Client Side Telephony (CST) Chip Software User's Guide

SPIRIT CORP
DSP Software Source
www.spiritDSP.com/CST

Literature Number: SPRU029A
March 2003

TEXAS INSTRUMENTS

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

This user's guide assists the user with programming the various components from SPIRIT™ Corp for the TMS320C54x platform. It provides instructions for integrating these software components and implementating various telephony devices based on TMS320C54x platform. All CST algorithims conform to the TMS320 DSP Algorithm Standard, also known as XDAIS.

How to Use This Manual

The contents of the Client Side Telephony (CST) Chip Software User's Guide are as follows:

❑ Chapter 1, *Introduction to Client Side Telephony (CST),* is a brief overview of the Client Side Telephony (CST) Chip Software User's Guide (CST), abbreviations and terms used throughout this document, and important copyright information.

❑ Chapter 2, *Getting Started,* provides quick steps to allow the user to immediately begin using a CST chip and its modes. Important notes concerning SDK installation procedures are also provided.

❑ Chapter 3, *Hardware Overview*, provides an overview of the CST chip and the C54CST EVM board, and its settings. A description of the UART interface with the C54CST is provided, as well as instructions for adapting a C54CST chip to user specific hardware.

❑ Chapter 4, *Software Overview,* is an overview of the Framework and Components of CST Software parts. This chapter also describes the benefits of using Flex mode to control CST chips.

❑ Chapter 5, *Flex Application Development Guidelines,* is a brief overview on how to develop user-specific applications. The benefits of Flex mode over Chipset is discussed.

❑ Chapter 6, *CST Framework and API Overview*, provides the user with overviews and descriptions of the different CST Layers, services, their API.

- Chapter 7, *CST Framework Components*, provides detailed descriptions of all CST framework components, their interface, and architecture.

- Chapter 8, *C54CST Resources: Registers Conventions, Memory, and MIPS*, is a summary of important information about C54CST chip resources and their use by CST framework and algorithms.

- Chapter 9, *AT Command Set Descriptions*, provides the user with description of AT commands, syntax, shielded codes, and result tokens.

- Chapter 10, *CST Host Utility,* provides the user with requirements and settings for running a CST host utility.

- Chapter 11, *Product Installation Procedure,* provides brief instructions on installation of the CST SDK, setup of the CST host to communicate with the C54CST EVM, and the setup of Windows™ to communicate with the C54CST as a generic modem.

- Chapter 12, *Chipset Mode Testing and Troubleshooting,* provides descriptions of several test procedures available for troubleshooting and testing functionality.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

```
0011  0005  0001         .field    1, 2
0012  0005  0003         .field    3, 4
0013  0005  0006         .field    6, 3
0014  0006               .even
```

  Here is an example of a system prompt and a command that you might enter:

```
C:  csr -a /user/ti/simuboard/utilities
```

Information About Cautions and Warnings

This book may contain cautions and warnings.

CAUTION

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

WARNING

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to <u>you</u>.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Related Documentation From Texas Instruments

Using the TMS320 DSP Algorithm Standard in a Static DSP System (SPRA577)

TMS320 DSP Algorithm Standard Rules and Guidelines (SPRU352)

TMS320 DSP Algorithm Standard API Reference (SPRU360)

Technical Overview of eXpressDSP-Compliant Algorithms for DSP Software Producers (SPRA579)

The TMS320 DSP Algorithm Standard (SPRA581)

Achieving Zero Overhead with the TMS320 DSP Algorithm Standard IALG Interface (SPRA716)

Reference Framework 3: A Flexible, Multi-Channel/Algorithm, Static System (SPRA793)

Reference Frameworks for eXpressDSP Software: API Reference (SPRA147)

TMS320 DSP/BIOS User's Guide, (SPRU423)

TMS320C5000 DSP/BIOS Application Programming Interface (API) Reference Guide (SPRU404)

Writing DSP/BIOS Device Drivers for Block I/O (SPRA802)

TMS320C54x Chip Support Library API Reference Guide, (SPRU420)

TMS320C54CST Client Side Telephony DSP (SPRS187)

TMS320VC5407 Bootloader Technical Reference (SPRA827)

Client Side Telephony (CST) Chipset Mode (SPRA859)

Client Side Telephony (CST) Chip Flex Mode Flex Examples Description (SPRA862)

Related Documentation

Si3044 User Guide. 3.3 V ENHANCED GLOBAL DIRECT ACCESS ARRANGEMENT. © Silicon Laboratories, 2000. http://www.silabs.com/products

ITU-T Recommendation V.250. Serial asynchronous automatic dialing and control, 07/97

ITU-T Recommendation V.253. Control of voice-related functions in a DCE by an asynchronous DTE, 02/98

TMS320C54CST Evaluation Module. Technical Reference. Spectrum Digital, Inc.

Using the Zero-Overhead model / Static memory example

Documentation for XDAIS Algorithms

Automatic Gain Control (AGC) Algorithm User's Guide (SPRU631)

Caller ID (CID) Algorithm User's Guide (SPRU632)

Comfort Noise Generator (CNG) Algorithm User's Guide (SPRU633)

Echo Canceller (EC) Algorithm User's Guide (SPRU634)

Voice Activity Detector (VAD) Algorithm User's Guide (SPRU635)

ModemIntegrator Algorithm User's Guide (SPRU636)

G726 Algorithm User's Guide (SPRU637)

Universal Multifrequency Tone Detector (UMTD) Algorithm User's Guide (SPRU638)

Universal Multifrequency Tone Generator (UMTG) Algorithm User's Guide (SPRU639)

Trademarks

Software Copyright

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/products/index.htm
DSP Solutions	http://www.ti.com/dsp
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm
Microcontroller Home Page	http://www.ti.com/sc/micro
Networking Home Page	http://www.ti.com/sc/docs/network/nbuhomex.htm
Military Memory Products Home Page	http://www.ti.com/sc/docs/military/product/memory/mem_1.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(972) 293-5050	Fax: (972) 293-5967	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
Microcontroller Hotline	(281) 274-2370	Fax: (281) 274-4203	Email: micro@ti.com
Microcontroller Modem BBS	(281) 274-3700 8-N-1		
DSP Hotline		Email: dsph@ti.com	
DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs			
Networking Hotline		Fax: (281) 274-4027 Email: TLANHOT@micro.ti.com	

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:		
Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32
Email: epic@ti.com		
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68	
English	+33 1 30 70 11 65	
Francais	+33 1 30 70 11 64	
Italiano	+33 1 30 70 11 67	
EPIC Modem BBS	+33 1 30 70 11 99	
European Factory Repair	+33 4 93 22 25 40	
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	
Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/		

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ **Documentation**

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail:	Texas Instruments Incorporated Technical Documentation Services, MS 702 P.O. Box 1443 Houston, Texas 77251-1443	Email: dsph@ti.com Email: micro@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

For product price & availability questions, please contact your local Product Information Center, or see www.ti.com/sc/support http://www.ti.com/sc/support for details.

For additional CST technical support, see the TI CST Home Page (www.ti.com/telephonyclientside) or the TI Semiconductor KnowledgeBase Home Page (www.ti.com/sc/knowledgebase).

If you have any problems with the Client Side Telephony software, please, read first the list of Frequently Asked Questions at http://www.spiritDSP.com/CST.

You can also visit this web site to obtain the latest updates of CST software & documentation.

Contents

Figures

Tables

Notes, Cautions, and Warnings

Chapter 1

Introduction to Client Side Telephony (CST)

This chapter provides a brief overview of the Client Side Telephony (CST) Chip Software User's Guide. It lists and explains abbreviations and terms used throughout this document, and contains copyright information.

Topic **Page**

1.1 CST Overview

CST Software consists of several eXpressDSP compliant telephony components and a special CST Framework, which ties them together and provides unified access to each of them. CST Software was ROM'ed into TMS320C54CST DSP chip from Texas Instruments.

There are two main modes of CST Chip operation – Chipset mode and Flex mode:

- ❑ In Chipset mode, only CST software is running inside CST Chip, controlled from outside via serial link by AT commands.
- ❑ In Flex mode, user code is running inside of CST Chip, controlling the CST Software in ROM using several different control layers of CST Framework.

The following components are included in CST Software (as standalone XDAIS algorithms):

- ❑ Data Modem (V.32bis/V.32, V.22bis/V.22, V.14, V.42, V.42bis)
- ❑ Voice processing (ADPCM G.726, G.711, G.168 Echo Canceller, VAD, CNG and AGC)
- ❑ Telephony Signals Processing (DTMF, CPTD, CID)

Besides, CST algorithms portofolio can be extented via a set of very memory-efficient CST Add-ons, supplied separately from CST chip:

- ❑ Fax G3 functionality (fax modem supporting V.17/V.29/V.27ter/V.21)
  V.29 Fast Connect (for POS terminals)
- ❑ Standard vocoders (G.729AB – 8 kbps, G.723.1 – 5.3 and 6.3 kbps)
- ❑ SPIRIT-proprietary 1200 bps vocoder

There is also an integration shell (CST Framework), which consists of several layers and forms very flexible and configurable framework. Each framework layer has its own intermediate interface, with its own level of abstraction. CST Framework consists of the following parts (supplied in open source code):

- ❑ AT Command Parser (Data and Voice commands, used mostly in Chipset mode)

- Several control layers (used in Flex mode only):
  - CST Action layer (to give the user control over CST Solution as whole through mapping all commands and messages to different CST sub-layers; eliminates the need to use AT Parser)
  - CST Commander layer (to give the user control over CST Solution through set of special command sequences)
  - CST Service layer (to provide data flow between different XDAIS components and device drivers, and to give the user unified access to CST XDAIS components through set of special messages)
- LIO- and CSL-compliant drivers and data flow controllers of UART and DAA codecs
- Memory management and other system services

The Framework was organized to give the user maximal flexibility. To achieve this, many Framework functions call one another via function pointers. This allows the user to override these functions as well as driver routines. It is also possible to create several instances of the framework.

The user can still directly use the standalone eXpressDSP compliant algorithms regardless of the CST Framework or use the Framework partially.

Besides the CST Software, there is also a start up Bootloader and a core code of DSP/BIOS ROM’ed into CST Chip.

1.2 Abbreviations and Acronyms

The following abbreviations are used in this document:

Table 1-1. Abbreviations and Acronyms

Name	Description
ADC	Analog-digital converter
ADPCM	Adaptive differential pulse code modulation. A type of waveform coding implemented in G.726 codec.
AFE	Analog front end. Hardware and/or software parts that convert signal waveform to a stream of samples. Delay introduces by buffering or hardware part in AFE may affect modem operation.
AGC	Automatic Gain Control
ALGRF	XDAIS algorithm creation/deletion functions (ALGorithm instantiation for Reference Frameworks). See also XDAIS, IALG and RF3.
BIOS	Used interchangeably with DSP/BIOS, see DSP/BIOS
CID	Caller ID
Chipset Mode	Mode of CST Chip operation when it is controlled only externally, via AT commands sent over serial link.
CNG	Comfort noise generator
CPTD	Call progress tone detector
CSL	Chip support library- TI’s standard library to support on-chip hardware
CST	Client side telephony, also means the CST Chip solution
CTS	Clear-to-send. UART interface signal that indicates readiness to receive data in one direction (see also RTS).
DAA	Data access arrangement, hardware interface with telephone line
DAC	Digital-analog converter
DARAM	Dual access RAM
DCE	Data communications equipment. Within the scope of this document it implies *EITHER* CST chip when used in chipset mode receiving commands from external host, *OR* CST software solution controlled by user-specific software inside CST chip.
DSP/BIOS	TI’s Real Time OS for DSPs

Table 1-1. Abbreviations and Acronyms (Continued)

Name	Description
DTE	Data terminal equipment. Within the scope of this document it implies *EITHER* a PC (or another host) sending commands to CST Chip via serial link, *OR* user-specific software inside CST chip (flex application), sending commands to AT parser via virtual UART, *OR* software unit providing DTE functionality.
DTMF	Dual-tone modulated frequency signal
EVM	TMS320C54CST evaluation module supplied by spectrum digital.
FCS	Frame check sequence
Flex Mode	Mode of CST chip operation when it is controlled internally by a user program loaded into internal or external memory of the CST Chip.
GSTN	General switched telephone network
IALG	Interface to define XDAIS algorithms' memory requirements (see also XDAIS and ALGRF)
ISP	Internet service provider
ISR	Interrupt service routine
LIO	"Low-level I/O" – TI's standard for drivers interface
MAU	Minimum addressable unit, whose size is usually equal to sizeof(char) in C.
MCU	Micro-controller unit
PCM	Pulse code modulation. This term means representation of a waveform by quantized digital signal using linear or logarithmic laws, rather than a modulation technique
PSTN	Public switched telephone network
RF3	Reference framework level 3. Includes into itself: XDAIS, ALGRF, DSP/BIOS
RTS	Request-to-send. UART interface signal that indicates readiness to receive data in another direction (see CTS).
SDK	Software development kit
SWI	DSP/BIOS software interrupt
TAM	Telephone answering machine
UART	Universal asynchronous receiver/transmitter, the chip which allows data exchange over serial link
UMTD	Universal multi-tone detector

Table 1-1. Abbreviations and Acronyms (Continued)

Name	**Description**
VAD	Voice activity detector
XDAIS	eXpressDSP^{TM} Algorithm standard (also known as TMS320 DSP Algorithm Standard).
NOTICE on DAA part number: Throughout the document, Silicon Lab's DAA is referred to as Si3016 or Si3021 chip. Here is the explanation of part names:	
Si3016	Line-side DAA, directly connected to telephone line. External chip.
Si3021	DSP-side DAA, connected to line-side only via capacitors. This part is on-chip in C54CST chip.
Si3044	Compound part name, denoting Si3016 and Si3021 together.
NOTICE on C54CST part number: TMS320C54CST chip is a current version of the chip, having CST bundle V2.0 in ROM, also referred as CST2. TMX320VC54CST chip is a previous version of the chip, having CST bundle V1.0 in ROM, also referred as CST1. Throughout the document, C54CST name refers to CST bundle V2.0, unless noted otherwise.	

1.3 Legal Disclaimer

Legal Disclaimer

The views, opinions and references expressed herein do not necessarily state or reflect those of the companies cited in section 1.3. The aforementioned companies in this User's Guide are in no way affiliated with SPIRIT CORP, or SPIRIT TECHNOLOGIES, INC.

Chapter 2

Getting Started

This chapter provides quick steps to allow the user to immediately begin using a CST chip and its modes. Important notes concerning SDK installation procedures are also provided.

Topic **Page**

2.1 Overview

There are two main modes of CST Chip operation:

- Flex mode
- Chipset mode

Please refer to section 3.2 for more information concerning these modes.

To learn how to control CST Chip in Flex mode and to write your own program for TMS320C54CST using CST solution, read the following chapters:

- Chapter 3: *Hardware Overview*
- Chapter 5: *Flex Application Development Guidelines*
- Chapter 8: *C54CST Resources: Register Conventions, Memory, and MIPS*
- CST Algorithm user guides found in *Related Documentation from Texas Instruments* section of the *Preface*
- *Client Side Telephony (CST) Chip Flex Mode Flex Examples Description* (SPRA862)

For instructions on learning to control CST Chip in Chipset mode, please consult the following chapters:

- Chapter 9: *AT Command Set Descriptions*
- Chapter 10: *CST Host Utility*
- Chapter 12: *Chipset Mode Testing & Troubleshooting*
- *Client Side Telephony (CST) Chipset Mode* (SPRA859)

To get yourself quickly acquainted with the CST Software, we strongly recommend trying out several examples described in the *Client Side Telephony (CST) Chip Flex Mode Flex Examples Description* (SPRA682). The fast way to run these applications using TMS320C54CST chip is to use TMS320C54CST EVM. You can also use the code of these examples, located in `Src\FlexExamples\`, to quickly create your own application for Flex mode. It’s also recommended that you look through the supplied CST Framework source code when learning the CST Framework components and other CST internals.

For installation procedure, please refer to chapter 11, *Product Installation Procedure*.

2.2 Running a CST Solution: Standalone Chipset Mode

To run CST solution in standalone Chipset mode, the following steps should be taken:

Step 1: Connect and set up TMS320C54CST EVM, connect its COM port to one of PC's COM ports (COM1 or COM2, for example), connect EVM to telephone line and also to power supply, but do not power it up yet.

Set jumpers on EVM as described in section 3.3.

Step 2: Start host terminal – specialized `CSTHost\CSTHost.exe`, or general-purpose HYPERTERMINAL or PROCOMM communication programs and open the COM port to which EVM is connected. Set this port for 115200 bps, 8 bits of data, 1 stop bit, no parity, Hardware flow-control.

If running `CSTHost`, open terminal window at `File->CST Terminal`, and press `Settings` button. Choose COM port to which EVM is connected, and press `Configure Port` button. Set the port according to the settings mentioned above, and press `OK`.

Step 3: Turn on power for EVM.

Step 4: Type “AT” in the terminal window, to tell CST bootloader to switch into Chipset mode, OR load a patch (if necessary) via `Settings->Load Patch` in CSTHost.

Step 5: The LEDs should blink several times (which indicates that the CST has successfully started in Chipset Mode) and the following greeting should be output to the terminal window (if COM port was configured correctly):

```
CST Bundle Rel 2.0
(c) SPIRIT CORP
Type AT$, AT$H or AT&V for help
```

Step 6: Type `AT$<Enter>` on the terminal – you should see help on the available S registers and their settings.

Step 7: Type `AT&V<Enter>` on the terminal – you should see current settings of CST.

Step 8: To test modem, type `ATDTxxxx`, where `xxxx` is the number you wish to dial and connect. Once modem connects, it reports the rate at which it connected.

Step 9: To test voice capabilities, press `Play Greeting and Record` button on CST Host, and call the number to which EVM is connected from another phone.

Step 10: If something does not work properly, you may need to tune DAA properly for the standards of your country by pressing `Settings`, `DAA International settings` (read more about this in chapter 11), and also read *Client Side Telephony (CST) Chipset Mode* (SPRA859).

2.3 Running a CST Solution: Flex Mode

Note: Compiling Flex examples and Code Composer Studio

Before taking these steps, please, read carefully installation procedure in chapter 11. To correctly compile a Flex example, the include path in project file may require tuning, and your Code Composer Studio may require a small update.

To run CST solution in user programmable Flex mode, the following steps should be taken:

Step 1: Connect and set up TMS320C54CST EVM, connect its COM port to one of PC's COM ports (if COM port is needed in your Flex mode application), connect EVM to telephone line and also to power supply, but do not power it up yet. Set jumpers on EVM described in section 3.3.

Step 2: Turn on power for EVM.

Step 3: Start Code Composer Studio.

Step 4: Load GEL file `EVM54CST.gel` from `Src\GEL`

Step 5: Open appropriate project file in one of the folders:

❑ `Src\FlexApp` (for non-DSP/BIOS based applications)

or

❑ `Src\FlexAppBIOS` (for DSP/BIOS based apps).

The project files are:

❑ `CSTFlexApp.pjt`

`or`

❑ `CSTFlexAppBIOS.pjt`, respectively.

Copy one of the Flex examples from folder `Src\FlexExamples` into either:

❑ `Src\FlexApp`

`or`

❑ `Src\FlexAppBIOS`

renaming it to `main.c`, in order to substitute the existing `main.c` file.

Step 6: Compile the application and load it to EVM.
If compilation fails, please, refer to chapter 11 on how to update CSL files in Code Composer, and on how to tune project file include path.

Step 7: Run the Program – EVM LEDs should blink several times, which indicates that the program loaded and initialized CST in Flex mode correctly.

Step 8: Call the number to which EVM is connected from another phone, and test the application running in CST Chip (read CST Flex Mode Application Note to learn more about Flex mode examples).

Step 9: If something does not work properly, you may need to tune DAA properly for the standards of your country by pressing `Settings`, `DAA International settings` (read more about this in chapter), and also read *Client Side Telephony (CST) Chip Flex Mode Flex Examples Description* (SPRA682).

Chapter 3

Hardware Overview

This chapter gives hardware overview of CST chip and C54CST EVM board and its settings. It also describes the UART interface of the C54CST.

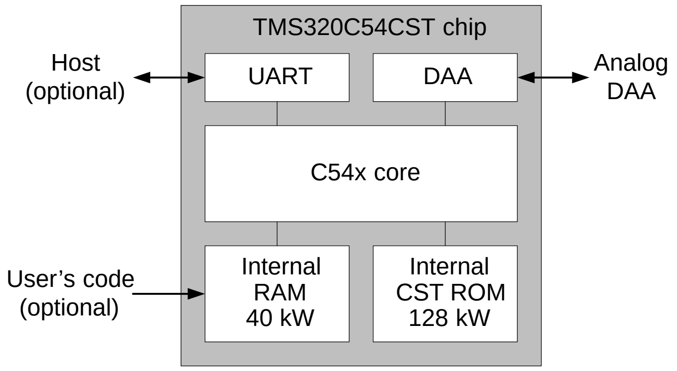
Section 3.6 provides an overview and instructions for adapting a C54CST Chip to User-Specific Hardware.

Topic **Page**

3.1 Introduction to the CST Chip

Texas Instrument’s TMS320C54CST is a generic C54x DSP with UART and Digital DAA integrated into it. It has 40 kW of RAM and 128 kW of ROM. Client Side Telephony software was developed by SPIRIT and ROM’ed into this DSP by TI, thus making it a TMS320C54CST device (see Figure 3-1).

Figure 3-1. CST Chip Overview



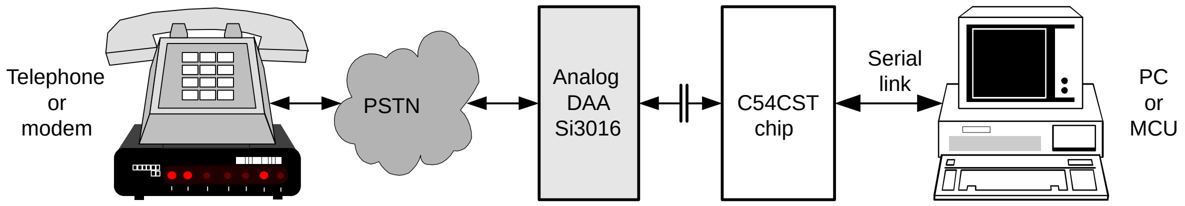
The most generic hardware setup for CST Chip is shown in Figure 3-2.

On the telephone line side, CST Chip can be connected to the Analog DAA chip from Silicon Laboratories, Si3016 (NOTE: CST chip can use any other DAA or Codec; DAA driver in CST can be reloaded easily, see section 7.7.7.2). This provides galvanic de-coupling with the telephone line, and DSP is connected to Analog DAA chip via two capacitors only.

On the host interface side, CST chip is connected to PC or any MCU controller via serial asynchronous port (RS232C). Host controls CST chip via AT command set both in data modem mode and in voice mode (most of the functions of CST Chip are controllable via AT commands).

CST Chip does not require any external RAM or other hardware to run CST tasks. At the same time, it is possible to load additional code to control CST chip into the internal RAM of TMS320C54CST, and use CST software in ROM as a library of XDAIS objects, thus eliminating the need for the host controller.

Figure 3-2. General Hardware Setup of CST Chip



This document concentrates mostly on software aspects of 54CST chip. For more information on hardware aspects read TMS320C54CST Client Side Telephony DSP Data Manual, SPRS187.

3.2 Main Modes of CST Chip

There are two main modes of CST Chip operation – Chipset mode and Flex mode.

In Chipset mode, only CST software is running inside CST Chip, controlled from outside via serial link by AT commands. In this mode, the CST chip can be used as standard data modem with voice features, including duplex voice transfer (all standard functionality of CST Software is accessible via AT commands).

In Flex mode, user code is loaded into and running inside of CST chip, using the CST Software in ROM as a library. This mode gives the user more flexible access to different levels of CST software, and allows the user to build applications using only CST Chip, without need for any host controller.

There are several ways to switch into Chipset mode from CST Bootloader:

1) High to low transition on the INT1 pin within 30 CPU cycles after reset;

2) Sending two symbols (“AT”) via UART, at 115200 bps, shortly after reset.

3) Writing a “magic” number `0x45` to memory location `0x7E` via HPI interface.

Even while in Chipset mode, the User still has possibility to load Flex application (user code) into RAM using special CST AT command (`AT#DATA`, see section 9.4.1.29).

If DSP starts in Chipset mode, it immediately runs CST solution from internal ROM. Otherwise, it starts in Flex mode, and tries to load user’s program through one of the external interfaces.

More information on CST Bootloader is given in section 5.4.2 of this document and the *TMS320C54CST Bootloader Technical Reference Guide* (SPRA827).

3.3 TMS320C54CST EVM Configuration

To run Spectrum Digital's EVM with TMS320C54CST processor, jumpers on EVM should be set the following way (`ON` – pins 1 and 2 connected, `OFF` – pins 2 and 3 connected):

```
JP1 – OFF    JP2 – OFF
JP3 – OFF    JP4 – ON
JP5 – OFF    JP6 – OFF
```

Initially EVM will start TMS320C54CST in Flex mode. Bootloader will run first, and will be waiting for Flex application from several of external sources (see section 5.4.2 for details).

To start TMS320C54CST in Chipset mode, just type “AT” symbols via terminal, connected to EVM's UART.

3.4 UART Hardware Flow Control

TMS320C54CST chip has only 2 dedicated pins for UART – RX and TX, the rest of UART lines have to use general purpose I/O lines of the chip, which are combined with HPI pins.

The CST UART driver implies that these additional UART lines are connected in the same way they are connected on EVM board (see the *TMS320C54CST Evaluation Module Technical Reference*, Spectrum Digital, Inc.).

Table 3-1. UART Lines

UART Line	External DSP Pin	Direction (for DSP)	Comments
DTR	HD0	Input	Controls CST behavior
RTS	HD1	Input	Used for HW flow control, tells CST that host is ready to receive data
CTS	HD2	Output	Used for HW flow control, tells host that CST is ready to receive data
DSR	HD3	Output	Not used by CST
DCD	HD4	Output	Reports modem online status
RI	HD5	Output	Reports RING event

If the user wants to connect UART lines to other pins or not to use these pins at all, it is necessary to modify CST UART driver by reloading some or all of its virtual methods in Flex mode (see section 7.7.7.2).

3.5 LED Indication

C54CST chip uses its I/O Port #0 to output indication information about some of the internal events. On Spectrum Digital's EVM this port is connected to 4 LEDs, DS3 through DS6.

The meaning of this indication is described in Table 3-2:

Table 3-2. Indication LEDs Meaning

Data Bit # in Port 0	EVM LED #	CST's LED #	Meaning
0	DS3	LED0	Not enough MIPS for real-time operation This LED is toggled every time a buffer in DAA driver or UART driver overflows. Buffer overflow usually happens when some parts of the code consume so many MIPS, that CST Framework consumes less data from these buffers than it is supposed to, according to real-time requirements (for example, 8000 samples per second from DAA).
1	DS4	LED1	Voice buffer underrun Voice controller has a buffer, storing bitstream to be decoded and played out in voice mode. This LED is toggled every time this buffer underruns. This happens when Host does not send bitstream to be played out fast enough, and this leads to interruptions in output voice signal and sometimes even to incorrect decoding of further bitstream.
2	DS5	LED2	CTS (clear-to-send) circuit state When CST's UART driver receive buffer gets filled to ¾ of its size (capacity), the driver turns OFF CTS circuit telling the Host to wait and not send data. When the buffer frees up to ½ of its size, the driver turns CTS circuit back ON.
3	DS6	LED3	DSP in IDLE mode (power saving) When Power saving mode is enabled (via ATP command), this LED is turned on when DSP enters IDLE mode, and turned off when DSP leaves IDLE mode. This LED allows the user to estimate roughly how loaded the DSP is MIPS-wise: the darker this LED is, the more time DSP spends processing CST's routines and less time it spends in IDLE mode. When Power saving mode is disabled, this LED should be off.

If the user needs to use I/O Port 0 for some other purposes, it is possible to reload CST peripheral driver in Flex mode (see section 7.7.7.4), and remove any indication code, which writes to Port #0.

3.6 Adapting the C54CST Chip for User-Specific Hardware

When going to User-specific hardware, CST software needs to be reconfigured in order to fit the new hardware environment, in areas where it is different from C54CST EVM.

This chapter gives only a brief overview of this topic, more information can be found in other chapters of this document, and in appropriate application notes.

The following areas need to be taken into consideration:

1) DAA connection

   a) Using external SiLab’s Si3021+Si3016 DAA
      CST DAA driver already includes support for an external Si3021 DAA, and also for multiple DAAs of this type. So the User can connect multiple DAAs to CST (either each DAA to each McBSP (which is what the CST DAA driver supports), or several DAAs to one McBSP – connected in daisy chain (for what a new DAA driver required)), and use the same driver to control them all. Read sections 7.7.5, 7.7.7.3 , and 7.7.7.5 for details.

   b) Using another external DAA
      CST DAA driver can be reloaded and reconfigured to support any other external DAA. Again, multiple DAAs can be connected to CST. Read sections 7.7.7.3 and 7.7.7.5 for details.

2) UART connection

   a) Reconfiguring UART control lines
      User may want to redefine the pins, which are used as UART control lines (CTS/RTS, DSR/DTR, DCD, RI). By default, GPIO pins HD0-HD5 are used for this purpose. To do this, some of the UART driver virtual functions need to be reloaded. Read section 7.7.7.2 for details.

   b) Using GPIO pins HD0-HD5 for other purposes
      To configure CST UART driver not to use GPIO pins, see section 7.7.7.2.
      However, CST framework writes couple of more times into GPIO register during initialization.
      This happens in functions `CST_DSPInit()` and `TargetBoardInit()` (they both set `GPIOCR` to `0`).
      So, if you want CST not to touch GPIO registers even during initialization, you have to reload functions `CST_DSPInit()` and `TargetBoardInit()`.

c) Connecting with Host via UART
To connect host computer or Host CPU to C54CST via UART, it is not required, although recommended, to use UART hardware flow control lines. If flow control lines are not connected, the User needs to make sure that 1) RTS line is tied to 1 (always high), so that CST would be allowed to send data to Host; 2) Host never sends too much data to CST chip, to prevent overflows of the internal UART buffer.
Also, C54CST's UART is capable of operating at higher rates than 115200 bps (at least 16 times faster). This may be useful in some applications.

3) HPI connection

a) Connecting with Host CPU via HPI
Host CPU can control CST via HPI port only. CST Bootloader supports booting from HPI. UART traffic (AT commands and data) can be redirected from UART to HPI. To make this redirection, the UART driver needs to be reloaded (see 7.7.7.1).
SPIRIT Corp. also is planning to provide a flex example on how to switch CST into "HPI-controlled Chipset Mode". Please, refer to the CST support web site.

4) McBSP connection

a) Connecting TDM channels (T1/E1)
C54CST can process data coming from TDM channel just as it processes data coming from DAA. In order to use CST framework in this case, DAA driver needs to be reloaded, as described in section 7.7.7.3. To process several slots (PCM channels), multiple instances of DAA driver and CST Framework should be created.

b) Connecting with Host CPU via McBSP
Host CPU can control CST via McBSP port only. CST Bootloader supports booting from McBSP. UART traffic (AT commands and data) can be redirected from UART to McBSP. To make this redirection, the UART driver needs to be reloaded (see 7.7.7.1).

5) LEDs control

a) CST Peripheral driver uses I/O Port #0 to output indication information about some of the internal events using C54CST EVM LEDs. To configure this indication differently, or to disable it, peripheral driver needs to be reloaded, as described in section 7.7.7.4.

6) Adding driver for a new device

   a) When adding a driver for a new device, the User can still benefit from the reach functionality of CST framework, if that driver is added inline with CST framework and driver concept.

      Read sections 7.3.5.1 and 7.3.5.2 for details.

7) Using another clock for DSP

   a) Clock Frequency
      If on-chip DAA is not used, C54CST can run at any frequency, up to 120 MHz.

      Internal DSP clock multiplier can be set by calling function `TargetBoardInit(…,int  Multiplier,…)` with appropriate parameter.

      By default, on C54CST EVM, this parameter is equal to 8, which sets DSP clock to 118 MHz with 14.7456 MHz input clock.

      To run C54CST EVM at 59 MHz, change this parameter to 4.

   b) Clock jitter
      If DSP clock is used to clock DAA (for on-chip DAA, this is the case), clock jitter needs to be very small, in order to enable robust modem operation. For this reason, it is recommended to use crystals without internal PLL, because otherwise crystal’s internal PLL in combination with DSP’s PLL leads to high jitter.

   c) Memory wait states considerations
      When DSP clock is higher than access time to external RAM/ROM, accesses to external memory are done with one or several wait states. This is done by programming a wait state register, `SWWSR`. CST peripheral driver does this in initialization function, that contains parameter which specifies the amount of wait states for external memory - `TargetBoardInit(…,…,int ExtWaitStates).`

      By default, this parameter is equal to 2 wait states, which is applicable for C54CST EVM with its 12 ns SRAM and when DSP is running at 118 MHz. If DSP is running at 59 MHz, this parameter should be equal to 1. For all port accesses (I/O), 7 wait states are set by `TargetBoardInit()`, and if this needs to be changed, this function has to be reloaded/overridden (see section 7.7.7.4).

   d) UART divisor considerations
      On-chip UART is clocked from DSP clock. So, if DSP clock changes, UART divisor needs to be changed to, to enable operation at standard baud rate. For this purpose, UART driver has global parameters, `UartParams.baud` and `UartParams.clkInput`. UART divisor is set as multiple of these two values, `DLAB=baud*clkInput`. For example, to set 115200 baud rate at 118 MHz DSP clock, these parameters are `baud=32` and `clkInput=2`.

8) Connecting to external SRAM

   a) Since C54CST device has a lot of internal RAM and ROM, and both RAM and ROM need to be visible in program and data space for CST software to function normally, external memory is visible/accessible only via certain address areas, as described in section 8.3. In order to map external SRAM to these areas of visibility, some external address decoding & page access logic is required. Look for a hardware application note on this on CST support web site.

9) Connecting to external ROM/Flash and booting from it

   a) Connection of external parallel Flash can be done in the similar way as it is done on Spectrum Digital's C54CST EVM (see schematics in *TMS320C54CST Evaluation Module Technical Reference*, Spectrum Digital, Inc.). Programming utility for C54CST EVM's Flash is provided as an example in CST SDK, at `Utilities\Flex2Flash`. Refer to its readme file on how to use it.

   b) Connection of other types of Flash and ROM chips is also possible. Look for a hardware application note on this on CST support web site.

10) Adding new algorithm

   a) Read section 7.3.5.2 for details.

   b) Remember, that CST algorithms portfolio can be extended via a set of very memory-efficient CST Add-ons, supplied separately from CST chip: Fax G3, V.29FC, G.729AB, G.723.1, 1200 bps vocoder.

Some of these topics will also be covered in upcoming application notes.

Chapter 4

Software Overview

This chapter explains the benefits of using the Flex mode to control a CST chip, and gives an overview of two main CST software parts – CST Framework and CST Telephony Components.

Topic **Page**

4.1 Flex Mode Applications

There are two main modes of CST chip operation:

- Chipset mode
- Flex mode

In Flex mode, user code is loaded into and running inside of CST Chip, using the CST Software in ROM as a library.

As an alternative for AT commands, CST offers a unified top-level software interface – the main interface for flex mode applications. The unified high-level interface - also called CST Action interface – fully covers and extends functionality of AT command approach used during serial connection of DCE and DTE.

CST Action interface can only be used in flex mode. The easy-to-use API offers a number of additional features and dramatically reduces program size and development time.

To design most of Flex applications, only the basic knowledge of CST architecture is required. This is why the document mostly attends to CST Action interface and main CST types.

When using the high-level interface, the whole development process can be considered as a couple of standard stages.

Typically, the first stage includes preliminary design of application logical structure.

The second stage is basically implementing the algorithm as a combination of main and callback functions.

A standard flex application roughly corresponds to a set of AT-command - based standard applications. Multichannel and multi-codec applications, as well as those with non-standard data flow, are considered non-standard flex applications.

Such non-standard applications can use auxiliary options of CST Framework, but in some cases it may be reasonable not to use CST Framework at all and get down to XDAIS libraries instead.

Note that CST SDK contains a broad range of examples on implementing different applications, which makes it even easier to create User-specific Flex applications.

Main guidelines on Flex application development are given in chapter 5, *Flex Application Development Guidelines*. Brief specifications of main CST types are given in section 6.3, *Framework API Overview*.

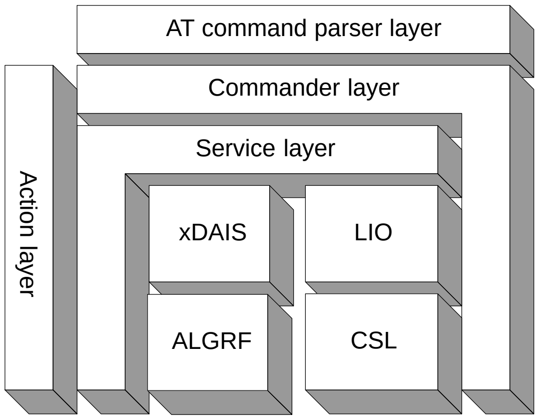
4.2 Framework Components

All components of CST solution are tied together by a CST Framework, which consists of several layers and forms very flexible and configurable framework. Each framework layer has its own intermediate interface which allows DSP developers to work with the solution as a whole or get down to any level of abstraction most suitable for their purposes.

CST Framework includes:

- AT command parser (data and voice commands)
- CST Action control layer (to give the user control over CST solution as a whole through mapping all commands and messages to different CST sublayers)
- CST Commander control layer (to give the user control over CST solution through a set of special command sequences)
- CST Service control layer (to provide data flow between different XDAIS components and device drivers, and to give the user unified access to CST XDAIS components through a set of special messages)
- LIO compliant drivers for UART and DAA codec.
- Memory management
- DSP/BIOS core

Figure 4-1. CST Framework Diagram



CST Framework has two alternative top layers and therefore, two alternative top-level interfaces.

In Chipset mode, CST Framework interacts with host via AT commands. This mode is used in Chipset mode and is configured as default.

However, the default status does not mean the preference of this interface for user integration in flex mode because it enforces the user to imitate host terminal, generate AT commands and decode responses, substitute a virtual UART driver instead of existing one and so on.

As a rule, standard CST Flex applications are not oriented on additional host terminals. If firmware under development does not require modem-oriented AT commands, it is not recommended to use AT parser for a top-level interface.

The alternative to AT command interface is action-based interface. The basic concept of this method is to eliminate AT parser and unify access to CST control layers.

CST Framework is a multichannel framework. All sub-service structures are grouped into a global structure. CST defines a global instance of this structure to be used for single channel applications.

CST Framework is OS-agnostic, in other words it can run under DSP/BIOS or other RTOS, or without it.

In non-RTOS environment, CST Framework is running as single thread application. However, CST fully supports multi-threaded mode, and if multi-threading capability is provided (if the user is using some kind of RTOS), CST Framework can benefit from it, by calling most of CST algorithms in a high-priority periodic process and posting low priority standard threads to run background tasks (such as V.42bis compression). If there is no multithreading in User’s environment, CST Framework will still operate OK, but the User will have to be more careful about evenness of MIPS load distribution.

For more flexibility, some important CST functions are called via pointers.

To learn more about CST Framework, please refer to chapter 5, *Flex Application Development Guidelines*.

4.3 Telephony Components

The CST solution integrates libraries of data communication, telephony signaling, and voice processing algorithms. All components of the solution are eXpressDSP compliant and share a standardized interface. All of these objects and device drivers are linked together by the CTS framework that provides unified access to the algorithms and eliminates compatibility and access issues.

The following algorithms are included in the CST solution:

- Modem algorithms
  - V.32bis/V.22bis(up to 14.4 kbps)
  - V.42 error correction
    - Embedded V.42bis compression
  - Modem integrator
    - Embedded V.14 async-to-sync conversion
  - Fax G3, supporting V.17/V.29/V.27ter/V.21 (up to 14.4 kbps) – as add-on only[1]
  - V.29 fast connect (for POS terminals) – as add-on only[1]
- Telephony algorithms
  - UMTG/UMTD (Universal multifrequency tone generator/detector)
    - DTMF generation/detection
    - Call progress tone generation/detection
  - Caller ID types 1 and 2
- Voice algorithms
  - G.168 line echo cancellation
  - G.726 ADPCM compression (16-40 kbps)
    - Embedded G.711 PCM
  - G.729AB vocoder (8 kbps) – as add-on only[1]
  - G.723.1 vocoder (5.3 and 6.3 kbps) – as add-on only[1]
  - SPIRIT-proprietary 1200 bps vocoder – as add-on only[1]
  - Automatic Gain Control (AGC)
  - Voice Activity Detection (VAD)
  - Comfort Noise Generator (CNG)

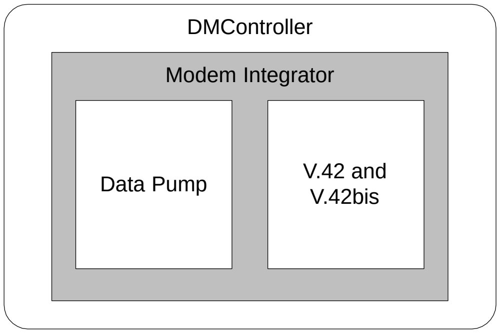
[1] This functionality can be added via a very memory-efficient CST Add-on, supplied separately from CST chip

4.3.1 Data Modem

Data modem consists of several components, each implemented as a separate XDAIS object. These objects are modem data pump (unifies ITU-T V.22/V.22bis/V.32/V.32bis and automode modem procedure), V.42 error correction protocol with embedded V.42bis data compression protocol, and a Modem Integrator object, which unifies access to all other modem algorithms (unified parameters, sample and data flows, extended status, etc), and interconnects them inside of itself.

Data Modem Controller (hereafter referred to DMController) is an upper layer that integrates Modem Integrator object into CST Framework.

Figure 4-2. Data Modem Objects



4.3.2 Voice Processing

Voice processing includes several components – waveform codec (PCM and ADPCM), line echo canceller, Automatic Gain Control (AGC) controlled by Voice Activity Detector (VAD), Comfort Noise Generator (CNG). All components have simple interface and operate with 14-bit 8 kHz samples.

CST’s G726G711 component implements ITU-T G.726 adaptive differential pulse code modulation (ADPCM) encoder and decoder of voice frequencies, as well as G.711 logarithmic conversion. It supports A-law or µ-law conversion to/from uniform (linear) PCM according to G.711; and compresses/decompresses linear samples to/from bitstream, based on selected compression rate – 16, 24, 32 or 40 kbps, according to G.726

CST’s Line Echo Canceller (LEC) is used for cancellation of electric echo introduced by telephone hybrid, and conforms to G.165 and G.168 ITU recommendations. It includes double talk detector and nonlinear processor. User can set the value of maximum echo path equal to 16, 32 or 64 msec.

CST’s Voice Activity Detector (VAD) detects the presence of speech in the signal. It has special adaptive algorithm to automatically adjust to the level of the noise in the signal, in order to provide robust operation even in the noisy speech. It has many user configurable parameters, allowing the algorithm to optimally tune itself for a specific application. VAD also outputs several coefficients that characterize spectral envelope of the noise (when no speech is detected), so that the regenerated noise would be similar to the original noise.

CST's Comfort Noise Generator (CNG) generates noise, distributed either uniformly or shaped according to the spectral envelope coefficients, which can be passed to CNG as parameters.

CST's Automatic Gain Control (AGC) is designed specifically to amplify voice signal, which has very non-stationary amplitude envelope. It operates much better in conjunction with VAD, which can tell AGC when there is no speech in the signal, so that AGC would not adapt in these periods.

4.3.3 Telephony Signals Processing

Telephony signals processing includes several components – UMTD (detects DTMF and CPT signals), UMTG (generates DTMF and CPT signals) and Client side Caller ID. All components have simple interface and operate with 16-bit 8 kHz samples (they have wider input dynamic range than voice processing components, which operate with 14-bit samples only).

CST includes Universal Multifrequency Tone Detector (UMTD) for detecting DTMF, Call Progress Tones (CPT) and many other telephony signals.

In brief, UMTD detector filters input samples, estimates spectrum of input signal, checks the cadences and pauses and makes the decision about presence of signaling tone. UMTD can be easily configured to fit the specific standard of any country.

CST's DTMF Detector operates in compliance to ITU-T Q.24 Recommendation.

CST's CPT detector supports a wide range of call progress tones fitting the standards of most countries according to Q.35 recommendation and some signals that do not fit into Q.35 recommendation. It is also possible to describe custom call progress tones.

CST includes Universal Multifrequency Tone Generator (UMTG) for DTMF, CPT and many other telephony signals generation. It can be set to generate tones according to the standards of different countries (tones' frequencies and cadences are adjustable).

UMTG-based DTMF Generator operates in compliance to ITU-T Q.23 Recommendation.

UMTG-based CPT generator produces output signals with cadences and frequencies specified in UMTG settings.

