state 72
playts
tools summary 11
port
glossary 6
port errors 29
programmers note
6UB5 EC input timeslot cannot be shared 143
calling API functions from callback functions 32
Calling API Functions from Callbacks in Async Modes 23
Calling API Functions from Callbacks in Synch Mode 25
calling API functions from signal handlers 31
calling VTS functions from callbacks 136
discarding important events in synch mode 25
disconnecting HTS timeslots 27
event processing in asynch modes 22
handling of incomplete E53_CfgChan 142
handling of incomplete S51_CfgChan 47
handling of incomplete S51_CmdChan 50
hardware limitation of HTS timeslots on 6UB5 board .27
memory usage by dynamically loaded grammars 124
mixing play and TTS 109
per-channel callback functions 47
take care when stopping recording 77
timeout events and timeout completion 22
using asynch signal mode 24
using VTS as input to ASR 124
pron
manual page 198

R

rear
manual page 200
reboot
see boot
reboot51
manual page 201
speech board types 39
recd41
tools summary 11
recdts
tools summary 11
record resource 74
allocation and configuration 74
commands 77
example 79
S51_REC_CFGVAL 76
S51_REC_CMDVAL 77
S51_REC_STAT 78
state 78
WAVE files 78
request
completion on speech channel 51
glossary 6
ResErrCode
speech channel request completion 52
resource
relation to channels and timeslots 20
resource cleanup
automatic 35
explicit 35
resource state
ASR 125
connect 99
CPT detector 96
DTMF detector 83
play 72
record 78
TDD detector 87
tone generator 91
transcoder 103
TTS 114
resources
allocation conflicts 34
ASR 117
connect 98
CPT detector 94
DTMF detector 80
glossary 6
play 69
record 74
speech channel 44
TDD detector 85
tone generator 89
transcoder 101
TTS 108
returned value
glossary 6
RTC 50
run time control
See RTC

S

S51_ABORT_CMDREQ
channel command description 49
S51_ASR_CFGVAL
description 119
S51_ASR_CMDVAL
description 124
S51_ASR_STAT
description 125
S51_BD_CFGVAL
BCC allocation and configuration 61
S51_BD_CMDVAL
BCC commands 63
S51_BD_STAT
BCC board state 64
S51_BOOT_CMDREQ
BCC commands 62
S51_CANCEL_CMDREQ
channel command description 48
S51_CfgChan
channel allocation/configuration description 45
manual page 161
speech API summary 43
speech channel resources 44
S51_CFGVAL
allocation/configuration requests 46
S51_CH
speech channel data description 55
S51_CloseChan
manual page 162
speech API summary 44
S51_CmdChan
channel command description 47
manual page 163
speech API summary 43
speech channel resources 44
S51_CMDVAL
channel command description 49
S51_CODER_CFGVAL
description 102
S51_CODER_CMDVAL
description 103
S51_CODER_STAT
description 103
S51_CON_CFGVAL
description 98
S51_CON_CMDVAL
description 99
S51_CON_STAT
description 99
S51_DETACH_CMDREQ
BCC commands 62
S51_DTMF_CFGVAL
description 82

S51_DTMF_CMDVAL
description 83
S51_DTMF_STAT
description 83
S51_EQP
speech API initialization 44
S51_EventChan
manual page 164
speech API summary 44
S51_EventChan()
summary 30
S51_EVT
speech channel event description 56
summary 30
S51_GetBd
manual page 166
S51_GETPERF_CMDREQ
BCC commands 63
S51_GetPort
manual page 166
S51_GetType
manual page 166
S51_Init
manual page 165
speech API summary 43
S51_INS_CMDREQ
BCC commands 62
S51_LED_CMDREQ
BCC commands 62
S51_MakeCid
channel allocation/configuration 45
channel command description 48
for controlling BCC 59
manual page 166
speech API summary 43
S51_OOS_CMDREQ
BCC commands 62
S51_PAUSE_CMDREQ
channel command description 49
S51_PING_CMDREQ
BCC commands 62
S51_PLAY_CFGVAL
description 69
S51_PLAY_CMDVAL
description 71
S51_PLAY_SET
description 70, 112
S51_PLAY_STAT
description 72
S51_READY_CMDREQ
channel command description 48
S51_REC_CFGVAL
description 76
S51_REC_CMDVAL
description 77
S51_REC_STAT
description 78
S51_REQ
allocation/configuration requests 45
channel command description 48
S51_RESUME_CMDREQ
channel command description 49
S51_SETVOLUME_CMDREQ
channel command description 49
S51_SKIPFWD_CMDREQ
channel command description 49
S51_START_CMDREQ
channel command description 48
S51_STOP_CMDREQ
channel command description 48
S51_TDD_CFGVAL
description 86
S51_TDD_CMDVAL
description 86
S51_TDD_STAT
description 87
S51_TDET_CFGVAL
description 95
S51_TDET_CMDVAL
description 96
S51_TDET_STAT
description 96
S51_Term
manual page 167
speech API summary 43
S51_TGEN_CFGVAL
description 90
S51_TGEN_CMDVAL
description 91
S51_TGEN_STAT
description 92
S51_TTS_CFGVAL
description 110
S51_TTS_CMDVAL
description 113
S51_TTS_STAT
description 114
s51boot
manual page 202
speech board types 39
SCbus timeslots
See HTS
signals - Unix 31
SIGPOLL 23
software packages - Unix 13
installation 15
installation order 16
lspssdk installation 15
speech API
channel data structure 54
command requests 47
configuration requests 45
description 39
event data structure 56
interface initialization 44
operation 53
request completion 51
summary 43