CST’s Client Side Caller ID Includes Type I and Type II Caller ID signal detection, compliant with standards of several providers and countries: Bellcore GR-30-CORE, SR-TSV-002476; British Telecom SIN227 and SIN242; ETSI ETS 300 659, ETS 300 778; Mercury Communications MNR 19. Client side caller ID supports on call waiting operation and parsing/converting the message into presentable format.

For more information see section 7.6, *Telephony Components Brief Specification*.

Chapter 5

Flex Application Development Guidelines

This chapter is design specifically to give a quick overview or quick start on developing a user-specific flex application, without overloading the User with lots of specific information about CST framework and its components. We strongly recommend to read this whole chapter before plunging into reading CST framework API or into building your own flex application based on one of the flex examples.

Section 5.1 explains the benefits of the Flex mode over Chipset mode for controlling CST chip.

Section 5.2 elaborates a little bit more on this topic, explaining why AT commands is not a convenient way to control modem- and voice-based applications when user code is inside the CST chip.

Section 5.3 describes three steps to build a user-specific flex application.

And finally, section 5.4 gives some specific details on how to actually compile and load flex application.

Topic **Page**

5.1 Chipset vs. Flex Mode

CST is a multifunctional solution that can be used as an external modem chip, as an embedded modem chip, or as a library of standalone XDAIS algorithms. CST chip is a full-scale modem on a single crystal.

A CST chip can be connected to the serial port of a personal computer running any terminal program. Using the default AT commands the user can test any of the standard CST modes, including voice mode.

The user is not required to have any prior knowledge of ‘C54x architecture or development tools in order to start using CST as an external modem chip. However, this kind of firmware presumes a separate host processor connected to CST via a serial port.

Software of the host processor should incorporate DTE interface of serial asynchronous automatic dialing and control that includes data exchange via serial port and flow control, algorithm for synchronization of AT commands, data and shield codes (if required), AT commands coding and recognition.

Using CST as an embedded modem (i.e. flex mode) is somewhat similar to the previous technique. Nearly each AT command corresponds to a CST API command (CST Action).

In this case the host processor can be absent or can perform entirely different functions. Program interface is simpler and more reliable than data transfer via serial port using AT commands.

The user is not required to have any knowledge of CST architecture, XDAIS algorithms, etc. in order to start using CST API, but basic knowledge of ‘C54x architecture and related development tools is needed.

It is also necessary to note that CST framework is a real time system, and therefore, the user may need to obtain some knowledge on real-time OS, for example TI’s DSP/BIOS.

In Flex mode, the user is able to utilize auxiliary features of CST, to modify parameters/functionality of various services and create non-standard applications. CST SDK (Software Development Kit) contains a large number of examples of CST usage in Flex mode.

Flex mode is also used to create patch-applications and to reconfigure CST the way AT commands wouldn’t allow.

In the following chapters, we will review CST Flex mode exclusively.

5.2 AT Commands vs Alternative Interfaces

DTE-DCE interface is logically implemented as a set of AT commands sent by DTE to DCE (all commands and parameters are coded as lines of ASCII characters) and DCE responses (the responses may be either verbal or numeric).

Using the AT command interface (from hereafter, AT interface) allows to design equipment compatible with modems of different manufacturers, but this is only true for the basic AT command set.

One should keep in mind that CST supports incomplete set of AT commands and registers defined in V.250 standard, also providing additional features not defined in scope of that standard.

The main problem of a classic DTE-DCE interface implementation is using the same information channel (UART) for both data transfers and for control signals exchanging.

It is supposed that interpretation of data being exchanged is strictly related to DCE state (off-line/on-line). This reduces flexibility of the interface significantly. This disadvantage is due to the minimalist implementation of RS-232 interface that is accepted as a standard.

However, certain modems (Comsphere from AT&T, DataLink from Penril) use a separate physical port for control/diagnostic purposes. The problem becomes more aggravated when the amount of control data increases.

Sharing the UART channel results in a drastic complication of the exchange protocol between DTE and DCE.

In general, simultaneous transfer of data and AT commands is impossible. An attempt to do this can lead to a number of consequences, such as:

- ❑ Lack of flexibility, especially obvious in voice and Caller ID applications
- ❑ Inability to control things continuously, inability to control at arbitrary instants
- ❑ Inability to distinguish between data and commands in certain situations, for example, if connection with a remote modem is broken prematurely
- ❑ Necessity for parsing the incoming data stream to separate the additional status/control information (in voice applications)

Implementation of a protocol for intense data and command transmission is a complicated issue in itself.

As an alternative for AT commands, CST offers a unified top-level software interface, referred to as CST Action interface, which fully covers and even extends functionality of the AT command approach used for serial connection between DCE and DTE.

CST Action interface can only be used in flex mode. The easy-to-use API offers a number of additional features and dramatically reduces program size and development time.

Advantages of the CST Action interface arise from separation of data and control channels, easy implementation of complicated operations (such as non-standard connection establishment scripts), automatic control and synchronization of processes being executed, etc.

Flex mode allows to exploit all additional features of CST, including multi-channeling, and considerably simplifies debugging.

Applications oriented on AT interface are easily transferable to CST Action interface, as most of basic AT commands are reflected to standard CST Actions.

CST SDK contains a broad range of examples on implementing different algorithms, which makes it even easier to create Flex applications.

5.3 Designing and Implementing Standard CST Applications

With CST it is easy to create applications for a broad range of tasks, such as:

- ❑ Embedded data transfer modems. Can be used in various remote sensors and security systems that require transfer of compressed digital data over phone lines
- ❑ Voice menu systems with remote control option
- ❑ Call centers and answering machines
- ❑ Internet appliances

Designing of the vast majority of Flex applications requires only the basic knowledge of the CST architecture. The whole development process can be divided into two important distinct stages.

The first stage includes preliminary design of application logical structure.

The second stage is basically implementing the application's algorithm as a combination of main and callback functions.

This chapter gives some recommendations on what we consider to be the optimal techniques for designing **standard** Flex applications.

A standard flex application roughly corresponds to a set of AT-command -based standard applications.

Multichannel and multi-codec applications, as well as those with non-standard data flow, are considered non-standard flex applications. Such non-standard applications can use auxiliary options of CST Framework, but in some cases it may be reasonable not to use CST framework at all and get down to XDAIS libraries instead.

Definition of a non-standard application is based on the fact that the non-standard application requires functionality beyond a unified high-level interface as described below.

5.3.1 Preliminary Application Design

Any Flex application algorithm should be represented as a set of elementary states, such as waiting for a ring, initializing the modem, waiting for a connection to be established, sending data, closing the connection, etc.

Each algorithmic state is usually associated with a single command (CST Action) sent to CST via the unified high-level interface. If a state of the User's algorithm requires performing multiple CST Actions, that state should be broken up into several elementary states, so that each state is only associated with a single CST Action.

Normally, each CST Action is associated with a single AT command. The most common CST Actions are listed below:

Table 5-1. CST Action Associations

CST Action	**Similar AT-command**	**Note**
Go off hook	ATH1	Go off hook, run CPTD and DTMF detectors.
Run the Caller ID after a ring end	-	Usually runs automatically
Run the Caller ID after a line reversal	-	Usually runs automatically
Call a remote modem	ATD	Go off hook, wait for a dial tone, dial a number and run the modem in the originate (calling) mode *(unless the ";" dial modifier is included in the number).*
Answer a call from a remote modem	ATA	Go off hook and run the modem in the answer (response) mode.
Dial and switch to the voice mode	ATD *in voice mode*	Go off hook, wait for a dial tone, dial a number, wait for a ring back signal appearance and then disappearance *(only if the "@" dial modifier is included in the number)* and enable voice mode
Answer and switch to the voice mode	ATA *in voice mode*	Go off hook and enable the voice mode (run Caller ID Type 2)
While in the voice mode, enable voice receiving procedures	AT#VRX	Run the G.726/G.711 encoder and all signal detectors (CPTD, DTMF).

Table 5-1. CST Action Associations(Continued)

CST Action	Similar AT-command	Note
While in the voice mode, enable voice sending procedures	AT#VTX	Run the G.726/G.711 decoder and all signal detectors (CPTD, DTMF).
While in the voice mode, enable both the voice receiving and sending procedures	AT#VRXTX	Run the G.726/G.711 encoder, decoder and all signal detectors (CPTD, DTMF).
While in the voice TX or RXTX modes, disable the voice receiving and sending procedures	<DLE><CAN>	Turn off the G.726/G.711 encoder, decoder, keep the signal detectors on (CPTD, DTMF and Caller ID).
Just make a call	ATDxx;	Go off hook, wait for a dial tone (if not disabled), and dial a number
Shut down the current process correctly	ATH	Correctly stop the current task, then turn off all other algorithms and go on hook.
Shut down the current process immediately	ATH	Turn off all algorithms, go on hook. Also used to abort operation.
Send data	-	Feed data to one of the transmitting algorithms (DTMF generator, modem, G.726/G.711 decoder)

It is necessary to note that the developer is not limited by the set of standard commands or actions described above. It is possible to create new Actions for specific needs.

Any such Action implies performing a sequence of elementary operations that can be interrupted, by the user or automatically, at any given time, but cannot be modified once it is initiated.

Normally, it takes some time to perform a CST Action; therefore, CST will ignore a new Action command if the current one is still being performed. Thus, developers should always be prepared for the system to reject sent command or data. In this case, the rejected command/data should be re-sent after some pause[2].

It is necessary to emphasize that CST provides full control over execution of telephony tasks and it automatically synchronizes commands/data being sent from the user with the current states of the CST processes, allowing the user to concentrate on the higher-level design of the application.

[2] Modification of process settings occurs instantaneously. However, in most cases, process settings modification does not influence the process that is running already, and is not considered a standalone CST Action.

The currently executed atomic command can be canceled by a special user's command or automatically (when it cannot be executed successfully). In the latter case, all unprocessed data is lost.

5.3.2 Detailed Application Design

Most of the data, coming from inside CST to the application, is transmitted in special egress messages. The function that receives these messages is to be **provided by the developer**. CST calls a function from CST services whenever it is necessary. Therefore, in order to implement a proprietary application, the developer needs to create only **two** functions:

1) Main function. This function contains the main logic of the application. Normally, the main function is implemented as a finite state machine. In each of the states the function may possibly send data/commands **to CST** and transit to a new state. Besides the state machine, the main function should also call a standard CST process routine, which carries out the internal CST processes.

2) Callback function. This function receives and processes messages coming **from the CST process routine and addressed to the application**[3].

The callback function receives messages from various CST services. The list of standard messages follows:

Table 5-2. Standard Callback Function Messages

Message Type	Description
Event detected by a peripheral	Attached data contains a peripheral driver message, such as: ring message, end of ring message, line reversal message.
Caller ID	Attached data contains a Caller ID result code (success or error code, such as time-out, invalid state, wrong check sum, illegal length, unknown type)
Detected DTMF symbol	Attached data contains a DTMF symbol
Detected call progress tone	Attached data contains a CPTD tone, such as: dial tone, (fast) busy tone, ring back tone and end of detected tone
Informs that modem just connected	
Informs that modem just disconnected	
Informs that voice just disconnected	Voice data receiving and transmitting has stopped but the system is still off-hook

[3] Note that sending commands/data to CST Framework via unified high-level interface does not perform any immediate actions and never leads to immediate calling of the user's callback function.

Table 5-2. Standard Callback Function Messages(Continued)

Message Type	Description
Voice data	Attached data is an array of voice data bytes
Modem data	Attached data is an array of modem data bytes
Auto turnoff request	Attached data characterizes the reason for turnoff request (which may be one of the following: Call Progress Tone Detection time-out, busy detection, modem disconnection, failure to create an XDAIS algorithm)
Tick message	Attached data is the number of DAA codec samples elapsed since the last tick message

The callback function is a suitable place to make transitions between the states of the main program's state machine. This is due to some of the state transitions being caused by messages that CST sends to the application. It is recommended to place some of the state transition code to the callback function.

The developer should first outline the application's logical structure and find the elementary states and actions of the application. This includes finding the correct sequence of state-to-state transitions and transition conditions. This is the preliminary development stage.

Once the preliminary design stage has been completed, the developer may turn his or her attention to CST and start developing the actual application. The implementation of the application mainly consists of creating the application's main function and the callback function.

Both functions will work according to a previously designed application state machine. The main function will send commands/actions to CST in the appropriate states and transit between some of the application states. The callback function will receive and handle messages coming from CST and perform the rest of transitions between the application states.

Implementing a CST application, however, is not limited only to creation of the code that sends commands to CST, receives messages from CST and handles the application state machine. It may be necessary to add a custom signal-processing algorithm or use one of the CST algorithms directly, bypassing the CST framework. It may be necessary to modify the behavior of CST, work with additional and different hardware, etc.

This all is still about developing a CST application, although it may involve considerably more effort than just creating the two functions as described above and the application may no longer be considered as a standard CST Flex application.

What CST offers to the user is a convenient global state machine that can be used to unify a number of telephony algorithms and routines. Anything beyond the standard CST functionality will require the appropriate changes to be made in CST and the application.

5.3.3 Implementation

The second stage of flex application design process is implementation of the application’s algorithm as a combination of the main and callback functions. Most standard CST applications do not require re-configuring CST services.

In this case, it is feasible to implement the above functions by means of the unified top-level interface provided by the CST Action Layer. The unified top-level interface allows access to the lower layers and, if necessary, are controllable.

The unified top-level interface provided by CST Action layer is represented by three functions:

Table 5-3. Top-Level Interface Functions

Name	Functionality
CSTAction_Init	CST initialization (does not include hardware init)
CSTAction_Process	Function to be called periodically
CSTAction	CST action execution. Sends a command or portion of data, and reads/ writes to configuration registers (S-registers)

The unified action-based interface is both BIOS- and single-thread ready. It also supports multiple channels. However, to support multichannel I/O some virtual functions in CST Framework have to be reloaded.

A generic single-threaded flex application, as well as a BIOS application, will be similar to this code example:

```
#include ”AppropriatePath\Framework\CSTChannel.h”
#include ”AppropriatePath\Framework\CSTAction.h”
//Select single-thread or DSP/BIOS-based application
#define BIOS_APPLICATION 1
  typedef enum {
    as_STATE_1,
    as_STATE_2,
    ...
  } tApplicationState;
  tApplicationState ApplicationState;
  //Prevent improper compilation
#if !BIOS_APPLICATION
  asm(”__sys_memory .usect  \”.sysmem\”,0”);
#endif //!BIOS_APPLICATION
  asm(”__STACK_BEGINNING .usect  \”.stack\”,0”);
bool MyCallback (tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,int Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {
    ...
    ApplicationState=as_STATE_4;
  }
  return 1;
}
void MyPeriodicThread ()
{
  CSTAction_Process (&Ch0);
  //////////////////
  //USER’S CODE...//
  //////////////////
  switch (ApplicationState)
  {
    ...
  }
}
void MyInitialization ()
{
  //////////////////
  //USER’S CODE...//
  //////////////////
}
#define EVM54CST_118MHZ_MULT 8
void main ()
```

```
{
  ////////////////////////////////
  ////STANDARD INITIALIZATION:////
  ////////////////////////////////
#if !BIOS_APPLICATION
  //Processor boot init.
  CST_DSPInit ();
#else
  initBiosConst();
#endif //BIOS_APPLICATION
      //CST internal data sections init.
  CST_bssInit ();
     //Particular board init.
     //You may need to change this according to your board specification
     //if it is different from standard EVM C54CST board
  TargetBoardInit (
    BIOS_APPLICATION,      // BIOS flag
    EVM54CST_118MHZ_MULT,  // internal PLL multiplier for external clock
    2);                    // Wait states for external memory
  //CST Framework init.
  CSTAction_Init (&Ch0,BIOS_APPLICATION,MyCallback);
  //Initialize CSL
  CSL_init();
  //Particular peripheral init.
  //You may need to change this according to your UART and codecs
  //if they are different from ’C54CST on-chip UART and DAA
  TargetPeriphInit (BIOS_APPLICATION,1);
  #if !BIOS_APPLICATION
    //Now it is safe to enable interrupts
    asm (” rsbx INTM”);
  #endif //BIOS_APPLICATION
  //Perform user’s specific initialization
  MyInitialization ();
  //////////////////////////
  ////MAIN LOCAL LOOP...////
  //////////////////////////
  #if !BIOS_APPLICATION
    while ( 1 )
    {
      MyPeriodicThread ();
    }
  #endif //BIOS_APPLICATION
}
```

The user's code downloaded to the CST chip must contain a new function `main()`. This function should perform all required hardware and software initializations and periodically call the CST periodic function `CSTAction_Process()`.

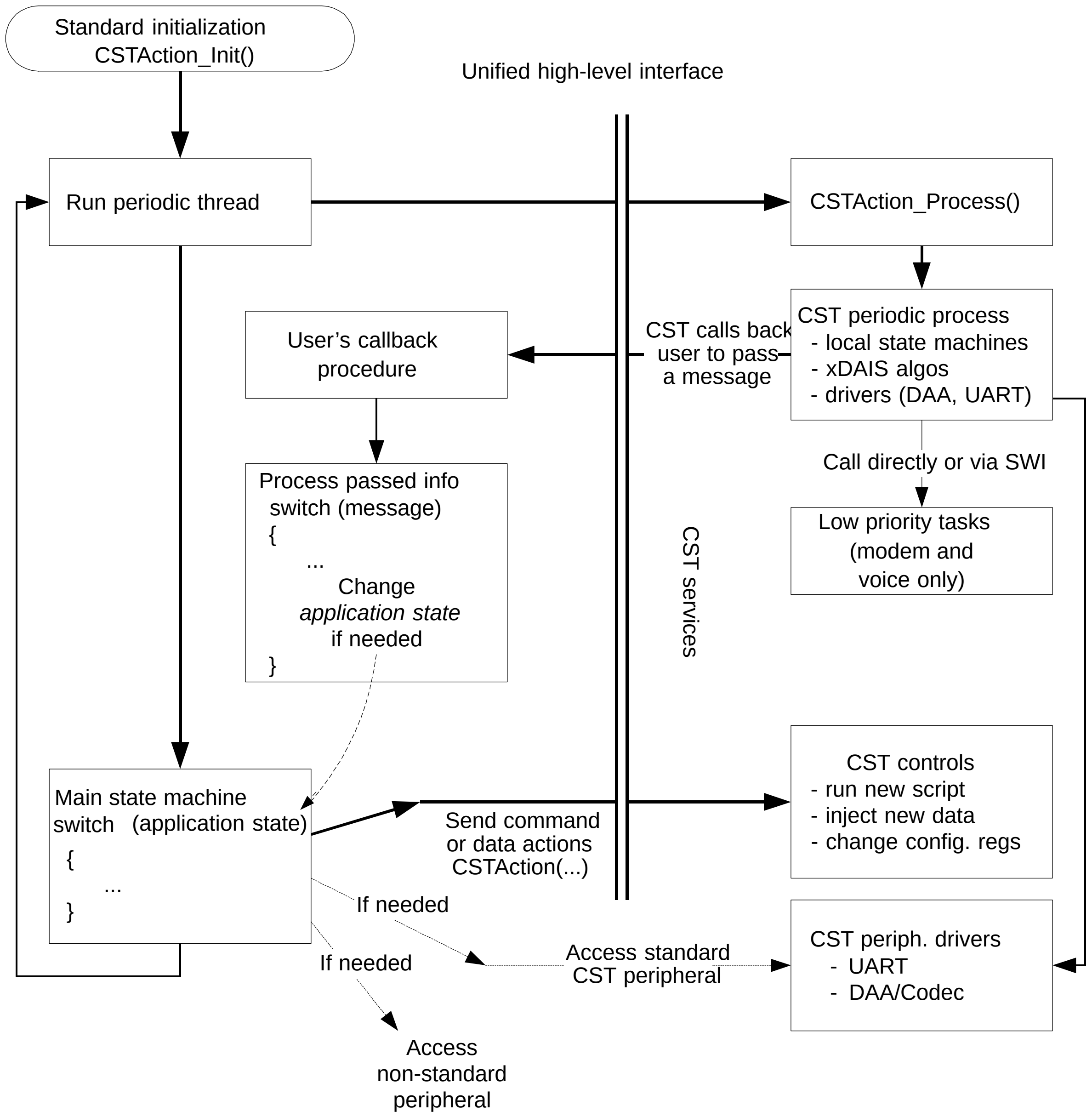
In chipset mode, the CST framework does not use DSP/BIOS functionality, however, the DSP/BIOS core is available in the ROM and it may be used in DSP/BIOS based applications.

In DSP/BIOS based applications, the user should periodically post an SWI that runs the `CSTAction_Process()` function. An example of doing this is contained in the file `BIOS\CSTBIOS.c` (this is a supplementary file for making single-channel applications for DSP/BIOS). The function `BIOSDAADataCallBack()` posts the SWI periodically (once per a certain amount of input DAA samples).

Thus, only `MyInitialization()`, `MyCallback()` and `MyPeriodicThread()` functions are to be extended by the user (only these three routines are overridden in all CST Flex application examples in the SDK package).

Figure 5-1 represents a generic CST flex application.

Figure 5-1. Generic CST Flex Application



Once the initialization is complete, interrupts should be enabled.

This moment marks the beginning of the main loop of the application. The main loop **must** periodically call the function `CSTAction_Process()`. It can be called as rarely as every 5 ms, but it is highly recommended to call it at least several times more frequently.

The function `CSTAction_Process()` runs all internal processes inside CST and calls back the user's message processing function. `CSTAction_Process()` is a high priority thread function. Usage of DSP/BIOS allows adding several auxiliary low-priority threads to utilize the processor resources more efficiently. It is also possible to allocate a separate auxiliary threads for low-priority modem and voice tasks (for example, for vocoders with large processing frame size, such as G.723.1, G.729, etc.), and user's control functions.

Three SWI's are defined in the file `FlexAppBIOS\CSTFlexAppBIOS.cdb`. The periodic thread corresponding to `MyPeriodicThread()` is to be invoked via the high priority SWI. The other two are modem and voice low priority SWI's.

Many of flex applications can be based on standard easy-to-read examples included in the SDK to demonstrate certain capabilities of the CST algorithms. Any CST flex application can run either under BIOS or in a single-threaded process.

5.3.4 Chapter Summary

- ❑ CST features a unified top-level interface that provides a unified bi-directional stream of commands and data.
- ❑ CST features a set of predefined top-level actions to solve standard telephony tasks.
- ❑ CST features a global state machine for solving the standard telephony tasks, and a number of independent system services.
- ❑ CST provides a reliable control over the process being executed. This eliminates the need for the user to code additional logic that synchronizes execution of standard telephony operations.
- ❑ Development of a standard CST application may be treated as development of the main and callback functions. The main function usually serves as the primary state machine for the application's logic and it transfers user's data to CST algorithms. The callback function serves to process messages sent by CST algorithms and contain data or control information on state of the process being executed.
- ❑ CST allows developing both single-threaded and multi-threaded applications using DSP/BIOS.
- ❑ Some multichannel, multicodec, and other non-standard applications require reconfiguration of some CST services. CST provides support for that, however in some cases it may be reasonable not to use CST Action Layer but get down to lower layers, even to XDAIS layer.

5.4 Building and Loading Flex Applications

This chapter gives some specific information on how to build a flex application, and load it to CST chip.

If compilation fails, please, refer to chapter 11, *Production Installation Procedure*, for instructions on updating CSL files in Code Composer, and on how to tune project file include path.

5.4.1 Projects for Building Flex Applications

All available flex examples are made in the same manner. As it mentioned in section 5.3, there is a suggested scheme for building a standard flex application, single- or multithreaded. There are a few functions prepared for the developer to be completed according to the task in hand.

There are two standard project files for building any of the offered flex application examples.

One is for single-threaded applications that will run without DSP/BIOS and the other one is for multi-threaded applications that will run under DSP/BIOS. Each of the two projects is already setup and tuned.

The developer only needs to create the main application C file (the project expects it to be `main.c`) or take the existing flex application example, put it into the directory with the project file and build the project. Of course, if there are extra files need to be included into the project, the project file may be altered.

The main files for the flex application examples are contained in the directory `FlexExamples`.

5.4.1.1 Project for Single-Threaded Applications

A standard project for single-threaded flex applications consists of five files:

`CSTFlexApp.pjt`	The project file for Code Composer Studio version 2.0 or higher
`CSTFlexApp.cmd`	The linker command file
`Main.c`	The main source code file of the flex application
`ROM\CSTRom.s54`	The CST ROM reference file
`ROM\rts_ext.lib`	TI’s Runtime Library (the version which was used to build CST)

The linker command file (`CSTFlexApp.cmd`) specifies locations and sizes of the application’s code and data sections. This file also contains sizes of the heap and stack.

To accommodate the developer needs, the linker command file may be modified.

After the project has been built, the produced out-file can be converted to a binary image to be loaded into the CST chip via the serial port (if a JTAG is not available).

To do this, run the file `hexCST.bat`, which will first obtain a hex file by running the utility `hex500.exe`, and then a binary image by running the utility `hextobin.exe`.

`hexCST.bat`	The batch file, which runs `hex500.exe`, and then `hextobin.exe`
`hex.cmd`	The command file for `hex500.exe`
`hextobin.exe`	ASCII to BINARY file conversion utility

The project file, linker command file, batch file and conversion utility reside in the directory `FlexApp`.

5.4.1.2 Project for Multi-Threaded Applications

A standard project for multi-threaded flex applications consists of the following files:

`CSTFlexAppBIOS.pjt`	The project file for Code Composer Studio version 2.0 or higher
`CSTFlexAppBIOS.cmd`	The linker command file
`CSTFlexAppBIOS.cdb`	The DSP/BIOS configuration file, preset for CST
`Main.c`	The main source code file of the flex application
`BIOS\CSTBIOS.c`	The DSP/BIOS support file
`BIOS\BIOSmemman.c`	The CST wrapper for the DSP/BIOS memory manager
`ROM\CSTRom.s54`	The CST ROM reference file, includes references to DSP/BIOS components in ROM
`ROM\rts_ext.lib`	TI’s Runtime Library (the version which was used to build CST)

The DSP/BIOS configuration file (`CSTFlexAppBIOS.cdb`) specifies locations and sizes of the application’s code and data sections. This file also contains sizes of the heap and stack.

To accommodate the developer needs, the DSP/BIOS configuration file may be modified (see *TMS320 DSP/BIOS User’s Guide* (SPRU423B)).

After the project has been built, the produced out-file can be converted to a binary image to be loaded into the CST chip via the serial port (if a JTAG is not available).

To do this, run the file `hexCST_BIOS.bat`, which will first obtain a hex file by running the utility `hex500.exe`, and then a binary image by running the utility `hextobin.exe`.

`hexCST_BIOS.bat`	The batch file, which runs `hex500.exe`, and then `hextobin.exe`
`hex_BIOS.cmd`	The command file for `hex500.exe`
`hextobin.exe`	ASCII to BINARY file conversion utility

The project file, linker command file, DSP/BIOS configuration file, batch file and conversion utility reside in the directory `FlexAppBIOS`.

5.4.2 CST Bootloader

The CST bootloader is used to transfer code from an external source into internal or external memory following power-up.

The bootloader provides a variety of ways to download code to accommodate different system requirements. This includes multiple types of both parallel bus and serial port boot modes, UART boot mode, bootloading through the HPI, and a special 54CST chipset boot mode. Bootloading in both 8-bit byte and 16-bit word modes are supported. To determine which boot mode to use, the bootloader uses various control signals including interrupts, BIO, and XF.

The following is a list of different boot modes implemented by the bootloader, as well as a summary of their functional operation:

❑ 54CST Chipset Boot Mode

This mode is used to start the 54CST device in the CST chipset mode. Upon detection of this mode, the bootloader automatically passes the control to the 54CST chipset application. No code is copied in this mode. There are several ways to switch into the Chipset mode from the CST Bootloader:

1) High to low transition on the INT1 pin within 30 CPU cycles after reset;
2) Sending two symbols (“AT”) via UART, at 115200 bps, shortly after reset.
3) Writing a “magic” number `0x45` to memory location `0x7E` via HPI interface.

Even while in the Chipset mode, the user still has the possibility to load a Flex application (user code) into RAM using a special CST AT command (`AT#DATA`, see section 9.4.1.29).

❑ Host Port Interface (HPI) Boot Mode

The code to be executed is loaded into on-chip memory by an external host processor via the Host Port Interface. Code execution begins once the execution address loading is completed.

- ❑ Parallel Boot Modes (8-bit and 16-bit supported)

  The bootloader reads the boot table from data space via the external parallel interface bus. The boot table contains the code sections to be loaded, the destination locations for each of the code sections, the execution address once loading is completed, and other configuration information.

- ❑ Standard Serial Port Boot Modes (8-bit and 16-bit supported)

  The bootloader receives the boot table from one of the multi-channel buffered serial ports (McBSP) operating in standard mode, and loads the code according to the information specified in the boot table. McBSP0 supports 16-bit serial receive mode. McBSP1 supports 8-bit serial receive mode.

- ❑ UART Boot Mode (8-bit supported)

  The bootloader receives the boot table from the on-chip UART and loads the code according to the information specified in the boot table. Below are the UART settings used:

  - 8 data bits
  - No Parity bit,
  - 1 Stop Bit,
  - No flow control

- ❑ 8-Bit Serial EEPROM Boot Mode

  The bootloader receives the boot table from a serial EEPROM connected to McBSP1 operating in clockstop mode, and loads the code according to the information specified in the boot table.

- ❑ I/O Boot Mode (8-bit and 16-bit supported)

  The bootloader reads the boot table from I/O port 0h via the external parallel interface bus employing an asynchronous handshake protocol using the XF and BIO pins. This allows data transfers to be performed at a rate dictated by the external device.

The bootloader also offers the following additional features:

- ❑ Reprogrammable Software Wait State Register

  In the parallel and I/O boot modes, the bootloader reconfigures the software wait state register based on a value read from the boot table during the bootload.

- ❑ Reprogrammable Bank Switching Control Register

  In the parallel and I/O boot modes, the bootloader reconfigures the bank switching control register based on a value read from the boot table during the bootload.

- ❑ Multiple-Section Boot

  The 54CST bootloader is capable of loading multiple separate code sections. These sections are not required to occupy a continuous memory space as in some previous C54x bootloaders.

For more information on CST bootloader, please, refer to *TMS320C54CST Bootloader Technical Reference* document (SPRA827).

Chapter 6

CST Framework and API Overview

This chapter provides the user with overviews and descriptions of the different CST layers, services, their API.

Topic **Page**

6.1 Overview

Besides standalone XDAIS algorithms, CST offers a hierarchy of services to the user.

Generally, the CST services can be divided into two groups:

- Basic, low-level, services:
  - CST Service layer
  - S-registers
  - high-level DAA driver, peripheral driver, low-level DAA and UART drivers, interrupt and memory management subsystems
- Advanced, high-level, services:
  - AT Parser
  - CST Action
  - CST Commander layers.

The AT Parser is not used in standard flex applications.

The purpose of the CST Commander layer is executing a complex process represented as a script, containing a sequence of commands, where each command corresponds to an elementary operation to carry out. The CST Commander interacts with the CST Service layer.

The CST Action layer supplements the CST Commander but it provides virtually no additional functionality.

The CST Service layer is the foundation of CST. Its interface is similar to CST Action layer but more XDAIS oriented.

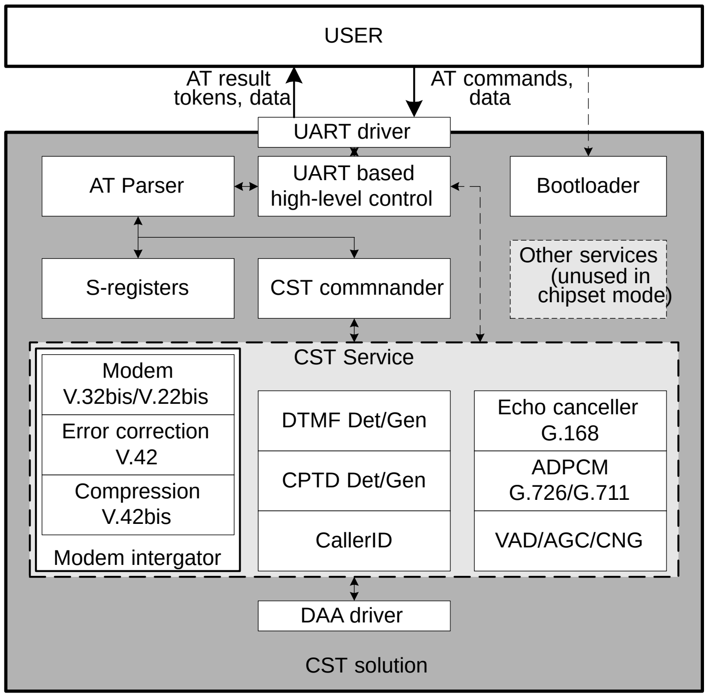
The user can skip the advanced, high-level, services and instead work with the basic, low-level, services directly.

6.2 CST Framework Layers

CST Framework has two alternative top layers and therefore, two alternative top-level interfaces.

In Chipset mode, CST Framework interacts with host via AT commands. This mode is used in chipset mode and is configured as default. General view of the CST Framework structure in initial configuration is shown in Figure 6-1.

Figure 6-1. CST Framework Controlled via AT Command Parser

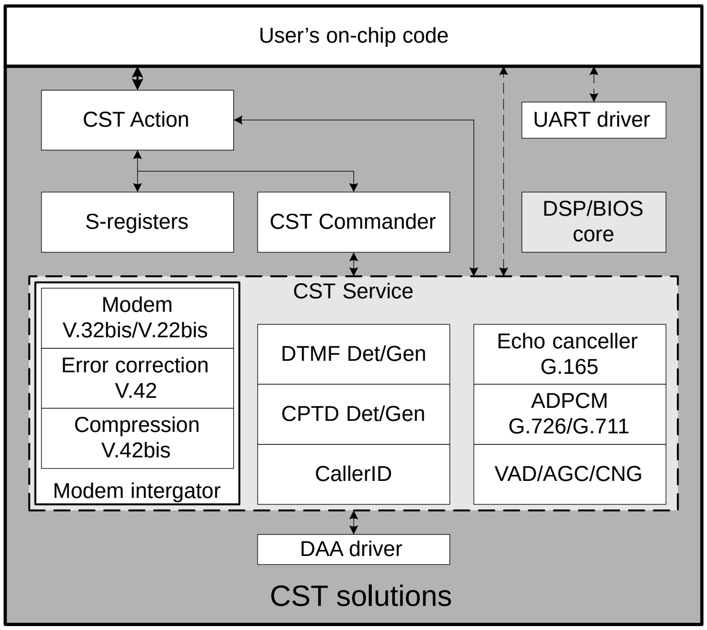


However, the default status does not mean the preference of this interface for user integration in Flex mode because it enforces the user to imitate a host terminal, generate AT commands and decode responses, substitute a virtual UART driver instead of existing one and so on.

Usually, standard CST Flex applications are not oriented on additional host terminals. If the developed firmware does not require modem-oriented AT commands, it is not recommended to use the AT parser as the top-level interface.

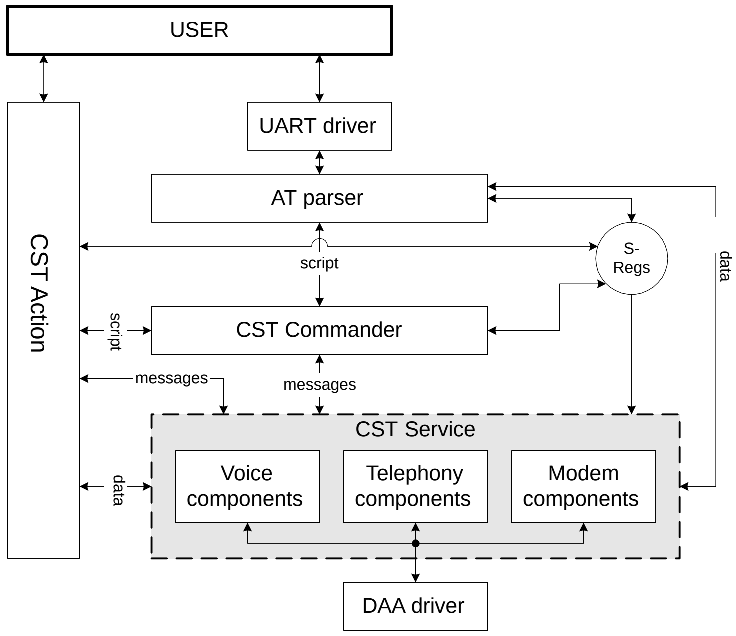
An alternative to the AT command interface is the Action-based interface. The basic concept of this method is to eliminate the AT parser and unify access to sublayers.

Figure 6-2. CST Framework Controlled via CST Action Layer



The action interface is based on interfaces of sublayers without essential modifications. The action interface allows the user to begin integration into the CST Framework.

Figure 6-3. Control Layers Interaction



CST incorporates a wide range of services. The CST Framework provides:

1) Interrupt management
2) Own and BIOS memory management
3) LIO compliant UART and DAA codec drivers, high-level DAA driver and peripheral driver (CST Service Layer)
4) A service that manages XDAIS object instances (XDAIS layer, CST Service layer)
5) A service that ties XDAIS layer with the drivers (CST Service layer)
6) A service to control and reconfigure processes on XDAIS layer (CST Service layer)
7) A service for unified digital data and control data flows organization (CST Service layer)
8) A service for automatic consecutive execution of standard operations needed for the telephony routines (CST Commander layer)
9) A service to handle CST Service messages (CST Commander layer) and to redirect message information upward (to AT parser or CST Action layer and then to the user)

10) A service for external user’s control (CST Commander layer, AT parser, optional for Flex mode)

11) An implementation of CST Commander and UART interconnection (AT parser, optional for Flex mode)

12) AT command parser (optional for Flex mode)

13) A high-level service providing a unified interface for the user to access CST data and transfer commands (CST Action layer)

When controlling the solution from an external controller via a serial port, the **AT Parser** is considered to be the most convenient, though simplified, control tool. Standard AT commands provide control over all CST solution components.

6.2.1 Action-Based Interface

The Action-based interface is an alternative to the AT parser to control the CST solution. It is a unified interface between the whole CST and the CST user application. The CST Action layer unifies all CST services being split to several sublayers in order to offer the user the most powerful and easy way for fast integration into the CST Framework. The CST Action layer interface is more convenient than AT commands and, unlike the AT command interface, it does not restrain the user from the direct access to other CST modules.

As it has been mentioned above, the CST Action layer is represented by the three functions: `CSTAction_Init()`, `CSTAction_Process()`, and `CSTAction()`.

The function `CSTAction_Init()` is used for initialization of the interface. The function `CSTAction_Process()` runs the internal CST processes, and is to be called periodically. The function `CSTAction()` is used for unified command/data delivery to CST services.

CSTAction performs actions of three types:

Table 6-1. CST Actions

CST Action	Description
Configure CST settings	This action results in reading or writing of a given system parameter mapped to one of the S registers (note that DAA registers, which are also mapped to the S registers, can't be set by the `CSTAction()` function). The action of this type will be executed immediately.
Run one standard (typical) telephony operation	This action results in preparing to execute a script consisting of a sequence of elementary operations called atomic commands. No immediate action is performed. The CST Commander later executes the atomic commands.
Transfer CST Service message	Generally used for data transfer.

Therefore, the CST Action layer integrates elements of the three modules: the S registers (configuring options), the CST Commander layer (running algorithms successively), and the CST Service layer (transferring data).

6.2.2 CST Commander Layer

The middle CST layer **- the CST Commander layer -** is intended to perform sequences of elementary operations and handling the internal state machine. In general, the CST Commander provides interaction between the user and the CST Service control layer by decoding user's instructions into separate atomic commands and transferring sequences of corresponding messages to the CST Service layer. This allows controlling the CST solution as a single object through standard and custom command scripts. The CST Commander contains a set of predefined standard command scripts, which cover most of standard operations needed for the telephony applications.

The CST Commander layer is an extendible layer that allows adding new functionality at any time, without having to change internal CST code.

At any given moment, the CST Commander focuses on execution of a single atomic command, which is active at the moment. If the command is executed successfully, the next command of the script becomes active.

The main atomic commands are as follows:

- ❑ Send a message to the CST Service that will immediately turn off all active algorithms
- ❑ Send a command to CST peripheral driver
- ❑ Sustain a pause
- ❑ Wait for appearance of a call progress tone
- ❑ Wait for disappearance or absence of a call progress tone
- ❑ Dial a telephone number stored in a string
- ❑ Send a message to the CST Service that will turn on the modem
- ❑ Send a message to the CST Service that will turn on stand alone voice loop (run echo canceller and activate voice path)
- ❑ Send a message to the CST Service that will turn on the voice pumping in rx or tx direction
- ❑ Send a message to the CST Service that will turn off the voice pumping in rx or tx direction
- ❑ Wait for the modem to establish a connection
- ❑ Send a message to the CST Service that will turn on the Caller ID
- ❑ Send a message to the CST Service that will turn on an algorithm. No algorithm specific parameters are included in the message. Used to run DTMF or CPTD detector.
- ❑ Correctly terminate the current task (usually it’s used to disconnect the modem).
- ❑ Write to an S-register
- ❑ Several system commands
- ❑ Several special AT parser oriented commands

The CST Commander layer knows how to perform each command, what to do in abnormal situations, including task cancellation. The ultimate purpose of the Commander layer is interaction with the CST Service layer.

When developing a non-standard flex application, the developer can create new custom atomic commands and scripts.

In general, the CST Commander layer performs the following operations:

1) It processes the current atomic command of the current script

2) If the current command requires sending of a message to the CST Service layer, it attempts to deliver that message to the Service layer, until the Service layer will be able to accept it or until the task is aborted

3) It processes messages from the CST Service layer, which contain data and control information

4) It also provides a service to handle CST Service messages and to redirect message information upward (to the AT Parser or CST Action layer and then to the user).

There are two important things to point out about the CST Commander:

1) The CST Commander does not support data transfers from the user to CST Service layer.

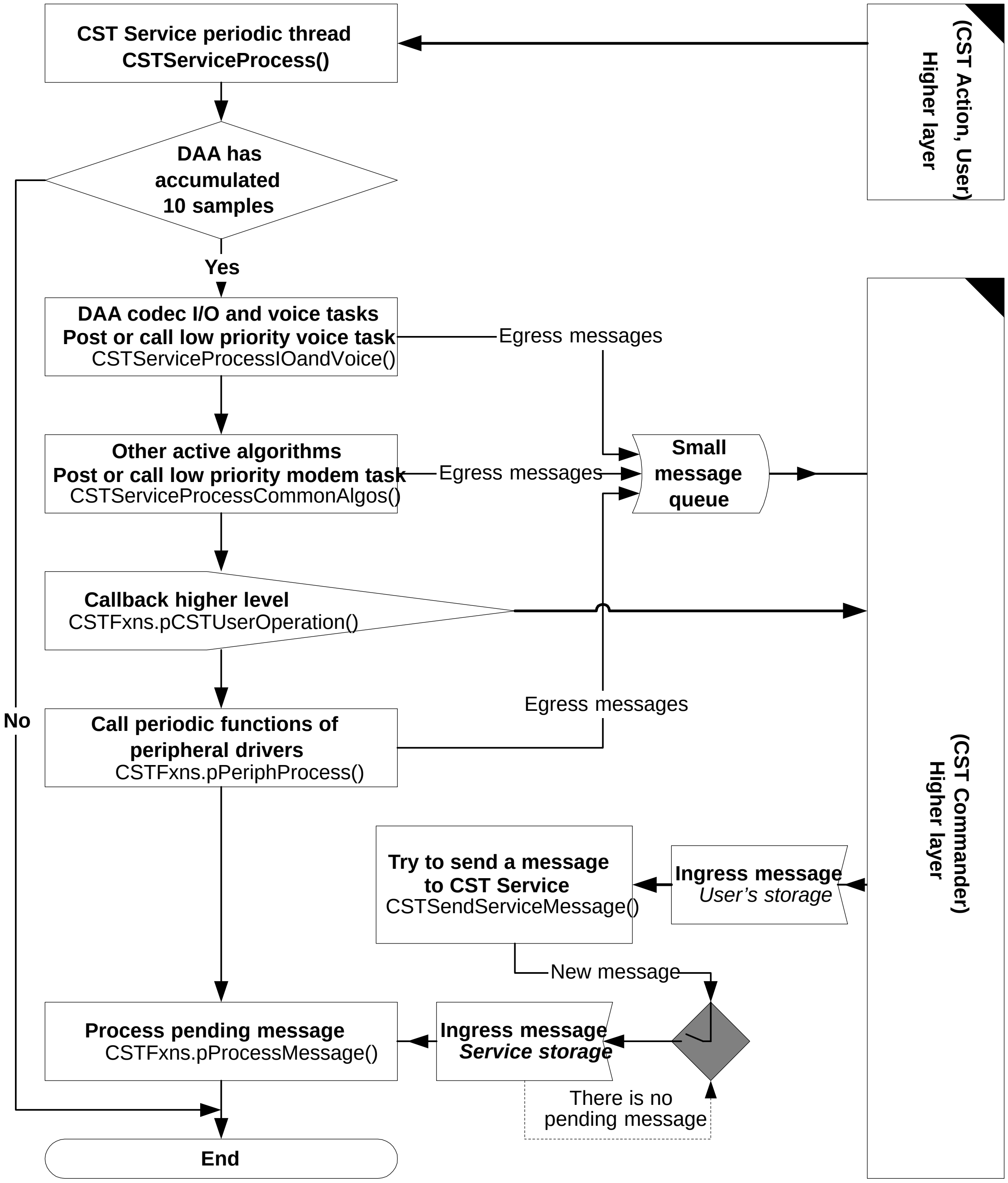
2) The user can skip both the AT Parser or CST Action layer and start working with the CST Commander directly, since the CST Commander does not depend on these layers.

6.2.3 CST Service Layer

The CST Service layer is the lowest control layer of the solution. It performs foreground processing by running XDAIS algorithms. Internally, the Service layer ties separate XDAIS objects, analog and digital data flows and control commands together. The CST Service and CST Commander layers interact with each other by dedicated messages. This allows controlling specific objects within the CST solution via unified interface, which also has some error protection capabilities.

Almost all control and data information between the CST Service and its user (the CST Commander) is carried by dedicated CST Service messages. A CST Service message is a multifunctional information packet. It can contain broadcast and dedicated control command, and data bytes. In case when a message is sent to the Service layer, it is actually a request, which the service can either accept or reject. At any time the Service layer can be executing only one request. The message being sent from the service is to inform the higher layer or submit a new portion of data. These messages are put into a small queue that allows keeping several messages issued from different tasks.

Figure 6-4. CST Service Periodic Thread


CST Service periodic thread
CSTServiceProcess()
(CST Action, User)
Higher layer
DAA has accumulated 10 samples
Yes
No
DAA codec I/O and voice tasks
Post or call low priority voice task
CSTServiceProcessIOandVoice()
Egress messages
Other active algorithms
Post or call low priority modem task
CSTServiceProcessCommonAlgos()
Egress messages
Small message queue
Callback higher level
CSTFxns.pCSTUserOperation()
Egress messages
Call periodic functions of peripheral drivers
CSTFxns.pPeriphProcess()
(CST Commander)
Higher layer
Try to send a message to CST Service
CSTSendServiceMessage()
Ingress message
User's storage
New message
Process pending message
CSTFxns.pProcessMessage()
Ingress message
Service storage
There is no pending message
End

The CST Service layer is the foundation of CST. Its main periodic function is `CSTServiceProcess()`, which is called directly from `CSTAction_Process()`. This function must be called from a high-priority periodical thread. The maximum allowed interval between the calls is 5 ms (e.g. a call once per 40 samples at 8KHz sampling rate), but it is recommended to call it several times more often. The CST Service control layer is synchronized with the DAA codec operating at 8 KHz, and, normally, the `CSTServiceProcess()` periodic function runs the internal processes if there are at least 10 new samples (which corresponds to 1.25 ms), but if there are fewer samples the function will do no processing. In multithreaded applications, `CSTAction_Process()` may be called from the user's periodic thread. In the file `BIOS\CSTBIOS.c` a high-priority SWI is periodically posted in a special procedure called from the DAA interrupt callback. This SWI calls the function `CSTAction_Process()`.

Similarly to the CST Commander, the user can work with the CST Service layer directly, skipping the AT Parser, CST Action and CST Commander layers. This is possible because the CST Service layer does not strongly depend on these layers.

If all of the above control layers lack the needed functionality for a specific task and a direct control over the CST objects is required, the XDAIS layer can be used directly. This layer includes a set of eXpressDSP compliant objects and standard XDAIS oriented framework (ALGRF) for correct operations with the objects.

CST Framework is a multichannel framework; therefore, all services are multichannel as well.

6.2.4 Other CST Parts and Services

Besides the AT-command parser, CST Action, CST Commander, CST Service and XDAIS layers, there are other parts in CST, namely: the S-registers service, high-level DAA driver, peripheral driver, low-level DAA codec and UART drivers.

The CST Framework supports a well-known modem S-registers interface. The S-registers can be thought of as object properties. The S-register service maps major CST configuration variables to the index-addressable parameter list. That means that each S-register can be assigned to an existing 16-bit variable. Writing or reading an S-register will result in writing or reading the variable associated with the S-register. Each S-register is referenced by its number. When developing a non-standard Flex application, the developer can define his/her own S-registers.

Additionally, for convenience, all DAA hardware registers are mapped to S registers starting from S register #100, so the user can configure DAA international and other settings via S registers.

A more detailed specification of S-registers can be found in sections 7.2.1.1, 9.4.5 and 9.4.6.

The high-level DAA driver is dedicated to perform the hardware-independent portion of DAA operations such as going off- and on-hook, dialing a digit (in the pulse mode), detecting rings and line reversals and more. The hardware independence is achieved by indirectly calling the low-level DAA driver functions through a well-defined LIO interface and the fact that the DAA hardware register numbers and values aren't hard-coded. This is why it has been possible to make a large portion of the entire DAA driver hardware-independent. When using a different DAA device or codec, the high-level DAA driver is not a subject to change.

For detailed information about the high-level DAA driver see section 7.7.3.

The peripheral driver is used to perform the hardware-specific initialization of CST and handling the hardware specific for the EVM54CST (LED signaling). The driver also overrides methods of the high-level DAA driver to extend them with hardware specific functionality.

For detailed information about the peripheral driver see section 7.7.2.

The low-level DAA and UART drivers give the CST and user access to the appropriate devices. The drivers export their functions through the unified LIO interface. The interface makes it possible to integrate drivers for new devices, override driver methods and alter their functionality, even at run time.

For detailed information about the low-level DAA and UART drivers see sections 7.7.5 and 7.7.6. The LIO interface is described in 7.7.4.

6.2.5 CST Layers Summary

CST Framework consists of the following parts:

- AT command parser (data and voice commands, used in chipset mode)
- Several control layers (used in Flex mode):
  - CST Action layer (to give the user control over CST solution as whole through mapping all commands and messages to different CST sublayers; eliminates the need to use the AT parser)
  - CST Commander layer (to give the user control over CST solution through a set of special command scripts)
  - CST Service layer (to provide data flow between different XDAIS components and device drivers, and to give the user unified access to CST XDAIS components through a set of special messages)

- ❑ High-level DAA driver (to perform the hardware-independent portion of DAA operations)
- ❑ Memory management subsystem
- ❑ Interrupt management subsystem
- ❑ Peripheral driver (responsible for low-level, hardware-specific initialization of CST)
- ❑ Low-level (LIO) DAA and UART device drivers

CST will use DSP/BIOS functions for handling interrupts, memory allocation and scheduling threads of different priorities if a Flex application is compiled for DSP/BIOS.

CST Service Layer

The low CST layer (CST Service layer) performs foreground processing by running XDAIS algorithms. The CST Service layer and the higher layer (the CST Commander layers) interact with each other by dedicated messages.

CST Commander Layer

The middle CST layer is intended to perform sequences of elementary operations and handling the internal state machine. In any time the CST Commander layer can be executing a single command of a script. The CST Commander knows how to perform each command, what to do in abnormal situations, including task cancellation. The purpose of the Commander layer is to interact with the CST Service layer.

AT Parser

An additional part of the CST Framework is the UART interconnection. It includes AT command parser and all related tasks such as modem escape sequence tracking (<pause> +++ <pause>), voice shielded (DLE) code processing, data flow organization, etc. The CST AT parser is a simplified service with reduced functionality. The AT parser and the high-level control functions represent an example of the user integration. The AT parser (as well as the Action layer) does not have any especially useful logic. Only the AT parser is aware of UART, it is the only layer using it.

CST Action Layer

The high-level CST Action layer represents a unified interface between the whole CST and the CST user application. The main interface function of the Action layer is `CSTAction()`. This layer does not add new functionality and only unifies the CST Service and CST Commander layer interfaces. The layer is an alternative to the AT command parser and UART interface. All data and commands to CST are sent through a CST Action unified message. All data and control information from CST is transferred through the only callback function (except for the modem and caller ID tasks). The CST Action layer is a very small service, and it is a good example of working directly with the CST Commander and CST Service layers.

6.3 Framework API

6.3.1 Main CST Types

Main CST types:

Type Name	Description
tCSTAction	Unified CST Action message (the main structure of CST Action interface). The message is a variant record, whose actual content depends on the message type. See 7.3.1.
tCSTActionType	CST Action message type (the type key). Selects the actual type of the CST Action message content. See 7.3.2.
tCSTStandardOperationType	Set of CST Action standard operations. Most operations correspond to standard AT commands, such as ATA, ATD etc. See 7.3.3.2.
tSRegDefinition	Set of defined CST S-registers. The S-registers configure CST settings and serve for ❑ XDAIS parameters setting (e.g. voice or modem speed) ❑ services parameters setting (e.g. voice gain, pause duration's) ❑ Various system parameters and controls See 7.2.1.1.
tCSTExternalMsgEvent	Set of CST Action (CST Commander) external event messages to be sent to the user's callback function. Either the AT parser or the user's flex application should receive and process such event messages. See 7.2.2.4.
tCSTAtomicCommand	Set of CST Commander atomic commands. These commands, when put together in a sequence, compose a script. Standard scripts correspond to CST Action standard operations. See 7.2.3.1.
tCSTServiceMessage	CST Service message structure (the main structure of CST Service interface). It is used for both messages addressed to CST Service and messages originating from CST. See 7.1.1.2
tCSTMessageResult	Set of CST Service (and CST Action) message result codes. Contains results of processing a message by the CST Service (or CST Action). See 7.1.1.7.
tCSTFxns	CST dynamic functions. They allow the user to extend functionality of CST services for non-standard Flex applications. All global dynamic functions are grouped in the structure `CSTFxns.` See 7.2.2.1.
tCSTChannel	A structure that keeps individual channel data. One such structure is defined in CST: `tCSTChannel Ch0`. In multichannel applications there will be a number of such structures.

Most services keep their variables (data) in separate structures. These structures are grouped together into a global structure `tCSTChannel` (defined in `CSTChannel.h`).

Therefore, data of the services is individual for each channel instance. CST defines a global variable `Ch0` to be used for single channel applications (file `CSTChannel.c`). At the same time, S-registers, DAA and UART drivers also have channel dependent data structures for each channel instance.

Nevertheless, there is also some global data common for all channels. It includes dynamic function structure, S-register descriptors, algorithm initial parameters (except for those mapped on S-registers), DAA and UART drivers' global data. This means that CST Framework can't be used for several different applications running simultaneously on the same CPU, however, it is perfectly fine to have a single multi-channel flex application.

6.3.2 S-Registers

CST defines several dozens of S-registers (`enum tSRegDefinition` defined in `CSTSReg.h`). Most of them may be split into the following semantic groups:

- ❑ Registers specifying dial operation (`srd_LONG_DIAL_DELAY, srd_DTMF_TONE_DURATION, srd_DEFAULT_DIAL_MODE` and several temporary registers for dial modifiers)
- ❑ Registers for modem settings (`srd_V42, srd_V42BIS, srd_MODEM_GAIN, srd_FAST_CONNECT, srd_DESIRED_MODEM_SPEED, srd_TIME_BEFORE_FORCED_HANGUP` and AT parser oriented register `srd_ESCAPE_PROMPT_DELAY`)
- ❑ Registers for voice settings (`srd_VOICE_GAIN, srd_ECAN, srd_VOICE_BPS, srd_VAD, srd_AGC, srd_DLECHAR`)
- ❑ Registers for other settings (`srd_INPUT_GAIN, srd_CID_MODE`)
- ❑ Registers controlling AT parser behavior
- ❑ System indication registers (`srd_STATISTICS_FLAGS, srd_AVAILABLE_ALGOS, srd_AVAILABLE_MEMORY, srd_STACK_FREE_SIZE, srd_PEAK_MIPS, srd_INPUT_POWER`)

When developing a non-standard application, the developer can add new S-registers. As it has been mentioned earlier, all channel instances have a common list of S-registers, but the values are individual. Most of S-registers are assigned to variables in `tCSTChannel` structure. New S-registers can be linked to memory that is reserved for user in array `tCSTChannel.aUserReservedWords`.

S-registers with numbers exceeding 100 are treated as physical peripheral registers (CST maps Si3021 DAA registers to S-registers 100…119). User should not access these registers via CST Action interface.

For more information about S-registers, read section 7.1.1.

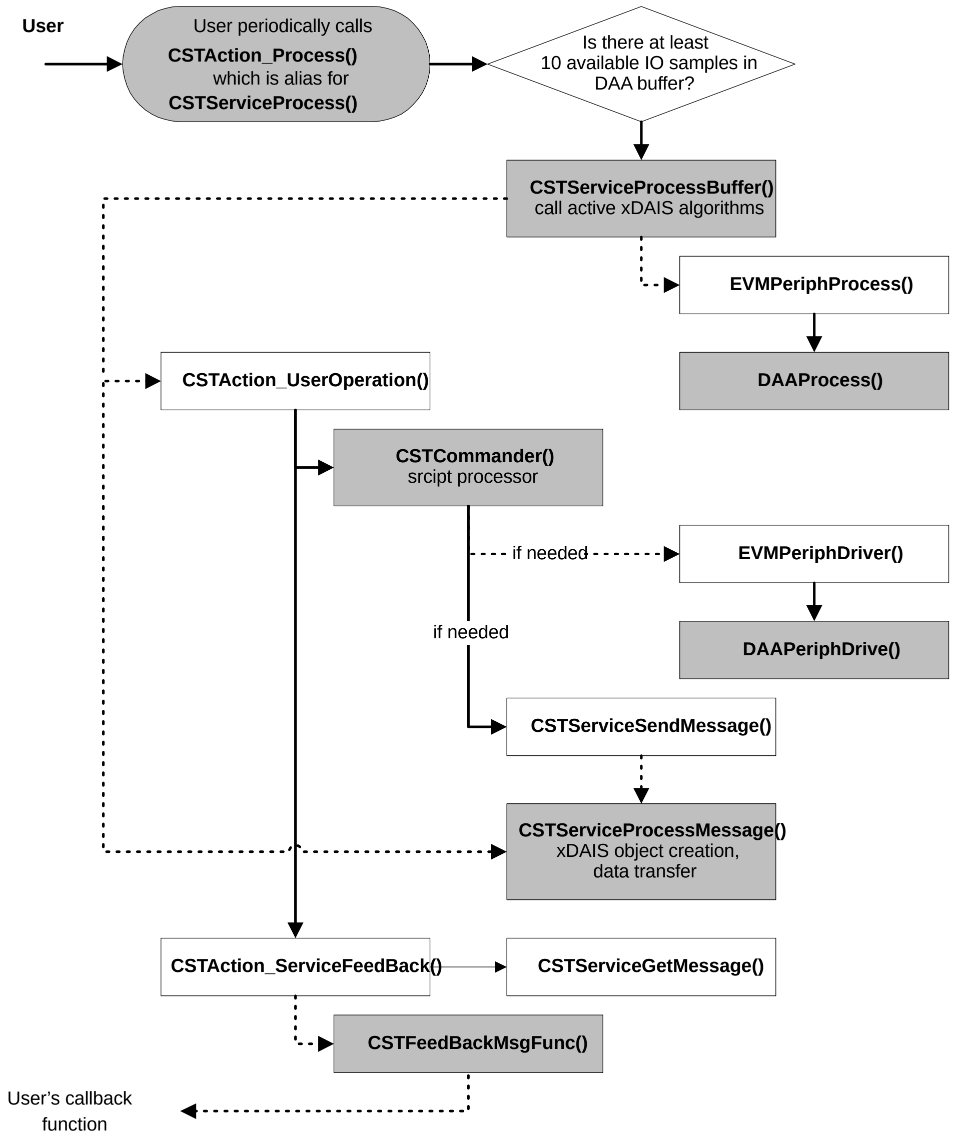
6.3.3 Call Tree

All CST algorithms run in a high-priority periodic process. The entry point of the periodic thread is defined as `CSTAction_Process()`. In multi-threaded mode, the CST Service control layer posts a couple of low priority threads. The user should not manually post them. During periodic thread execution, a number of internal CST routines are to be called and in between them the user's callback function is invoked. For more flexibility, some important CST functions are called via pointers. Once initialized, the pointers never change. This allows the user to redefine the virtual CST functions and, thus, alter or add functionality of various services.

Most of the pointers for virtual/dynamic functions are defined in the global structure `CSTFxns` (of the `tCSTFxns` type, declared and defined in `CSTCommander.h`, `CSTCommander.c`). See sections 6.3.8 and 7.2.2.1.

A schematic diagram of the periodic thread call tree is shown in Figure 6-5.

Figure 6-5. Schematic Diagram of CST Periodic Thread Call Tree



The key functions are painted in gray. Dotted arrows represent dynamic function calls, e.g. the functions are called through the `CSTFxns` pointers. The Action, Commander, Service control layers, high-level DAA driver and the peripheral driver are presented by the following functions:

- ❑ CSTAction_Process(), CSTAction_UserOperation() and CSTAction_ServiceFeedBack() belong to the CST Action control layer.
- ❑ CSTCommander() and CSTFeedBackMsgFunc() belong to the CST Commander layer.
- ❑ CSTServiceProcess(), CSTServiceProcessBuffer(), CSTServiceSendMessage(), CSTServiceProcessMessage(),CSTServiceGetMessage() belong to the CST Service control layer
- ❑ EVMPeriphProcess(), DAAProcess(), EVMPeriphDriver(), DAAPeriphDrive() belong to the peripheral and high-level DAA drivers.

The functions CSTServiceSendMessage(), EVMPeriphDriver() and CSTAction() (the last one is never called in the periodic thread) implement main control for respective layers. A call to any of these functions may initiate a process, which will take some time to complete. Therefore, a subsequent call to the function may not always be accepted, as there may still be an ongoing process that hasn’t finished yet. If a service cannot accept a user request, the user should repeat the request during (or after) next periodic thread iteration.

For more information concerning this topic, please read about *The Main Periodic High-Priority Thread Function* in 7.1.1.9.

6.3.4 Controlling CST Through Action Layer Interface

The CST Action control layer unifies access to the three services: S-registers, CST Commander and CST Service control layers. To access any of these services, a CST Action message should be sent to the CST Action layer with the function `CSTAction()`. There are four message type keys defined (`enum tCSTActionType` defined in `CSTAction.h`): `cat_SET_REGISTER`, `cat_GET_REGISTER`, `cat_STANDARD_OPERATION` and `cat_CSTSERVICE_MESSAGE`. Therefore, depending on the type of the message, the function `CSTAction()` will do one of the following: read/write an S-register, run a script (a sequence of atomic commands) or send a message directly to the CST Service layer (usually used to transfer data).

The CST Action can run both standard and custom user-defined scripts.

The set of standard scripts is defined by the `enum tCSTStandardOperationType` (file `CSTAction.h`). Each of the set elements corresponds to a standard script. The standard scripts may be split into the following semantic groups (the parenthesis contain the applicable set elements):

- ❑ Scripts for standard modem operations (sot_TURNON_MODEM_CALL_X, sot_TURNON_MODEM_ANS)
- ❑ Scripts for standard voice operations (sot_TURNON_VOICE_CALL_X, sot_TURNON_VOICE_ANS, sot_TURNON_VOICE_RXDATA, sot_TURNON_VOICE_TXDATA, sot_TURNON_VOICE_RXTXDATA, sot_TURNOFF_VOICE_DATA)
- ❑ Scripts for standard caller ID operations (sot_CID_AFTER_RINGEND, sot_CID_AFTER_LINE_REVERSAL)
- ❑ Scripts for standard simple telephone operations (sot_OFF_HOOK, sot_JUST_CALL_X)
- ❑ Scripts for task termination (sot_SOFT_TURNOFF_ALL, sot_TURN-OFF_ALL)

Other standard scripts are sot_CSTSERVICE_TURNOFF_ALL, sot_CUS-TOM_ATOMIC_CHAIN_X and sot_SET_DIAL_STRING_X.

All of the identifiers with the postfix '_X' need an additional parameter (a dial number string or the user-defined sequence of atomic commands, e.g. a user-defined script). The additional parameter is also stored in the CST Action message.

The internal CST processes, some of which may have been initiated by CST Action messages, will eventually send event and data messages into the user's callback function. The Action based layer takes part in transferring these messages.

The CST Action layer redirects the CST Commander's feedback messages to the user's callback function.

The layer also sends periodic timer messages to the user's callback. Additionally, the CST Action layer defines a dedicated modem callback function to be able to get the received data from the modem and pass the data as a message to the user's callback function. This is a default configuration. However, it is recommended to use another modem callback function for intensive modem data transmission, because the CST Action interface does not allow the user to reject the received data. If the user can't take the data, the data will be lost.

All these messages, which will be sent to the user's callback, are defined in enum tCSTExternalMsgEvent (file CSTCommander.h).

The feedback messages for the user's callback may be split into the following semantic groups:

- ❑ Messages with received information (eme_PERIPH_DATA, eme_DTMF_DATA, eme_CPTD_DATA, eme_VOICE_DATA, eme_MODEM_DATA, eme_CID_DATA)
- ❑ Event messages (eme_TICK, eme_MODEM_CONNECT, eme_MODEM_DISCONNECT, eme_VOICE_DISCONNECT)
- ❑ System data messages

Strictly speaking, the above lists are not exact. Even though the message eme_CID_DATA should carry CID data, it does not. The reason for this is the amount of CID data (which may exceed the size of the feedback message) and their formatting. Therefore, the data is not attached to the message and other functions should be used to read the CID data. The message eme_TICK does not carry any data except the time (measured in 8KHz samples) since the last eme_TICK message. Because of that and the fact that this message is periodic (the period is related to the time between subsequent calls to CSTAction_Process() or, what is the same, to CSTServiceProcess()), it may be treated as an event message.

6.3.5 Standard and Custom Atomic Commands

The CST Commander control layer is designed to process complex scripts. Each script is a sequence of atomic commands (see standard scripts in CSTAtomic.c). CST predefines approximately thirty atomic commands (enum tCSTAtomicCommand defined in CSTAtomic.h), which may be split into the following semantic groups:

- ❑ Commands to turn on an algorithm and activate a Service task (cac_TURNON_VOICE_LOOP, cac_TURNON_VOICE_DATA_X, cac_TURNON_MODEM, cac_TURNON_CID_X, cac_TURNON_SIMPLE_X)
- ❑ Commands to turn off one or several algorithms and deactivate one or several Service tasks (cac_TURNOFF_ALL, cac_TURNOFF_VOICE_DATA_X, cac_SOFT_STOP_TASK)
- ❑ Conditional/unconditional pauses (cac_PAUSE_X, cac_WAIT_CPTD_APPEARANCE_XX, cac_WAIT_CPTD_DISAPPEARANCE_X, cac_MODEM_CONNECT_WAIT)
- ❑ Commands for standard telephone operations (cac_PERIPH_SIMPLE_X, cac_DIALING)
- ❑ A number of system commands
- ❑ A number of AT parser oriented commands

Every script must end with a `cac_NONE` command. All of the identifiers with the postfix '_X' expect an additional parameter placed in the next word. All of the identifiers with the postfix '_XX' expect two additional parameters placed in the next two words (again, see standard scripts in `CSTAtomic.c`).

The developer can alter the behavior of the standard commands and add new ones by extending the `CSTCommander()` function (see section 7.4.2).

6.3.6 Command Execution at Different CST Layers

Let us consider processing of a basic command which dials the phone number "532" and establishes a modem connection. In Chipset mode, this is initiated by the `ATDT532` command; in Flex mode it is initiated by CST Action message, passed via the `CSTAction(&Ch0, &Action)` function, with the `Action` message equal to:

- `Action.ActionType =` **`cat_STANDARD_OPERATION;`**
- `Action.Action.CSTStandardOperation.OperationType =` **`sot_TURNON_MODEM_CALL_X;`**
- `Action.Action.CSTStandardOperation.aData[0..3] =` **`"532"`**`;    // pseudo-code`

This message results in processing of a script in the CST Commander. The script is contained in the array `aTurnOnModemCall[]`, which is selected by the value `sot_TURNON_MODEM_CALL_X` (see this and other standard scripts in `CSTAtomic.c`). Following is the script with some commands being omitted:

```
const tCSTAtomicCommand aTurnOnModemCall[]=
{
  //Make sure that CST Service has no task
  cac_TURNOFF_ALL,
  //Send to peripheral driver (DAA) a command.
  //Second word denotes the command type, "go off hook".
  cac_PERIPH_SIMPLE_X,
         (tCSTAtomicCommand) pdc_OFF_HOOK,
  //Turn on CPTD (rx) algorithm. This command performs
    initializing an algorithm,
  //  which does not require parameter specification
  //Second word contains the algorithm (task) ID in CST
    Service.
    cac_TURNON_SIMPLE_X,
           (tCSTAtomicCommand) cstst_CPTD,
    //Wait for dial tone detection.
    //Second word specifies the tone type.
    //Third word contains the pause value. If the value is
      in the range of
    //  0 .. than csp_SPECIAL_PAUSE_AMOUNT-1, it is
       treated as an index in
    //  aCSTSpecialPauses array (the array of programmable
       pause values).
    cac_WAIT_CPTD_APPEARANCE_XX,
     (tCSTAtomicCommand) ICPTDDET_DIAL,
     (tCSTAtomicCommand) csp_CPTD_DIALTONE_TIMEOUT,
    //Dial number. It takes number string from
      aDialNumber.
    cac_DIALING,
    //Turn on modem task. It takes several init.
      parameters from S-registers
    cac_TURNON_MODEM,
    //Wait until modem connection established
    cac_MODEM_CONNECT_WAIT,
    //End of script (atomic command chain) end. Must end
      each script.
    cac_NONE
  };
```

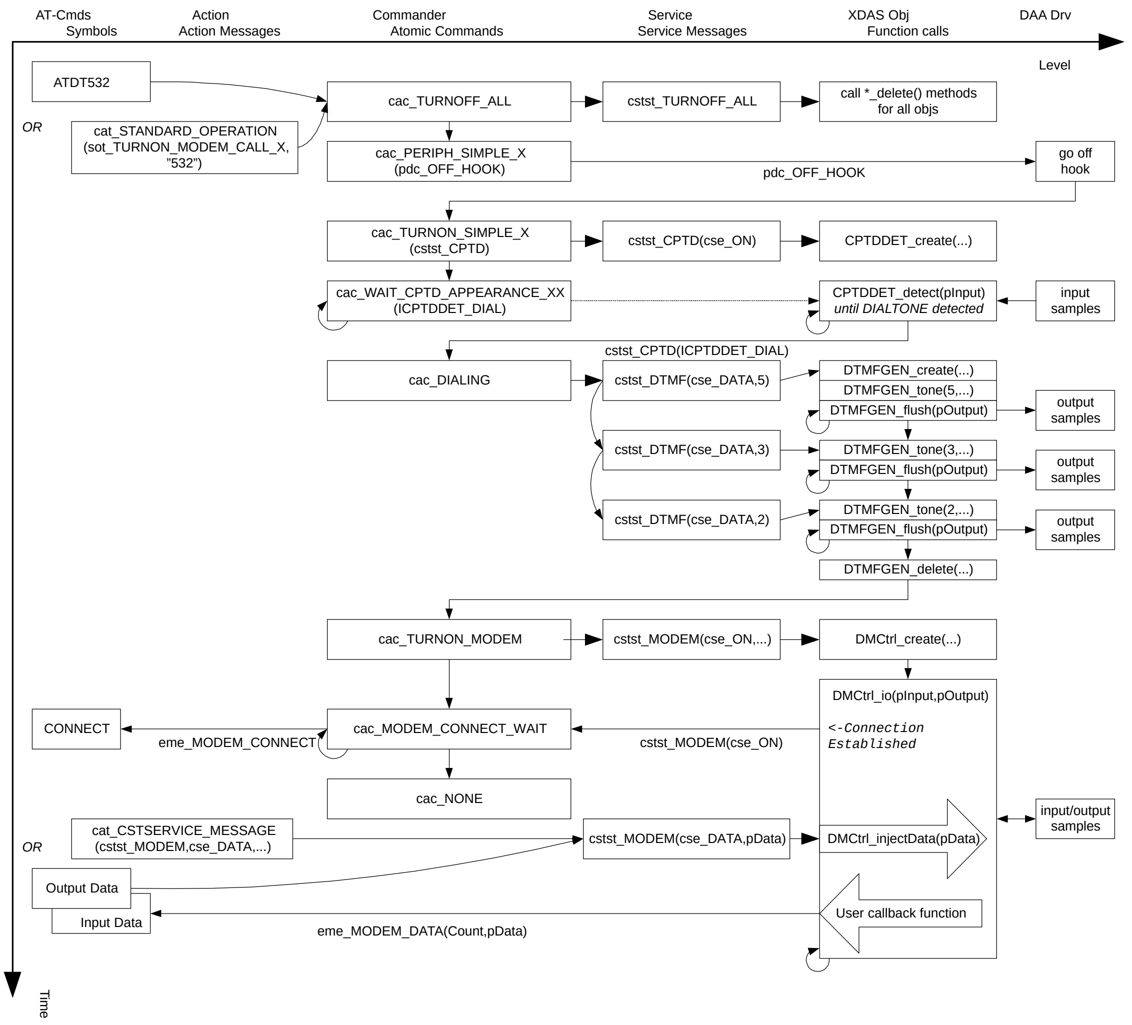
The CST Commander processes each of these atomic commands. Some of the commands transform to messages to the CST Service, some of the commands transform to the peripheral driver commands, and some of them just modify S-registers.

In turn, the CST Service messages are processed by the CST Service layer, which results in running the XDAIS telephony algorithms/objects.

Considering the opposite direction of message flow, result codes from the peripheral drivers and XDAIS objects are processed by either CST Service or CST Commander, which leads to some change in their state, such as moving to next atomic command.

The process described above is shown in Figure 6-6

Figure 6-6. Example of Command Execution at Different CST Layers



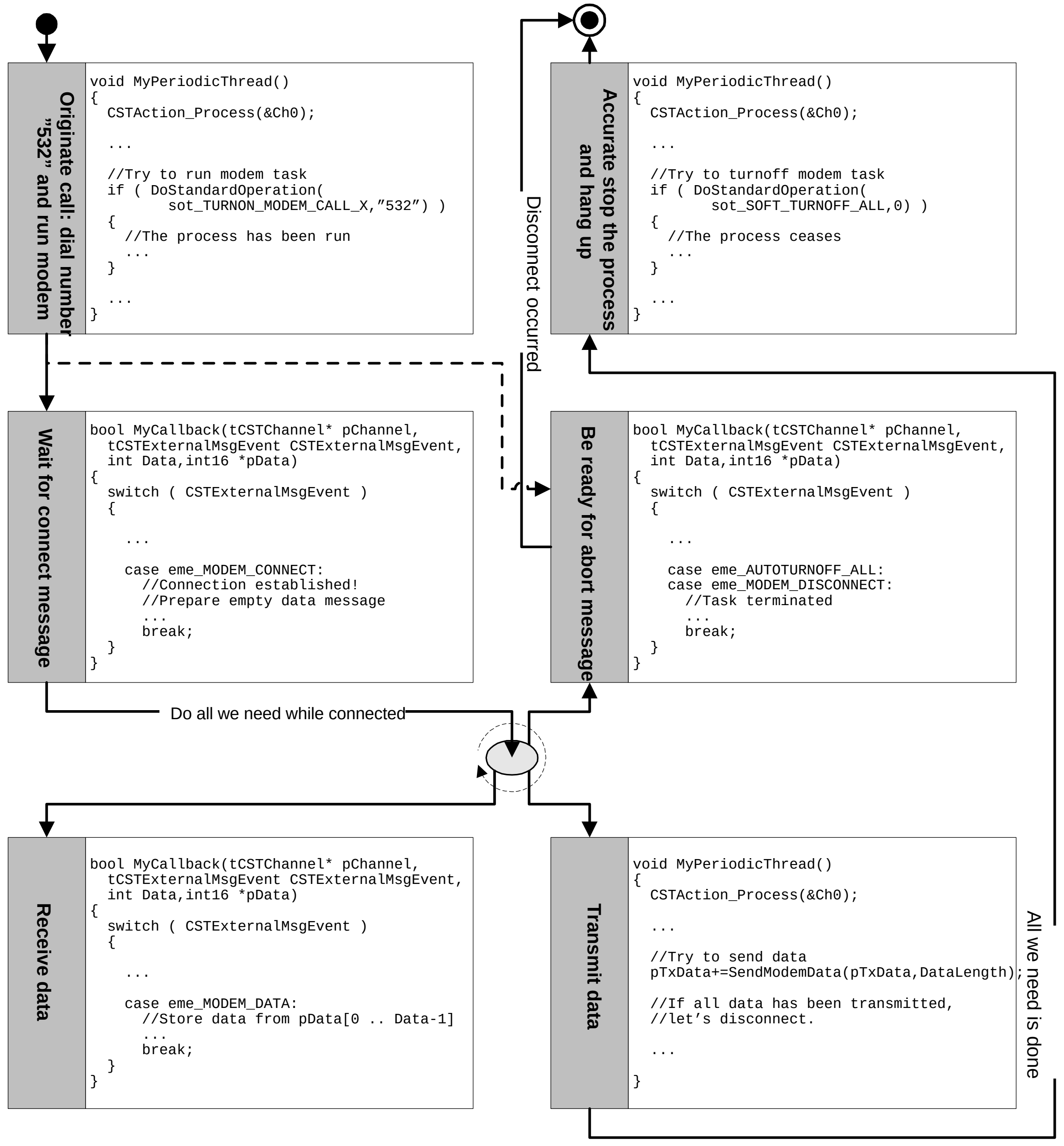
Once again, either AT commands (usually, in Chipset mode) or CST Actions (in Flex mode) are used. These are the two mutually exclusive ways to control the CST solution.

6.3.7 CST Action Interface Usage

As it has been said earlier, the CST SDK contains a number of flex application examples. All the examples are designed in a similar manner and propose to implement the user's code by filling bodies of the three functions: `MyInitialization()`, `MyCallback()` and `MyPeriodicThread()`.

For example, a modem flex application can be implemented by putting several relatively standard pieces of code into the functions mentioned above:

Figure 6-7. Fragments of Modem Call Code


Originate call: dial number "532" and run modem
Wait for connect message
Disconnect occurred
Accurate stop the process and hang up
Be ready for abort message
Do all we need while connected
Receive data
Transmit data
All we need is done

```
void MyPeriodicThread()
{
  CSTAction_Process(&Ch0);

  ...

  //Try to run modem task
  if ( DoStandardOperation(
         sot_TURNON_MODEM_CALL_X,"532") )
  {
    //The process has been run
    ...
  }

  ...
}
```

```
bool MyCallback(tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,
  int Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {

    ...

    case eme_MODEM_CONNECT:
      //Connection established!
      //Prepare empty data message
      ...
      break;
  }
}
```

```
void MyPeriodicThread()
{
  CSTAction_Process(&Ch0);

  ...

  //Try to turnoff modem task
  if ( DoStandardOperation(
         sot_SOFT_TURNOFF_ALL,0) )
  {
    //The process ceases
    ...
  }

  ...
}
```

```
bool MyCallback(tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,
  int Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {

    ...

    case eme_AUTOTURNOFF_ALL:
    case eme_MODEM_DISCONNECT:
      //Task terminated
      ...
      break;
  }
}
```

```
bool MyCallback(tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,
  int Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {

    ...

    case eme_MODEM_DATA:
      //Store data from pData[0 .. Data-1]
      ...
      break;
  }
}
```

```
void MyPeriodicThread()
{
  CSTAction_Process(&Ch0);

  ...

  //Try to send data
  pTxData+=SendModemData(pTxData,DataLength);

  //If all data has been transmitted,
  //let's disconnect.

  ...

}
```

The picture represents several pieces of code of the functions `MyCallback()` and `MyPeriodicThread()`, corresponding to the standard tasks highlighted in gray. Both unlisted static subroutines `DoStandardOperation()`and `SendModemData()`, which are used in several of the flex application examples in the CST SDK, call the `CSTAction()` function with a filled CST Action message. The message type is `cat_STANDARD_OPERATION` and `cat_CSTSERVICE_MESSAGE` respectively (for more details see 7.3.2). However, it is important to note that there may be an additional callback mechanism (direct call from the data modem controller, see 7.6.1) for intensive modem data transfers in ARQ mode, because the CST Action interface does not allow the user to reject the received data. If the user is unable to take the data, the data will be lost.

6.3.8 CST Dynamic Functions

For more flexibility, several important CST functions are called through pointers. Once initialized, the pointers never change. Most of the pointers to dynamic functions (e.g. functions to be called through pointers) reside in the global structure `CSTFxns` (of `tCSTFxns` type, declared and defined in CSTCommander.h, CSTCommander.c). However, several dynamic functions are defined directly in XDAIS algorithm parameters or peripheral drivers (modem callbacks and voice controller callbacks). Besides, several high-level interface functions are never called directly inside CST, and thus, the developer can create his/her own implementation of the top level CST methods based on the open source code. The full list of dynamic functions is given below:

- Functions to be invoked in CST Flex mode:
  - Dynamic functions to manage CST services
    - The main callback function `CSTFxns.pCSTExternalMsgEvent`.
    - High-level functions at CST Action, Commander and Service layers (`CSTFxns.pCSTUserOperation`, `CSTFxns.pCSTFeedBackMsgFunc` and `CSTFxns.pProcessMessage` respectively). Overriding the functions allows to add or modify atomic commands execution; DAA input/output sample processing (run a new external algorithm); add/or modify processing of CST Service messages (in both directions). Thus, the three dynamic methods provide a standard way to run and control an external algorithm.
    - Peripheral driver functions (`CSTFxns.pPeriphProcess`, `CSTFxns.pPeriphDriver`). Overriding these functions allows quick adaptation of CST Framework and user application to specific target platform.

  - Low-level drivers methods. The methods of the low-level UART and DAA drivers are also accessible through pointers and may be changed to make CST work with hardware different from the default CST hardware (e.g. internal UART and DAA, EVM54CST).
  - Multithread-related functions (`CSTFxns.pVControllerPost`, `CSTFxns.pVControllerProcess`, `CSTFxns.pLowPriorityModem`, `DMControllerSubfxns.pPreemptionControl`, `BIOSDAADataCallBack()`). Multithreaded applications need to override these routines. An example of this for DSP/BIOS is given in file `BIOS\CSTBIOS.c`.
  - Other functions (`CSTFxns.pVControllerSelectVocoder`, `DMControllerSubfxns.pTransferData`). These methods allow adding an external vocoder and correctly getting intensive data flow from a remote modem.
- Top-level CST methods called by user only (`CSTAction()`, `CSTAction_Process()` or `CSTServiceProcess()`).

❑ Functions intended for AT Parser (Chipset mode).

Chapter 7

CST Framework Components

This chapter gives detailed description of each of the CST Framework components, their interface and architecture.

First three sections describe CST control layers.

Section 7.4 describes AT parser implementation.

Section 7.5 describes the SPIRIT proprietary memory manager, used in CST by default, and how to reload this manager with user-specific (from DSP/BIOS or other).

Section 7.6 gives an overview of XDAIS components and describes modem and voice controllers.

Section 7.7 describes CST drivers and how to reload existing/add new drivers to CST.

Topic **Page**

7.1 CST Service Layer

7.1.1 Files CSTSReg.c, CSTSReg.h

7.1.1.1 Exchanging Messages With CST Service Layer

The lowest CST layer (the CST Service layer) performs foreground processing by running XDAIS algorithms. The CST Service and a higher control layer (normally, the CST Commander layer) interact with each other by dedicated messages.

In case when a message is sent to the Service layer, it is actually a request, which the service can either accept or reject. At any time the Service layer can be executing only one request. If the service cannot execute a new request message immediately (e.g. there is a pending process) it keeps the message as delayed. If there is already a delayed message, the new message request will be rejected without any effect. That means, “Try again later”. The same behavior is correct for data messages. Therefore, the CST Service layer provides a protection logic that guards the correct order of command execution. As an exception, there is the only message that can discard another delayed message, terminate the pending process and turn off all algorithms, i.e. reset the CST Service layer to idle state.

The message being sent from the service is to inform the higher layer or submit a new portion of data. These messages are put into a small queue that allows keeping several messages issued from different tasks. However, sending messages from the service is not the only way to transfer data. For some intensive data flows, a direct connection between an XDAIS algorithm and a client can be used (in CST Framework, it is the case with the Modem Controller).

7.1.1.2 CST Service Message

Description Almost all control and data information between the CST Service and its user (the CST Commander) is carried by dedicated CST Service messages. A CST Service message is a multifunctional information packet. It can contain broadcast and dedicated control command, and data bytes.

Structure `typedef struct tCSTServiceMessage {`

Table 7-1. CST Service message

Field Type	Field Name	Description
tCSTServiceTask	Task	Destination/source task or special broadcast command (see 7.1.1.3)
bool	IsItTxTask	Selects receiver/transmitter if needed
tCSTSubEvent	SubEvent	Determines type of the message (see 7.1.1.4)
int16	DataLength	If `SubEvent` is equal to `cse_DATA`, this field sets the data length
int16	aData[ ]	If `SubEvent` is equal to `cse_DATA`, this array contains the data. If `SubEvent` is equal to `cse_ON`, this array may contain task dependent initialization parameters

`} tCSTServiceMessage;`

Type `tCSTServiceMessage` is defined in CSTService.h.

7.1.1.3 Set of CST Service Tasks

Description The type `tCSTServiceTask` is used to denote a destination task if the message is sent from outside (ingress message) the CST Service, or a source task if the message is sent from the CST Service (egress message). A value of the type `tCSTServiceTask` is a task ID. Following are the task IDs used in CST:

Enum Definition `typedef enum tCSTServiceTask {`

Table 7-2. Set of CST Service Tasks

Name	**Value**	**Description**
`cstst_NOTASK`	0	Symbolizes a void message, serving for empty message return by `CSTServiceGetMessage()`. This message can be sent to the service and it will be accepted, but it will have no effect. (see 7.1.1.9) If `CSTServiceStatus.ActiveTxTask` is equal to `cstst_NOTASK`, there is no active tx task. (see 7.1.1.6)
`cstst_PERIPH`	1	A message with this task ID can be sent from the service only. `aData[0]` contains an event detected by the peripheral hardware. (see 7.7.2.3)
`cstst_TURNOFF_ALL`	2	When sent to the service, a message with this task ID is addressed to all algorithms and it is intended to turn them off. All other fields in the message structure are ignored. The message with this command will be executed immediately. When sent from the service (upon message processing failure), a message with this task ID reports that an algorithm could not be created. Upon receiving of the message from the CST Service it is suggested to terminate all active processes.
`cstst_CID`	3	Caller ID
`cstst_CPTD`	4	Call Progress Tone Detector (as well as generator)
`cstst_DTMF`	5	Dial Tone Modulated Frequency
`cstst_MODEM`	6	Data Modem
`cstst_VOICE_LOOP`	7	Handset – DAA full duplex loop (Handset is not present in CST chip, but the User may want to add it as an external codec, and that's why CST reserves some support for it). Voice loop and echo canceller (if enabled via S register settings) are activated.
`cstst_VOICE_DATA`	8	Voice data link
`cstst_PARAM_DATA`	9	Special system task for loading external flex application image (data) into CST chip. User should never use it

```
} tCSTServiceTask;
```

Type `tCSTServiceTask` is defined in CSTService.h.

7.1.1.4 Set of CST Service Message Types

Description The CST Service message type determines the message meaning and tells how to treat other message fields. Depending on the direction (ingress or egress), the type values have different meanings. Standard type values are described in Table 7-3:

Enum Definition typedef enum tCSTSubEvent {

Table 7-3. Set of CST Service Message Types

Name	Value	Description
`cse_ON`	0	For ingress case (e.g. for messages being sent **to** the **CST Service**) this is a command to turn on a task. For egress case (e.g. for messages being **from** the **CST Service**) this is a special notification, e.g. modem just connected
`cse_OFF`	1	For ingress case this is a command to turn off a task. For egress case this is a special notification, e.g. modem just disconnected
`cse_DATA`	2	Defines a data packet (used in both directions). This message is usually used in CST Action interface to carry user's data. For `cstst_DTMF` task it automatically creates and destroys the appropriate DTMF (UMTG) instance.
`cse_OTHEREVENTS`	3	Reserved

`} tCSTSubEvent;`

Type tCSTSubEvent is defined in CSTService.h.

7.1.1.5 CST Service Message Summary

Table 7-4 summarizes the CST Service message content. Notice, the meaning of some fields depends on the content of preceding fields.