system architecture 3
timers 53
speech boards
booting 42
configuration and booting 39
type at system startup 42
types and configuration 39
status51
manual page 203
speech board types 39
syn_playd
examples summary 9
synchronous mode 24
events 30
signals 31
speech API operation 53

T

tag
unparsed 127
tags
parsing metatags 128
TDD detector resource 85
allocation and configuration 85
commands 86
example 88
S51_TDD_CFGVAL 86
S51_TDD_CMDVAL 86
S51_TDD_STAT 87
state 87
Term
function summary 19
TermReason
speech channel request completion 52
test programs and tools
summary 10
text to speech
See TTS
timers
AtComplEnableResTimers 51
AtStopEnableResTimers 51
EST (end silence timer) 54
InhTimers flag 54
IST (initial silence timer) 54
MLT (maximum listen timer) 54
speech resources 53
timeslot
relation to channels and resources 20
timeslots 25
common standard speech coding on all timeslots 26
ETS 25, 27
glossary 6
HTS 25, 26
TID (timeslot ID) 25
VTS 26, 27
tone generator resource 89
allocation and configuration 90
commands 91
example 93
S51_TGEN_CFGVAL 90
S51_TGEN_CMDVAL 91
S51_TGEN_STATL 92
state 91
tools
installation on SPARC 17
installation on WinNT 17
transcoder resource 101
allocation and configuration 101
commands 103
example 103
S51_CODER_CFGVAL 102
S51_CODER_CMDVAL 103
S51_CODER_STAT 103
state 103
the only way to use it 26
TTS
software packages - Unix 13
tools summary 11
TTS resource 108
allocation and configuration 109
commands 112
example 116
S51_PLAY_SET 112
S51_TTS_CFGVAL 110
S51_TTS_CMDVAL 113
S51_TTS_STAT 114
state 114
ttsenna
software packages - Unix 13
speech board types 40

U

unsolicited event
summary 30
USR_PrintTrace
manual page 168
USR_Trace 37
manual page 168
USR_TraceSw
manual page 168
UsrData
speech channel data structure 47
UW2compat 17

V

V5_AbortVts
description 133
manual page 169
V5_CloseVts
description 133
manual page 170
V5_COMPL_EVT 135
V5_CreateVts
description 133
manual page 171

V5_DataEndVts
description 133
manual page 172
V5_DESTROY_EVT 135
V5_DestroyVts
description 133
manual page 173
V5_EQP data structure 133
V5_EventChan()
summary 30
V5_EventVts
description 133
manual page 174
V5_EVT
summary 30
V5_EVT data structure 135
V5_Init
description 133
V5_InitVts
manual page 175
V5_OpenVts
description 133
manual page 176
V5_READMORE_EVT 135
V5_ReadVts
description 133
manual page 177
V5_Term
description 133
V5_TermVts
manual page 178
V5_VTS data structure 134
V5_WRITEMORE_EVT 135
V5_WriteVts
description 133
manual page 179
virtual timeslot
see VTS
VTS 131
API interface summary 132
description 27, 131
differences from HTS 131
drain and discard speech 132
example 136
summary 26
V5_COMPL_EVT event 135
V5_DESTROY_EVT event 135
V5_EQP data structure 133
V5_EVT data structure 135
V5_READMORE_EVT event 135
V5_VTS data structure 134
V5_WRITEMORE_EVT event 135
vts_loop
tools summary 11
vts_test
tools summary 11
vts_tts
tools summary 11

W

WAVE files
playing 70
recording 78

X

xplot
manual page 204