Table 7-4. CST Service Message Summary

CST Service Message Fields					Direction	Content Meaning
Task	**IsItTx-Task (is it for TX or RX Task?)**	**SubEvent**	**aData[0]**	**aData[1.. DataLength-1]**		
cstst_NOTASK	-	-	-	-	-	Empty message
cstst_PERIPH	- (There is only Rx)	cse_DATA	value	-	egress	Event from peripheral driver
cstst_TURNOFF_ ALL	-	-	-	-	ingress	Immediately turn off all algorithms
		-	value	-	egress	Request failed
cstst_CID	- (There is only RX)	cse_ON	mode	-	ingress	Turn on caller ID reception
		cse_OFF	-	-		Turn off caller ID reception
		cse_DATA	Error code or a repeat dummy code	-	egress	Received caller ID data
cstst_CPTD	False (There is only RX)	cse_ON	-	-	ingress	Turn on CPTD
		cse_OFF	-	-		Turn off CPTD
		cse_DATA	value	values (unusual)	egress	Detected CPT
cstst_DTMF	True (TX)	cse_DATA	DTMF symbols and durations		ingress	Turn on DTMF generator, generate DTMF symbols, turn off DTMF generator
	False (RX)	cse_ON	-	-	ingress	Turn on DTMF detector
		cse_OFF	-	-	ingress	Turn off DTMF detector

Table 7-4. CST Service Message Summary (Continued)

CST Service Message Fields					Direction	Content Meaning
Task	**IsItTx-Task (is it for TX or RX Task?)**	**SubEvent**	**aData[0]**	**aData[1.. DataLength-1]**		
		cse_DATA	value	values (unusual)	egress	Detected DTMF symbol(s)
cstst_MODEM	- (Both)	cse_ON	initialization parameters		ingress	Turn on modem
		cse_OFF	-	-		Turn off modem
		cse_DATA	data			A number of tx. data bytes
		cse_ON	-	-	egress	Modem just connected (handshake succeeded)
		cse_OFF	-	-		Modem just disconnected
cstst_VOICE_ LOOP	-	cse_ON	echo canceller mode	-	ingress	Turn on voice loop
		cse_OFF	-	-		Turn off voice loop
cstst_VOICE_ DATA	True *(TX)*	cse_ON	BPS	-	ingress	Turn on voice tx path
	False *(RX)*		BPS	-		Turn on voice rx path
	True *(TX)*	cse_OFF	-	-		Turn off voice tx path
	True *(RX)*		-	-		Turn off voice rx path
	True *(TX)*	cse_DATA	data			A number of tx data bytes
	False *(RX)*	cse_OFF	-	-	egress	Both rx and tx paths turned off
	False *(RX)*	cse_DATA	data			A number of rx data bytes

7.1.1.6 CST Service Status

Description The CST Service layer has an internal status, which is used by the service itself and can be used by outsiders.

```
typedef struct tCSTServiceStatus {
```

Table 7-5. CST Service Status

Field Type	Field Name	Description
`tCSTServiceTask`	`ActiveTxTask`	Current transmitting task (see 7.1.1.3)
`tCSTMessageResult`	`MessageResult`	Status/result code for the last message (see 7.1.1.7). This is a runtime information about message performing.
`int`	`TxMessageCounter`	Amount of egress messages waiting in the queue for delivery
`bool`	`IsRxDataOverflow`	Flag signaling that the egress message queue overflowed (at least once)
`char`	`DLEChar`	Value of the common DLE character (shield code), mapped to `S46` (see 7.2.1.1)
`int`	`InputGain`	Attenuator for input samples from the DAA. Permitted range is [0 .. 30], which corresponds to a negative gain of 0 .. –30dBm. Mapped to `S31` (see 7.2.1.1)
`int`	`InputPower`	Averaged level of DAA input, dBm. Mapped to `S65` (see 7.2.1.1)
`bool`	`IsTxMessageInUse`	Used in DSP/BIOS (multi-threaded) mode for correct egress message handling
`bool`	`IsPendingLowPriorityTask`	Protects low priority tasks from premature destruction. To be set/reset externally (see `CSTBIOS.c`).
`bool`	`IsProcessMsgNeeded`	Forces CST Service to process a message upon completion of a low priority procedure, which helps to terminate active CST algorithms and processes safely.

```
} tCSTServiceStatus;
```

Type tCSTServiceStatus is defined in CSTService.h.

7.1.1.7 Set of CST Service Message Result Codes

Description The set of CST Service message result codes inform about CST Service availability for new message and, if the last message has been executed, the result characterizes the effect of the message. The CST Service message result type is used both as runtime CST Service message status and as an immediate result of high-level CST Action (see Table 7-6).

Enum Definition typedef enum tCSTMessageResult {

Table 7-6. Set of CST Service Message Result Codes

Name	Value	Description
`cmr_EXECUTING`	0	The message (or high-level action) is in progress. Other messages, except “turn off all”, will not be accepted.
`cmr_REJECT`	1	The last message (or high-level action) has been rejected for some reason, e.g. the destination task is inactive
`cmr_RESULTOK`	2	The last message (or high-level action) has been successfully processed
`cmr_FAIL`	3	The last message has failed, e.g. an algorithm could not be created
`cmr_TRY_AGAIN`	4	There is already a high-level action being processed at the moment, which does not allow accepting a new message. This result is used in CST Action layer only (see 7.3.4.2).

```
} tCSTMessageResult;
```

Type tCSTMessageResult is defined in CSTService.h.

7.1.1.8 CST XDAIS Algorithms

All CST algorithms are eXpressDSP compliant (to learn about XDAIS, read *Using the TMS320 DSP Algorithm Standard in a Static DSP System* (SPRA577).

Most of pointers to CST algorithms are defined in the `tCSTService` substructure. (see the file `CSTService.h`). Pointers to G.726/G.729, VAD and AGC instances are allocated in a separate structure `tVControllerStr`. Being complex standalone tasks, the modem and voice modules are implemented as separate controllers outside the file `CSTService.c`, but only the voice controller has a separate instance structure because it integrates several algorithms, while the modem controller manages only one algorithm, Modem Integrator, which in turn embeds a modem data pump and V.42/V.42bis algorithms.

Table 7-7. List of XDAIS Algorithms

Name	Allocation and Description
`UMTG_Handle pDTMFGenHandle`	DTMF generator. `tCSTService` structure, defined in `CSTService.h`
`UMTD_Handle pDTMFDetHandle`	DTMF detector. `tCSTService` structure, defined in `CSTService.h`
`UMTD_Handle pCPTDDetHandle`	Call Progress Tone Detector. `tCSTService` structure, defined in `CSTService.h`
`UMTG_Handle pCPTDGenHandle`	Call Progress Tone Generator. `tCSTService` structure, defined in `CSTService.h`
`CID_Handle pCIDHandle`	Caller ID detector/receiver. `tCSTService` structure, defined in `CSTService.h`
`EC_Handle pECHandle`	Echo canceller. `tCSTService` structure, defined in `CSTService.h`
`MODINT_Handle pMODINTHandle`	Modem Integrator (data pump, V.42 and V.42bis are embedded). `DMController.c, tCSTService` structure, defined in `CSTService.h`
`void *pVocoder`	Voice G.726/G.711. `VController.c, tVControllerStr` structure, defined in `VController.h`
`AGC_Handle pAGC`	Automatic Gain Control. `tVControllerStr` structure, defined in `VController.h`
`CNG_Handle pCNG`	Comfort Noise Generator. `tVControllerStr` structure, defined in `VController.h`
`VAD_Handle pVAD`	Voice Activity Detector. `tVControllerStr` structure, defined in `VController.h`

7.1.1.9 Brief Description of CSTService Function Interface

The most part of CST Service layer is implemented in the files `CSTService.c` and `CSTService.h`. The main interface functions are the following:

Table 7-8. CST Service Interface Functions

Name	**Functionality**
`CSTServiceInit()`	CST Service layer initialization
`CSTServiceProcessIOandVoice()`	Called from `CSTServiceProcessBuffer()`. Reads/writes a number of DAA and potentially handset I/O samples through the data controllers. Runs voice algorithms and routes DAA and potentially handset samples flows.
`CSTServiceProcessCommonAlgos()`	Called from `CSTServiceProcessBuffer()`. Runs all other (non-voice) active XDAIS algorithms. Performs single-byte oriented data exchanges via CST Service messages
`CSTServiceProcess()`	The main periodic high-priority thread function. `CSTServiceProcess()`is self-synchronizing relative to DAA interrupt. The function should be periodically called at least once each 5 milliseconds.
`CSTServiceProcessBuffer()`	A subroutine for `CSTServiceProcess()`. It runs all processes for a timeframe of 10 I/O samples from the DAA.
`CSTServiceSendMessage()`	Attempts to send a new message to the Service tasks. The false result means that the previous message has not been delivered yet
`CSTServiceGetMessage()`	Returns a new message from service tasks. If there are no new messages, the method returns a message with the task ID cstst_NOTASK
`CSTServiceProcessMessage()`	Main function for ingress message processing. Attempts to process a pending message obtained by `CSTSendServiceMessage()`. Called by `CSTSendServiceMessage()` and `CSTServiceProcessBuffer()` through the pointer `CSTFxns.pProcessMessage`.
`CSTServiceProcessMessageLow()`	Special Message Execution In Low Priority Task. Called from a low priority task and attempts to execute a pending data message obtained by `CSTSendServiceMessage()`.

Initialization The CST Service layer initialization

Function void CSTServiceInit(tCSTChannel* pChannel);

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

DAA/Handset I/O, Voice Operations

Called from `CSTServiceProcess()`. Reads/writes a number of DAA and potentially handset IO samples. Runs voice algorithms and routes DAA and potentially handset samples flows.

Function `void CSTServiceProcessIOandVoice(tCSTChannel* pChannel,int *pInput,int *pOutput);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pInput`	Pointer to a buffer of 10 input samples to be read from AFE and processed (modified) by the voice (if active). The voice (if active) includes the echo canceller and G.726/G.711 encoder/decoder
`pOutput`	Pointer to a buffer of 10 output samples (from the previous iteration) to be written to AFE. After writing, the sample buffer is reset to zero and regenerated by the voice encoder (if active)

Return Value None

Running PSTN Oriented Algorithms

Runs all other (non-voice) active XDAIS algorithms. Performs single-byte oriented data exhanges via CST Service messages.

Function `void CSTServiceProcessCommonAlgos(tCSTChannel* pChannel,int *pInput,int *pOutput);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pInput`	Pointer to a buffer of 10 valid input samples to be processed by standard PSTN algorithms, which are DTMF, CPTD, Caller ID and Data Modem.
`pOutput`	Pointer to a buffer of 10 valid output samples to be updated by those algorithms

Return Value None

The Main Periodic High-Priority Thread Function

The function `CSTServiceProcess()`is self-synchronizing relative to DAA samples. The function should be periodically called at least once each 5 milliseconds. When called, it drives several different processes that can be split into the following three semantic groups: drivers (low-level DAA driver, peripheral driver and high-level DAA driver), algorithm oriented operations and high-level control operations. In order to achieve the maximum flexibility, many functions are called through pointers. The main operation, which `CSTServiceProcess()` performs, is processing a number of DAA I/O samples by running all active XDAIS algorithms. The synchronization between various CST modules is achieved via a simple messaging mechanism that represents an intermediate interface between the CST Service layer and the CST Commander layer (see section 7.1.1.2).

As it is shown in Figure 6-5, `CSTServiceProcess()`calls back a dedicated function (the `CSTAction_UserOperation()` function in the figure called through the pointer `CSTFxns.pCSTUserOperation`) that should perform user-defined control operations and manage data flows (from the CST Service layer's point of view, the user is AT parser/CST Action and CST Commander layers).

The `CSTFxns.pCSTUserOperation` function processes UART data (not shown on the picture since CST may not use UART in the Flex mode; by default, the function processes UART data only in the Chipset mode), processes messaged data from XDAIS objects (the function `CSTFeedBackMsgFunc()`in Figure 6-5) and runs the CST Commander scripts. Figure 6-5 represents final call tree for standard Flex mode application.

In the Chipset mode CST is controlled by AT commands, and in the Flex mode it is also possible to control the CST Framework operations and data flows through the DTE emulation. This means that a flex application can be controlled by the AT commands. However, **it is strongly recommended not to use the DTE emulation**. The user can control the CST Framework legally through the several CST interface layers. In both cases (DTE emulation and legal CST control), the user can modify CST behavior by creating his/her own implementation of some CST methods.

Function `void CSTServiceProcess(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

Subroutine Called From `CSTServiceProcess()`

This routine runs all processes for a block of 10 (`INPUT_OUTPUT_LENGTH`) I/O samples from the DAA.

Function

```
void   CSTServiceProcessBuffer(tCSTChannel*   pChannel,int
*pInput,int  *pOutput);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pInput`	Pointer to a buffer of 10 valid input samples to be processed by standard PSTN algorithms, which are DTMF, CPTD, Caller ID and Data Modem.
`pOutput`	Pointer to a buffer of 10 valid output samples to be generated by one of those algorithms

Return Value None

Sending Messages to CST Service

CSTSendServiceMessage() attempts to send a new message to the Service tasks. The false result means that the previous message has not been delivered yet.

Function

```
bool CSTServiceSendMessage(tCSTChannel*
pChannel,tCSTServiceMessage Message);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`Message`	Universal CST Service message (see 7.1.1.1)

Return Value Acceptance flag. The false result means that the message has not been accepted and needs to be sent again.

Receiving Messages from CST Service

Returns a new message from the Service tasks. If there are no new messages, the function returns a message with the task ID equal to `cstst_NOTASK`.

Function `tCSTServiceMessage CSTServiceGetMessage(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value Universal CST Service message (see 7.1.1.1). Outgoing messages are queued in a small FIFO queue. If there are no messages, an empty message with the task ID equal to `cstst_NOTASK` is returned.

Main Routine for Service Message Processing

Attempts to process a pending message obtained by `CSTSendServiceMessage()`.

Function `void CSTServiceProcessMessage(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

Message Processing in Low-Priority Threads

Attempts to execute a pending data message obtained by `CSTSendServiceMessage()`. Called from a low priority thread function.

Function `void CSTServiceProcessMessageLow(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

7.2 CST Commander

7.2.1 Files CSTSReg.c, CSTSReg.h

7.2.1.1 Set of S-Registers and Their Implementation

The CST Framework supports a well-known modem S-registers interface. The S-registers can be thought of as object properties. The S-register service maps major CST configuration variables to the index-addressable parameter list. That means that each S-register can be assigned to an existing 16-bit variable. Writing or reading an S-register will result in writing or reading the variable associated with the S-register. Each S-register is referenced by its number. (see Table 7-10 and Table 9-9).

The CST Framework provides a set of S-registers that can be extended by the developer. The CST Framework can operate with several arrays of S-register descriptors (see Table 7-9). Each array should be provided with a handle of the type `tSimpleMap` and registered by a dedicated function (see Table 7-13).

The main array with S-register descriptors is located in the file `CSTSReg.c`, its name is `aInternalSRegAlias[]`.

Each S-register's descriptor has a pointer to a physical variable, in which S-register's contents is stored. Most of these variables are located in the structure `tCSTSettings` (see 7.2.1.2).

```
typedef struct tInternalSRegAlias {
```

Table 7-9. S-Register Descriptor

Field Type	Field Name	Description
int	`RegNumber`	Unique number of the S-register
int*	`pValue`	Pointer to a physical variable, in which the contents of the S register is stored
char*	`pHint`	Pointer to a help string for this register being printed by the AT parser upon request

```
} tInternalSRegAlias;
```

Type tInternalSRegAlias is defined in CSTSReg.h.

Enum Definition `typedef enum tSRegDefinition {`

Table 7-10. Set of Defined CST S-Registers

No	**Name**/Alias Points to Variable	**Description**
-4	**`srd_IS_VOICE_RINGBACK_SKIP`** `CSTCommanderPrivate.IsVoiceRingbackSkip`	Internal temporary S-register – skips ring back signal appearance/disappearance detection. Default value 1. (Set automatically before a new script runs, reset by the ‘@’ dial modifier).
-3	**`srd_CALL_WITHOUT_MODEM`** `CSTCommanderPrivate.IsCallWithoutModem`	Internal temporary S-register – skips modem algorithm creation. Default value 0. (Reset automatically before a new script runs, set by the ‘;’ dial modifier).
-2	**`srd_CURRENT_DIAL_MODE`** `CSTCommanderPrivate.IsPulseMode`	Internal temporary S-register – Current Dialing Mode: 0 - tone mode, 1 – pulse mode. Automatically set to the value of `srd_DEFAULT_DIAL_MODE` S-register before a new script runs.
-1	**`srd_IS_ORIGINATOR`** `CSTCommanderPrivate.IsOriginator`	Internal temporary S-register to select whether it is an originating modem or not.
(0)	**none** `CSTSettings.S0`	Automatic Answer; does not affect CST behavior
3	**none** `CSTSettings.S3`	Command Line Termination Character <CR> By default, equal to 13.
4	**none** `CSTSettings.S4`	Response Formatting Character <LF> By default, equal to 10.
5	**none** `CSTSettings.S5`	Command Line Editing Character, backspace <BS> By default, equal to 8.
6	**none** `aCSTSpecialPauses[csp_MODEM_START_PAUSE].Duration`	Pause Before Blind Dialing, in seconds In CST this register contains the duration of the delay inserted after going off-hook and before any other action. By default, equal to 1 sec.
(7)	**none** `CSTSettings.S7`	Connection Completion Timeout, in seconds If a modem can’t establish a connection for the period of this timeout. CST will stop connecting and will go on hook. By default, equal to 60 sec.

Notes:
1) A register number in parenthesis means that the corresponding register is implemented only for compatibility and its value does not affect CST behavior. S-registers with negative numbers are not printed out in help list, and are not accessible via standard ATS command.
2) Some S-registers (labeled with Note 2) are referencing variables, which are not multichannel and are global for the whole CST. All other variables are multichannel and should be accessed via Channel structure `tCSTChannel` (see section 6.3.1).

Table 7-10. Set of Defined CST S-Registers (Continued)

No	**Name**/Alias Points to Variable	**Description**

8	**`srd_LONG_DIAL_DELAY`**
	`aCSTSpecialPauses[csp_LONG_DIAL_PAUSE].Duration`
		Comma Dial Modifier pause duration, in seconds
		Dialing string may contain the comma character, which sustains a pause in dialing for the specified amount of seconds. By default, equal to 2 sec.

(10)	**none** `CSTSettings.S10`	Automatic Disconnect Delay; does not affect CST behavior
11	**`srd_DTMF_TONE_DURATION`** `CSTSettings.DtmfToneDuration`	DTMF tone/space duration, msec. The duration of a DTMF tone and the pause between the DTMF tones. By default, equal to 80.
12	**`srd_ESCAPE_PROMPT_DELAY`** `CSTSettings.EscapePromptDelay`	Guard pause before and after '+++' (escape sequence) in 1/8th of msec Escape sequence is guarded with 2 periods of inactivity, when DTE should not send anything to DCE. If these periods exist before and after '+++' sequence, the AT Parser will consider the incoming sequence as Escape Sequence, and will switch to the Modem Online Command Mode. By default, equal to 8000 (1 sec).
26	**`srd_V42`** `CSTSettings.IsV42`	Boolean flag enabling V.42 mode (when disabled, V.14 mode is used). See section 9.4.3.10 for details. By default, equal to 1.
27	**`srd_V42BIS`** `CSTSettings.IsV42bis`	V.42bis compression selection. Bit 0 enables V.42bis compressor, bit 1 enables V.42bis decompressor. See section 9.4.3.11 for details. By default, equal to 3.
28	**`srd_MODEM_GAIN`** `CSTSettings.ModemGain`	Modem output signal attenuation in decibels (0..17 dB), treated as negative value. See section 9.4.3.13 for details. By default, equal to 9.
29	**`srd_FAST_CONNECT`** `CSTSettings.IsFastConnect`	Enables the fast connect mode. See section 9.4.3.15 for details. By default, equal to 0.
30	**`srd_VOICE_GAIN`** `VController.VoiceGain`	Output voice signal attenuation in decibels (0..30 dB), treated as negative value. See section 9.4.4.2 for details. By default, equal to 0.
31	**`srd_INPUT_GAIN`** `CSTService.CSTServiceStatus.InputGain`	Common input signal attenuation in decibels (0..30 dB), treated as negative value. Used only in Voice mode. See section 9.4.4.2 for details. By default, equal to 0.

Notes:

1) A register number in parenthesis means that the corresponding register is implemented only for compatibility and its value does not affect CST behavior. S-registers with negative numbers are not printed out in help list, and are not accessible via standard ATS command.
2) Some S-registers (labeled with Note 2) are referencing variables, which are not multichannel and are global for the whole CST. All other variables are multichannel and should be accessed via Channel structure `tCSTChannel` (see section 6.3.1).

Table 7-10. Set of Defined CST S-Registers (Continued)

No	**Name**/Alias Points to Variable	**Description**
37	**`srd_DESIRED_MODEM_SPEED`** `CSTSettings.MaxModemRateBc`	The maximum desired modem rate. See section 9.4.3.14 for details. 0,1 – Automodem; 2 – V.22 1200; … 8 – V.32bis 14400. By default, equal to 0.
38	**`srd_TIME_BEFORE_FORCED_HANGUP`** `CSTSettings.ModemForcedHangUpDelay`	An extra pause before V.42 session completion, in seconds. The modem waits this amount of time before V.42 connection is terminated, in order to flush data from internal buffers. By default, equal to 2.
40	**`srd_DEFAULT_DIAL_MODE`** `CSTSettings.DialingMode`	Default dialing mode: 0 - tone mode, 1 – pulse mode; Used when dialing string does not contain explicit dialing mode modifier. See sections 9.4.1.13 and 9.4.1.21 for details. By default, equal to 0.
41	**`srd_ECAN`** `CSTSettings.EchoCancellerMode`	Line echo canceller mode: 0 – EC off; 1 – EC on without NLP; 2 – EC on with NLP. By default, equal to 1.
42	**`srd_VOICE_BPS`** `CSTSettings.VoiceBitPerSample`	Voice Bit Per Second rate. Can be 2, 3, 4, 5 or 8, which corresponds to rates 16, 24, 32 and 40 kbps for G.726 and 64 kbps for G.711. See section 9.4.4.2 for details. By default, equal to 8.
43	**`srd_CID_MODE`** `CSTSettings.CIDEnablingMode`	Caller ID mode. Selects: 1 – formatted CID, 2 - unformatted CID information printing; 0 - disables it. See section 9.4.2.4 for details. By default, equal to 1.
44	**`srd_VAD`** `CSTSettings.VADEnabled`	Enables VAD in voice mode. By default, equal to 1.
45	**`srd_AGC`** `CSTSettings.AGCEnabled`	Enables AGC in voice mode. By default, equal to 1.
46	**`srd_DLECHAR`** `CSTService.CSTServiceStatus.DLEChar`	Shield code value. By default, equal to 0x10 (<DLE>).
47	**`srd_CID_CRCERROR_BEHAVIOR`** `CSTSettings.CIDErrorBehavior`	Enable Caller ID report even if data have been received with incorrect CRC. By default, equal to 0.

Notes:

1) A register number in parenthesis means that the corresponding register is implemented only for compatibility and its value does not affect CST behavior. S-registers with negative numbers are not printed out in help list, and are not accessible via standard ATS command.

2) Some S-registers (labeled with Note 2) are referencing variables, which are not multichannel and are global for the whole CST. All other variables are multichannel and should be accessed via Channel structure `tCSTChannel` (see section 6.3.1).

Table 7-10. Set of Defined CST S-Registers (Continued)

No	**Name**/Alias Points to Variable	**Description**
50	**`srd_AT_ECHO_MODE`** `CSTSettings.IsEchoMode`	Boolean flag to enable AT Parser echo. See section 9.4.1.7 for details.
51	**`srd_AT_AUTOBAUD`** `CSTSettings.IsAutoBaudOn`	Enables automatic adjustment of the UART baud rate. By default, equal to 1 (auto baud enabled). The UART driver performs auto baud detection only during AT command input, based on “AT” characters.
60	**`srd_STATISTICS_FLAGS`** `CSTStatistics.Flags` Note2	Statistics enable flags. Bit 0 enables MIPS measurement, bit 1 enables heap free size measurement, and bit 2 enables stack free size measurement. By default, equal to 7 (all flags enabled).
61	**`srd_AVAILABLE_ALGOS`** `CSTStatistics.AvailableAlgos` Note2	Contains a number of currently active (created) XDAIS algorithms. Read only.
62	**`srd_AVAILABLE_MEMORY`** `CSTStatistics.AvailableMemory` Note2	Contains free heap size in words. Read only.
63	**`srd_STACK_FREE_SIZE`** `CSTStatistics.StackFreeSize` Note2	Contains free stack size in words. Read only.
64	**`srd_PEAK_MIPS`** `CSTStatistics.PeakMIPS` Note2	Peak MIPS tracked since last reset (averaged on 4 msec block). Writing to this register a zero value resets it.
65	**`srd_INPUT_POWER`** `CSTService.CSTServiceStatus.InputPower`	Contains average value of input signal power, in dBm. Read only.
70	**`srd_CHANNEL`** `CSTCurrentChannel` Note2	Active channel number. Not used.

```
} tSRegDefinition;
```

Notes:

1) A register number in parenthesis means that the corresponding register is implemented only for compatibility and its value does not affect CST behavior. S-registers with negative numbers are not printed out in help list, and are not accessible via standard ATS command.

2) Some S-registers (labeled with Note 2) are referencing variables, which are not multichannel and are global for the whole CST. All other variables are multichannel and should be accessed via Channel structure `tCSTChannel` (see section 6.3.1).

Type tSRegDefinition is defined in CSTSReg.h.

The CST Framework does not provide boundary check when writing to S-register. S-registers with number greater than 100 are treated as physical hardware DAA registers (in CST Framework, they are mapped to Si3021 DAA registers) and are redirected (mapped) to the CST Peripheral module, and then to the DAA Driver. The user should not access these registers via CST Action interface, because in this interface reading/writing an S register is done immediately by accessing a variable in the DSP memory, which is not possible for SiLab’s DAA registers.

7.2.1.2 CST Settings

CST (particularly the CST Commander) holds global settings for such tasks as voice, modem etc. The user can directly modify the settings and is responsible for correctness of this modification. Some of these settings are to be passed as parameters during algorithm initialization or to be used internally, e.g. for dialing. The settings are also accessible through AT commands, and most of them are accessible via S-registers (see Table 7-11).

Structure `typedef struct tCSTSettings {`

Table 7-11. CST Settings

Field Type	Field Name	S-Reg	Description
`int`	`S3`	S3	Command Line Termination Character <CR> By default, equal to 13.
`int`	`S4`	S4	Response Formatting Character <LF> By default, equal to 10.
`int`	`S5`	S5	Command Line Editing Character, backspace <BS> By default, equal to 8.
`int`	`S0`	S0	Automatic Answer; does not affect CST behavior
`int`	`S7`	S7	Connection Completion Timeout, in seconds If a modem can’t establish a connection for the period of this timeout. CST will stop connecting and will go on hook. By default, equal to 60 sec.
`int`	`S10`	S10	Automatic Disconnect Delay; does not affect CST behavior
`int`	`DCEResponseMode`	-	AT Parser: Corresponds to the command ATV: 1-verbose result code, 0-numeric. See 9.4.1.22 for details. By default, equal to 1.
`int`	`ResultCodeSupression`	-	AT Parser: Corresponds to the ATQ command: 1-supress result code. See 9.4.1.14 for details. By default, equal to 0.

Table 7-11. CST Settings (Continued)

Field Type	Field Name	S-Reg	Description
`int`	`ResultCodeSelection`	-	AT Parser: Corresponds to the ATX command. See 9.4.1.21 for details. By default, equal to 4.
`int`	`DTRbehaviour`	-	DTR (108/2) line control. See 9.4.1.2 for details. By default, equal to 2.
`int`	`RSDbehaviour`	-	RSD (109) line control. See 9.4.1.1 for details. By default, equal to 1.
`tDialingMode`	`DialingMode`	S40	Default dialing mode: 0 - tone mode, 1 - pulse mode; Used when dialing string does not contain explicit dialing mode modifier. See sections 9.4.1.13 and 9.4.1.21 for details. By default, equal to 0.
`int`	`IsEchoMode`	S50	Boolean flag to enable AT Parser echo. See section 9.4.1.7 for details.
`int`	`IsAutoBaudOn`	S51	Enables automatic adjustment of the UART baud rate. By default, equal to 1 (auto baud enabled).
`char*`	`ManufacturerInfo`	-	Corresponds to ATI: manufacturer information returned by ATI.
`int`	`EscapePromptDelay`	S12	Guard pause before and after ’+++’ (escape sequence) in 1/8th of msec Escape sequence is guarded with 2 periods of inactivity, when DTE should not send anything to DCE. If these periods exist before and after ’+++’ sequence, the AT Parser will consider the incoming sequence as Escape Sequence, and will switch to the Modem Online Command Mode. By default, equal to 8000 (1 sec).
`int`	`MaxModemRate`	-	The maximum desired modem rate. Can be 1200, 2400, 4800, 7200, 9600, 12000, 14400. By default, equal to 14400. Changes along with `MaxModemRateB` variable.
`int`	`MaxModemRateB`	S37	Changes along with `MaxModemRate` variable.
`int`	`ModemGain`	S28	Modem output signal attenuation in decibels (0..17 dB), treated as negative value. See section 9.4.3.13 for details. By default, equal to 9.
`int`	`IsV42`	S26	Boolean flag enabling V.42 mode (when disabled, V.14 mode is used). See section 9.4.3.10 for details. By default, equal to 1.
`int`	`IsV42bis`	S27	V.42bis compression selection. Bit 0 enables V.42bis compressor, bit 1 enables V.42bis decompressor. See section 9.4.3.11 for details. By default, equal to 3.

Table 7-11. CST Settings (Continued)

Field Type	Field Name	S-Reg	Description
`int`	`IsFastConnect`	S29	Enables the fast connect mode. See section 9.4.3.15 for details. By default, equal to 0.
`tECMode`	`EchoCancellerMode`	S41	Line echo canceller mode: 0 - EC off; 1 - EC on without NLP; 2 - EC on with NLP. By default, equal to 1.
`int`	`DtmfToneDuration`	S11	DTMF tone/space duration, msec. The duration of a DTMF tone and the pause between the DTMF tones. By default, equal to 80.
`int`	`ModemForcedHangUpDelay`	S38	An extra pause before V.42 session completion, in seconds. The modem waits this amount of time before V.42 connection is terminated, in order to flush data from internal buffers. By default, equal to 2.
`int`	`VoiceBitPerSample`	S42	Voice Bit Per Second rate. Can be 2, 3, 4, 5 or 8, which corresponds to rates 16, 24, 32 and 40 kbps for G.726 and 64 kbps for G.711. See section 9.4.4.3 for details. By default, equal to 8.
`tCIDEnablingMode`	`CIDEnablingMode`	S43	Caller ID mode. Selects: 1 - formatted CID, 2 - unformatted CID information printing; 0 - disables it. See section 9.4.2.4 for details. By default, equal to 1.
`int`	`CIDErrorBehavior`	S47	Enable Caller ID report even if data have been received with incorrect CRC. By default, equal to 0.
`char`	`aDialNumber[]`	-	Current dial number ASCII string for `cac_DIALING` (see). Allowed symbols are digits and characters ‘#’, ‘*’, ‘P’ (pulse dialing), ‘T’ (tone dialing), ‘W’ (wait for dial tone), ‘,’ (long pause), ‘/’ (short pause), ‘!’ (flash), ‘R’ (response/answer mode), ‘;’ (don’t initiate modem), ‘@’ (wait for ring back appearance and disappearance).
`int`	`VADEnabled`	S44	Enables VAD in voice mode. By default, equal to 1.
`int`	`AGCEnabled`	S45	Enables AGC in voice mode. By default, equal to 1.

```
} tCSTSettings;
```

Type tCSTSettings is defined in CSTSReg.h.

7.2.1.3 Brief Description of CSTSReg Function Interface

Table 7-12. Brief Description of CST S-Registers Function Interface

Name	Functionality
`SregistersInit()`	Default S-register initialization
`SregistersAdd()`	Adds an array of S-register descriptors (see Table 7-9)
`SregisterSet()`	Attempts to set an S-register
`SregisterGet()`	Attempts to read an S-register
`SregistersFind()`	Searches for an S-register

Initialization

Default S-register intialization

Function `void SRegistersInit(tCSTChannel* pChannel);`

Parameter(s) `pChannel` Pointer to a global CST channel structure

Return Value None

Add New S-register Array

Adds an array of S-register descriptors.

Function `void SRegistersAdd(tCSTChannel* pChannel,const tSimpleMap *pSReg);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pSReg`	Pointer to a handler, containing a pointer to the array of S-register descriptors (see Table 7-13 and Table 7-9)

Return Value None

```
typedef struct tSimpleMap {
```

Table 7-13. Simple Map Structure

Field Type	Field Name	Description
`const void *`	`mpMap`	Pointer to a map/array of units, which have some fixed size and can be indexed by an integer. Type “void” is because this structure is universally used with maps/arrays of different type
`size_t`	`mUnitSize`	Size of a single unit in the map
`int`	`mLength`	Total number of units in the map

```
} tSimpleMap;
```

Type tSimpleMap is defined in CSTstd.h.

Set S-Register Attempts to set an S-register.

Function `bool SRegisterSet(tCSTChannel* pChannel,tSRegRequest *pSRegRequest)`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pSRegRequest`	Pointer to a request structure

Structure `typedef struct tSRegRequest {`

Table 7-14. S-Register Request Descriptor

Field Type	Field Name	Description
`int`	`RegNumber`	S-register number
`int`	`BitNumber`	Selects bit to be read or written. If the bit number is negative, the whole register is to be read or written.
`int16`	`Value`	16-bit value, if bit number is negative, or one bit value in MSB if bit number is positive

`} tSRegRequest;`

Type tSRegRequest is defined in CSTSReg.h.

Return Value Completion flag. The false result means that it is necessary to call this function again to push the process (this may happen when writing to peripheral registers). The function does not return false if the S-register does not exist.

Read S-Register

Attempts to read an S-register (“Attempts” because some of the register’s values, such as DAA-mapped S registers, may not be read immediately, and that is why the user should call this function several times until it returns “true” and the value can be read).

Function `bool SRegisterGet(tCSTChannel* pChannel,tSRegRequest *pSRegRequest);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pSRegRequest`	Pointer to a request structure (see Table 7-14)

Return Value Completion flag. The false result means that it is necessary to call this function again to push the process (this may happen when reading from peripheral registers). The function does not return false if the S-register does not exist. The read value is returned in the request structure.

Search for S-Register

Searches for an S-register.

Function

```
tInternalSRegAlias* SRegistersFind (tCSTChannel*
pChannel,int RegNumber);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`RegNumber`	Number of the S-register to be searched

Return Value Pointer to the descriptor of the S-register or zero if the register does not exist.

7.2.2 Files CSTCommander.c, CSTCommander.h

7.2.2.1 CST Dynamic Functions

To make the CST Framework configuration more convenient and to allow maximum re-use of the code in ROM, some important functions are called through pointers.

All these pointers to dynamic functions are grouped together in the following global structure `CSTFxns`:

Structure

```
typedef struct tCSTFxns {
```

Table 7-15. CST Dynamic Functions

Params Type	Params Name	Description
`void (*) (tCSTChannel* pChannel, int16 *pIn, int16 *pOut, int AmountOf8KHzSamples)`	`pCSTUserOperation`	User’s (overridden by the AT Parser or CST Action layers) callback function called from the CST Service. Desired area to make extra operations with I/O samples, time measurement and all control operations that are beyond the CST Service (e.g. user-defined control operations and data flow management) In the Chipset mode, the default value is `CSTUserOperation()`. In the Flex mode, the default value is `CSTAction_UserOperation()`. In the Chipset mode, this function interconnects the CST Service with UART stream being treated as AT commands and data stream.
`void (*) (tCSTChannel* pChannel)`	`pCSTServiceFeedBack`	The routine is called from `CSTUserOperation()` or `CSTAction_UserOperation()`. It gets a new message from the CST Service and calls `pCSTFeedBackMsgFunc` method to process the message. In the Chipset mode, the default value is `ATParser_CSTServiceFeedBack().` In the Flex mode, the default value is `CSTAction_ServiceFeedBack().`
`void (*) (tCSTChannel* pChannel, tCSTServiceMessage *pMessage)`	`pCSTFeedBackMsgFunc`	A good heritable method intended to process CST Service egress messages. Called from `ATParser_CSTServiceFeedBack()` in the Chipset mode and from `CSTAction_ServiceFeedBack()` in the Flex mode. The default value is `CSTFeedBackMsgFunc()`.
`void (*) (tCSTChannel* pChannel, int AmountOf8KHzSamples)`	`pCSTUserMonitor`	The routine is called from `CSTUserOperation()` and can be used for additional control monitoring. The parameter is a time stamp in 8KHz samples, which is the time passed since last call to this function. The default value is `CSTUserMonitor()`. Unused in Flex mode.

Table 7-15. CST Dynamic Functions (Continued)

Params Type	Params Name	Description

`bool (*)`
`(tCSTChannel* pChan-`
`nel,`
`tCSTExternalMsgEvent`
`CSTExternalMsgEvent,`
`int Data,`
`int16 *pData)`
	`pCSTExternalMsgEvent`	The routine is usually called from `CSTFeedBackMsgFunc()` to pass obtained data to the client (to AT parser in the Chipset mode and to the user in the Flex mode).
		In the Chipset mode, the default value is `CSTExternalMsgEvent()`. In the Flex mode it has to be initialized by the user's callback function.

`tCSTPeriphEvent (*)`
`(tCSTChannel* pChan-`
`nel,`
`int AmountOf8KHzSam-`
`ples)`
	`pPeriphProcess`	The routine is called from the CST Service and should perform all hardware related background operations.
		The default value after CST Service initialization is `DAAPeriphProcess()`, however, EVM board specific initialization overloads it to `EVMPeriphProcess()`.

`long (*)`
`(tCSTChannel* pChan-`
`nel,`
`tPeriphDriverCommand`
`Command,`
`int Param1,`
`int Param2)`
	`pPeriphDriver`	Attempts to perform a hardware-related operation.
		The default value after CST Service initialization is `DAAPeriphDriver()`, however, EVM board specific initialization overloads it to `EVMPeriphDriver()`.

`void (*)`
`(tCSTChannel* pChan-`
`nel)`
	`pCSTGlobalReset`	Reset the whole CST solution (called upon the ATZ command in the Chipset mode) – "soft restart".
		The default value is CSTGlobalReset().
		The user may want to overload this method in order to control exactly how CST restarts, for example, to keep pre-loaded patch code at software reset.

`void (*)`
`(tCSTChannel* pChan-`
`nel,`
`char Data)`
	`pUARTRxMonitor`	Inform a client (AT parser) about a new byte obtained from UART.
		In the Chipset mode, the default value is `UARTRxMonitor()`, which keeps a track of modem escape character sequences. In the Flex mode, the default value is `CSTAction_UARTRxMonitor()`, which does nothing.

`void (*)`
`(tCSTChannel* pChan-`
`nel)`
	`pProcessMessage`	CST Service message processing.
		The default value is `CSTServiceProcessMessage()`

Table 7-15. CST Dynamic Functions (Continued)

Params Type	Params Name	Description
`void (*) (tCSTChannel* pChannel)`	`pLowPriorityModem`	Called from CST Service to post a low priority modem thread function. Used in multi-threaded applications only. The default value is `NULL` (undefined). In DSP/BIOS-oriented applications, it should be initialized with a real function address (see `CSTBIOS.c`)
`void (*) (tCSTChannel* pChannel, int16 *pIn, int16 *pOut, int Count)`	`pVControllerHigh PriorityProcess`	High priority voice processing method. The default value is `VControllerHighPriorityProcess ()`.
`void (*) (tCSTChannel* pChannel, int param)`	`pVControllerSelect Vocoder`	Vocoder selection function. The default value is `VControllerSelectVocoder()`
`void (*) (tCSTChannel* pChannel)`	`pVControllerProcess`	Low priority voice processing method. The default value is `VControllerProcess()`. In DSP/BIOS-oriented applications, it can be reinitialized with another function (see `CSTBIOS.c`)

```
} tCSTFxns;
```

Type tCSTFxns is defined in CSTCommander.h.

See also memory management dynamic functions defined in section 7.5 and sections devoted to the low-level drivers: 7.7.4, 7.7.7, and 7.7.7.3 in general.

Callback Function Called From CST Service

User's (overridden by the AT Parser or CST Action layers) callback function being called from the CST Service. This is a good place to make extra operations with I/O samples, time measurement and all control operations that are beyond the CST Service (e.g. user-defined control operations and data flow management)

In the Chipset mode, the default value is `CSTUserOperation()`. In the Flex mode, the default value is `CSTAction_UserOperation()`. In the Chipset mode, this function interconnects the CST Service with UART stream being treated as AT commands and data stream.

Function

```
void  (*pCSTUserOperation)
(tCSTChannel* pChannel,
 int16 *pIn,
 int16 *pOut,
 int AmountOf8KHzSamples)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pIn`	Pointer to a buffer of valid input samples
`pOut`	Pointer to a buffer of valid output samples
`AmountOf8KhzSamples`	Amount of samples in the buffers

Return Value None

Getting Egress Message From CST Service Message

The routine is called from `CSTUserOperation()` or `CSTAction_UserOperation()`. It gets a new message from the CST Service and calls `pCSTFeedBackMsgFunc` method to process the message.

In the Chipset mode, the default value is `ATParser_CSTServiceFeedBack().` In the Flex mode, the default value is `CSTAction_ServiceFeedBack().`

Function

```
void  (*pCSTServiceFeedBack)
(tCSTChannel* pChannel)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

Processing CST Service Egress Message

A good heritable method intended to process CST Service egress messages. Called from `ATParser_CSTServiceFeedBack()` in the Chipset mode and from `CSTAction_ServiceFeedBack()` in the Flex mode.

The default value is `CSTFeedBackMsgFunc()`.

Function

```
void  (*pCSTFeedBackMsgFunc)
(tCSTChannel* pChannel,
 tCSTServiceMessage *pMessage)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pMessage`	The message from the CST Service

Return Value None

Additional Monitor Function

The routine is called from `CSTUserOperation()` and can be used for additional control monitoring. The parameter is a time stamp in 8KHz samples, which is the time passed since last call to this function.

The default value is `CSTUserMonitor()`. Not used in Flex mode.

Function

```
void  (*pCSTUserMonitor)
(tCSTChannel* pChannel,
 int AmountOf8KHzSamples)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`AmountOf8KhzSamples`	A time stamp in 8KHz samples that informs the time passed since last call

Return Value None

User’s Callback Function to Process CST Commander Messages

The routine is usually called from `CSTFeedBackMsgFunc()` to pass obtained data to the client (to AT Parser in the Chipset mode and to the user in the Flex mode).

In the Chipset mode, the default value is `CSTExternalMsgEvent()`. In the Flex mode it has to be initialized by the user’s callback function.

Function

```
bool  (*pCSTExternalMsgEvent)
(tCSTChannel* pChannel,
 tCSTExternalMsgEvent CSTExternalMsgEvent,
 int Data,
 int16 *pData)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`CSTExternalMsgEvent`	Event info (see 7.2.2.4)
`Data`	Depending on the event, it can be data byte or length of data bytes in `pData` buffer or nothing
`pData`	Depending on the event, it can be data buffer or nothing

Return Value

The false value means that the message event needs to be repeated again. The false value cancels a possible switch to another atomic command script caused by the corresponding CST Service message and sends a special repeat event USER_REPEAT_DATA to itself. This is also used in the AT Parser to output the Caller ID message information in parts.

Peripheral Background Periodic Function

The routine is called from the CST Service and should perform all hardware related background operations.

The default value after CST Service initialization is `DAAPeriphProcess()`, however, EVM board specific initialization overloads it to `EVMPeriphProcess()`.

Function

```
tCSTPeriphEvent  (*pPeriphProcess)
(tCSTChannel* pChannel,
 int AmountOf8KHzSamples)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`AmountOf8KhzSamples`	Time stamp in 8KHz samples that informs the time passed since last call

Return Value

A peripheral event (see 7.7.2.3)

Peripheral Driver Command Function

Attempts to perform a hardware-related operation.

The default value after CST Service initialization is `DAAPeriphDriver()`, however, EVM board specific initialization overloads it to `EVMPeriphDriver()`.

Function

```
long  (*pPeriphDriver)
(tCSTChannel* pChannel,
 tPeriphDriverCommand Command,
 int Param1,
 int Param2)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`Command`	Peripheral command (see 7.7.2.2)
`Param1`	First auxiliary parameter for the command
`Param2`	Second auxiliary parameter for the command

Return Value

Result of the command execution. Zero means that the command has not yet finished executing (the user has to send the command again to push the process). Nonzero result means that the execution has completed. For example, when the user sends `pdc_PULSE_GEN` command to dial a digit in pulse mode, the driver will return zero until the dialing of this digit is completed.

If the command is to read a DAA hardware register (`pdc_READ_REG`), the returned 32-bit integer value will contain the result of the execution in the high word and the read register value in the low word. If the high and low words are equal to zero, the register has not been read yet. Otherwise, the high word becomes non-zero and low word contains the register value.

CST Framework Reset Function

Reset the whole CST solution (called upon the ATZ command in the Chipset mode) – “soft restart”.

The default value is `CSTGlobalReset()`.

The user may want to overload this method in order to control exactly how CST restarts, for example, to keep pre-loaded patch code at software reset.

Function

```
void  (*pCSTGlobalReset)
(tCSTChannel* pChannel)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value

None

UART Byte Monitor Function

Inform a client (AT Parser) about a new byte obtained from UART.

In the Chipset mode, the default value is `UARTRxMonitor()`, which keeps a track of modem escape character sequences. In the Flex mode, the default value is `CSTAction_UARTRxMonitor()`, which does nothing.

Function

```
void (*pUARTRxMonitor)
(tCSTChannel* pChannel,
 char Data)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`Data`	UART byte

Return Value None

Note: This function informs AT Parser of the bytes that are received right away, before putting these bytes into a buffer, so that in case data is stuck in the buffer (when modems are in retrain mode, for example), escape sequence ("+++" with guard periods) could still be received and processed by AT Parser.

At the same time, the AT Parser aksks the UART driver to perform auto-baud detection if the AT Parser is in the Command Mode.

Processing CST Service Ingress Messages

Processes messages coming to the CST Service.

The default value is `CSTServiceProcessMessage()`

Function

```
void (*pProcessMessage)
(tCSTChannel* pChannel)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

Posting Low Priority Modem Thread

Called from CST Service to post a low priority modem thread function. Used in multi-threaded applications only.

The default value is `NULL` (undefined). In DSP/BIOS-oriented applications, it should be initialized with a real function address (see `CSTBIOS.c`)

Function

```
void (*pLowPriorityModem)
(tCSTChannel* pChannel)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

High Priority Voice Processsing Function

The default value is `VControllerHighPriorityProcess()`.

For more details on this function, read section 7.6.2.2.

Function

```
void (*pVControllerHighPriorityProcess)
(tCSTChannel* pChannel,
 int16 *pIn,
 int16 *pOut,
 int Count)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pIn`	Pointer to a buffer of valid input samples
`pOut`	Pointer to a buffer of valid output samples
`Count`	Amount of samples in the buffers

Return Value None

Vocoder Selection Function

The default value is `VControllerSelectVocoder()`

For more details on this function, read section 7.6.2.2.

Function

```
void (*pVControllerSelectVocoder)
(tCSTChannel* pChannel,
 int param)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`param`	Bit Per Sample

Return Value None

Low Priority Voice Processing Function

The default value is `VControllerProcess()`.

In DSP/BIOS-oriented applications, it can be reinitialized with another function (see `CSTBIOS.c`)

For more details on this function, read section 7.6.2.2.

Function

```
void (*pVControllerProcess)
(tCSTChannel* pChannel)
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

7.2.2.2 Main Control Fields of CST Commander

The CST Commander control interface consists of several important fields, which determine the state of the CST Framework.

`typedef struct tCSTCommanderGeneral {`

Table 7-16. CST Commander General Control Interface

Field Type	Field Name	Description
`tCSTAtomicCommand*`	`pCSTAtomicCommand`	Pointer to the current atomic command (see 7.2.3.1). Zero means that there is no command currently being executed.
`tCSTCommandMode`	`CommandMode`	The CST Commander mode (see 7.2.2.3 mostly used by the AT-command parser. The CST Commander has an atomic command to change this field.
`int`	`CLSMode`	The AT Parser's mode. The CST Commander never reads this field internally, but can reset it when `CommandMode` becomes `ccm_STANDARD_COMMAND`. Nevertheless `CLSMode` is included in the structure as an important CST Framework variable.

`} tCSTCommanderGeneral;`

Type tCSTCommanderGeneral is defined in CSTCommander.h.

7.2.2.3 Set of CST Commander Modes

One of the CST Commander's control fields, `CommandMode`, defines the global state of the CST Framework as a whole. This state is mostly used by the AT command parser. Possible command modes are listed in the following table.

Enum Definition `typedef enum tCSTCommandMode {`

Table 7-17. Set of CST Commander Modes

Value	Name	Description
0	ccm_STANDARD_ COMMAND	Standard command mode. CST Framework is neither pumping data nor connecting. AT Parser recognizes and accepts standard AT commands only
1	ccm_ONLINE_ COMMAND_MODE	CST Framework is in online command mode. Modem stays connected to the line, while the user has the possibility to enter AT commands
2	ccm_VOICE_MODE	CST Framework is in voice command mode. Voice AT commands are permitted
3	ccm_ANYKEY_BREAK_ MODEM	Modem connection establishment is in progress. Any incoming byte from UART causes break of connection and switches the framework back to standard command mode

Table 7-17. Set of CST Commander Modes (Continued)

Value	Name	Description
4	ccm_ANY-KEY_BREAK_VOICE	Voice connection in progress. Any incoming byte from UART causes break of connection and switches the framework back to standard command mode
5	ccm_MODEM_DATA	CST Framework is pumping modem data
6	ccm_VOICE_DATA	CST Framework is pumping voice data. Note that this mode disables auto turnoff upon busy detection. Thus, 'voice data' command state affects not only AT Parser behavior, but CST Commander as well.
7	ccm_PARAM_DATA	Special system mode for loading external flex application image (data) into CST chip

```
} tCSTCommandMode;
```

Type tCSTCommandMode is defined in CSTCommander.h.

CST command mode field can be viewed as a shared interface between the CST Commander layer and AT Parser (in Flex mode it is used too). It represents three main CST Framework modes: standard command, connecting and voice/modem data transfer.

7.2.2.4 CST Commander Extended Message Events

Upon message reception from the CST Service, the CST Commander processes this message itself and partially converts the information into another kind of messages to be passed to the CST Commander user (AT Parser in the Chipset mode). Some message events contain an additional data byte or data array. (see section 7.1.1.2)

Enum Definition

```
typedef enum tCSTExternalMsgEvent {
```

Table 7-18. Set of CST Commander External Message Events

Value	Name	Description
0	eme_NONE	No event
1	eme_PERIPH_DATA	Event detected by the peripheral driver. Attached data contains peripheral driver message (see 7.7.2.3)
2	eme_CID_DATA	Caller ID result code. Attached data contains Caller ID result code (see sections 7.6.2.3 and 9.4.2.4)
3	eme_DTMF_DATA	Recognized DTMF symbol. Attached data contains DTMF symbol

Table 7-18. Set of CST Commander External Message Events (Continued)

Value	Name	Description
4	eme_CPTD_DATA	Detected Call Progress Tone. Attached data contains CPTD tone.
5	eme_MODEM_ CONNECT	Informs that the modem just connected (no attached data)
6	eme_MODEM_ DISCONNECT	Informs that the modem just disconnected (no attached data)
7	eme_VOICE_ DISCONNECT	Informs that voice just disconnected (i.e. turned off in both directions, no attached data)
8	eme_VOICE_DATA	Voice data. Attached data is an array of voice data bytes (in each 16-bit word only lower 8 bits are used)
9	eme_PARAM_DATA	Special system message for loading external flex application image (data) into CST chip. It just echoes back loaded bytes
10	eme_PARAM_DATA_ TURN	Special system message informing the AT parser that loading of external flex application is over
11	eme_AUTOTURNOFF_ ALL	CST Commander message event being generated upon auto switch to the `aTurnOffAll` script (see 7.2.3.2). Attached data is a unique ID indicating the reason for turnoff request (belongs to `atk_CPTD_TIMEOUT`, `atk_BUSY`, `atk_MODEMDISCONNECT`, `atk_ALGCREATE_FAIL`). User can cancel (refuse) turning off by returning false (zero).
12	eme_MODEM_DATA	Modem data. Attached array is modem data bytes (in each 16-bit word only lower 8 bits are used). This event is used by CST Action layer only. Note that it is not very suitable way for intensive data reception in ARQ mode. (see section 7.6.1)
13	eme_TICK	Tick message (used by CST Action layer only). Attached data contains number of processed DAA codec samples since last eme_TICK.

```
} tCSTExternalMsgEvent;
```

Type tCSTExternalMsgEvent is defined in CSTCommander.h.

7.2.2.5 Brief Description of CST Commander Function Interface

Table 7-19. Brief Description of CST Commander Function

Name	Functionality
`CSTCommanderInit()`	CST Commander initialization
`CSTCommanderSoftReset()`	Reset some runtime varriables
`CSTCommander()`	The main CST Commander function

Initialization CST Commander initialization

Function `void CSTCommanderInit (tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

Reset Runtime Variables

Reset some runtime varriables to be ready to start a new script of atomic commands.

Function `void CSTCommanderSoftReset (tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

The Main CST Commander Function

The purpose of the function is to execute the current atomic command being pointed to by the `pCSTAtomicCommand` pointer (see 7.2.2.2). To be more precise, this function kind of “attempts” to execute an atomic command, until it succeeds, because there may be some other pending tasks/commands to be completed, or because execution of an atomic command takes time.

The function is inherited by the AT Parser.

Function

```
void CSTCommander (tCSTChannel* pChannel,int
AmountOf8KHzSamples);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`AmountOf8KHzSamples`	Time stamp in 8KHz samples that indicates the time passed since last call

Return Value None

7.2.3 Files CSTAtomic.c, CSTAtomic.h

7.2.3.1 CST Commander Atomic Commands

The CST Commander is always attempting to perform a control operation, an *atomic command*. After the current command has been executed, the CST Commander increments `pCSTAtomicCommand` pointer (see 7.2.2.2) to select the next atomic command in a script, which is a sequence of atomic commands. There is a couple of exceptions: the `cac_NONE` command stops the script execution and must be the last command in the script; the `cac_DIALING` command may temporarily interrupt execution of the script, where it has been encountered, and force the CST Commander to start execution of another script, which must not contain another `cac_DIALING` command. Some commands require an extra word or two as their parameters.

Enum Definition

```
typedef enum tCSTAtomicCommand {
```

Table 7-20. Set of CST Commander Atomic Commands

Value	Name	Description
0	cac_NONE	Marks the end of an atomic command script. If there is an interrupted script, it is resumed. Otherwise, if there’s no interrupted script, further atomic command execution is stopped by setting `pCSTAtomicCommand` to zero.
1	cac_TURNOFF_ALL	Sends a `cstst_TURNOFF_ALL` message to the CST Service that will immediately turn off all active algorithms (see the `cstst_TURNOFF_ALL` message). (see section 7.1.1.3)

Table 7-20. Set of CST Commander Atomic Commands (Continued)

Value	Name	Description
2	cac_PERIPH_SIMPLE_X	Executes a peripheral driver command (see 7.7.2.1), the command is passed as a parameter in the next word. The next atomic command will be processed only after successful execution of this peripheral command.
3	cac_PAUSE_X	Sustains a pause, whose duration is passed as a parameter in the next word. If the parameter is less than `csp_SPECIAL_PAUSE_AMOUNT`, it is treated as an index into the `aCSTSpecialPauses` array that contains real pause duration values and scales (seeTable 7-21). Otherwise, the parameter contains immediate pause value in milliseconds.
4	cac_SET_COMMAND_ MODE_X	Sets `CommandMode` to a new value (see 7.2.2.3), passed as a parameter in the next word.
5	cac_WAIT_CPTD_ APPEARANCE_XX	Waits for appearance of a Call Progress Tone signal, whose type is passed as a parameter in the next word. Detection timeout is passed in the second parameter, having the same format as `cac_PAUSE_X`. Waiting for a call progress tone should be done after CPTD object is created using the `cac_TURNON_SIMPLE_X` command.
6	cac_WAIT_CPTD_ DISAPPEARANCE_X	Waits for disappearance or absence of a call progress tone signal, whose type is passed as a parameter in the next word.
7	cac_TURNON_VOICE_ LOOP	Sends a `cstst_VOICE_LOOP` message to the CST Service to turn on a stand alone voice loop (enable echo canceller and Caller ID for 'on call waiting' mode) (see 7.1.1.3 and 7.1.1.5).
8	cac_TURNON_VOICE_ DATA_X	Sends a `cstst_VOICE_DATA` message to the CST Service to turn on the voice pumping in direction, defined by a parameter in the next word (the parameter is a value of `IsItTxTask` field/ (see 7.1.1.5)
9	cac_TURNOFF_VOICE_ DATA_X	Sends a `cstst_VOICE_DATA` message to the CST Service to turn off the voice pumping in direction, defined by a parameter in the next word (the parameter is a value of `IsItTxTask` field, (see 7.1.1.5).
10	cac_LOAD_PARAM_DATA_ LOOP	Special system command to start loading an external flex application image (data) into the CST chip
11	cac_LOAD_PARAM_DATA	Special system command for loading an external flex application image (data) into the CST chip
12	cac_DIALING	Dials a complete telephone number, defined by ASCII string in the `aDialNumber` field (see 7.2.1.2).

Table 7-20. Set of CST Commander Atomic Commands (Continued)

Value	Name	Description
13	cac_TURNON_MODEM	Sends a cstst_MODEM message to CST Service to turn on the modem (see 7.1.1.3 and 7.1.1.5)
14	cac_MODEM_CONNECT_WAIT	Waits for a modem connection. The next atomic command will be accepted only after the modem has established a connection.
15	cac_TURNON_CID_X	Sends a message to the CST Service to turn on the Caller ID in a mode, defined by a parameter in the next word. The parameter is of `tCIDStdSeq` type and selects a scenario of expected Caller ID.
16	cac_TURNON_SIMPLE_X	Sends a message to the CST Service to turn on an algorithm (see 7.1.1.3), defined by a parameter in the next word. No algorithm specific parameters are included in the message. Used to start running DTMF and CPTD detectors.
17	cac_SOFT_STOP_TASK	Correctly terminates the current task. If the current algorithm is a modem, it calls the `MODINT_disconnect` method (see section 7.6.1).
18	cac_OPERA-TIVE_WRITE_SREG_XX	Sets an S-register, whose number is specified as a parameter in the next word, to a value, specified by another parameter in the second word.
19	cac_BRANCH_IF_SREG_XX	Skips a number of words/commands in the script (the number is passed as a parameter in the next word) if the value of an S-register (specified by a parameter in the second word) is nonzero.
22	cac_RESET	Resets the whole CST solution (called upon the command ATZ) – “soft restart”. The pointer `pCSTGlobalReset` points to the function to be called (by default, it points to `CSTGlobalReset()`).
NOTE: All the following commands are meaningful only for the AT Parser. The CST Commander is not aware of them.		
20	cac_READ_S_REG	Reads an S-register
21	cac_WRITE_S_REG	Writes an S-register
23	cac_PRINT_RESPONSE	Prints a final response
24	cac_PRINTING_INTER-NAL_SREG_ALIAS_START	Starts printing an S-registers table (AT$)
25	cac_PRINTING_INTERNAL_SREG_ALIAS	Continues printing the S-registers table
26	cac_PRINTING_CMD_START	Starts printing the AT commands list (AT$H)

Table 7-20. Set of CST Commander Atomic Commands (Continued)

Value	Name	Description
27	cac_PRINTING_CMD	Continues printing the AT commands list
28	cac_PRINTING_SET-TINGS_START	Starts printing the current settings of the AT commands (AT&V)
29	cac_PRINTING_SETTINGS	Continues printing the current settings of the AT commands

```
} tCSTAtomicCommand;
```

Type tCSTAtomicCommand is defined in CSTAtomic.h.

The CST Commander supports 12 (`csp_SPECIAL_PAUSE_AMOUNT`) different special pauses, which are set in the `tSpecPauseDescr aCSTSpecialPauses[csp_SPECIAL_PAUSE_AMOUNT]` array. Each record of the array contains two fields: time duration and time scale in milliseconds.

Enum Definition

```
typedef enum tCSTSpecialPauses {
```

Table 7-21. CST Commander Special Pauses

Value	Name	Description
0	csp_LONG_DIAL_PAUSE	Pause for the comma ‘,’ character in a dial number, in seconds. The default value is 2 sec
1	csp_SHORT_DIAL_PAUSE	Pause for the slash ‘/’ character in a dial number, in milliseconds. The default value is 125 ms
2	csp_MODEM_START_PAUSE	Pause before the modem calls/answers and a voice call starts. The default value 1 sec
3	csp_FLASH_ONHOOK_PAUSE	Pause duration for the on hook state during the flash procedure. The default value is 300 ms
4	csp_FLASH_OFFHOOK_PAUSE	Pause duration for the off hook state during the flash procedure. The default value is 0 ms
5	csp_CPTD_DIALTONE_TIMEOUT	Timeout for dial tone detection. Timeout results in the abortion of the current process. The default value is 10 sec
6	csp_CPTD_RINGBACK_TIMEOUT	Timeout for ringback detection. Timeout results in the abortion of the current process. Default value is 60 sec

Table 7-21. CST Commander Special Pauses (Continued)

Value	Name	Description
7-11	cps_FREE_PAUSE_1 - cps_FREE_PAUSE_5	Reserved for the User
12	csp_SPECIAL_PAUSE_AMOUNT	Amount of special pauses in `aCSTSpecialPauses` array.

```
} tCSTSpecialPauses;
```

Type tCSTSpecialPauses is defined in CSTCommander.h.

7.2.3.2 Basic Predefined CST Commander Atomic Command Scripts

The CST Commander layer contains a big set of predefined atomic command scripts that allow the user to perform standard telephone operations easily. The relevant subset follows:

Table 7-22. Basic Predefined CST Commander Atomic Command Scripts

Name	Functionality
`aOffHook`	Go off hook, run the CPTD and DTMF detectors, don't run the Caller ID (to run the Caller ID, use special scripts, described below). Corresponds to the `ATH1` command.
`aCIDAfterRingEnd`	Run the Caller ID after a ring end
`aCIDAfterLineReversal`	Run the Caller ID after a line reversal
`aTurnOnModemCall`	Go off hook, wait for a dial tone, dial the number and run the modem in the originating (calling) mode. The semicolon ';' dial modifier disables running the modem after the number has been dialed. Corresponds to the `ATD` command.
`aTurnOnModemAns`	Go off hook and run the modem in the answer (called) mode. Corresponds to the `ATA` command.
`aTurnOnVoiceCall`	Go off hook, wait for a dial tone, dial the number, wait for a ring back signal appearance/disappearance (only if the '@' dial modifier found) and run the voice pump (echo canceller and Caller ID). Corresponds to the `ATD` command in the voice mode.
`aTurnOnVoiceAns`	Go off hook and run the voice pump (echo canceller and Caller ID). Corresponds to the `ATA` command in the voice mode.
`aTurnOnVoiceRxData`	Run the G.726/G.711 encoder and all signal detectors (CPTD, DTMF). Corresponds to the `AT#VRX` command.

Table 7-22. Basic Predefined CST Commander Atomic Command Scripts (Continued)

Name	Functionality
`aTurnOnVoiceTxData`	Run the G.726/G.711 decoder and all signal detectors (CPTD, DTMF). Corresponds to the `AT#VTX` command.
`aTurnOnVoiceRxTxData`	Run the G.726/G.711 encoder, decoder and all signal detectors (CPTD, DTMF). Corresponds to the `AT#VRXTX` command.
`aTurnOffVoiceData`	Turn off the G.726/G.711 encoder, decoder, do not turn off signal detectors (CPTD, DTMF). Corresponds to the <DLE><3> sequence in the AT-parser stream.
`aJustCall`	Go off hook, wait for a dial tone, dial the number. Corresponds to the `ATDxx;` command
`aSoftTurnOffAll`	Correctly stop the current task, then turn off all other algorithms and go on hook. Corresponds to the `ATH` command.
`aTurnOffAll`	Turn off all algorithms, go on hook. Corresponds to the `ATH` command. Also used for abort operation.
`aCSTServiceTurnOffAll`	Turn off all algorithms without going on hook.

The predefined atomic scripts are defined in CSTAtomic.c.

7.3 CST Action

7.3.1 Unified CST Action Message

A CST Action message combines several interface messages and commands into a single packet.

Structure `typedef struct tCSTAction {`

Table 7-23. Unified CST Action Message

Field Type	Field Name	Description
`tCSTActionType`	`ActionType`	CST Action type key (see 7.3.2)
`union Action{`		
`tCSTConfigCommand`	`CSTConfigCommand`	Corresponds to cat_CONFIG_COMMAND. To configure CST settings. The CST Action message of this type will be executed immediately (see 7.3.3.1
`tCSTStandard Operation`	`CSTStandard Operation`	Corresponds to cat_STANDARD_OPERATION. To run one standard (typical) operation which belongs to tCSTStandardOperationType (see 7.3.3.2)
`tCSTServiceMessage`	`CSTServiceMessage`	Corresponds to cat_CSTSERVICE_MESSAGE. To transfer CST Service message (see 7.1.1.2) directly via CSTSendServiceMessage() method (see 7.1.1.9)
`}`		

`} tCSTAction;`

Type tCSTAction is defined in CSTAction.h.

7.3.2 CST Action Message Type Key

The key type selects the actual type of the CST Action message content.

Enum Definition `typedef enum tCSTActionType {`

Table 7-24. CST Action Message Type Key

Value	Name	Description
0	`cat_SET_REGISTER`	Indicates that the rest of the Action message is described by the `CSTConfigCommand` structure (see 7.3.3.1).
1	`cat_GET_REGISTER`	Indicates that the rest of the Action message is described by the `CSTConfigCommand` structure (see 7.3.3.1).

Table 7-24. CST Action Message Type Key (Continued)

Value	Name	Description
2	`cat_STANDARD_OPERATION`	Indicates that the rest of the Action message is described by the `CSTStandardOperation` structure (see 7.3.3.2).
3	`cat_CSTSERVICE_MESSAGE`	Indicates that the rest of the Action message is described by the `CSTServiceMessage` structure (see 7.1.1.2).

```
} tCSTActionType;
```

Type tCSTActionType is defined in CSTAction.h.

7.3.3 CST Action Message Contents

7.3.3.1 Configuration Commands

Corresponds to `cat_SET_REGISTER` and `cat_GET_REGISTER` types, used to set and get CST settings through the S-registers. The CST Action message of this type will be executed immediately.

Structure `typedef struct tCSTConfigCommand {`

Table 7-25. The `tCSTConfigCommand` Structure

Field Type	Field Name	Description
`int`	`InternalSReg`	S-register number to be read/set
`int`	`Value`	Retrieved value/New value to be set

```
} tCSTConfigCommand;
```

Type tCSTConfigCommand is defined in CSTAction.h.

7.3.3.2 Standard Commands

Correspond to `cat_STANDARD_OPERATION`, used to run one standard (typical) operation of the type `tCSTStandardOperationType`. This action is intended to run or configure an atomic command script.

Structure `typedef struct tCSTStandardOperation {`

Table 7-26. The `tCSTStandardOperation` Structure

Field Type	Field Name	Description
`tCSTStandardOperationType`	`OperationType`	Select the operation type (see Table 7-27)
`uint8`	`aData[]`	Attached data depending on the standard operation

```
} tCSTStandardOperation;
```

Type tCSTStandardOperation is defined in CSTAction.h.

Enum Definition typedef tCSTStandardOperationType {

Table 7-27. Set of CST Action Standard Operations

Value	Name	Description
0	sot_OFF_HOOK	Corresponds to the `aOffHook` script (see 7.2.3.2 for this and all remaining values of Table 7-27)
1	sot_CID_AFTER_RINGEND	Corresponds to the `aCIDAfterRingEnd` script
2	sot_CID_AFTER_LINE_ REVERSAL	Corresponds to the `aCIDAfterLineReversal` script
3	sot_TURNON_MODEM_CALL_X	Corresponds to the `aTurnOnModemCall` script. The `aData` field should contain the dialing number
4	sot_TURNON_MODEM_ANS	Corresponds to the `aTurnOnModemAns` script
5	sot_TURNON_VOICE_CALL_X	Corresponds to the `aTurnOnVoiceCall` script. The `aData` field should contain the dialing number
6	sot_TURNON_VOICE_ANS	Corresponds to the `aTurnOnVoiceAns` script
7	sot_TURNON_VOICE_RXDATA	Corresponds to the `aTurnOnVoiceRxData` script
8	sot_TURNON_VOICE_TXDATA	Corresponds to the `aTurnOnVoiceTxData` script
9	sot_TURNON_VOICE_ RXTXDATA	Corresponds to the `aTurnOnVoiceRxTxData` script
10	sot_TURNOFF_VOICE_DATA	Corresponds to the `aTurnOffVoiceData` script
11	sot_JUST_CALL_X	Corresponds to the `aJustCall` script
11	sot_SOFT_TURNOFF_ALL	Corresponds to the `aSoftTurnOffAll` script
12	sot_TURNOFF_ALL	Corresponds to the `aTurnOffAll` script
13	sot_CSTSERVICE_TURN-OFF_ALL	Corresponds to the aCSTServiceTurnOffAll script
14	sot_CUSTOM_ATOM-IC_CHAIN_X	Custom atomic command script. The script is coded directly in the `aData` field
15	sot_SET_DIAL_STRING_X	Just set a new dialing number being given via `aData` field. This operation (if accepted) is executed immediately

```
} tCSTStandardOperationType;
```

Type tCSTStandardOperationType is defined in CSTAction.h.

7.3.4 Brief Description of CST Action Function Interface

Table 7-28. CST Action Function Interface

Name	Functionality
`CSTAction_Init()`	Standard CST initialization (does not inlude any hardware initialization)
`CSTAction()`	CST Action execution
`CSTAction_Process()`	The main high-priority thread function. Should be called periodically

7.3.4.1 CST Initialization

CST initialization (does not inlude hardware init). This function performs a required set of operations to correctly initialize the CST Service , CST Commander and CST Action layers. The full CST initialization can be done as follows:

```
TargetBoardInit
```

```
CSTAction_Init
```

```
TargetPeriphInit
```

Function

```
void CSTAction_Init
(tCSTChannel* pChannel,
 bool IsBIOSUsed,
 bool (*pUserCallback) (tCSTChannel* pChannel,
    tCSTExternalMsgEvent CSTExternalMsgEvent,
    int Data,
    int16 *pData));
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`IsBIOSUsed`	Single threaded/multi-threaded (DSP/BIOS) mode selection
`pUserCallback`	The user callback function to be called by the CST Framework. The function performs transfer of data and control information from the CST Commander (see description of the `pCSTExternalMsgEvent` function in section 7.2.2.1). The CST Action, however, may also transfer modem data from the CST Service through this function. The function should normally return true.

Return Value None

7.3.4.2 CST Action Execution

The main interface function of the Action layer is `CSTAction()`. Similarly to the CST Service message, the CST Action message can be accepted or rejected. The configuration command (`cat_CONFIG_COMMAND`) is to be accepted immediately. The standard operation (`cat_STANDARD_OPERATION`) is accepted, if there is no other pending operation. Acceptance of the standard operation usually starts a process, which will not allow accepting the next command or a direct CST Service message being passed, until the process is done.

The message `cat_CSTSERVICE_MESSAGE` means an attempt to send a message directly to the CST Service, bypassing the command layer. For example, it can be a data message. If the CST Service refused the message due to another message being processed at the moment, `cmr_TRY_AGAIN` result would be returned.

Function `tCSTMessageResult CSTAction(tCSTChannel* pChannel,tCSTAction *pAction);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pAction`	Pointer to a CST Action message (see 7.3.1)

Return Value Immediate result (see 7.1.1.7). In case of `cat_GET_REGISTER` key type, the result is S-register value.

7.3.4.3 Main High-priority Thread

The main high-priority thread routine (periodically called function). It is self-synchronized with respect to DAA interrupts. In fact, it just calls `CSTService-Process()` function (see 7.1.1.9). The routine should be periodically called at least once each 5 milliseconds, but it is strongly recommended to call it several times more often.

Function `inline void CSTAction_Process (tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

7.3.5 Using CST Action Interface, Practical Aspects

7.3.5.1 Standard Applications

This chapter illustrates use of the CST Action interface by a set of practical examples. Section 6.3.7 has illustrated implementation of an algorithm making a modem call, sending and receiving data and disconnecting. The implemented algorithm includes the following subtasks:

❑ **Originating a call:** dialing a number (say, 532) and running the modem This operation is similar to ATD532 command and implies going off hook, sustaining a pause, waiting for dial tone, dialing the number, detecting busy tones, turning on the modem and waiting for connection establishment. The whole process can be run as follows:

- **Starting the process**
  The following code should a be part of the function `MyPeriodicThread()`. It sets all necessary Action fields, sends the Action message and checks whether the message has been accepted. If the message was rejected, the operation should be repeated again.

```
tCSTAction Action;

Action.ActionType = cat_STANDARD_OPERATION;
Action.Action.CSTStandardOperation.OperationType  =  sot_TUR-
NON_MODEM_CALL_X;
Action.Action.CSTStandardOperation.aData[0] = ’5’;
Action.Action.CSTStandardOperation.aData[1] = ’3’;
Action.Action.CSTStandardOperation.aData[2] = ’2’;
Action.Action.CSTStandardOperation.aData[3] = 0;

if ( CSTAction (&Ch0,&Action) == cmr_TRY_AGAIN )
  //Repeat again!
else
  //Process has started. Wait for connect.
```

- **Waiting for the connection**
  The following code should be a part of the function `MyCallBack()`(`CSTExternalMsgEvent` is a parameter of the function). The function analyzes the message from the CST Service and sees if the message is a connect or disconnect/termination event. Note that this code should remain active even after the connection establishment.

```
if ( CSTExternalMsgEvent == eme_MODEM_CONNECT )
  //Modem successfully connected!

if ( (CSTExternalMsgEvent == eme_AUTOTURNOFF_ALL) ||
     (CSTExternalMsgEvent == eme_MODEM_DISCONNECT) )
  //Connection failed. All operations aborted
```

- **Preparing a static CST Action message for data sending**
  The following code should be a part of the function `MyPeriodic-Thread()`. It initializes a data message to be filled by transmitted data.

```
static tCSTAction DataAction;

DataAction.ActionType = cat_CSTSERVICE_MESSAGE;
DataAction.Action.CSTServiceMessage.Task = cstst_MODEM;
DataAction.Action.CSTServiceMessage.IsItTxTask = 1;
DataAction.Action.CSTServiceMessage.SubEvent = cse_DATA;
DataAction.Action.CSTServiceMessage.DataLength = 0; //empty
//Now we are ready to send and receive data
```

❑ **Sending and receiving data**
Now that the connection has been established and we are ready to send and receive data. Here it is assumed that the user has data to be sent and expects data to be received. Just in this particular example, data will be sent byte by byte.

- **Sending a data byte**
  The following code should be a part of the function `MyPeriodic-Thread()`. It fills the data message and tries to transfer current content into CST. If the message was accepted, all contained data has been copied into CST local memory and is being sent. This means that the `DataAction` variable can be filled again by next data.

```
tCSTMessageResult ActionResult;

if ( DataAction.Action.CSTServiceMessage.DataLength < CST_MAXDA-
TALENGTH )
  DataAction.Action.CSTServiceMessage.aData[
    DataAction.Action.CSTServiceMessage.DataLength++] = <next
byte to be sent>;

ActionResult = CSTAction (&Ch0,&DataAction);

if ( (ActionResult == cmr_RESULTOK) || (ActionResult == cmr_EX-
ECUTING) )
  DataAction.Action.CSTServiceMessage.DataLength = 0;
  //Send next data byte
```

- **Receiving data bytes**

  The following code should be a part of the function `MyCallBack()` (`CSTExternalMsgEvent`, `pData` and `Data` are parameters of the function). The CST Action interface forces the user to take all received data. To be able to take the received data partially, use a direct callback from the modem integrator (see 7.6.1).

  ```
  int DataCount = Data;

  if ( CSTExternalMsgEvent == eme_MODEM_DATA )
    while ( DataCount-- )
      <pointer to a static buffer>++ = *pData++;
  ```

❑ **Disconnecting**

Having sent and received all data, the modem should terminate the connection. This operation is similar to the ATH command and it implies a soft modem disconnection and going on hook. The whole process can be executed as follows:

- **Accurate disconnection**

  The following code should be a part of the function `MyPeriodicThread()`. It sets all necessary Action fields, sends the Action message and checks whether the message has been accepted.

```
tCSTAction Action;

Action.ActionType = cat_STANDARD_OPERATION;
Action.Action.CSTStandardOperation.OperationType = sot_SOFT_TURN-
OFF_ALL;

if ( CSTAction (&Ch0,&Action) == cmr_TRY_AGAIN )
  //Repeat again!
else
  //Go to initial state and expect eme_MODEM_DISCONNECT or
  //eme_AUTOTURNOFF_ALL events
```

- **Going to the initial state**

  The modem has started disconnecting. However, the process may take a while depending on V.42 activity and the value of the `srd_TIME_BEFORE_FORCED_HANGUP` S-register. So, there may be a delay between disconnection initiation and actually going to the initial state.

It is important to note again, that there may be an additional callback mechanism (direct call from the data modem controller, see 7.6.1) for intensive modem data transfers in ARQ mode, because the CST Action interface does not allow the user to reject the received data. If the user is unable to take the data, the data will be lost.

The following example illustrates a simple voice mail:

- **Initial state**
  It is a default on-hook state. Upon detection of a number of consecutive rings, the voice mail should be activated.

  - **Waiting for a ring**
    The following code should be a part of the function `MyCallBack()` (`CSTExternalMsgEvent` and `Data` are parameters of the function). It analyzes the message from the CST Service and sees if it's a peripheral event message for ring (or, for example, ring end) event.

```
if ( CSTExternalMsgEvent == eme_PERIPH_DATA ) && ( Data ==
cpe_RING )
   //If nobody takes the handset for a while (e.g. for three
rings)
  //go off hook and run voice mail
```

  - **Going off hook**
    The following code should be a part of the function `MyPeriodic-Thread()`. It sets all necessary Action fields, sends the Action message and checks whether the message was accepted.

```
tCSTAction Action;

Action.ActionType = cat_STANDARD_OPERATION;
Action.Action.CSTStandardOperation.OperationType            =
ms_GO_OFF_HOOK;

if ( CSTAction (&Ch0,&Action) == cmr_TRY_AGAIN )
  //Repeat again!
else
  //Run voice mail
```

- **Running voice mail (playing the greeting)**
  Now we are going to play a greeting stored in external memory.

  - **Selecting a coder**[4]
    The following code should be a part of the function `MyPeriodicThread()`. Prior to activation of the voice path, let's select the G.726 waveform coder at 16K rate (economical bit rate). The resulting bit per sample rate is equal to 2.

[4] All CST algorithms and integration services can be configured before initialization. Most important parameters are channel-dependent, in other words each channel (in case of multichannel application and several instances of CST Framework) contains its own set of settings mapped to S-registers. All other parameters are stored in global channel-independent structures, in other words they are applicable for all channels, and can be modified directly.

```
tCSTAction Action;

Action.ActionType = cat_SET_REGISTER;
Action.Action.CSTConfigCommand.InternalSReg = srd_VOICE_BPS;
Action.Action.CSTConfigCommand.Value = 2;
CSTAction (&Ch0,&Action);
//Ok, let's run playback
```

- **Running the G.726 decoder and related services**
  The following code should be part of `MyPeriodicThread()` function. It sets all necessary Action fields, sends the Action message and checks whether the message was accepted. If the message was rejected, the operation should be repeated next time.

```
tCSTAction Action;

Action.ActionType = cat_STANDARD_OPERATION;
Action.Action.CSTStandardOperation.OperationType   =   sot_TUR-
NON_VOICE_TXDATA;

if ( CSTAction (&Ch0,&Action) == cmr_TRY_AGAIN )
  //Repeat again!
else
   //Voice  tx.  path  activated.  Prepare  a  static  message  for
data sending
```

- **Preparing a static CST Action message for data sending**
  The following code is a part of the function `MyPeriodicThread()`. It initializes a data message to be filled by data to be sent.

```
static tCSTAction DataAction;

DataAction.ActionType = cat_CSTSERVICE_MESSAGE;
DataAction.Action.CSTServiceMessage.Task = cstst_VOICE_DATA
DataAction.Action.CSTServiceMessage.IsItTxTask = 1;
DataAction.Action.CSTServiceMessage.SubEvent = cse_DATA;
DataAction.Action.CSTServiceMessage.DataLength = 0; //empty
//Now we are ready to send voice data
```

- **Playing the greeting**
  Now we are ready to send voice data. Just in this particular example, data will be sent sent byte by byte.

- **Sending a data byte**
  The following code should be a part of the function `MyPeriodic-Thread()`. It fills the data message and tries to transfer the current contents into CST. If the message is accepted, all contained data will be copied into CST local memory and it will be sent. This means that the `DataAction` variable can be filled again by the next portion of data.

```
tCSTMessageResult ActionResult;

if   (   (DataAction.Action.CSTServiceMessage.DataLength   <
CST_MAXDATALENGTH) )
  DataAction.Action.CSTServiceMessage.aData[
     DataAction.Action.CSTServiceMessage.DataLength++] = <next
byte to be sent>;

ActionResult = CSTAction (&Ch0,&DataAction);

if  (  (ActionResult  ==  cmr_RESULTOK)  ||  (ActionResult  ==
cmr_EXECUTING) )
  DataAction.Action.CSTServiceMessage.DataLength = 0;
  //Send next data byte
  //If greeting has been played, stop playing
```

- **Auto hang up control**
  The following code should be a part of the function `MyCallBack()` (`CSTExternalMsgEvent` is a parameter of the function). It analyzes the message from the CST Service by looking for an auto turnoff event (auto hang up). It will happen if the abonent hangs up. Note that this code should be active during both greeting playback and message recording.

```
if ( CSTExternalMsgEvent == eme_AUTOTURNOFF_ALL )
  //Connect fail. All operations aborted
```

- **Stopping the playback**
  The following code should be a part of the function `MyPeriodic-Thread()`. It sets all necessary Action fields, sends the Action message and checks whether the message was accepted. If the message is rejected, the operation should be repeated again.

```
tCSTAction Action;

Action.ActionType = cat_STANDARD_OPERATION;
Action.Action.CSTStandardOperation.OperationType  =  sot_TURN-
OFF_VOICE_DATA;

if ( CSTAction (&Ch0,&Action) == cmr_TRY_AGAIN )
  //Repeat it again!
else
  //Let's start recording
```

- ❑ **Recording a voice message**
  Now we are going to record the incoming voice. Recording can be stopped manually or automatically upon busy tone detection.

  - ■ **Running the G.726 encoder and related services**
    The following code should be a part of the function `MyPeriodicThread()`. It sets all necessary Action fields, sends the Action message and checks whether the message was accepted. If the message is rejected, the operation should be repeated again.

```
tCSTAction Action;

Action.ActionType = cat_STANDARD_OPERATION;
Action.Action.CSTStandardOperation.OperationType  =  sot_TUR-
NON_VOICE_RXDATA;

if ( CSTAction (&Ch0,&Action) == cmr_TRY_AGAIN )
  //Repeat it again!
else
  //Voice rx path activated. Let's catch voice data!
```

  - ■ **Receiving data bytes**
    The following code should be a part of the function `MyCallBack()` (`CSTExternalMsgEvent`, `pData` and `data` are parameters of the function). The user should store all passed data. Unlike the modem, voice throughput is known and data is passed in chunks of even size.

```
int DataCount = Data;

if ( CSTExternalMsgEvent == eme_VOICE_DATA )
  while ( DataCount-- )
    <pointer to a static buffer>++ = *pData++;
//If max allowed time elapsed, stop message record
```

- **Stopping recording and going on hook**
  The following code should be a part of the function `MyPeriodic-Thread()`. It sets all necessary Action fields, sends the Action message and checks whether the message was accepted. If the message is rejected, the operation should be repeated again.

```
tCSTAction Action;

Action.ActionType = cat_STANDARD_OPERATION;
Action.Action.CSTStandardOperation.OperationType  =  sot_TURN-
OFF_ALL;

if ( CSTAction (&Ch0,&Action) == cmr_TRY_AGAIN )
  //Repeat it again!
else
  //All operations will be stopped
```

- **Going to the initial state**
  The process will stop shortly and the voice mail device will go on hook.

7.3.5.2 Non-Standard Applications

In a non-standard application the user may need to add his/her own algorithms, change the behavior of certain CST Services and so on. As it has been mentioned in section 6.3.8 and section 7.2.2.1 , CST provides a set of dynamic functions to be overridden by the user if necessary. Some typical reasons for the dynamic function overloading are listed below:

Adding New Algorithms

The user can add his/her own algorithm and run it together with standard CST algorithms. In most cases it is enough to overload the function `CSTAction_UserOperation()` that allows the user to process I/O samples from the DAA.

```
void MyUserOperation(tCSTChannel* pChannel, int16* pIn,int16
*pOut,int AmountOf8KHzSamples)
{
  CSTAction_UserOperation (pChannel,pIn,pOut,AmountOf8KHzSamples);
  ...
  <My algorithm I/O processing> (pIn,pOut,AmountOf8KHzSamples);
}
```

In `MyInitialization()` the user should include the following code:

```
CSTFxns.pCSTUserOperation = MyUserOperation;
```

Running Several Audio Channels

The CST framework is multichannel-ready, though implementation of a multichannel application involves some adjustments in the CST framework. Since TMS320C54CST contains only one on-chip DAA, a number of external codecs may be connected to McBSP0 and McBSP2 or to the other peripherals (HPI, GPIO etc). There can be various input/output techniques. To connect the user's I/O stream to the CST framework, the following steps should be taken:

- **Reload hardware drivers**
  If you want to use the existing DAA and add to this additional DAAs (or codecs) or want to use other than the Silicon Lab's DAA peripherals, you will need to reload the DAA driver and the peripheral driver. The same applies to the UART driver. It will need to be reloaded if there will be additional UARTs used or some other peripherals will replace the UART. For more information on CST drivers and reloading them read section 7.7, specifically subsection 7.7.7.

- **Create and initialize additional CST channels**
  To create an additional CST channel structure, define a structure of the type `tCSTChannel` or allocate memory dynamically by calling the memory manager allocation function as follows:

  ```
  pCh1     =     (tCSTChannel*)     CSTMemManager.allocate(CST_DE-
  FAULT_MEM_SPACE, sizeof(tCSTChannel), 1);
  ```

  To initialize the additional CST channel, use the following function `CSTAction_Init_ForExtraChannel()`. Note that this initialization should be done after initialization of the channel 0 (`Ch0`).

  ```
  void CSTAction_Init_ForExtraChannel (tCSTChannel* pChannel)
  {
     CSTAction_InitForSingleThread (pChannel, 0, CSTFxns.pCSTExter-
  nalMsgEvent);
      pChannel->CSTService.IsModemPreemption  =  Ch0.CSTService.IsMo-
  demPreemption;
  }
  ```

- **Overload dynamic and overloadable functions if need be**
  create your own `CSTServiceProcess()` and `CSTServiceProcessBuffer()` functions. A new function, combined of these two functions, may look like this:

  ```
  void MyCSTServiceProcess (tCSTChannel* pChannel)
  ```

```
{
  int AmountOf8KHzSamples;
  // Process only if enough samples accumulated
  AmountOf8KHzSamples = DAAAvail (pChannel->DAAChanHandle);
  while ( AmountOf8KHzSamples>=INPUT_OUTPUT_LENGTH )
  {
    AmountOf8KHzSamples-=INPUT_OUTPUT_LENGTH;
    {
      tCSTService* pCSTService = &pChannel->CSTService;
      tCSTPeriphEvent PeriphEvent;

      // Get the new portion of analog input samples
      // and put the previous portion of output samples
      memcpy (aInput, pCSTService->aOutput, INPUT_OUTPUT_LENGTH);
        DAAReadWrite (pChannel->DAAChanHandle, aInput, INPUT_OUT-
PUT_LENGTH);

          CSTServiceProcessIOandVoice  (pChannel,  aInput,pCSTSer-
vice->aOutput);
          CSTServiceProcessCommonAlgos  (pChannel,  aInput,pCSTSer-
vice->aOutput);
            CSTFxns.pCSTUserOperation  (pChannel,  aInput,pCSTSer-
vice->aOutput,INPUT_OUTPUT_LENGTH);
         PeriphEvent=CSTFxns.pPeriphProcess (pChannel, INPUT_OUT-
PUT_LENGTH);
      if ( PeriphEvent!=cpe_NONE )
          CSTServicePriv_PutDataIntoMessage (pChannel, cstst_PER-
IPH,PeriphEvent);
      CSTFxns.pProcessMessage(pChannel);
    }
  }

  if ( pChannel->CSTService.CSTServiceStatus.IsProcessMsgNeeded )
  {
    CSTFxns.pProcessMessage (pChannel);
    pChannel->CSTService.CSTServiceStatus.IsProcessMsgNeeded=0;
  }
}
```

The above is mostly a copy of the original CST Service functions `CSTSer-viceProcess()` and `CSTServiceProcessBuffer()`. If you need to modify voice and other CST Service algorithm processing, you may do this here, as this is the place, where the CST Service processing functions are

called.
Note that if you want to use a custom CSTServiceProcess(), like the above, the original functions `CSTAction_Process()` and `CSTServiceProcess()`must not be called anymore in the application.

- **BIOS**
  Modify the functions `DMControllerLowPriorityModemSWI()` and `VCtrlLowPriorityProcessSWI()` (see `BIOS\CSTBIOS.c` and `BIOS\CSTBIOS.h`). The functions must be made multichannel.

Running Several Modem Channels

The technique is the same as for voice channels. However, V.32bis usage limits the maximum local loop delay (a delay, in which transmitted sample appears as near echo at the input of a modem). Therefore, the user should make the local loop delay as short as possible. Finally, the user should measure the local loop delay and update the corresponding parameter in the modem initialization structure (see section 7.6.1). If V.42bis is used and the modem is running in single thread (which is not recommended), the user should overload the function called by the `DMControllerSubfxns.pIsRealtimeShortage` pointer. (See section 7.6.1)

An example Flex application that works with two modems is described in section 7.7.7.5.

7.4 CST AT Parser

The AT Commands parser is partially supplied in open source code in C language. It is located in the folder `CST\Src\Framework` in the files `ATParser.c, ATExtended.c, CSTSReg.c` (see 7.1.1).

The AT parser can be considered as an example of controlling the CST Framework through the CST Commander, although it usually should not be used in standard Flex applications. The AT parser consists of the AT command line parser, small UART controller and an integrator of these parts with the CST Commander layer, CST service layer and peripherals.

The AT parser is an extendable service that allows the user to add his own AT commands and S-registers, and modify the application's behavior, but such operations are not described and it is suggested not to reconfigure the AT paser interior without a real need.

It has been mentioned that the AT parser is not used together with CST Action layer. Therefore using AT commands is uncommon for typical Flex applications.

7.4.1 AT Command Line Parser

The AT parser keeps AT command strings in a list of arrays of dedicated descriptors of the type `tATParameter`.

Structure `typedef struct tATParameter {`

Table 7-29. AT Command Descriptor

Field Type	Field Name	Description
`int`	`ast`	A unique ID of the type tATStringType combined with flags defined in ast_XXX constants, which defines the type of the AT command and its properties. This ID is used as reference in additional tables, which describe when the AT command is applicable and associated with it CST Commander atomic command script (see 7.2.3.1).
`char*`	`string`	AT command sub-string (the "AT" prefix is omitted)
`int*`	`pParameter`	Pointer to an integer parameter that can be set (or read) by the command. NULL means absence of integer variable. If the ast field includes the bit-attribute ast_LIST set (e.g. (ast & ast_LIST) != 0), then the address is treated as the address of an array of integer parameters. Amount of the parameters in the array is stored in the array's very first element. The union d defines the allowed values of the parameter(s).
`union d {}`:		

Table 7-29. AT Command Descriptor (Continued)

Field Type	Field Name	Description
`int`	`limits`	Defines the boundary values of the integer parameter. The upper 8 bits define the maximum value; the lower 8 bits define the minimum value. If the value of the upper bits is the same as the value of the lower bits, it is the default value
`int*`	`range`	Defines the values range of the integer parameter. This pointer points to an array containing a number of subranges. For example, the array {3, 2,5, 8,8, 10,12} of integers would define a range, which is a union of 3 subranges: 2…5, 8…8 and 10…12. E.g. the resulting range would include the values: 2, 3, 4, 5, 8, 10, 11 and 12. Note, the field range is used only if (ast & ast_RANGE) != 0. Otherwise the field limits is used.

```
} tATParameter;
```

Type tATParameter is defined in ATParser.h.

The user can add his/her own arrays of the AT command descriptors, but cannot modify the existing ones. Thus, the AT parser is not fully extendable. All extended arrays of AT command descriptors are treated as always applicable regardless of `CommandMode` (see 7.2.2.3). Besides, the AT parser does not support any mechanism for external linkage of reference tables and cannot automatically run a user's process associated with an AT command. Thus, adding a new AT command is not a trivial operation in general, however, it is easy to add AT commands that just set or read some variables.

The AT parser is active and can consume ASCII characters when the function `IsUARTtoATParserReady()` returns a nonzero result. It means that an ASCII character can be passed to the function `ATAddCharToCmdLine()` to be added to the end of the current line. When the end of line character / carriage return character is entered, line parsing and command processing start.

7.4.2 AT Command Execution

Each AT command can start execution of only one atomic command script. The script is selected by the `ast` field of the AT command descriptor (see 7.4.1).

The AT parser extends the main CST commander function (see 7.2.2.5) in order to change execution of some existing atomic commands and to add processing of a number of new ones not supported by the CST commander layer. In the code it is implemented in the function `AT_CSTCommander()`, which inherits the CST commander main function. The function `AT_CSTCommand-`

`er()` is called from the function `CSTUserOperation()`, which, in turn, is called through the pointer `CSTFxns.pCSTUserOperation`. To print execution result token, the `cac_NONE` termination atomic command is used (see 7.2.3.1), whose processing is different when using the AT parser.

7.4.3 Brief Description of AT Command Line Parser Interface

Table 7-30. Some of the AT Parser Interface Functions

Name	Functionality
`CSTATParserInit()`	AT parser initialization
`CSTATParserAdd()`	Add an array of new AT command descriptors
`AT_CSTCommander()`	Inherited `CSTCommander()` method. It redefines execution of some atomic commands and adds several new ones (see 7.2.2.5).

7.4.3.1 AT Parser Initialization

Initializes the AT parser.

Function

```
void CSTATParserInit (tCSTChannel* pChannel);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

7.4.3.2 Adding New AT Commands

Adds new AT commands. The examples of this procedure can be found in the AT parser's source code files, `ATParser.c` and `ATExtended.c`. The main function for the CST Chipset mode (see the file `ROM\main.c`) adds extended AT commands to the standard set.

Function

```
void CSTATParserAdd
(tCSTChannel* pChannel, const tSimpleMap*
pExtendedATCommands);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pExtendedATCommands`	A pointer to a handler, initialized with an address of the new AT command table being added, total number of commands in the table, and size of a single command (see Table 7-13).

Return Value None

7.5 Memory Management

7.5.1 Overview

The CST Framework provides a memory manager that can be used by both the framework itself and the user. The memory manager functions are accessible through pointers, which makes it possible to reload the original memory manager functions by the DSP/BIOS memory manager functions or other user-defined memory management functions.

The memory management subsystem in the CST Framework has the following features:

❑ The memory management functions can be reloaded as they're accessed in the framework indirectly through a set of dedicated pointers. This makes it possible to use the DSP/BIOS memory management functions or user-defined memory manager functions.

❑ The available heap memory size can be estimated.

❑ The user may change the default heap size and location.

❑ The user may define several separate memory segments/heaps mapped to physically different RAMs (e.g. DSP internal DARAM and external data RAM).

Creating and deleting XDAIS algorithms, which is done by the functions `ALGRF_create()` and `ALGRF_delete()`, involves calling the memory manager functions through the pointers (see also 7.5.4).

The C standard functions `malloc()`, `calloc()`, `free()` and `realloc()` are redefined in the CST memory manager so that they also call the actual memory manager functions through the pointers.

Thus, all memory allocation in CST is carried out by the memory manager, whose functions are accessible through the dedicated pointers.

7.5.2 Memory Manager Function Interface

The memory management functions are accessible through the pointers in the structure `tCSTMemManager CSTMemManager` (see `malloc.h`).

```
typedef struct {
```

Table 7-31. Memory Manager Function Interface Types

Type	Name	Description
`void* (*)(IALG_MemSpace, size_t, size_t)`	allocate	Virtual function to allocate memory
`void (*)(IALG_MemSpace, void*, size_t)`	free	Virtual function to free memory
`size_t (*)()`	getAvailableMemory	Virtual function to estimate free memory size

```
} tCSTMemManager;
```

7.5.2.1 Memory Allocation

This memory allocation function returns a pointer to an aligned memory block (according to the `alignment` parameter, its size and memory space).

Function

```
void*   (*allocate) (IALG_MemSpace space,
                     size_t alignment,
                     size_t size);
```

Parameter(s)

`space`	Memory space, where allocated block should be placed. Available memory spaces types are defined in file `CST\Framework\XDAS\ialg.h`. The default CST memory space is `CST_DEFAULT_MEM_SPACE`, which is equal to `IALG_DARAM0`. This corresponds to the DSP internal DARAM (see 8.3).
`alignment`	Memory block alignment in MAUs.
`size`	Size of allocated block is in MAUs.

Return Value Address of allocated memory block or `NULL` on failure.

7.5.2.2 Memory Deallocation

This function is invoked to free a memory block that has been previously allocated by the `allocate()` function, defined in .

Function

```
void  (*free) (IALG_MemSpace space,
              void *memory,
              size_t size);
```

Parameter(s)

`space`	Memory space, where allocated block should be placed. Available memory spaces types are defined in file `CST\Framework\XDAS\ialg.h`. The default CST memory space is `CST_DEFAULT_MEM_SPACE`, which is equal to `IALG_DARAM0`. This corresponds to the DSP internal DARAM (see 8.2).
`memory`	The address of a previously allocated memory block.
`size`	Size of the allocated block is in MAUs. The value of this parameter is not required by default. The parameter is used mainly to make the function interface close to that of DSP/BIOS' `MEM_free()`.

Return Value None

7.5.2.3 Free Memory Size Estimation

This function is used to get an estimate of the free memory.

Function `size_t  (*getAvailableMemory)();`

Parameter(s) None

Return Value Returns an estimate of free heap memory in MAUs. The returned value is a sum of the free sizes of all heaps (there may be several heaps, up to 6). Note that in multithreaded applications this value can't be relied on. That's because the current thread may be preemted by another thread, which, in turn, may change the memory allocation state.

7.5.3 Possible Memory Configurations

There are 3 basic memory configurations supported by the CST solution. In any of these configurations, the user has to make sure that the size of memory available for CST algorithms satisfies requirements listed in Table 8-4. Otherwise, some functionality may become unavailable.

Table 7-32. Basic Memory Configurations

Configuration	Initialization	Usage Restrictions
CST Memory Manager Used by All		

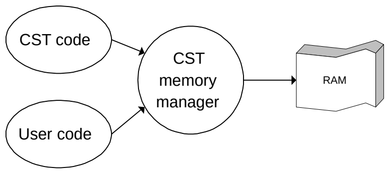
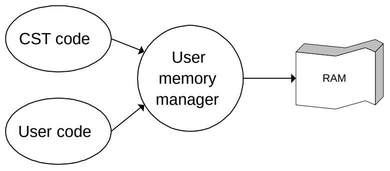
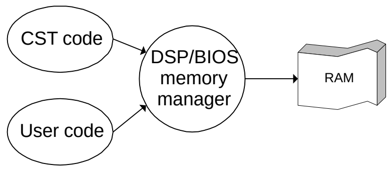
	❑ Preset heap table with available memory segments, if needed
	❑ Init the CST memory manager `CSTMemManagerInit()`
	See 7.5.3.1
		❑ ROMed functions:
		`malloc()`
		`calloc()`
		`realloc()`
		`free()`
		`ALGRF_create()`
		`ALGRF_delete()`

Table 7-32. Basic Memory Configurations (Continued)

User Memory Manager Used by All		

	❑ Perform initialization of user-defined memory manager
	❑ Redefine dynamic methods in `CSTMemManager` to redirect allocation requests
	❑ Disable CST Memory Manager
	
	See 7.5.3.2
		❑ ROMed functions:
		`malloc()`
		`calloc()`
		`realloc()`
		`free()`
		`ALGRF_create()`
		`ALGRF_delete()`
		❑ Any proprietary user functions

DSP/BIOS Memory Manager Used by All		

	❑ Preset heap table with available memory segments, if needed
	❑ Redefine dynamic methods in `CSTMemManager` to redirect allocation requests
	❑ Disable CST Memory Manager
	
	See 7.5.3.3
		ROMed functions:
		`malloc()`
		`calloc()`
		`realloc()`
		`free()`
		`ALGRF_create()`
		`ALGRF_delete()`
		`MEM_alloc()`
		`MEM_free()`
		`MEM_stat()`

7.5.3.1 Using CST Memory Manager

The CST memory manager is used by CST in the Chipset mode. This same memory manager can still be used in the flex mode for both CST Framework and a user’s Flex application. The CST memory manager is normally used in a non-BIOS environment. In the DSP/BIOS environment, the DSP/BIOS memory manager should be used (see 7.5.3.3)

In a standard Flex application, which uses the CST Action layer, the heap location and size is set up automatically. The function `CSTAction_Init()` (see `CSTAction.h`) initializes the very first entry in the array `tCSTMemSpace CSTMemSpace[6]` (see `malloc.h`, Table 7-33) with the base address of the heap and its size. The memory manager uses this array to find the memory that is used for the heap. The address and size of the standard heap become available at link time. A linker command file should specify the location and size of the heap. See the example project and linker command files for standard single-threaded Flex applications (more info on this can be found in 5.4.1). If your application needs different location and/or size of the heap, the linker command file may be adjusted appropriately.

Creating Second Heap

If your application needs to use two different heaps, for example, one heap is located in the DSP internal DARAM, while the other is located in the external data RAM connected to the DSP, you must initialize the next entry in the array `tCSTMemSpace CSTMemSpace[6]` manually. That is, the function `CSTAction_Init()` will initialize `CSTMemSpace[0]` for the first heap (whose address and size would be specified in the linker command file). The user will initialize `CSTMemSpace[1]` for the second heap. This initialization should be done before the call to the function `CSTAction_Init()` because this function calls the CST memory manager initialization function, `CSTMemManager-Init()`.

```
typedef struct {
```

Table 7-33. CST Memory Space Segment Structure

Type	Name	Description
`void*`	`base`	Base address of a contiguous memory segment
`size_t`	`size`	Size of the memory segment in MAUs
`IALG_MemSpace`	`space`	Space descriptor (see XDAIS core file `CST\Framework\XDAS\ialg.h` for options). The default CST memory space is `CST_DEFAULT_MEM_SPACE`, which is equal to `IALG_DARAM0`. This corresponds to the DSP internal DARAM (see section 8.2). The external memory may be denoted as `IALG_EXTERNAL`.

```
} tCSTMemSpace;
```

Type tCSTMemSpace is defined in malloc.h.

Hence, a portion of the Flex application initialization code will look something like this instead of a single call to `CSTAction_Init()`:

```
CSTMemSpace[1].base = (void*) 0xA000; // external data
memory starts at 0xA000

CSTMemSpace[1].size = 0x2000; // external data memory size
is 0x2000

CSTMemSpace[1].space = IALG_EXTERNAL; // denotes external
data memory

CSTAction_Init (&Ch0, 0, MyCallback);
```

7.5.3.2 Using User-Defined Memory Manager

To use a user-defined instead of the CST memory manager, just a few simple steps should be taken:

- ❑ Initialization of the user-defined memory manager
- ❑ Reloading of the pointers to the memory manager functions (see 7.5.2)
- ❑ Clearing of the array CSTMemSpace[6] to prevent the CST memory manager from any heap initializations

Therefore, a portion of the Flex application initialization code will look something like this instead of a single call to `CSTAction_Init()`:

```
MyMemManagerInit(...); // Init the user-defined memory
manager
CSTMemManager.allocate = MyMemManagerAllocate; // reload
functions
CSTMemManager.free = MyMemManagerFree;
CSTMemManager.getAvailableMemory = MyMemManagerGetAvaila-
bleMemory;
memset (&CSTMemSpace[0], 0, sizeof(CSTMemSpace)); // dis-
able CST memory manager
CSTAction_Init (&Ch0, 0, MyCallback);
```

7.5.3.3 Using DSP/BIOS Memory Manager

To use the DSP/BIOS memory manager with CST, the user should take the following steps:

- ❑ Reload the pointers to the memory manager functions
- ❑ Clear the array `CSTMemSpace[6]` to prevent the CST memory manager from any heap initializations

In a standard Flex application, which uses the CST Action layer, the heap location and size is defined in a DSP/BIOS configuration file. The heap is created in a data memory section and assigned the heap identifier label, `_SEG0` that will be used when calling the DSP/BIOS memory manager functions `MEM_alloc()`, `MEM_free()` and `MEM_stat()` (see *TMS320C5000 DSP/BIOS Application Programming Interface (API) Ref Guide* (SPRU404)). During the initialization of the Flex application, the function `CSTAction_Init()` (see `CSTAction.h`) reloads the pointers to the new memory manager functions and stores the address of the heap identifier `SEG0` for later use.

The address and size of the heap become available at link time when the DSP/BIOS configuration file is compiled. See the example project and configuration files for standard multi-threaded Flex applications (more info on this can be found in section 5.4.1). If your application needs different location and/or size of the heap, the DSP/BIOS configuration file may be adjusted appropriately.

Wrapper Functions

During initialization, the function `CSTAction_Init()` invokes the function `CSTBIOSMemManInit()`, which makes the pointers to point to the wrapper functions `CSTBIOSAllocate()`, `CSTBIOSFree()` and `CSTBIOSGetAvailableMemory()` (see the files `BIOS\CSTBIOSmemman.c` and `BIOS\CSTBIOSmemman.h`). The wrapper functions eventually call the DSP/BIOS memory manager functions to allocate, free and count memory.

These wrapper functions initially were introduced to solve the following two problems:

- Preemption of the DSP/BIOS memory management functions
- Need to keep/remember the size of each allocated block for `MEM_free()`

The first problem arises because the DSP/BIOS memory manager functions are to be called from DSP/BIOS tasks only and they may cause a task/context switch. This is undesirable for CST because the CST Service, which is to be run in a high-priority SWI, dynamically allocates memory for XDAIS algorithms and some data.

The second problem, which is much less severe, required rewriting portions of the code that could call the standard C `free()` function or the CST memory manager function `CSTMemManager.free()` with just a pointer to the allocated block but not the block size.

The preemption problem was solved by disabling the software interrupts (SWIs) before calling the functions `MEM_alloc()`, `MEM_free()` and reenabling them afterwards. The wrapper functions took care of this disabling and enabling of SWIs. It was not long before the CST release when this problem was solved on the DSP/BIOS side. The DSP/BIOS introduced two new functions: `MEM_register_lock()` and `MEM_register_unlock()`, which are specific for the C54CST chip. The functions let the DSP/BIOS memory manager know which functions to call to prevent context switching during memory allocatioin/deallocation processes. The function `CSTBIOSMemManInit()` sets `SWI_disable()` and `SWI_enable()` (see *TMS320C5000 DSP/BIOS Application Programming Interface (API) Ref Guide* (SPRU404)) functions as such lock/unlock functions.

The allocated block size problem was solved by allocating larger memory blocks than requested and storing the size in this additionally allocated memory. This is the second purpose of the wrapper functions. This modification enabled to call the standard C function `free()` without the size. However, the CST Framework was modified as well to call the memory manager function `CSTMemManager.free()` with the correct size.

To learn more about the wrapper functions, see the files `BIOS\CSTBIOSmemman.c` and `BIOS\CSTBIOSmemman.h`.

Creating Second Heap

It is possible to have two heaps, if your application must use two different heaps. For example, one heap is located in the DSP internal DARAM, while the other is located in the external data RAM connected to the DSP. You must create an additional heap in the DSP/BIOS configuration file and assign it the heap identifier label, `_SEG1` that will be used when calling the DSP/BIOS memory manager functions `MEM_alloc()`, `MEM_free()` and `MEM_stat()`. Having done this, the user will also need to let the wrapper functions know about this second heap. This is done by modifying the array `tCSTBIOSMemSpace CSTBIOSMemSpace[6]` (see `BIOS\CSTBIOSmemman.c`, Table 7-34). The very first entry in this array (index 0) is initialized by the function `CSTAction_Init()` and it corresponds to the heap in the DSP internal DARAM. The next entry should be initialized by the user prior to calling the function `CSTAction_Init()`.

```
typedef struct {
```

Table 7-34. CST BIOS Memory Space Segment Structure

Type	Name	Description
`Int*`	`psegid`	Pointer to a DSP/BIOS heap segment ID.
`IALG_MemSpace`	`mSpace`	Space descriptor (see XDAIS core file `CST\Framework\XDAS\ialg.h` for options). The default CST memory space is `CST_DEFAULT_MEM_SPACE`, which is equal to `IALG_DARAM0`. This corresponds to the DSP internal DARAM (see section 8.3). The external memory may be denoted as `IALG_EXTERNAL`.

```
} tCSTBIOSMemSpace;
```

Type tCSTBIOSMemSpace is defined in BIOS\BIOSmemman.h.

Therefore, a portion of the Flex application initialization code will look something like this instead of a single call to `CSTAction_Init()`:

```
CSTBIOSMemSpace[1].psegid = &SEG1; // external data memory heap

CSTBIOSMemSpace[1].mSpace = IALG_EXTERNAL; // denotes external data memory

CSTAction_Init (&Ch0, 1, MyCallback);
```

7.5.4 More About Algorithm Creation and Deletion

The CST Framework includes XDAIS algorithm creation/deletion functions, also known as ALGRF library (ALGorithm instantiation for Reference Frameworks), having the following features:

- The following ALGRF functions are in CST ROM:
  ALGRF_create(),
  ALGRF_delete()
- The ALGRF functions in CST ROM do not support scratch memory (if the scratch support is required, the user can use the functions from the files ALGRF\algrf_creScratchSupport.c & algrf_delScratchSupport.c).
- The creation function ALGRF_create() uses a stack-based memTab[] array, in order to avoid heap fragmentation, and thus it is limited to only 16 memory records during algorithm creation (parameter ALGRF_MAXMEMRECS is equal to 16 in ROM). So, an algorithm should not request more than 16 memory records.
- CST Framework accesses the ALGRF functions only via stub-functions ALG_create_wrapper() and ALG_delete_wrapper(). The functions CSTStatisticsOnCreateAlg() and CSTStatisticsOnDeleteAlg() are called from those stub-functions upon successful algorithm creation and upon algorithm deletion, for collecting statistics on the amount of existing algorithms.
- Statistics report supports up to 20 algorithms existing simultaneously

To find out more about algorithm creation and deletion, IALG and ALFGRF, please read *Reference Frameworks for eXpressDSP Software: API Reference* (SPRA147).

7.6 Telephony Components Brief Specification

The following components are included in CST software:

- ❑ Data Modem (V.32bis/V.32, V.22bis/V.22, V.14, V.42, V.42bis)
- ❑ G.726/G.711 encoder/decoder
- ❑ G.168 Echo Canceller
- ❑ VAD, CNG and AGC
- ❑ UMTD (DTMF and CPTD) detector/generator
- ❑ CID receiver

Detailed description for each of the software components can be found in the corresponding documentation (User Guides and Product Annotations for specific components). This chapter gives only a brief overview of these components.

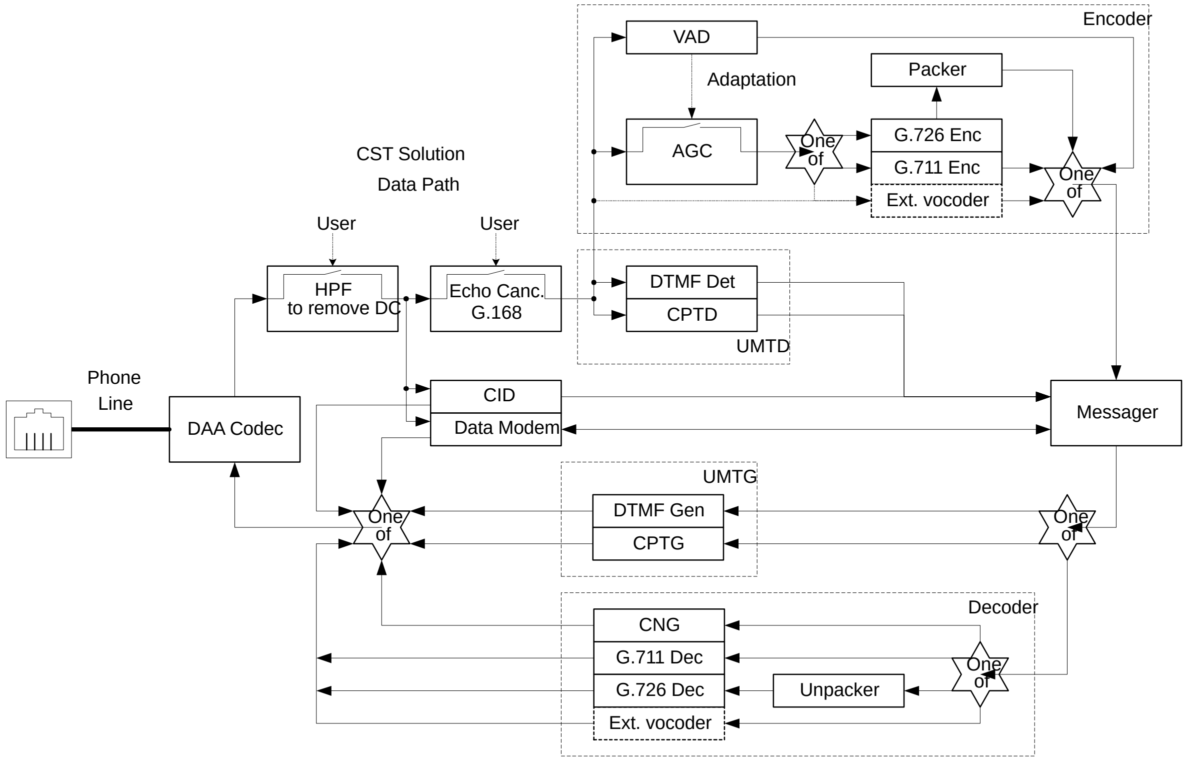
Besides, CST algorithms portofolio can be extented via a set of very memory-efficient CST Add-ons, sold separately from CST chip:

- ❑ Fax G3 (V.17/V.29/V.27ter/V.21) and V.29FastConnect (for POS-term.)
- ❑ Standard vocoders (G.729AB abd G.723.1) and SPIRIT-proprietary 1200 bps vocoder

Description of these CST add-ons is beyond the scope of this document.

The interconnection of XDAIS algorithms inside the CST Service layer is shown in Figure 7-1.

Figure 7-1. CST Solution Data Path



The user can control the way components are connected inside the CST framework and what components are currently active via AT commands or, in flex mode, via messages to one of CST control layers.

7.6.1 Data Modem

The data modem consists of several components, each implemented as a separate XDAIS object. These objects are modem data pump (MDP), V.42 error correction protocol with embedded V.42bis data compression protocol and a Modem Integrator object, which unifies access to all other modem algorithms (unified parameters, sample and data flows, extended status, etc), and interconnects them inside of itself.

The data modem controller (from hereof, DMController) is the upper layer that integrates the modem integrator object into the CST framework.

In brief, the modem integrator performs the following operations and has the following features:

- Implicitly creates all required XDAIS objects and performs their preliminary linking to each other. Depending on parameters, one of the three configurations is available: MDP + V.14 only, MDP + V.14/V.42 and MDP + V.14/V.42/V.42bis.
- Modem data pump auto rate and retrain control
- V.14 based asynchronous-to-synchronous conversion
- V.14/V.42 switch, connect and disconnect condition report
- V.42bis can be configured both in the symmetric (standard) mode and in the asymmetric mode
- Both single- and two-threaded mode support
- Fast connect capability
- Unified data flow, unified status

During the initialization, the user can configure the following important parameters of this object:

- General options:
  - Single threaded/multithreaded (preemption) mode
  - Answering or originating mode
  - A few callback functions for data transfers, preemption control (optional) and real-time control (optional)
- Data pump related options:
  - Fast connect mode
  - AFE delay (to produce correct V.32bis start up timing).
  - Output transmit level (can also be adjusted via S-register `S28` or via AT command `AT%L` in chipset mode, see section 9.4.3.13).
  - Maximum supported round trip delay in milliseconds (influences the amount of memory reserved for far echo bulk delay). In most cases the round trip delay does not exceed 100 ms. The amount of memory reserved for the far echo bulk delay buffer is calculated as follows: Supported_Far_Echo_Delay_ms*2.4. For example, for 100 ms, 240 words are required.
    However, for satellite connections the far echo delay may reach up 2 sec, and in such cases this parameter has to be set appropriately.
    The round trip delay can also be adjusted in the chipset mode, by the command `AT+ARTD` (see section 9.4.3.12).
  - Speedup and slowdown initiation permission

- V.42 related options:
  - Heap size (for storing received and sent packets)
  - Window size (number of stored last sent data packets)
  - System timeouts
  - 32bit FCS
- V.42bis related options
  - Dictionary size
  - Maximum string length
  - Compressor and decompressor enabling/disabling

When the fast connect is enabled (can be set via S-register S29 or by the AT command AT#F, see section 9.4.3.15), the modem will not transmit nor will it wait for the answer tone (2100 Hz tone in the beginning of modem connection), and training time for V.32bis/V.32 modem echo canceller will be reduced to minimum (0.5 sec).

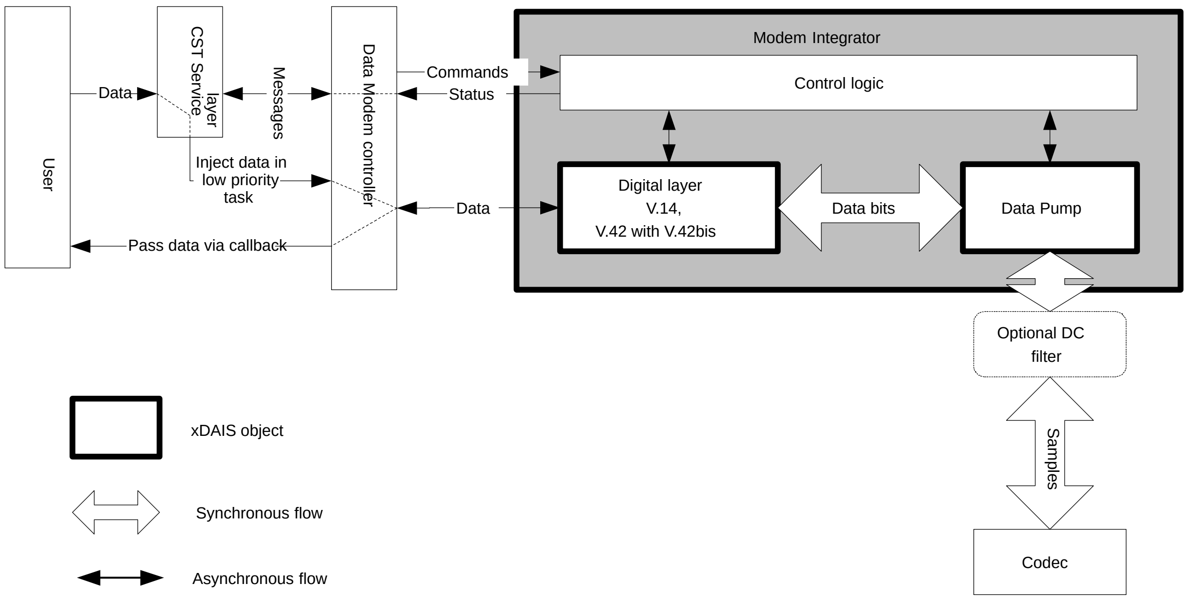
It is important to emphasize that the Modem Integrator supersedes the Data Pump/V.42/V.42bis interfaces, therefore, the user should interact with the Modem Integrator only.

The Data Modem Controller (DMController) finally integrates the modem into the CST framework. Since receiving compressed data may result in decompression of a large portion of data bytes, a special callback mechanism is used to transfer the received data to the user. Error correction also affects receiving the data since it may introduce some irregularity because of data rerequests.

7.6.1.1 Data Flow

The data modem can run in two modes: single-threaded and multi-threaded mode. The multithreaded mode makes it possible to move the compression procedures into a dedicated low-priority thread. The data flow is shown in Figure 7-2.

Figure 7-2. Modem Data Flow



As it has been mentioned, the modem passes the decoded data through a call-back function. Besides this data callback function, there are two other (option-al) callback functions to be implemented by user.

```
typedef struct IMODINT_ClientSubfxns {
```

Table 7-35. Data Flow Parameters

Params Type	Params Name	Description

`Int (*)`
 	
`(XDAS_Void* pClient,`
`Int instanceID,`
`XDAS_UInt8* pBuffer,`
`Int count)`
	`pTransferData`	Passes the decoded data from the remote modem. It is the only mandatory function to be implemented by the user. To set the address of called function, use the function `DMController_setTransferDataFunc()`.

Table 7-35. Data Flow Parameters(Continued)

Params Type	Params Name	Description
`XDAS_Bool (*)` `(XDAS_Void* pClient, Int instanceID, XDAS_Bool isPermitted)`	`pPreemption Control`	Requests to temporarily disable or reenable possible switching to the high-priority thread. It protects critical operations inside V.42/V.42bis modules. This method is invoked in multithread mode only, assuming V.42 is active.
`XDAS_Bool (*)` `(XDAS_Void* pClient, Int instanceID)`	`pIsRealtime Shortage`	Asks if the V.42bis compression/decompression should be continued for the next byte. This function is intended to prevent missing real-time. It is invoked in the single-threaded mode only. However it does not guarantee that burst activation of V.42bis would not cause real-time loss. This is because of big time slices in V.42bis iterations.

```
} IMODINT_ClientSubfxns;
```

Type IMODINT_ClientSubfxns is defined in imodint.h.

7.6.1.2 Brief Description of Data Modem Contrroller Interface (File DMController.c)

Table 7-36. Brief Description of CST S-Registers Function Interface

Name	Functionality
`DMController_create`	Creates a Modem Integrator instance (including all related algorithms)
`DMController_delete`	Delete a Modem Integrator instance (and all related algorithms)
`DMController_io`	Process a number of I/O samples (high-priority thread function)
`DMController_ setTransferDataFunc`	Set user's callback function to pull received data
`DMController_injectData`	Push a portion of the user's data into the modem

Creation Creates a modem instance

Function

```
bool DMController_create(tCSTChannel* pChannel,
 Int MaxSpeed,
 Int TxLevel,
 bool IsOriginator,
 bool IsV42,
 int IsV42bis,
 bool isFastConnect);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`MaxSpeed`	Maximum permitted speed (BPS)
`TxLevel`	Output signal amplitude in absolute units
`IsOriginator`	Originator/answerer mode
`IsV42`	Enable V.42
`IsV42bis`	Enable V.42bis: 0 – V.42bis disabled 1 – enable compressor only 2 – enable decompressor only 3 – enable both compressor and decompressor
`isFastConnect`	Reduce the connection time

Return Value Success/fail result

Deletion Deletes a modem instance

Function `void DMController_delete(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a global CST channel structure

Return Value None

Modem Process Function

Processes a number of I/O samples (high-priority thread function)

Function

```
void DMController_io(tCSTChannel* pChannel,
 int16 *pIn,
 int16 *pOut,
 int Count);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pIn`	Pointer to a buffer of valid input samples
`pOut`	Pointer to a buffer of valid output samples
`Count`	Amount of samples in the buffers

Return Value None

Set Callback Function to Pull Received Data

Sets the user’s callback function to pull the received data

Function

```
void DMController_setTransferDataFunc(tCSTChannel*
pChannel,
 Int (*pTransferData)
   (XDAS_Void* pClient,int,XDAS_UInt8*,Int));
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pTransferData`	Address of the user’s callback function

Return Value None

Push Data Push a portion of the user’s data into the modem

Function

```
int DMController_injectData(tCSTChannel* pChannel,
 uint8 *pData,
 int Count);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`pData`	Pointer to the data buffer
`Count`	Total number of data bytes

Return Value Number of taken bytes

7.6.1.3 V.32/V.32bis and V.22/V.22bis Data Pump

The modem data pump is implemented according to the ITU-T recommendations V.32bis/V.32 and V.22bis/V.22, and supports all their features and options, including retrain, rate renegotiation request and automodem for interoperability. It is a member of a family of SPIRIT complete fax/data modem data pumps, and includes unified implementation of these protocols.

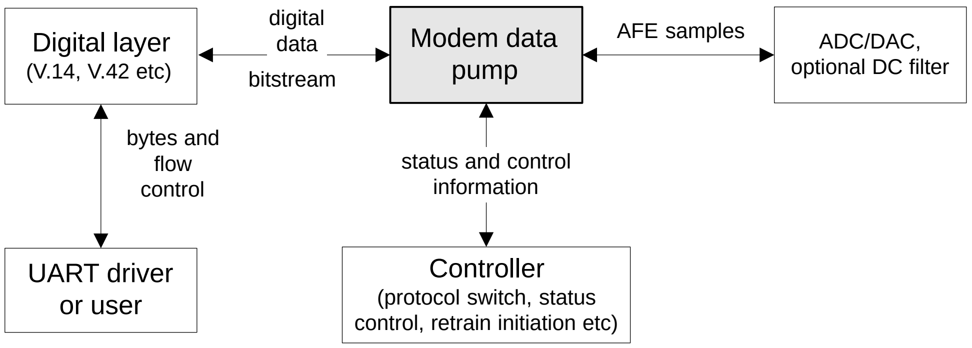
It has a simple interface and can be easily connected to an analog line (8 kHz 16-bit samples) and to an HDLC client (e.g. V.42/V.42bis). The interface is fully compatible with other SPIRIT data pumps and HDLC clients.

In brief, the V.22bis/V.32bis module performs the following operations:

- ❑ Processing of a number of samples from ADC and generation of new samples to be output to DAC
- ❑ Sending and receiving data bits through callback functions to the high-level client (modem integrator)
- ❑ Accepting and executing a number of control commands
- ❑ Reporting status and informing the client about status changes

Figure 7-3 represents typical V.22bis/V.32bis modem data pump operating environment.

Figure 7-3. Modem Data Pump Operating Environment



After the initialization, the user can send a command (in Flex Mode) to the data pump, telling at what maximum rate the pump may try to connect with a remote modem. The maximum rate can also be selected by the `ATB<0-8>` command (see section 9.4.3.14).

The Modem Integrator encapsulates and integrates the V.32/V.32bis, V.22/V.22bis and the data pump objects inside itself. Thus, when using the Modem Integrator, the user should not interact with the pump directly. All interaction should be done through the Modem Integrator interface.

7.6.1.4 Asynchronous to Synchronous Data Conversion, V.14

This protocol is implemented inside the Modem Integrator object. When the V.42 protocol is disabled or fails to connect with a remote counterpart, the Modem Integrator uses the protocol V.14 instead to provide conversion from asynchronous data flow from serial port to synchronous data flow, required by the modem data pump.

The V.14 protocol can be forced by setting the S-register `srd_V42` to zero (see 7.2.2.1) or by issuing the command `AT\N0` (see section 9.4.3.10).

If the V.42 operation is enabled (see 7.6.1.5) but the remote modem does not support V.42 or the V.42 handshake falls throuhg, the modem falls to the V.14 asynchronous mode anyway.

The Modem Integrator implements and integrates the V.14 protocol. There are no means to access V.14 functions directly.

7.6.1.5 Error Correction, V.42

The CST's V.42 component implements an ITU-T V.42 compliant HDLC client and provides an error correction protocol between a software-emulated unbuffered DTE and a V-series duplex data pump.

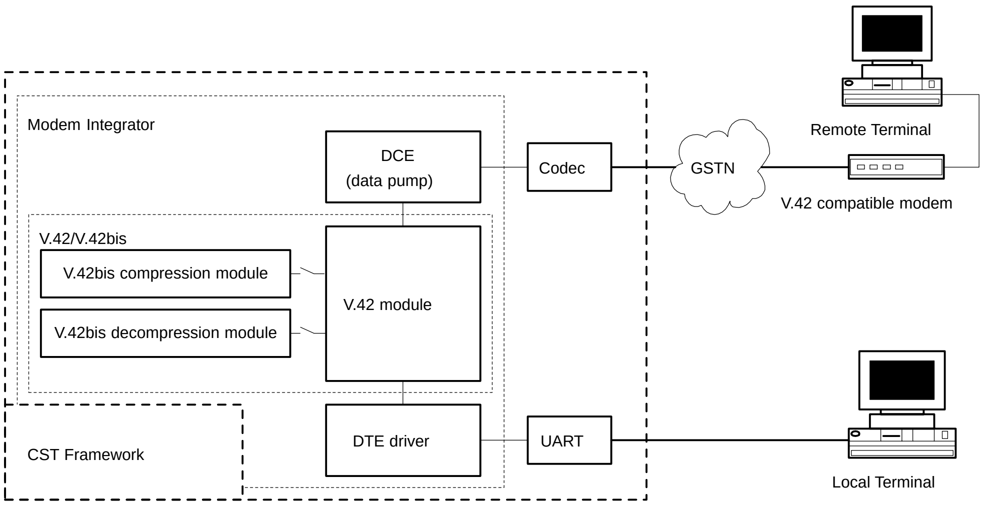
The Modem Integrator incorporates into itself the data pump and it also represents the DTE. Thus, when using the Modem Integrator, the user should not interact with the V.42 instance directly.

In brief, the V.42 module features the following functions and properties:

- Embedding the V.42bis compression/decompression module (see Figure 7-4 below)
- Getting the byte stream from the DTE (the Modem Integrator);
- Compressing it by the internally linked V.42bis module;
- Packing the bytes into frames;
- Converting the frames into the bit stream ready to be transmitted by the synchronous DCE (the modem data pump);
- Resending the frames (within the frame buffer) on demand, providing flow control, error correction and stream integrity;
- Receiving the bit stream from the synchronous DCE and converting it into frames;
- Parsing the frames and unpacking them into bytes;
- Decompressing the bytes by the internally linked V.42bis module;
- Sending the byte stream to the DTE.

The Modem Integrator interconnects the V.42 module to the data pump and an analog of a DTE driver (see Figure 7-4). As it has been mentioned above, the user can optionally disable the V.42 module in whole or in part.

Figure 7-4. V.42 Operating Environment



During initialization, the user has to tell the V.42 module how much memory to use for its internal heap (which is mostly used for storing received and sent packets so that V.42 could resend them upon a request). Normally, the V.42 has to store at least 15 sent packets (this number is a changeable parameter), each 133 bytes long. There are several other buffers that the V.42 has to store as well. Typically, the V.42 needs around 1.5 kW of dynamic memory (1 kW minimum) for its internal heap (this is a changeable parameter), plus about 0.8 kW for the V.42 object itself.

This internal heap memory size can also be set by sending the command `AT+EHEAP` to the AT parser (see section 9.4.3.9). The greater the size of this heap is, the more efficiently the V.42 module will do the error correction.

Some of the AT commands related to V.42 operations are introduced only for compatibility reasons with the V.250 standard (see *ITU-T Recommendation V.250. Serial asynchronous automatic dialing and control*, 07/97), but do not actually affect anything. These commands are: +EB, +ER, +ES, +ESR (see sections 9.4.3.2 through 9.4.3.6).

The V.42 protocol is enabled by setting the `srd_V42` S-register to a non-zero value (see 7.2.1.1) or by issuing the command `AT\N1` (see section 9.4.3.10).

7.6.1.6 Data Compression V.42bis

The CST's V.42bis component is integrated into the V.42 module and implements an ITU-T V.42bis compliant data compression/decompression functions and is designed to operate with an HDLC client, such as V.42.

In brief, the V.42bis module performs the following operations:

- Compressing (or decompressing) bytes to/from bit stream with codewords;
- Accepting and processing special control primitives like C_INIT and C_FLUSH.

The V.42bis module is already connected to V.42 module as shown in Figure 7-4.

The CST's V.42 module can use the V.42bis component in several modes: completely disabled, compression for TX only, decompression for RX only, and both compression and decompression. Note that some modems do not correctly support the asymmetric mode.

These modes can be selected by setting the S-register `srd_V42BIS` (see 7.2.1.1) or by the command AT%C<`0-3`> (see section 9.4.3.11) or by the command +DS (see 9.4.3.1).

`AT%C0`	No compression/decompression; V.42bis disabled
`AT%C1`	Only compresses transmitted data
`AT%C2`	Only decompresses received data
`AT%C3`	Both compression and decompression enabled

During initialization, it's possible to set the dictionary size for V.42bis. There's a parameter for that. The greater the size of the dictionary is, the more efficiently V.42bis can compress the data.

In the Chipset Mode, this parameter is equal to 512, and it is impossible to make the size greater. This is due to the memory organization in the Chipset Mode, which initially has been configured with 16KW DARAM in mind. As it turned out, the C54CST chip got 40KW of internal DARAM, so the above limitation holds only for the unchanged chipset CST application. It is possible to initialize V.42bis in the Flex Mode with a greater dictionary size (or load a special patching flex application that will reconfigure CST chipset application), and thus make the compression more efficient. For each direction (for compression or decompression), the memory size needed for the dictionary is calculated as `Dictionary_Size*3+256` words.

Disabling V.42bis saves about 2.6 kW of memory during modem connection, and it is even recommended to disable V.42bis, if higher level protocols have their own compression enabled, or if low-compressible data is to be transferred.

7.6.2 Voice Processing

Voice processing includes several components – waveform codec (PCM and ADPCM), line echo canceller, Automatic Gain Control (AGC) controlled by Voice Activity Detector (VAD), Comfort Noise Generator (CNG). All components have simple interfaces and operate with 14-bit 8 kHz samples.

The CST Solution also has a simple voice controller, partially supplied in open source code, which provides voice bit stream packing/unpacking, continuous voice play-out and ADPCM encoder/decoder creation/deletion.

The CST Service layer includes the voice controller for easy connection of XDAIS algorithms for voice processing. The voice controller performs data flow/exchange control (to/from the CST Service layer).

All of the CST voice controller functions are contained in the file `VController.c`. In the AT parser, voice commands become accessible only in the voice mode, which is turned on by the command `AT#CLS=8`. The command `AT#VRX` starts recording a signal from the telephone line, the command `AT#VTX` starts playing-out a signal to the line, and the command `AT#VRXTX` starts duplex voice exchange (see section 9.4.4 for details). The voice bitstream, transferred between the host and CST chip via the serial link, consists of bits packed into bytes. Control events (such as “stop play-out”, or “DTMF Digit 7 detected”, or “BUSY signal detected”) are transferred as special shielded codes (see section 9.5 for details) inside this bytes stream.

The LPC coefficients and other parameters for the Comfort Noise Generator (CNG packets) are transferred along with with PCM/ADPCM packets, shielded with corresponding DLE symbols. This is applicable for the Flex Mode too.

7.6.2.1 Files VController.c, VController.h

Voice Controller Main Structure Definition

The voice controller has an internal structure, which contains the controller’s current state and can be useful for outsiders.

typedef struct tVControllerStr {

Table 7-37. Voice Controller Main Structure Definition

Field Type	Field Name	Description
`void*`	`pUserData`	Pointer to user's data.
`bool`	`IsCoder`	This flag shows voice encoder availability.
`bool`	`IsDecoder`	This flag shows voice decoder availability.
`int`	`BPS`	This field stores the vocoder's bit rate.
`tCSTFIFO`	`UARTIngressData`	Circular buffer (FIFO) to store ingress packed voice samples (from UART or HPI).
`tCSTFIFO`	`VoiceIngressData`	Circular buffer (FIFO) to store ingress voice samples.
`tCSTFIFO`	`VoiceEgressData`	Circular buffer (FIFO) to store egress voice samples.
`void*`	`pVocoder`	Pointer to vocoder object instance.
`AGC_Handle`	`pAGC`	Automatic gain control object handle.
`CNG_Handle`	`pCNG`	Comfort noise generator object handle.
`VAD_Handle`	`pVAD`	Voice activity detector object handle.
`tVocoderFxns*`	`pVocoderFxns`	Pointer to virtual function table, containing pointers to vocoder create, delete, encode, decode wrapper functions.
`tDLEParser`	`DLEParser`	DLE parser structure.
`eVCtrlDLE`	`LastDLE`	Last DLE symbol processed by the voice controller.
`int [CNG_ PARAM_LEN]`	`CNGParamBuffer`	Buffer for receiving CNG parameters
`int`	`CNGParamIndex`	Temporary variable to store current position in CNG parameters buffer while receiving CNG parameters.
`int`	`LPCStrSize`	Calculated size of CNG parameters to receive.
`bool`	`CNGParamsReady`	This flag is set when CNG parameters received.
`int`	`LPCOrder`	Current order (amount) of LPC coefficients. This value is used only when VAD is enabled.
`int`	`VoiceGain`	Output voice gain.

```
} tVControllerStr;
```

Type tVControllerStr is defined in VController.h.

7.6.2.2 Brief Description of Voice Controller Function Interface

The most of the CST voice controller is implemented in the files `VController.c` and `VController.h`. The main interface functions are the following:

Table 7-38. Brief Description of Voice Controller Function Interface

Name	Functionality
`VcontrollerInit()`	Voice controller initialization.
`VcontrollerInjectData()`	Tries to put ingress packed voice data into the dedicated circular buffer (FIFO).
`VcontrollerProcess()`	Performs voice encoding and decoding. In single-threaded mode is called from `VcontrollerHighPriorityProcess()`. In case of two-threaded mode should be called in a software interrupt (SWI), which is to be posted from a hardware interrupt (HWI/ISR).
`VcontrollerDeleteRx()`	Deletes voice-encoding path.
`VcontrollerDeleteTx()`	Deletes voice-decoding path.
`VcontrollerCreateRx()`	Creates voice-encoding path.
`VcontrollerCreateTx()`	Creates voice-decoding path.
`VcontrollerIsVoiceData()`	Verifies if a vocoder instance is created.
`VcontrollerHighPriorityProcess()`	This functions should be called from a high-priority thread. It stores voice samples to the ingress circular buffer (FIFO) and gets decoded voice samples from the egress voice circular buffer (FIFO).
`VcontrollerSelectVocoder()`	Default function for selecting a vocoder. Sets the pointer to the virtual function table to point to the G726 and G711 vocoder wrapper functions.
`VcontrollerAllocBuffers()`	Tries to allocate memory for the circular buffers. Should be called from the vocoder’s virtual create function.
`VcontrollerFreeBuffers()`	Deallocates memory previously allocated for the circular buffers. Should be called from the vocoder’s virtual delete function.
`VControllerGetAndSendLPC()`	This function gets CNG parameters and send them as a byte array.

Initialization Initialization of the voice controller

Function `void VControllerInit(struct tCSTChannel* pChannel);`

Parameter(s)

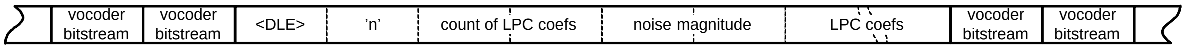
`pChannel`	Pointer to a channel structure.

Return Value None

Packed Voice Data Reception

Receives ingress packed voice data and places it to the vocoder’s buffer. The format of the bit stream for vocoders G726 and G711 is as follows:

Figure 7-5. G726 and G711 Bitstream Format



Each vocoder (any kind of PCM coder) frame, as well as a “noise” (CNG) frame, covers the same time interval.

G.726 bitstream is composed of packed ADPCM results (minimum 120 dibits per frame, maximum 120 pentabits per frame, i.e. from 30 to 75 bytes per frame, depending on the vocoder’s bitrate).

G.711 bitstream is composed of encoded PCM bytes (120 bytes per frame).

The structure of a CNG frame is shown in the above figure.

Function

```
int VcontrollerInjectData
(tCSTChannel* pChannel,
 uint8 *pData,
 int Count);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`pData`	Pointer to a byte array, containing packed voice data.
`Count`	Amount of samples in array pointed by `pData`.

Return Value Function returns the number of data bytes it has taken from the array.

High Priority Processing

This function should be called from a high priority thread. It stores voice samples to ingress circular buffer and gets decoded voice samples from the egress voice circular buffer. In the single-threaded mode it calls the function `VcontrollerProcess()` to process samples, in case of two-threaded mode it should post a software interrupt (SWI) for background processing.

Function

```
void VcontrollerHighPriorityProcess
(tCSTChannel* pChannel,
 int16 *pInput,
 int16 *pOutput,
 int Count);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`pInput`	Pointer to an array of input voice samples.
`pOutput`	Pointer to an array for output voice samples.
`Count`	Amount of samples in each array.

Return Value None

Low Priority Processing

This function is called from the function `VcontrollerHighPriorityProcess()` directly (in single-threaded mode) or posted from it via SWI (in two-threaded mode). It executes virtual wrapper functions for voice encoding and decoding when available.

Function

```
void VControllerProcess (tCSTChannel* pChannel);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.

Return Value None

Vocoder Selection

This function is used to select a vocoder. By default it selects G726 and G711 coders. When adding extra vocoders, the user should override this function and set `tVControllerStr.pVocoderFxns` pointer to its own vocoder's wrapper functions structure.

Function

```
void  VControllerSelectVocoder  (tCSTChannel*  pChannel,int
param);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`param`	This parameter is used to select a vocoder.

Return Value None

Transferring Compressed Voice Samples to CST Service Layer

This function transfers data from the vocoder to the CST Service layer. It performs DLE stuffing and noise frames marking by using DLE symbols.

Function

```
void VcontrollerTransferVoiceData
(tCSTChannel* pChannel,
 uint8 *pData,
 int Count,
 tVCtrlDLE Type);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`pData`	Pointer to byte array.
`Count`	Size of byte array.
`Type`	The type of transmitted samples (voice data or noise parameters).

Return Value None

Sending CNG Parameters

This function retrieves LPC coefficients from the VAD object and sends them using the function `VControllerGetAndSendLPC()`. The `Gain` parameter is used to correct the noise magnitude.

Function

```
void  VControllerGetAndSendLPC  (tCSTChannel*  pChannel,int
Gain);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`Gain`	Current AGC gain coefficient.

Return Value None

Buffers Allocation for Vocoder

This function is used to allocate memory for buffers to be used by a vocoder for samples/bitstream buffering.

It also handles exception conditions.

Function

```
bool  VcontrollerAllocBuffers
(tCSTChannel* pChannel,
 int IngressUARTSize,
 int IngressVoiceSize,
 int EgressVoiceSize);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`IngressUARTSize`	Size of the buffer for packed voice data.
`IngressVoiceSize`	Size of the buffer for ingress voice samples.
`EgressVoiceSize`	Size of the buffer for egress voice samples.

Return Value Returns nonzero value when all buffers are allocated successfully.

Buffers Deallocation for Vocoder

This function is used to free the memory allocated for buffers, used by a vocoder for samples/bitstream buffering.

Function

```
void VControllerFreeBuffers (tCSTChannel* pChannel,
 int IngressUARTSize,
 int IngressVoiceSize,
 int EgressVoiceSize);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`IngressUARTSize`	Size of the buffer for packed voice data.
`IngressVoiceSize`	Size of the buffer for received voice samples.
`EgressVoiceSize`	Size of the buffer for transmitted voice samples.

Return Value None

Voice Encoder Creation

This function tries to create vocoder encoder path in case it has not been created earlier.

Function

```
bool VControllerCreateRx (tCSTChannel* pChannel,int
CoderBPS);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`CoderBPS`	Bitrate.

Return Value Returns nonzero value when the voice encoder is created and ready for use.

Voice Decoder Creation

This function tries to create vocoder decoder path in case it has not been created earlier.

Function

```
bool VControllerCreateTx (tCSTChannel* pChannel,int
DecoderBPS);
```

Parameter(s)

`pChannel`	Pointer to a channel structure.
`DecoderBPS`	Bitrate.

Return Value Returns nonzero value when the voice decoder is created ready for use.

Voice Encoder Deletion

This function deletes the vocoder instance if the decoder is marked as unused. Otherwise it just marks the encoder as unused.

It also deletes the VAD and AGC instances, if they have been created earlier.

Function `void VControllerDeleteRx (tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a channel structure.

Return Value None

Voice Decoder Deletion

This function deletes the vocoder instance if the encoder is marked as unused. Otherwise it just marks the decoder as unused.

It also deletes the CNG instance, if it has been created earlier.

Function `void VControllerDeleteTx (tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a channel structure.

Return Value None

7.6.2.3 Wrapper Functions

The user may change all wrapper function pointers in case he/she wants to use different coders (for example, G.723 or G.729 vocoder).

```
typedef struct tVocoderFxns {
```

Table 7-39. Structure Definition

Return Value	Function Name	First Parameter Type	Description
`void*`	`*pfCreate`	`tCSTChannel*`	Creates vocoder object instance
`int`	`*pfEncode`	`tCSTChannel*`	Voice encoding function
`int`	`*pfDecode`	`tCSTChannel*`	Voice decoding function
`void`	`*pfDelete`	`tCSTChannel*`	Vocoder instance deletion function

```
} tVocoderFxns;
```

Wrapper for Creation Function

This wrapper function should call the function `VControllerAllocBuffers()`to allocate memory for circular buffers.

It can allocate an additional memory block for private use (`tVControllerStr.pUserData` can store a pointer to this block).

Having allocated the buffers, the function may finally create a vocoder object instance.

Function `void* (*pfCreate)(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a channel structure.

Return Value Vocoder handle (successful creation) or NULL (failure).

Wrapper for Deletion Function

This wrapper function should call the function `VControllerFreeBuffers` to free memory allocated for the circular buffers. It should also free all allocated additional memory blocks (if any). And finally, it should delete the vocoder object instance.

Function `void (*pfDelete)(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a channel structure.

Return Value None

Wrapper for Encoding Function

This wrapper function should wait until the needed amount of samples is available for processing in the ingress voice buffer. When there's enough samples, it should process the samples and send packed voice data using the function `VControllerTransferVoiceData()`.

Function `int  (*pfEncode)(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a channel structure.

Return Value Not used now.

Wrapper for Decoding Function

This wrapper function should wait until the needed amount of packed samples is available and until the voice egress buffer has enough space to receive an unpacked voice frame. When there's enough samples and space, the function should process the samples and store the output to the voice egress buffer.

Function `int  (*pfDecode)(tCSTChannel* pChannel);`

Parameter(s)

`pChannel`	Pointer to a channel structure.

Return Value Not used now.

7.6.2.4 ADPCM/PCM Encoder/Decoder G.726/G.711

The CST's G726G711 component implements the ITU-T G.726 adaptive differential pulse code modulation (ADPCM) encoder and decoder of voice frequencies, as well as G.711 logarithmic conversion.

In brief, the G726G711 module performs the following operations:

- ❑ Optional converting of an A-law or μ-law PCM input signal to uniform (linear) PCM or vice versa according to G.711;
- ❑ Optional compressesing/decompressesing of linear samples to/from bitstream, based on the selected bit rate – 16, 24, 32 or 40 kbps, according to G.726

This algorithm is designed to process a signal sample by sample, not in the frame-based manner. However, its external interface allows to process samples by blocks of any length.

The user can either use only G.711, or only G.726, or both of these algorithms to process the voice signal.

The bitstream may contain 2 to 8 bits per sample. The voice controller does packing of these bits into bytes as well as it does unpacking.

Compression ratio can be selected by the command `AT#VBS<2,3,4,5,8>` (see section 9.4.4.3), which chooses 16, 24, 32, 40 kbps G.726 or 64 kbps PCM μ-law respectively.

7.6.2.5 Echo Canceller G.168

The CST's Line Echo Canceller (LEC) is used for cancellation of the electric echo created by the telephone hybrid. The LEC conforms to the G.165 and G.168 ITU recommendations. It includes a double talk detector and a nonlinear processor. The user can set the value of the maximum echo path equal to 16 or 32 msec.

For correct operation of the LEC, the input samples should be linear PCM samples with absolute values less than 8159 (this is the maximum value for linear samples after μ-law expansion). The CST Service performs the scaling needed for LEC automatically.

The echo canceller can be enabled/disabled in the CST Framework voice path by the command `AT#VEC` (see section 9.4.4.1).

7.6.2.6 VAD, CNG and AGC

The CST’s voice activity detector (VAD) detects the presence of speech in the signal. It has a special adaptive algorithm to automatically adjust to the level of the noise in the signal, in order to provide robust operation even in the noisy speech. It has many user configurable parameters, allowing the algorithm to optimally tune itself for a specific application. The VAD also outputs several coefficients that characterize the spectral envelope of the noise (when no speech is detected), so that the regenerated noise would appear similar to the original noise.

The CST’s Comfort Noise Generator (CNG) generates noise, distributed either uniformly or shaped according to the spectral envelope coefficients, which can be passed to the CNG as parameters.

The CST’s Automatic Gain Control (AGC) is designed specifically to amplify the voice signal, which has very non-stationary amplitude envelope. It operates much better in conjunction with the VAD, which can tell the AGC when there is no speech in the signal, so that the AGC would not adapt in these periods.

In the Chipset Mode, the VAD may report about speech absence and send the noise spectral envelope coefficients via special shielded codes, included in the voice data. Either vocoded (wave form coded) digital data or the VAD/CNG parameters are produced for each coded timeframe. The CNG can be enabled or disabled also via special shielded codes, received over the serial link (from the Host to CST). Read section 9.5 about these shielded codes.

7.6.3 Telephony Signals Processing

Telephony signals processing includes several components – UMTD (detects DTMF and CPT signals), UMTG (generates DTMF and CPT signals) and client side Caller ID. All components have simple interfaces and operate with 16-bit 8 kHz samples (they have wider input dynamic range than voice processing components, which usually operate with 14-bit samples only).

7.6.3.1 Universal Multifrequency Tone Detector (DTMF/CPT/etc.)

CST includes the Universal Multifrequency Tone Detector (UMTD) for detecting DTMF, Call Progress Tones (CPT) and many other telephony signals.

In brief, the UMTD detector filters the input samples, estimates the spectrum of the input signal, checks the cadences and pauses and makes the decision about presence of signaling tones. The UMTD can be easily configured to fit the specific standard of any country.

DTMF Detector

The CST's DTMF Detector operates in compliance to ITU-T Q.24 Recommendation.

The CST's DTMF has good talk-on performance, detecting the tone even in noisy signal, and good talk-off performance, avoiding false detection in the presence of speech or music. Good talk-off performance allows turning the DTMF detector on right away when going off hook and turning it off only when going back on hook. In the Chipset Mode, the DTMF detector is activated only in the voice mode of the AT parser, and detector results are sent to the Host via special shielded codes (see section 9.5 for details).

CPT Detector

The CST's CPT detector, in default configuration, accepts a wide range of call progress tones fitting the standards of most countries and detects the following signals/events:

Table 7-40. Detected CPT Signals

Configuration	Signal	Freq., Hz	Durations, sec
Q.35	dialtone	340-500	Continuous, more than 2.6
	busy	340-500	[0.07–0.70]–[0.01–0.80]
	ringback	340-500	[0.67–2.50]–[3.00–6.00]
	longtone	340-500	Continuous, more than 1
Q.35 extended	dialtone	340-500	Continuous, more than 2.6
	busy	340-500	[0.07–0.70]–[0.01–0.80]
		340-500	[0.70–0.80]–[0.70–0.80]
	ringback	340-500	[0.67–2.50]–[2.00–6.00]
		340-500	[0.35–0.55]–[0.15–0.30]–[0.35–0.55]–[1.95–6.05]
	longtone	340-500	Continuous, more than 1

CST contains two default configurations, which are defined in the array `CPTD_Configurations[4]` (see the files `umtd_signals.h` and `umtd_signals.c`). The first configuration detects all signals strictly according to the Q.35 recommendation. The second configuration detects all signals according to the Q.35 recommendation and some signals that do not fit the Q.35 recommendation (see Table 7-41). The third and fourth configurations are left empty by default, and the user can attach his/her own configurations here.

In the Chipset Mode, the current configuration can be selected by the command `AT+CNTRY` (see section 9.4.1.31)

Table 7-41. CPTD Configurations

Country	Configuration	Country	Configuration
Argentina	Q.35	Italy	Q.35 ext.
Australia	Q.35 ext.	Japan	Q.35 ext.
Austria	Q.35	Korea	Q.35 ext.
Belgium	Q.35	Netherlands	Q.35
Brazil	Q.35	Norway	Q.35
Canada	Q.35 ext.	Singapore	Q.35 ext.
China	Q.35	Sweden	Q.35
Denmark	Q.35	Switzerland	Q.35
Finland	Q.35	Taiwan	Q.35 ext.
France	Q.35	United Arab Emirates	Q.35 ext.
Germany	Q.35	United Kingdom	Q.35 ext.
Great Britain	Q.35 ext.	United States	Q.35
Israel	Q.35		

In the modem mode of the AT parser, the CPT detector's behavior can be controlled by the command ATX (see section 9.4.1.23 for details). Use ATX1 to disable, and ATX4 to enable both busy and dial tone detection.

In the voice mode, the CPTD events are sent to the Host via special shielded codes (see section 9.5 for details).

7.6.3.2 Universal Multifrequency Tone Generator (DTMF/CPTD/etc.)

CST includes the Universal Multifrequency Tone Generator (UMTG) for DTMF, CPT and many other telephony signals generation. It can be set to generate tones according to the standards of different countries (tones' frequencies and cadences are adjustable).

DTMF Generator

The UMTG-based DTMF Generator operates in compliance to ITU-T Q.23 Recommendation.

The UMTG-based DTMF generator produces output DTMF tones with the duration and pause specified by the user. During initialization, the user may also enable output the bandpass filter to remove clicks at the beginning and end of generated tones.

In all modes of the AT parser, the DTMF generator is controlled by the command ATDT (see section 9.4.1.5 for details). The duration of DTMF tones and pauses is controlled by the register S11 (the value is in milliseconds).

CPT Generator

The UMTG-based CPT generator produces output signals with cadences and frequencies specified in UMTG settings.

The following CPT signals are generated, with the following characteristics by default:

Table 7-42. Generated CPT Signals Parameters

Signal	Freq, Hz	Durations, sec
dial	350+440	continuous
busy	480+620	0.5–0.5
fast busy	480+620	0.3–0.3
ringback	440+480	2.0–4.0

The user can add new signals, and change their frequencies and cadences, so the CPT generator can be tuned to a standard of virtually any country. The CPT generator can be controlled only in the Flex Mode.

7.6.3.3 Client Side CID

The CST’s Client Side Caller ID includes Type I and Type II Caller ID signal detection, compliant with standards of several providers and countries (Bellcore, British Telecom, ETSI (European Countries), Australia, China, etc.). It has the following features:

❑ Complies with Bellcore GR-30-CORE, SR-TSV-002476; British Telecom SIN227 and SIN242; ETSI ETS 300 659, ETS 300 778; Mercury Communications MNR 19

- Supports Caller ID On Call Waiting operation
- Supports Single, Multiple Data Message Formats and VMWI
- Delivers completely decoded CID messages at a presentation layer, including forwarding call information, network operator messages, etc. The message parser is supplied in open source code, see the file `CST\CST\CID\CIDParser.c`.
- Supplied with simple high-level state machine wrapper (in open source code, see the file `CST\CST\CID\CIDWrapper.c`) to make integration and control easier.
- Can be switched to several different states by the user:
  DT-AS (CAS) signal detector,
  FSK carrier detector,
  FSK message detector,
  TE-ACK signal generator
- Allows the user to configure the software at run time, including carrier thresholds, signal levels, etc.

In the Flex Mode of the CST chip, there are many parameters for each signal detected or generated by CID, which can be adjusted.

The detailed description of the Client Side CID is given in the “Caller ID User’s Guide”. Please, be aware that the CST ROM contains this object with limited functionality (for example, only the Client Side is implemented, PBX side is not implemented).

It is also possible to tune some parameters of some CID signals: DT-AS, TE-ACK and FSK by using AT commands (see section 9.4.2).

In the CST Framework, the CID is enabled after each RING signal, detected by the DAA driver, and after going off hook (to detect CID On Call Waiting). To disable CID, use the command `AT#CID0` (see section 9.4.2.4 for details). To enable the CID and turn the formatted output on, use the command `AT#CID1`.

To parse a CID message, the CST Framework temporarily reserves a 442-words-long buffer.

7.6.4 Telephony Components Summary

The following table illustrates a relation between integrated CST algorithms, CST Service tasks, CST Commander atomic commands and CST Action standard operations.

Table 7-43. Relationship Between CST Algorithms, Service Tasks, Atomic Commands and CST Actions

Algorithm	Corresponding CST Service Task	Corresponding CST Commander Atomic Command	Corresponding CST Action Standard Operations
Modem data pump V.22/V.22bis, V.32/V.32bi	cstst_MODEM	cac_TURNON_MODEM	sot_TURNON_MODEM_CALL_X, sot_TURNON_MODEM_ANS
Error correction, data compression V.42/V.42bis			
ADPCM codec G.726	cstst_VOICE_DATA(rx/tx)	cac_TURNON_VOICE_DATA_X	sot_TURNON_VOICE_RXDATA, sot_TURNON_VOICE_TXDATA, sot_TURNON_VOICE_RXTXDATA
PCM codec G.711			
Electrical echo canceller G.168	cstst_VOICE_LOOP	cac_TURNON_VOICE_LOOP	sot_TURNON_VOICE_CALL_X, sot_TURNON_VOICE_ANS, sot_TURNON_VOICE_RXDATA, sot_TURNON_VOICE_TXDATA, sot_TURNON_VOICE_RXTXDATA
VAD/AGC/CNG	cstst_VOICE_DATA	cac_TURNON_VOICE_DATA_X	sot_TURNON_VOICE_RXDATA, sot_TURNON_VOICE_TXDATA, sot_TURNON_VOICE_RXTXDATA
UMTG/D (DTMF)	cstst_DTMF(rx)	cac_TURNON_SIMPLE_X(cstst_DTMF)	sot_OFF_HOOK, sot_TURNON_VOICE_RXDATA, sot_TURNON_VOICE_TXDATA, sot_TURNON_VOICE_RXTXDATA
	cstst_DTMF(tx) with cse_DATA(…)	cac_DIALING	sot_TURNON_MODEM_CALL_X, sot_TURNON_VOICE_CALL_X, sot_JUST_CALL_X

Table 7-43. Relationship Between CST Algorithms, Service Tasks, Atomic Commands and CST Actions (Continued)

Algorithm	Corresponding CST Service Task	Corresponding CST Commander Atomic Command	Corresponding CST Action Standard Operations
UMTD (CPTD)	cstst_CPTD(rx)	cac_TURNON_SIMPLE_X(cstst_CPTD), cac_WAIT_CPTD_APPEARANCE_XX	sot_OFF_HOOK, sot_TURNON_MODEM_CALL_X, sot_TURNON_VOICE_CALL_X, sot_TURNON_VOICE_RXDATA, sot_TURNON_VOICE_TXDATA, sot_TURNON_VOICE_RXTXDATA, sot_JUST_CALL_X
Caller ID	cstst_CID	cac_TURNON_CID_X	sot_CID_AFTER_RINGEND, sot_CID_AFTER_LINE_REVERSAL

7.7 CST Drivers

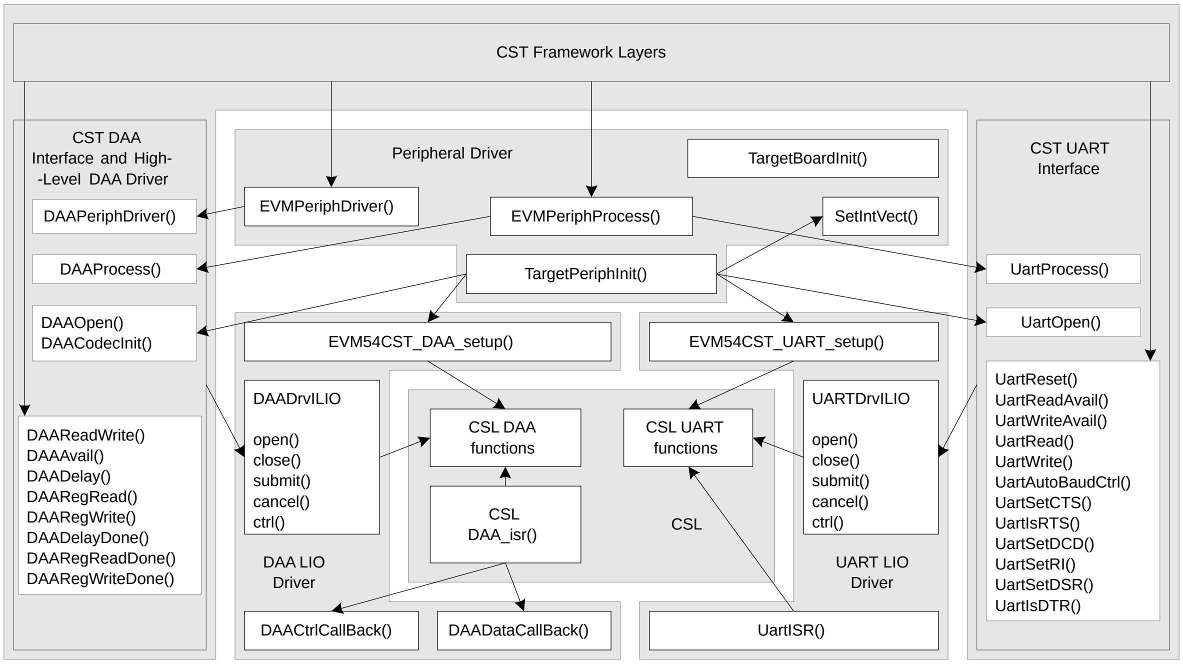
7.7.1 Overview, Interface Functions and Function Call Diagram

There are several drivers used in CST: a high-level DAA driver, a low-level (LIO) DAA driver, a low-level (LIO) UART driver, and a peripheral driver. The high-level DAA driver is hardware-independent and it is a part of the CST Framework. The other drivers are hardware-specific.

The CST Framework defines prototypes for a few peripheral drivers' functions and contains a number of interface functions to access to the low-level (LIO) DAA and UART drivers from the CST Framework itself.

Figure 7-6 illustrates the CST Framework and the CST driver subsystem and their interconnections. The arrows denote the calls between the functions and function blocks.

Figure 7-6. CST Drivers Function Call Diagram



Each of the drivers shown on the diagram is described in the later sections of this document.

7.7.1.1 CST DAA Interface Functions. Files DAADrv.c, DAADrv.h

The CST Framework defines a number of interface functions to access to the low-level (LIO) DAA driver. These functions are provided mainly to simplify invocation of the LIO driver functions (see section 7.7.4) inside the CST Framework. These functions do not contain any extra logic and serve as a bridge to the LIO driver functions.

Table 7-44. CST DAA Interface Functions

Function	**Description**
`DAAOpen`	Opens a DAA I/O channel
`DAAReadWrite`	Reads and writes a number of samples from/to the DAA channel
`DAAAvail`	Returns count of samples that can be read/written from/to the DAA channel
`DAADelay`	Starts a delay in the DAA driver
`DAARegRead`	Starts a DAA device hardware register read
`DAARegWrite`	Starts a DAA device hardware register write
`DAADelayDone`	Checks if the delay completed
`DAARegReadDone`	Checks if the DAA device hardware register read completed and returns the register value
`DAARegWriteDone`	Checks if the DAA device hardware register write completed

The other 3 DAA functions belong to the high-level DAA driver and are described in a greater detail in section 7.7.3.4:

Table 7-45. High-Level DAA Driver Functions

Function	**Description**
`DAACodecInit`	High-level DAA driver initialization. It does not include hardware inititalization.
`DAAProcess`	Performs periodic background DAA operations.
`DAAPeriphDriver`	Executes a peripheral command

DAAOpen function

Function `void tpVoid DAAOpen (tpVoid DaaChanHandle);`

Parameter(s)

`DaaChanHandle`	Pointer to an LIO channel object.

Type `tpVoid` is a pointer to void, defined in `CSTCommon.h`.

Return Value Channel handle, pointer to the channel state object/structure (the same value as `DaaChanHandle`) on success. NULL if function failed.

See also `EVM54CSTDrv.c`.

DAAReadWrite Function

Function `bool DAAReadWrite (tpVoid DaaChanHandle, int *pbuf, int count);`

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`
`pbuf`	Pointer to the user buffer. The buffer must contain `count` samples to be output through the DAA. If the function is accepted, the buffer will be filled with newly obtained samples from the DAA. This is because input and output through the DAA are synchronous processes, so we can use the same buffer for I/O.
`count`	Amount of samples to be input/output

Return Value Nonzero, if the operation has been accepted and there’re new samples in the buffer. Zero, if the operation has been denied (e.g. there’s not enough samples to read from the DAA) and the user should try again later.

DAAAvail Function

Function `int DAAAvail (tpVoid DaaChanHandle);`

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`

Return Value Number of samples that can be read off the DAA and written to the DAA by the `DAAReadWrite()` function. It’s not necessary to call this function before `DAAReadWrite()` since the latter function makes sure there’s enough samples to be read off the DAA before accepting the user’s buffer.

DAADelay Function

This function starts a delay of specified number of samples in the DAA driver. This delay does not affect the I/O process, nor does it sit in a tight loop. It just starts the process, counting samples. Upon passing of the specified number of samples (in either direction) through the DAA, the delay will be completed. To find out if the delay completed, use the `DAADelayDone()` function.

Function

```
bool DAADelay (tpVoid DaaChanHandle, int count);
```

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`
`count`	Number of samples, making up the delay time

Return Value Nonzero, if the delay started. Zero, if the delay can’t be started because of another delay being in progress.

DAARegRead Function

Function

```
bool DAARegRead (tpVoid DaaChanHandle, unsigned int reg);
```

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`
`reg`	DAA hardware register number

Return Value Nonzero, if the register read started. Zero, if the read can’t be started because there is another register read or write in progress.

DAARegWrite Function

Function

```
bool DAARegWrite (tpVoid DaaChanHandle, unsigned int reg,
unsigned int value);
```

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`
`reg`	DAA hardware register number
`value`	New value for the register

Return Value Nonzero, if the register write started. Zero, if the write can’t be started because there is another register write or read in progress.

DAADelayDone Function

Function `bool DAADelayDone (tpVoid DaaChanHandle);`

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`

Return Value Nonzero, if the delay, initiated by `DAADelay()`, has completed. Zero, otherwise.

DAARegReadDone Function

Function `int DAARegReadDone (tpVoid DaaChanHandle);`

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`

Return Value Nonnegative value, the value of the register, if the register read has completed. Negative value otherwise.

DAARegWriteDone Function

Function `bool DAARegWriteDone (tpVoid DaaChanHandle);`

Parameter(s)

`DaaChanHandle`	Channel handle, returned by `DAAOpen()`

Return Value Nonzero, if the register write has completed. Zero otherwise.

7.7.1.2 CST UART Interface Functions. Files UartDrv.c, UartDrv.h

The CST Framework defines a number of interface functions to access to the low-level (LIO) UART driver. These functions are provided mainly to simplify invocation of the LIO driver functions (see 7.7.4) inside the CST Framework. These functions do not contain any extra logic and serve as a bridge to the LIO driver functions.

Table 7-46. CST UART Interface Functions

Function	Description
`UartOpen`	Opens input and output UART channels
`UartReset`	Resets one or both channels
`UartReadAvail`	Returns count of characters that can be read from the input channel
`UartWriteAvail`	Returns count of characters that can be written to the output channel
`UartRead`	Reads a number of characters from the input channel
`UartWrite`	Writes a number of characters to the output channel
`UartProcess`	Periodic UART process function. Takes care of the hardware control flow and related to it functions.
`UartAutoBaudCtrl`	Enables/disables the autobaud function. The autobaud function helps to set up a correct baud rate if the baud rates of the two connected UARTs mismatch
`UartSetCTS`	Sets the CTS pin to a specified state
`UartIsRTS`	Reads and returns the state of the RTS pin
`UartSetDCD`	Sets the DCD pin to a specified state
`UartSetRI`	Sets the RI pin to a specified state
`UartSetDSR`	Sets the DSR pin to a specified state
`UartIsDTR`	Reads and returns the state of the DTR pin

UartOpen Function

Function `void UartOpen (tpVoid* pUartRxChanHandle, tpVoid* pUartTxChanHandle);`

Parameter(s)

`pUartRxChanHandle`	Pointer to a pointer to an LIO UART input channel object
`pUartTxChanHandle`	Pointer to a pointer to an LIO UART output channel object

Type `tpVoid` is a pointer to void, defined in `CSTCommon.h`.

Return Value None

See also `EVM54CSTDrv.c`.

UartReset Function

Function `void UartReset (tpVoid UartRxChanHandle, tpVoid UartTxChanHandle);`

Parameter(s)

`UartRxChanHandle`	Pointer to an LIO UART input channel object if the input channel needs to be reset, NULL if the reset is not needed.
`UartTxChanHandle`	Pointer to an LIO UART output channel object if the output channel needs to be reset, NULL if the reset is not needed.

Return Value None

UartReadAvail Function

Function `int UartReadAvail (tpVoid UartRxChanHandle);`

Parameter(s)

`UartRxChanHandle`	Pointer to an LIO UART input channel object

Return Value Number of characters that can be read from the input UART channel.

UartWriteAvail Function

Function `int UartWriteAvail (tpVoid UartTxChanHandle);`

Parameter(s)

`UartTxChanHandle`	Pointer to an LIO UART output channel object

Return Value Number of characters that can be written to the output UART channel.

UartRead Function

Function `int UartRead (tpVoid UartRxChanHandle, unsigned char *pbuf, int count);`

Parameter(s)

`UartRxChanHandle`	Pointer to an LIO UART input channel object
`pbuf`	Pointer to the user's buffer
`count`	Number of characters to be read and put into the buffer

Return Value Zero if there are no `count` characters available yet, the user's buffer is empty and the user should try calling this function again later. Nonzero if `count` samples have been written to the user's buffer.

UartWrite Function

Function `int UartWrite (tpVoid UartTxChanHandle, const unsigned char *pbuf, int count);`

Parameter(s)

`UartTxChanHandle`	Pointer to an LIO UART output channel object
`pbuf`	Pointer to the user's buffer
`count`	Number of characters in the user's buffer to be output

Return Value Zero if there is no room for `count` characters available yet in the driver's FIFO, and the user should try calling this function again later. Nonzero if `count` samples have been written to the user's buffer.

UartProcess Function

Function `void UartProcess (tpVoid UartRxChanHandle, tpVoid UartTxChanHandle);`

Parameter(s)

`UartRxChanHandle`	Pointer to an LIO UART input channel object
`UartTxChanHandle`	Pointer to an LIO UART output channel object

Return Value None

UartAutoBaudCtrl Function

Function `int UartAutoBaudCtrl (tpVoid UartRxChanHandle, int enable);`

Parameter(s)

`UartRxChanHandle`	Pointer to an LIO UART input channel object
`enable`	1 to enable the autobaud function or 0 to disable the autobaud function

Return Value Nonzero on success, zero on failure.

UartSetCTS Function

Function `void UartSetCTS (tpVoid UartRxTxChanHandle, int state);`

Parameter(s)

`UartRxTxChanHandle`	Pointer to an LIO UART input or output channel object
`state`	1 to set the pin high, 0 to set the pin low

Return Value None

UartIsRTS Function

Function `int UartIsRTS (tpVoid UartRxTxChanHandle);`

Parameter(s)

`UartRxTxChanHandle`	Pointer to an LIO UART input or output channel object

Return Value Zero if pin state is low, nonzero if pin state is high.

UartSetDCD Function

Function `void UartSetDCD (tpVoid UartRxTxChanHandle, int state);`

Parameter(s)

`UartRxTxChanHandle`	Pointer to an LIO UART input or output channel object
`state`	1 to set the pin high, 0 to set the pin low

Return Value None

UartSetRI Function

Function `void UartSetRI (tpVoid UartRxTxChanHandle, int state);`

Parameter(s)

`UartRxTxChanHandle`	Pointer to an LIO UART input or output channel object
`state`	1 to set the pin high, 0 to set the pin low

Return Value None

UartSetDSR Function

Function `void UartSetDSR (tpVoid UartRxTxChanHandle, int state);`

Parameter(s)

`UartRxTxChanHandle`	Pointer to an LIO UART input or output channel object
`state`	1 to set the pin high, 0 to set the pin low

Return Value None

UartIsDTR Function

Function `int UartIsDTR (tpVoid UartRxTxChanHandle);`

Parameter(s)

`UartRxTxChanHandle`	Pointer to an LIO UART input or output channel object

Return Value Zero if pin state is low, nonzero if pin state is high.

7.7.2 Peripheral Driver. Files CSTPeriph.h, EVM54CSTDrv.c, EVM54CSTDrv.h

7.7.2.1 Task of the Peripheral Driver

The peripheral driver is dedicated to carrying out the basic peripheral functions such as managing the hook state of the DAA (going off-hook/on-hook and dialing a digit (in pulse mode)), detecting ring signals and line reversals and more. The peripheral driver makes it easy to perform these generic telephony functions. The peripheral driver is also used to perform the hardware-specific initialization of CST and handling the hardware specific for the EVM54CST (LED signaling).

7.7.2.2 Set of Commands For Peripheral Driver

The CST Service peripheral driver interface provides the user with a set of standard commands to be executed in a background task. The Service can execute only one peripheral command at a time. If the user wants to send a new command to CST Service Peripheral Driver, he needs to continually try to push the command until the driver accepts it. Available peripheral driver commands are defined in the `tPeriphDriverCommand` enumeration.

Enum Definition

```
typedef enum tPeriphDriverCommand {
```

Table 7-47. Set of Peripheral Driver Commands

Value	Name	Description
0	pdc_OFF_HOOK	Go off hook. No parameters
1	pdc_ON_HOOK	Go on hook. No parameters
2	pdc_ON_HOOK_LINE_ MONITOR	Go to special phone line monitor state for Caller ID (in this state, DAA can listen to phone line signal while staying on-hook). No parameters
3	pdc_PULSE_GEN	Dial a digit in pulse mode. 1st parameter – digit to dial, from 0 to 9
4	pdc_READ_REG	Read DAA hardware register. 1st parameter – register number, from 0 to 18. Detailed description of DAA registers is given in the documents *TMS320C54CST Client Side Telephony* (SPRS187) *and Si3044 User Guide. 3.3 V ENHANCED GLOBAL DIRECT ACCESS ARRANGEMENT,* Silicon Laboratories, 2000. .

Table 7-47. Set of Peripheral Driver Commands (Continued)

Value	Name	Description	
5	pdc_WRITE_REG	Write DAA hardware register. 1st parameter – register number, from 0 to 18. 2nd parameter - value to write to the register Detailed description of DAA registers is given in *TMS320C54CST Client Side Telephony* (SPRS187).	
6	pdc_LED_SIGNAL	Indicate some event by LEDs on EVM board (this command is specific for C54CST EVM). 1st parameter – event, defined in `tLedSignalEvents` enumeration. Can be one of the following:	
		lse_BOOT_ILLUMINATION implemented yet.	Blink all LEDs at power on; not
		lse_DAA_HANDSET_RESET	Indicate DAA or Handset codec buffer overflow
		lse_UART_RX_RESET	Indicate UART RX buffer reset
		lse_UART_TX_RESET	Indicate UART TX buffer reset
		lse_UART_CTS_OFF	CTS circuit turned off (Host should wait)
		lse_UART_CTS_ON	CTS circuit turned on (Host can send data)
		lse_VOICE_UNDERRUN	Voice buffer underrun in voice controller
		lse_IDLE_START	DSP entered IDLE1 mode
		lse_IDLE_END	DSP left IDLE mode
6	pdc_GET_UART_ STATUS_LINES	Reserved. Not used in the current version	

```
} tPeriphDriverCommand;
```

Type tPeriphDriverCommand is defined in CSTPeriph.h.

7.7.2.3 Set of Events From Peripheral Driver

Upon detection of an event (for example, a ring), the CST Service generates a message originating from the `cstst_PERIPH` task (see section 7.1.1.3). Available peripheral driver events are defined in the `tCSTPeriphEvent` enumeration and described in Table 7-48.

Enum Definition typedef enum tCSTPeriphEvent {

Table 7-48. Set of Events From the Peripheral Driver

Value	Name	Description
0	cpe_NONE	No event
1	cpe_RING	Ring signal detected
2	cpe_RING_END	Ring signal lost
3	cpe_AUTO_RING_END	Automatic ring-end event generated upon going off-hook.
4	cpe_LINE_REVERSAL	Line reversal detected

```
} tCSTPeriphEvent;
```

Type tCSTPeriphEvent is defined in CSTPeriph.h.

7.7.2.4 Peripheral Driver Function Interface

The CST Framework declares several platform-specific peripheral driver functions to be called along with the whole CST initialization. The `CSTAction_Init()` function (see 7.3.4.1) performs an internal platform-independent CST initialization. The user is responsible for the specific hardware initialization, implementing and setting interrupt service routines, etc. The CST Framework offers an example of such platform-specific peripheral driver functions for the following hardware: TMS320C54CST chip and EVM board. These functions are described in Table 7-49.

Table 7-49. Peripheral Driver Function Interface

Name	Functionality
`TargetBoardInit`	To be implemented according to hardware specifics. Intended for primary hardware initialization.
`TargetPeriphInit`	To be implemented according to hardware specifics. Intended for final hardware initialization. In the CST Framework it initializes interrupt management, DAA and UART peripheral and data controllers.
`SetIntVect`	Set new interrupt service routine

Primary Hardware Initialization

To be implemented according to hardware specifics. Intended for primary hardware initialization.

Function

```
void TargetBoardInit (bool IsBIOSUsed, int Multiplier, int
ExtWaitStates);
```

Parameter(s)

`IsBIOSUsed`	Tells this function not to initialize DSP peripheral registers (CLKMD, BSCR, SWCR, SWWSR). Should be non-zero if DSP/BIOS is used as these registers are normally configured through the DSP/BIOS configuration.
`Multiplier`	Multiplier for DSP Clock PLL (external clock applied to DSP will be multiplied by this value, and DSP will run at the resulting frequency). The function presets the CLKMD DSP register. For C54CST's DAA to operate properly, input clock should be 14.7456 MHz. The multiplier should be 4 or 8, resulting in 59 or 118 MHz CPU clock.
`ExtWaitStates`	Amount of wait states to access external memory, both in program and data space (the amount of wait states to access to I/O space is not affected by this parameter and it is set to 7). The function presets the SWWSR DSP register.

Return Value None

Final Hardware Initialization

To be implemented according to hardware specifics. Intended for final hardware initialization. In the CST Framework it initializes interrupt management subsystem, DAA and UART peripherals and overrides CSTFxns.pPeriphProcess and CSTFxns.pPeriphDriver methods (see sections 6.3.8 and 7.2.2.1).

Function `void TargetPeriphInit (bool IsBIOSUsed, int TimerToBeUsed;`

Parameter(s)

`IsBIOSUsed`	Tells this function not to initialize the interrupt-related registers, interrupt vector table and service routine. Should be non-zero if DSP/BIOS is used as all interrupt management is normally configured by the DSP/BIOS configuration.
`TimerToBeUsed`	Select timer for optional system functions (MIPS measurement): `1` - timer 1; `0` - timer 0; -1 - disabled. These timer-related system functions are available in a non-BIOS environment only.

Return Value None

Setting an ISR

Set a new Interrupt service Routine. Used only in non-DSP/BIOS mode.

When DSP/BIOS is used, interrupts should be set through a DSP/BIOS configuration.

Function `void SetIntVect (int Number, void (*pIntVector) ());`

Parameter(s)

`Number`	Interrupt vector number (0 to 31)
`pIntVector`	Address of the Interrupt service Routine. The routine should not be declared as `interrupt`, because the CST software provides a single interrupt entrance for all vectors.

Return Value None

See also `EVM54CSTDrv.c`, `EVM54CSTDrv.h`.

7.7.3 High-Level DAA Driver. Files DAADrv.c, DAADrv.h

7.7.3.1 Task of the High-Level DAA Driver0

DAA operations can be split in to two parts: hardware-specific and hardware-independent. The independent part is provided by the high-level DAA driver. The CST Service calls only the high-level DAA driver functions. In other words, the Service never interacts directly with the low-level hardware DAA driver. It is the high-level DAA driver that interacts with the low-level hardware driver via low level I/O interface (LIO). The high-level DAA driver can execute a set of standard operations by processing special command scripts. The scripts specify sequences of commands; each sequence is hardware specific, as its commands may read/write/analyze different DAA hardware registers (see *TMS320C54CST Client Side Telephony* (SPRS187)).

7.7.3.2 Set of Standard Operations

Each operation corresponds to a standard script to be supplied by the low-level DAA driver. See also 7.7.3.3.

Enum Definition `typedef enum tDAAStdRequest {`

Table 7-50. Set of Standard Operations of High-Level DAA Driver

Value	Name	Description
0	dsr_ON_HOOK_LINE_ MONITOR	Go on hook, but enable line monitoring
1	dsr_ON_HOOK	Go on hook
2	dsr_OFF_HOOK	Go off hook
3	dsr_RING_DETECTION	Check ring
4	dsr_BEGIN_PULSING	Begin pulsing for a digit (assuming first part of inter-digit pause)
5	dsr_SINGLE_PULSE	Single pulse
6	dsr_END_PULSING	End pulsing for a digit (assuming second part of inter-digit pause)

```
} tDAAStdRequest;
```

Type tDAAStdRequest is defined in DAADrv.h.

7.7.3.3 Set of commands

There is a set of commands the high-level DAA driver can execute. These commands compose the scripts, specifying how to perform the high-level DAA driver standard operations. Most of the commands have a parameter.

Structure `typedef struct {`

Table 7-51. High-level DAA Driver Commands to Compose Scripts

Type	Name	Description
`tCodecDrvStageSwitch`	`Switch`	Command code
`int`	`Param`	Parameter

`} tCodecDrvStage;`

Type tCodecDrvStage is defined in DAADrv.h.

Addresses of the scripts should be put into an array, whose address should be stored in a global pointer, `tCodecDrvStage **pDAAStdRequests` (see `DAADrv.c` and `DAADrv.h`). By default, the scripts (for the C54CST's DAA) are contained in the array `tCodecDrvStage *apDAAStdRequests[]` and `pDAAStdRequests` is made pointing to it during initialization (see 7.7.5.2).

Enum Definition typedef enum tCodecDrvStageSwitch {

Table 7-52. Set of Commands of High-Level DAA Driver

Value	Name	Description
0	cdss_READ_REG	Read a hardware DAA register (Rx) to a working register (X) –The working register is just a variable. The old value of the working register (e.g. value before reading DAA register) is saved into another variable (X_{-1}) for other uses. The parameter (x) contains the DAA register number: $X_{-1} = X$ X = Rx
1	cdss_WRITE_REG	Write a hardware DAA register from the working register, the parameter (x) contains the DAA register number: Rx = X
2	cdss_SAVE_REG_VALUE	Copy the working register into a 'saving' array. The parameter (x) contains the index into the 'saving' array: Save[x] = X
3	cdss_RESTORE_REG_VALUE	Copy a value from the 'saving' array into a working register. The parameter (x) contains the index into the 'saving' array: X = Save[x]

Table 7-52. Set of Commands of High-Level DAA Driver (Continued)

Value	Name	Description
4	cdss_DO_IF_MASK	Calculate bitwise AND of the working register and a mask defined in the parameter (mask). The working register stays unchanged. If the result of the calculation is zero, the script execution terminates: if ((X & mask) == 0) then break
5	cdss_AND_MASK	Bitwise mask the working register. The parameter (mask) defines the mask value: X &= mask
6	cdss_OR_MASK	Set bits in the working register according to a mask. The parameter (mask) defines the mask value: X |= mask
7	cdss_SET_MASK	Set the working register to a value. The parameter (mask) defines the value: X = mask
8	cdss_WAIT	Unconditional wait. The parameter defines the time in 8KHz samples to wait before a next command will start.
9	cdss_WAIT_DIFFERENCE	Repeat previous command (normally, `cdss_READ_REG`) and wait until the previous (X_{-1}) and current values (X) of the working register differ or there's a timeout. If there's a timeout, the script execution terminates: repeat until timeout or X_{-1} == X if timeout break
10	cdss_SET_RESULT	Set the result of execution of the script but don't terminate the script yet
11	cdss_PDC_READ_REG	Special implementation of cdss_READ_REG for the peripheral driver's command `pdc_READ_REG` (see 7.7.2.2): X = Rx
12	cdss_PDC_WRITE_REG	Special implementation of cdss_WRITE_REG for the peripheral driver's command `pdc_READ_REG` (see 7.7.2.2): Rx = X
13	cdss_NONE	Script terminator. Must end each script

```
} tCodecDrvStageSwitch;
```

Type tCodecDrvStageSwitch is defined in DAADrv.h.

Note that the only allowed numbers of DAA registers are 0 through 18, that is, the high-level DAA driver supports maximum or 19 registers.

See also `DAADrv54CST.c`, `DAADrv54CST.h`, `Si3044Stages.c`, `EVM54CSTDrv.c`.

7.7.3.4 High-Level DAA Driver Function Interface

Table 7-53. High-Level DAA Driver Function Interface

Name	Functionality
`DAACodecInit`	High-level DAA driver initialization. It does not include hardware inititalization.
`DAAProcess`	Performs periodic background DAA operations such as ring detection, hook control, etc. Returns a peripheral event (see 7.7.2.3). This routine is dynamically called via `pPeriphProcess` method (see 7.7.2.1)
`DAAPeriphDriver`	Executes a peripheral command (see 7.7.2.2) and returns a result of the execution. This routine is dynamically called via `pPeriphDriver` method (see 7.7.2.1)

Initialization

High-level DAA driver initialization. It does not include low-level hardware DAA initialization.

Function `void DAACodecInit (tCSTChannel* pChannel);`

Parameter(s)	pChannel	Pointer to a global CST channel structure

Return Value None

DAA Periodic Routine

Performs periodic background DAA driver operations. Returns a peripheral event (see 7.7.2.3). This routine is called from `EVMPeriphProcess()`, which, in turn, is dynamically called via `pPeriphProcess` method (see 7.7.2.1)

Function `tCSTPeriphEvent DAAProcess (tCSTChannel* pChannel, int AmountOf8KHzSamples);`

Parameter(s)	pChannel	Pointer to a global CST channel structure
	`AmountOf8KhzSamples`	Time stamp in 8kHz samples that informs the time passed since last call

Return Value A peripheral event (see 7.7.2.3)

DAA Driver Command Execution

Executes a peripheral command and returns a result of the execution.This routine is called from `EVMPeriphDriver()`, which, in turn, is dynamically called via `pPeriphDriver` method (see 7.7.2.1)

Function

```
long DAAPeriphDriver (tCSTChannel* pChannel,
tPeriphDriverCommand Command, int Param1, int Param2);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`Command`	A peripheral command (see 7.7.2.2, commands from 0 to 5)
`Param1`	First auxiliary parameter for the command
`Param2`	Second auxiliary parameter for the command

Return Value

Result of the command execution. Zero means that the command has not yet finished executing (the user has to send the command again to push the process). Nonzero result means that the execution has completed. For example, when the user sends `pdc_PULSE_GEN` command to dial a digit in pulse mode, the driver will return zero until the dialing of this digit is completed.

If the command is to read a DAA hardware register (`pdc_READ_REG`), the returned 32-bit integer value will contain the result of the execution in the high word and the read register value in the low word. If the high and low words are equal to zero, the register has not been read yet. Otherwise, the high word becomes non-zero and low word contains the register value.

7.7.4 Brief Description of the Low-level I/O (LIO) Interface

This section contains information on the LIO interface used in the CST device drivers. This information is key to understanding the CST drivers. Additional information on the LIO interface and writing device drivers for block I/O can be found in *Writing DSP/BIOS Device Drivers for Block I/O* (SPRA802).

The LIO interface is intended to be a simple uniform interface for drivers. The interface makes it easy to integrate drivers for new devices, override driver methods and alter their functionality, even at run time.

The LIO interface is defined by a function table, which consists of 5 function pointers. Each pointer points to a dedicated function and can be changed.

The function table’s structure is shown in Table 7-54.

Structure

```
typedef struct LIO_Fxns {
```

Table 7-54. LIO Function Table

Function Type	Function Name	Description
`LIO_Tcancel`	`cancel`	Cancels all I/O jobs started by the submit() function.
`LIO_Tclose`	`close`	Closes an I/O channel.
`LIO_Tctrl`	`ctrl`	A control function to carry out implementation-specific operations.
`LIO_Topen`	`open`	Opens an I/O channel.
`LIO_Tsubmit`	`submit`	Submits a buffer for I/O and starts an I/O process. A completion notification will be delivered to a callback function registered by the open() function.

```
} LIO_Fxns;
```

Type LIO_Fxns is defined in LIO.h.

LIO Open Function

The function initializes a channel object for specified direction of data flow (input vs. output), handles the name parameter and implementation-specific argument, stores the callback function address and callback function argument in the channel object and marks the channel object as in use.

Once the channel has been opened, it may then be used for I/O with the `submit()` function. Normally, after the processing initiated by `submit()` completes, the callback function will be called with the callback function argument to notify the user.

The channel may be closed when there's no need to continue I/O. This is done by the `close()` function.

Function

```
Ptr open (String name, LIO_Mode mode, Arg arg, LIO_Tcallback cb, Arg cbArg);
```

Parameter(s)

`name`	Can be used to specify channel ID
`mode`	Specifies mode (input vs. output) of the channel
`arg`	Implementation-specific argument
`cb`	A callback function to be called with `cbArg` argument when I/O operation completes
`cbArg`	An argument to be used when calling the callback function

Note that the types `Void`, `Bool`, `String`, `Uns`, `Arg` and `Ptr` are defined in the file `std.h`.`Uns` is some unsigned integer type, `Ptr` is a void pointer, `Arg` is a type that is big enough to hold either of an integer and a pointer.

Enum Definition

```
typedef enum LIO_Mode {
```

Table 7-55. LIO Open Function Modes

Name	Value	Description
LIO_INPUT	0	Open channel in input mode
LIO_OUTPUT	1	Open channel in output mode

```
} LIO_Mode;
```

Type LIO_Mode is defined in LIO.h.

Return Value Channel handle, pointer to the channel state object/structure, or NULL if function failed.

LIO Close Function

The function closes a previously opened channel and marks the channel object as not in use. The channel may be opened with `open()` once again after it has been closed by `close()`.

Function

```
Bool close (Ptr chanp);
```

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.

Return Value TRUE on success, FALSE if failed.

LIO Submit Function

The function takes the user buffer to be output (or input to) and starts the I/O process. Upon completion of the process, the callback function (which has been previously registered in the open() function) will be called to notify the user.

The ongoing I/O process may be stopped by the `cancel()` function.

The `submit()` function should be callable from an ISR.

Function

```
Bool submit (Ptr chanp, Ptr buf, Uns nmaus);
```

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.
`buf`	User’s buffer with or for data
`nmaus`	Size of the buffer in MAUs. MAU is a minimum addressable unit, whose size is usually equal to sizeof(char) in C.

Return Value TRUE if the request has been taken and the I/O process started, FALSE if the request can’t be satisfied.

LIO Cancel Function

The function is intended to stop an ongoing I/O process, initiated by the `submit()` function.

Function `Bool cancel (Ptr chanp);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.

Return Value TRUE if the job initiated by `submit()` has been successfully stopped or cancelled, FALSE otherwise.

LIO ctrl Function

The function is dedicated to carry out implementation-specific operations. Any driver functions that are beyond the interface of the open(), close(), submit() and cancel() functions should be done through the use of this function.

For example, for a DAA device it would be logically to implement hardware registers reading and writing in this function.

If the operations implemented in this function are needed inside of the other LIO functions, it may be desirable to call this control function via a pointer from the driver’s LIO function table (see Table 7-54).

Function `Bool ctrl (Ptr chanp, Uns cmd, Arg arg);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.
`cmd`	Implementation-defined command parameter
`arg`	Implementation-defined argument parameter

Return Value TRUE if the implementation-defined command has been successfully accepted, FALSE otherwise.

LIO User’s Callback Function

The user of the LIO driver should provide the callback function to find out when the I/O processes complete.

Normally, the callback function is called from the driver’s ISR. The function should do whatever is required to start getting the newly obtained data (if the channel is in input mode) or prepare new data to be sent (if the channel is in output mode).

The function and its argument (the first function argument) are registered in the `open()` function when opening the channel. The function will be called with the registered argument.

Function `Void callback (Arg arg, Uns nmaus);`

Parameter(s)

`Arg`	Registered argument, may be the address of the data buffer
`nmaus`	size of data in MAUs. MAU is a minimum addressable unit, whose size is usually equal to sizeof(char) in C.

Return Value None

7.7.5 Low-level (LIO) DAA Driver. Files DAADrv54CST.c, Si3044Stages.c.

7.7.5.1 Task of the Low-level DAA Driver

The low-level DAA driver does the actual work with the underlying hardware, in our case C54CST’s DAA. The driver exports its functions through the Low-level I/O (LIO) interface. The driver makes use of the CSL DAA functions (see *TMS320C54x Chip Support Library API Reference Guide* (SPRU420)).

7.7.5.2 Scripts for High-Level DAA Driver. File Si3044Stages.c

The scripts, corresponding to the high-level DAA driver standard operations (see 7.7.3.2), are defined in the file `Si3044Stages.c`. These scripts are specific to the C54CST's DAA. The pointers to the scripts are contained in the `tCodecDrvStage *apDAAStdRequests[]` structure. The scripts contain register numbers specific to the C54CST's DAA and logic specific to processing their values.

7.7.5.3 Low-Level DAA Driver Hardware Setup Function. Files DAADrv54CST.c, DAADrv54CST.h

The hardware setup function is intended to perform the initialization of the DAA devices, e.g. to set up the DAA sample rate, preset analog rx/tx gain/attenuation and international registers, set up the appropriate McBSP, initialize non-static LIO channel objects, etc. This function just calls the CSL `DAA_setup()` function.

This function is to be called prior to use of any of the LIO functions (see 7.7.2.4) of the driver.

Function `void EVM54CST_DAA_setup (DAA_Setup *daaSetupStruct);`

Parameter(s)

`daaSetupStruct`	A pointer to a CSL multiple DAA device structure (Table 7-56). The structure contains number of devices to initialize and a pointer to an array of pointers to individual device setup structures.

Return Value None

A good example of using this function is available in the file `EVM54CSTDrv.c`.

Structure `typedef struct {`

Table 7-56. Multiple DAA Device Setup Structure

Field Type	Field Name	Description
`Uint16`	`numDevs`	Number of devices to be set up
`DAA_DevSetup`	`**dev`	Pointer to array of device setup structure pointers (see Table 7-54)

`} DAA_Setup;`

Type DAA_Setup is defined in csl_daa.h.

Structure `typedef struct {`

Table 7-57. DAA Device Setup Structure

Field Type	Field Name	Description
`DAA_Params`	`*params`	Pointer to a structure with DAA device parameters (initial register values, see Table 7-58, *TMS320C54CST Client Side Telephony* (SPRS187) and, *Si3044 User Guide. 3.3 V ENHANCED GLOBAL DIRECT ACCESS ARRANGEMENT.*© Silicon Laboratories, 2000)
`DAA_Handle`	`daaHandle`	Pointer to a DAA device state object created by the user
`Uint16`	`mcbspPort`	Number of an McBSP port the DAA device is connected to (MCBSP_PORT2 for internal DAA)
`Int16`	`*pCircBuf`	Pointer to a circular buffer that will contain samples for I/O
`Uint16`	`circBufSize`	Circular buffer size
`Uint16`	`circBufOffset`	Initial circular buffer offset (write pointer offset of the read pointer). Also used as the multiple of samples to skip when overflowing/underflowing, which may be crucial for modem applications.
`Uint16`	`dataLength`	Callback data size. The data callback will be called if there are this many samples available to read/write.
`void`	`*pID`	Some pointer to channel ID. In CST, this pointer points to a CST global channel structure of type `tCSTChannel` (see 6.3.1) associated with this device.
`DAA_CallBack`	`dataCallBack`	Pointer to data callback function
`DAA_CallBack`	`ctrlCallBack`	Pointer to control callback function
`DAA_RstFxn`	`reset`	Pointer to DAA device hardware reset control function.

```
} DAA_DevSetup;
```

Type DAA_DevSetup is defined in csl_daa.h.

Structure

```
typedef struct {
```

Table 7-58. Initial DAA Device Registers Values

Field Type	Field Name	Description
`Uint16`	`txAttenuation`	Analog transmit attenuation value, “OR”ed with `rxGain` defines value of the TX/RX Gain Control register (see *TMS320C54CST Client Side Telephony* (SPRS187))
`Uint16`	`rxGain`	Analog receive gain value, “OR”ed with `txAttenuation` defines value of the TX/RX Gain Control register
`Uint16`	`sampleRateReg7`	Sample Rate Control Register 7 value
`Uint16`	`sampleRateReg8`	Sample Rate Control Register 8 value
`Uint16`	`sampleRateReg9`	Sample Rate Control Register 9 value
`Uint16`	`sampleRateReg10`	Sample Rate Control Register 10 value
`Uint16`	`ictrl1`	International Control Register 1 value
`Uint16`	`ictrl2`	International Control Register 2 value
`Uint16`	`ictrl3`	International Control Register 3 value

```
} DAA_Params;
```

Type DAA_Params is defined in csl_daa.h.

DAA CSL Callback Functions

The low-level DAA driver creates two callback functions, a data callback, and a control callback. These functions will be called from the CSL ISR.

The data callback is used to notify the user of data samples availability, e.g. when it’s OK to read/write a new portion of samples.

The control callback is used to notify the user of completion of device register reads and writes. The read notification also delivers the value of the register just read. There’re two other control notifications, one for notifying the user of the circular buffer overflow (which may happen when the system goes off the real-time) and another one is for notifying the user when a delay completed (the delay function is important for going off-hook and on-hook because this takes a certain amount of time (or samples to pass through) before the new hook state becomes valid).

The DAA device setup structure (see Table 7-57) should have pointers to these callback functions.

CSL DAA Callback Function Prototype

```
void CallBack (void* pID, Uint16 task, Uint16 arg);
```

Function

Parameter(s)

`pID` — Pointer to channel ID, given in the DAA device setup structure (see Table 7-57). In CST, this pointer points to a CST global channel structure of type `tCSTChannel` (see 6.3.1) associated with this device.

`task` — For data callback:

- ❑ DAA_DATA – there’re samples ready to be read/written

For control callback:

A bit field with the following bits set/reset:

- ❑ DAA_OVERFLOW – the circular buffer overflowed notification
- ❑ DAA_REG_READ – a DAA device register read completed
- ❑ DAA_REG_WRITE – a DAA device register write completed
- ❑ DAA_DELAY – a delay completed

`arg` — For data callback:

Pointer to a DAA device state object created by the user. The pointer of type `DAA_Handle` typecast to `Uint16`.

For control callback:

Register value just read, if `(task & _DAA_REG_READ)!=0`

Table 7-59. Bit Fields of the Task Parameter

Bit Field Name	Bit Field Mask Value
`_DAA_REG_READ`	`0x0001`
`_DAA_REG_WRITE`	`0x0002`
`_DAA_DATA`	`0x0004`
`_DAA_DELAY`	`0x0010`
`_DAA_OVERFLOW`	`0x0020`

The bit fields are defined in `csl_daa.h`.

Return Value None

These CSL DAA callbacks are implemented in the functions `DAADataCall-Back()` and `DAACtrlCallBack()`.

DAA CSL Device Hardware Reset Control Function

During the initialization of the DAA devices, the reset control functions are used to put the devices into the reset state and take them back out of the rest as part of normal initialization procedure. The CSL provides one reset function for the internal C54CST DAA device. This is function `DAA_reset()` and it is a part of the CSL. In case there are external Si3021 DAAs connected to the CST chip, the individual DAA hardware reset functions should be provided for the extra DAAs.

The DAA device setup structure (see Table 7-57) should have a pointer to this function.

7.7.5.4 CSL DAA Device Hardware Reset Control Function Prototype

```
void DAA_RstFxn (Uint16 flag);
```

Parameter(s)

`flag`	Nonzero value puts the device into the reset state; zero value takes the device out of the reset state.

7.7.5.5 LIO Functions of Low-Level DAA Driver. Files DAADrv54CST.c, DAADrv54CST.h

The 5 LIO functions (see 7.7.4) are exported in the `LIO_Fxns DAADrvILIO` structure of the driver.

DAA LIO Open Function

The function initializes a channel object and opens a channel for I/O of the DAA samples. In CST, this function is called just once during the final hardware initialization (see 7.7.2.4).

Function

```
Ptr open (String name, LIO_Mode mode, Arg arg,
LIO_Tcallback cb, Arg cbArg);
```

Parameter(s)

`name`	Can be used to specify channel ID. Ignored.
`mode`	Specifies mode (input vs. output) of the channel. Ignored (and may be either of `LIO_INPUT` and `LIO_OUTPUT`), because the DAA device is a synchronous device, which inputs and outputs data samples at the same rate. Therefore, the implementation of the `submit()` function is made such that the function takes a buffer filled with samples to be output and upon completion of the function, this same buffer will be filled with new input samples. That is, output samples are taken and replaced by new input samples.
`arg`	Implementation-specific argument. Currently, this argument is used to pass a pointer to the LIO channel object to the function. This helps to avoid unwanted allocation of static data and to make the driver fully multichannel.
`cb`	A callback function to be called with `cbArg` argument when I/O operation completes. Ignored, because CST usually asks the driver if there're enough samples to be read/written.
`cbArg`	An argument to be used when calling the callback function. Ignored, by the same reason as the above parameter.

Return Value Channel handle, pointer to the channel state object/structure (the same value as `arg`), or NULL if function failed.

DAA LIO Close Function

The function closes a previously opened DAA channel and marks the channel object as not in use. This function is never called in CST.

Function `Bool close (Ptr chanp);`

Function

`chanp`	Channel handle, previously returned by `open()`.

Return Value TRUE on success, FALSE if failed.

DAA LIO Submit Function

The function takes a buffer filled with samples to be output and upon completion of the function, this same buffer will be filled with new input samples. That is, output samples are taken and replaced by new input samples. This is because the DAA device is a synchronous device, which inputs and outputs data samples at the same rate.

Function `Bool submit (Ptr chanp, Ptr buf, Uns nmaus);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.
`buf`	Pointer to the user's buffer with data samples to be output by the device. Upon completion of the function the buffer is filled with new input samples.
`nmaus`	Size of the buffer in MAUs. MAU is a minimum addressable unit, whose size is usually equal to sizeof(char) in C. On C54xx DSPs `sizeof(char)==sizeof(int)==1`. So, this parameter specifies number of samples to be read/written.

Return Value TRUE if the request has been accepted and the buffer processed, FALSE if the request can't be satisfied (request was early or nmaus==0).

DAA LIO Cancel Function

Even though, this function should stop an ongoing I/O process, initiated by the `submit()` function, it does not do so. This is because CST always continuously inputs and outputs samples and never stops.

Function `Bool cancel (Ptr chanp);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.

Return Value FALSE

DAA LIO ctrl Function

The function is dedicated to carry out implementation-specific operations. Any driver functions that are beyond the interface of the open(), close(), submit() and cancel() functions should be done through the use of this function. There are a few of such specific operations…

Function `Bool ctrl (Ptr chanp, Uns cmd, Arg arg);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.
`cmd`	Command parameter (see Table 7-60)
`arg`	Optional argument parameter, whose meaning depends on `cmd`.

Enum Definition `typedef enum tDAADrvCmd {`

Table 7-60. DAA LIO Driver Commands

Name	Value	Description
DAA_IOAVAILABILITY	0	Command to find out how many samples can be read/written at the moment.
DAA_REG_READ	1	Command to start reading from a DAA device hardware register.
DAA_REG_WRITE	2	Command to start writing to a DAA device hardware register.
DAA_DELAY	3	Command to start a delay. The delay does not stop samples I/O nor does it affect reading and writing of the device registers. It’s more like an alarm clock or a timer.
DAA_REG_READ_DONE	4	Command to get a value of the register, if already available.
DAA_REG_WRITE_DONE	5	Command to see if the register write completed.
DAA_DELAY_DONE	6	Command to see if the delay completed.

`} tDAADrvCmd;`

Type tDAADrvCmd is defined in DAADrv.h.

Return Value Depends on the `cmd` parameter. `FALSE` if the command has not been recognized.

The following table summarizes all commands, describes the meaning and use of the optional parameter `arg` and associated returned value of the `ctrl()` function:

Table 7-61. DAA LIO Driver Parameter - Result Map

cmd	arg Used for, Treated as	arg Use	Returned Value
DAA_IOAVAILABILITY	Output, Int16*	*(Int16*)arg is assigned a count of samples that can be read/written at the moment	TRUE
DAA_REG_READ	Input, Uint16	arg is the number of the register to be read	TRUE, if register read started; FALSE if not (another read/write is in progress)
DAA_REG_WRITE	Input, tDAADrvRegWriteArg*	*(tDAADrvRegWriteArg*)arg contains the register number to be written to and its new value (see Table 7-62)	TRUE, if register write started; FALSE if not (another read/write is in progress)
DAA_DELAY	Input, Int16	`arg` is the number of samples that make up the delay time	TRUE, if delay started; FALSE if not (another delay is in progress)
DAA_REG_READ_DONE	Output, Int16*	*(Int16*)arg is assigned to the register value just read	TRUE, if read completed; FALSE if not.
DAA_REG_WRITE_DONE	Nothing	None	TRUE, if write completed; FALSE if not.
DAA_DELAY_DONE	Nothing	None	TRUE, if delay completed; FALSE if not.

Structure `typedef struct tDAADrvRegWriteArg {`

Table 7-62. DAA LIO Driver Register Write Structure

Field Type	Field Name	Description
`unsigned int`	`Reg`	DAA device hardware register number
`unsigned int`	`RegValue`	Value to be written to the register

`} tDAADrvRegWriteArg;`

Type tDAADrvRegWriteArg is defined in DAADrv.h.

DAA LIO Callback Function

CST does not use LIO callback functions for DAA because CST usually asks the driver if there're enough samples to be read/written.

7.7.6 Low-Level (LIO) UART Driver. Files Uart550Drv.c, UartAutoBaud.c

7.7.6.1 Task of the Low-Level UART Driver

The low-level UART driver does the actual work with the underlying hardware, in our case C54CST's UART. The driver exports its functions through the Low-level I/O (LIO) interface. The driver makes use of the CSL UART functions (see *TMS320C54x Chip Support Library API Reference Guide* (SPRU420)).

7.7.6.2 Low-Level UART Driver Hardware Setup Function. Files Uart550Drv.c, Uart550Drv.h

The hardware setup function is intended to perform the initialization of the UART device, e.g. to set up the UART baud rate, character size, parity settings, control flow and initialize non-static LIO channel objects. This function calls the CSL `UART_init()` function.

This function is to be called prior to use of any of the LIO functions (see 7.7.2.4) of the driver.

Function

```
void EVM54CST_UART_setup (tUartDrvSetupStruct
uartDrvSetupStruct);
```

Parameter(s)

`uartDrvSetupStruct`	This structure contains the address of the function that tracks modem escape sequence characters (see 7.2.2.1).

An example of using this function is available in the file `EVM54CSTDrv.c`.

Structure

```
typedef struct {
```

Table 7-63. UART Setup Function to Track Modem Escape Sequence Characters

Field Type	Field Name	Description
`tUARTUserFxn`	`UserFxn`	This structure contains the address of the function that tracks modem escape sequence characters

```
} tUartDrvSetupStruct;
```

Type

tUartDrvSetupStruct is defined in UartDrv.h.

Return Value

None

Default setup settings of the UART driver are summarized in the following table:

Table 7-64. Default Setup Settings of the UART Driver

Setting	Value
Baud rate	115200 (bit/s)
Character size	8 (bits)
Number of stop bits	1
Parity	None, disabled

These default settings are contained in the variables `UART_Params UartParams` and `tUartAutoBaud UartAutoBaudParams` of the driver. And if the defaults need to be changed, the changes should be made in the mentioned variables prior to calling the driver hardware setup function.

UART Rx Monitor/Escape Sequence Tracking Function

This function is called each time a new character from the UART is received. The function is intended to track special character sequences (modem escape sequences). These sequences are used to switch a modem between the data and online command modes (see 7.2.2.1).

Function

```
void UARTUserFxn (struct tCSTChannel* pChannel, char data);
```

Parameter(s)

`pChannel`	Pointer to a global CST channel structure
`data`	Received character

Return Value None

7.7.6.3 LIO Functions of Low-Level UART Driver. Files Uart550Drv.c, Uart550Drv.h

The 5 LIO functions (see 7.7.4) are exported in the `LIO_Fxns UartDrvILIO` structure of the driver.

UART LIO Open Function

The function initializes a channel object and opens a channel for input or output of the UART characters. In CST, this function is called just once during the final hardware initialization (see 7.7.2.4).

Function `Ptr open (String name, LIO_Mode mode, Arg ignored, LIO_Tcallback cb, Arg cbArg);`

Parameter(s)

`name`	Can be used to specify channel ID. Ignored.
`mode`	Specifies mode (input vs. output) of the channel. Possible values are: `LIO_INPUT`, `LIO_OUTPUT`.
`ignored`	Implementation-specific argument. Ignored.
`cb`	A callback function to be called with `cbArg` argument when I/O operation completes. Ignored, because CST usually asks the driver if it's possible to read/write a certain amount of characters.
`cbArg`	An argument to be used when calling the callback function. Ignored, by the same reason as the above parameter.

Return Value Channel handle, pointer to the channel state object/structure, or `NULL` if function failed.

UART LIO Close Function

The function closes a previously opened UART channel and marks the channel object as not in use. This function is never called in CST.

Function `Bool close (Ptr chanp);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.

Return Value TRUE on success, FALSE if failed.

UART LIO Submit Function

The function takes a buffer from the user and either fills it with received data characters (if the channel is configured for input) or takes from it data characters to be sent (if the channel is configured for output).

Function `Bool submit (Ptr chanp, Ptr bufp, Uns nmaus);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.
`bufp`	Pointer to the user's buffer with/for data characters. This pointer is treated as a pointer to an array of characters, in other words, `unsigned char*`.
`nmaus`	Size of the buffer in MAUs. MAU is a minimum addressable unit, whose size is usually equal to sizeof(char) in C. On C54xx DSPs `sizeof(char)==sizeof(int)==1`. So, this parameter specifies number of characters to be read/written. The characters are *not* packed, even though a `char` is 16-bit wide and may keep more information. The most significant 8 bits of the characters are simply ignored.

Return Value TRUE if the request has been accepted and the user buffer processed, FALSE if the request can't be satisfied (reasons: there's no room for that many characters in the driver's FIFO, there're not enough characters in the driver's FIFO).

Note: in the current implementation of the driver, the FIFO size is fixed and equal to 298 characters. E.g. `submit()` will not take more than 298 characters to be output nor will it return more than 298 input characters.

UART LIO Cancel Function

The function is intended to stop an ongoing I/O process. The function resets the channel FIFO. For input channel it also reinitializes the hardware flow control so the host may start sending data again. For output channel it stops an ongoing transmission of characters.

Function `Bool cancel (Ptr chanp);`

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.

Return Value TRUE if the function has been successfully executed, FALSE otherwise.

UART LIO ctrl Function

The function is dedicated to carry out implementation-specific operations. Any driver functions that are beyond the interface of the open(), close(), submit() and cancel() functions should be done through the use of this function. There are a few of such specific operations…

Function

```
Bool ctrl (Ptr chanp, Uns cmd, Arg arg);
```

Parameter(s)

`chanp`	Channel handle, previously returned by `open()`.
`cmd`	Command parameter (see Table 7-65)
`arg`	Optional argument parameter, whose meaning depends on `cmd`.

Enum Definition

```
typedef enum tUartDrvCmd {
```

Table 7-65. UART LIO Driver Commands

Name	Value	Description
`UART_RESET`	0	Command to reset the channel FIFO. For an input channel, this also reinitializes the hardware control flow so the host may start sending data again. For an output channel, this stops an ongoing character transmission as well.
`UART_IOAVAILABILITY`	1	Command to find out how many characters can be read (for input channel) or written (for output channel) at the moment.
`UART_AUTOBAUD_FLAG`	2	Enables the autobaud function (applicable for an input channel only). The autobaud function tries to find and set the correct baud rate if the baud rates of the two connected UARTs mismatch. By default, the autobaud function is disabled and to enable it this command should be explicitly given to the driver.
`UART_PROCESS`	3	Command to perform background UART processes: hardware flow control related functions. This includes restarting transmission when the host sets RTS high again and setting CTS high when there's enough space in the input FIFO again.
`UART_CTS_PIN`	4	Sets the CTS pin high or low.
`UART_RTS_PIN`	5	Returns the state of the RTS pin.
`UART_DCD_PIN`	6	Sets the DCD pin high or low.
`UART_RI_PIN`	7	Sets the RI pin high or low.
`UART_DSR_PIN`	8	Sets the DSR pin high or low.
`UART_DTR_PIN`	9	Returns the state of the DTR pin.

```
} tUartDrvCmd;
```

Type tUartDrvCmd is defined in UartDrv.h.

Return Value Depends on the `cmd` parameter. `FALSE` if the command has not been recognized.

The following table summarizes all commands, describes the meaning and use of the optional parameter `arg` and associated returned value of the `ctrl()` function:

Table 7-66. UART LIO Driver Parameter - Result Map

cmd	arg Used for, Treated as	arg Use	Returned Value
UART_RESET	Nothing	None	TRUE, if successful reset; FALSE otherwise.
UART_IOAVAILABILITY	Output, Int16*	*(Int16*)arg is assigned a count of characters that can be read (if input channel) or written (if output channel) at the moment	TRUE
UART_AUTOBAUD_ FLAG	Input, tUartDrvCmdArg	if (arg==ENABLE) enables the autobaud function, otherwise disables it	TRUE, if successful enabling/disabling; FALSE otherwise.
UART_PROCESS	Nothing	None	TRUE
UART_CTS_PIN	Input, tUartDrvCmdArg	if (arg==PIN_ON) sets the CTS pin high, otherwise sets it low	TRUE
UART_RTS_PIN	Nothing	None	RTS pin state
UART_DCD_PIN	Input, tUartDrvCmdArg	if (arg==PIN_ON) sets the DCD pin high, otherwise sets it low	TRUE
UART_RI_PIN	Input, tUartDrvCmdArg	if (arg==PIN_ON) sets the RI pin high, otherwise sets it low	TRUE
UART_DSR_PIN	Input, tUartDrvCmdArg	if (arg==PIN_ON) sets the DSR pin high, otherwise sets it low	TRUE
UART_DTR_PIN	Nothing	None	DTR pin state

See also `UartAutoBaud.c`, `UartAutoBaud.h`.

UART LIO Callback Function

CST does not use LIO callback functions for UART because CST usually asks the driver if it can give or take a certain amount of characters.

7.7.7 Reloading Drivers

Reloading the CST drivers is normally an easy procedure. The following sections contain information about reloading the standard CST UART, DAA and peripheral drivers as well as using multiple DAA devices in multi-channel CST flex applications.

7.7.7.1 Reloading the UART Driver

To reload the existing CST UART driver, the user should first create his own driver for UART or whatever device will be used to replace the C54CST's UART.

The following steps should be taken:

1) An LIO channel object type should be defined for both input and output channels. The channel object/structure should contain the state variables of the channel (an example of the structure is available in file Uart550Drv.h, type tUARTChanObj). Let's say, it will be tMyUartChanObj.

2) A function analogous to `EVM54CST_UART_setup()` should be implemented (see 7.7.6.2). Let's say, it will be function `My_UART_setup()`.

3) All 5 LIO functions should be implemented. Their implementation is hardware specific but these functions have to behave the same way as the original CST UART driver's LIO functions. Remember that the UART LIO `submit()` function works with 8-bit characters, which are not packed (see 7.7.6.3).

   It is important that the new driver accepts all of the commands (through the LIO `ctrl()` function) listed in the . Even if there're no certain UART lines like DTR and RI or the autobaud function is not available, the commands that correspond to these unavailable functions have to be simulated. and the code of the original UART LIO driver will help to understand how to simulate the commands. A template for the UART driver can be found in the files `DriversTemplates\MyUartDrv.c` and `DriversTemplates\MyUartDrv.h`.

4) The LIO function addresses should be put into a variable:

```
LIO_Fxns MyUartDrvILIO =
{
  &cancel,
  &close,
  &ctrl,
  &open,
  &submit
};
```

This structure will be copied to the structure `UartDrvILIO` (see `Uart550Drv.c`) in the beginning of the new peripheral driver's final hardware initialization function. If any of the LIO functions need to call the other one, they should do that by reference through the pointers of the `UartDrvILIO` structure.

5) An ISR function specific to the hardware should be implemented and it should use the LIO channel objects **either** directly (if they're statically allocated in the driver as in `Uart550Drv.c`) **or** the ISR function should take their addresses as arguments. In this latter case, a `void(void)` wrapper function may be needed to call the actual ISR with the appropriate arguments.

So, there're two options, direct and indirect access to the channel objects:

```
// direct access:
/////////////////
// channel objects defined inside the UART driver:
tMyUartChanObj MyRxUartChanObj, MyTxUartChanObj;
void MyUartIsr() // executed upon interrupt
{
  // Use MyRxUartChanObj and MyTxUartChanObj here
}
// indirect access:
///////////////////
void MyUartIsr (tMyUartChanObj* pMyRxUartChanObj,
                tMyUartChanObj* pMyTxUartChanObj)
{
  // Use *pMyRxUartChanObj and *pMyTxUartChanObj here
}
// channel objects defined outside the UART driver:
tMyUartChanObj MyRxUartChanObj, MyTxUartChanObj;
void MyUartIsrWrapper() // executed upon interrupt
{
  MyUartIsr (&MyRxUartChanObj, &MyTxUartChanObj);
}
```

6) Finally, a new peripheral driver will be needed to initialize the new hardware, set the ISR and open UART channels for I/O. For example, a new `MyTargetPeriphInit()` function (analogous to `TargetPeriphInit()`, see `EVM54CSTDrv.c`) may look like this (important parts are in bold):

```
void MyTargetPeriphInit (bool IsBIOSUsed, int TimerTo-
BeUsed)
  // if TimerToBeUsed<0, no timer will be used for MIPS
measurements
{
  tUartDrvSetupStruct UartDrvSetupStruct;
  /* Override the PeriphProcess and PeriphDriver func-
tions */
  CSTFxns.pPeriphProcess=MyPeriphProcess;
  CSTFxns.pPeriphDriver=MyPeriphDriver;
  if (!IsBIOSUsed)
  {
    /* Initialize the ISR handling code */
    IntInit();
    if(TimerToBeUsed>=0)
    {
      #undef IRQ_EVT_TINT1
      #define IRQ_EVT_TINT1 (24)
      /* Init the timer and install the timer ISR */
      CSTTimerInit(TimerToBeUsed?TIMER_DEV1:TIM-
ER_DEV0);
      SetIntVect (TimerTo-
BeUsed?IRQ_EVT_TINT1:IRQ_EVT_TINT0, &CSTTimerISR);
    }
  }
  /* Initialize the UART driver */
// default settings (115200, 8N1) are already in place,
though:
/*
  UartAutoBaudParams.ClkInput = UartParams.clkInput =
UART_CLK_INPUT_117;
  UartAutoBaudParams.AutoBaudRate = UartParams.baud =
UART_BAUD_115200;
*/
  UartDrvSetupStruct.UserFxn = CSTFxns.pUARTRxMonitor;
  // Copy LIO function table
  UartDrvILIO = MyUartDrvILIO;
  // ISR accesses UART LIO channel objects directly:
  //////////////////////////////////////////////////
  if (!IsBIOSUsed)
  {
    /* Install the UART ISR */
    SetIntVect (UART_INTERRUPT_VECTOR_NUMBER, &MyUar-
tIsr);
  }
  /* Initialize the UART hardware device */
  My_UART_setup (UartDrvSetupStruct);
  /* Open both Rx and Tx channels (LIO) */
  UartOpen (&Ch0.UartRxChanHandle, &Ch0.UartTxChan-
Handle);
  /* Initialize the DAA driver */
  ...
}
```

7.7.7.2 Reloading UART Flow Control Functions

To prevent the UART driver from using the GPIO pins HD0-HD5 for UART control lines (CTS/RTS, DSR/DTR, DCD, RI) the `tUARTDrvCtrlFxns UartDrvCtrlFxns` structure (see the file `Uart550Drv.c`) should be modified. This structure contains pointers to functions that set and read the states of the mentioned UART lines. An example of redefining these function is available as an almost empty flex application example in the file `main11 (reloading UART).c`.

7.7.7.3 Reloading the DAA Driver

To reload the existing CST DAA driver, the user should first create his own driver for DAA or whatever device will be used to replace the C54CST's DAA.

The following steps should be taken:

1) An LIO channel object type should be defined for the input/output channel (there's a single channel for both input and output, see 7.7.5.4). The channel object/structure should contain the state variables of the channel (an example of the structure is available in file DAADrv54CST.h, type tDAAChanObj). Let's say, it will be tMyDAAChanObj.

2) A function analogous to `EVM54CST_DAA_setup()` should be implemented (see 7.7.5.3). Let's say, it will be function `My_DAA_setup()`.

3) All 5 LIO functions should be implemented. Their implementation is hardware specific but these functions have to behave the same way as the original CST DAA driver's LIO functions.
   It is important that the new driver implements all of the commands (through the LIO `ctrl()` function) listed in Table 7-65 and Table 7-66 and the code of the original DAA LIO driver will help to understand how to implement the commands. A template for the UART driver can be found in the files `DriversTemplates\MyDAADrv.c` and `DriversTemplates\MyDAADrv.h`.

4) The LIO function addresses should be put into a variable:
   ```
   LIO_Fxns MyDAADrvILIO =
   {
     &cancel,
     &close,
     &ctrl,
     &open,
     &submit
   };
   ```
   This structure will be copied to the structure `DAADrvILIO` (see `DAADrv54CST.c`) in the beginning of the new peripheral driver's final hardware initialization function. If any of the LIO functions need to call the other one, they should do that by reference through the pointers of the `DAADrvILIO` structure.

5) An ISR function specific to the hardware should be implemented and it should use the LIO channel object either directly or the ISR function should take their addresses as arguments. In this latter case, a `void(void)` wrapper function may be needed to call the actual ISR with the appropriate arguments (an example of such a wrapper is the function `DAAISRWrapper()` in `EVM54CSTDrv.c`).
   So, there're two options, direct and indirect access to the channel objects:
   ```
   // direct access:
   /////////////////
   // channel object defined inside the DAA driver:
   tMyDAAChanObj MyDAAChanObj;
   void MyDAAIsr() // executed upon interrupt
   {
     // Use MyDAAChanObj here
   }
   // indirect access:
   ///////////////////
   ```

```
void MyDAAIsr (tMyDAAChanObj* pMyDAAChanObj)
{
  // Use *pMyDAAChanObj here
}
// channel objects defined outside the DAA driver:
tMyDAAChanObj MyDAAChanObj;
void MyDAAIsrWrapper() // executed upon interrupt
{
  MyDAAIsr (&MyDAAChanObj);
}
```

6) Implement DAA scripts (see 7.7.3.2 and 7.7.5.2) specific to your DAA device. Store the pointers to the scripts in the array `tCodecDrvStage *MyapDAAStdRequests[7]`. This array will be copied to the array `apDAAStdRequests[]` (see `Si3044Stages.c`) in the beginning of the new peripheral driver's final hardware initialization function. A template file for the DAA scripts is available in the file `DriversTemplates\MyDAAStages.c`.

7) Finally, a new peripheral driver will be needed to initialize the new hardware, set the ISR and open the DAA channel for I/O. For example, a new `MyTargetPeriphInit()` function (analogous to `TargetPeriphInit()`, see `EVM54CSTDrv.c`) may now look like this (important parts are in bold):

```
void TargetPeriphInit (bool IsBIOSUsed, int TimerTo-
BeUsed)
  // if TimerToBeUsed<0, no timer will be used for MIPS
measurements
{
  tUartDrvSetupStruct UartDrvSetupStruct;
  /* Override the PeriphProcess and PeriphDriver func-
tions */
  CSTFxns.pPeriphProcess=MyPeriphProcess;
  CSTFxns.pPeriphDriver=MyPeriphDriver;
  if (!IsBIOSUsed)
  {
    /* Initialize the ISR handling code */
    IntInit();
    if(TimerToBeUsed>=0)
    {
      #undef IRQ_EVT_TINT1
      #define IRQ_EVT_TINT1 (24)
      /* Init the timer and install the timer ISR */
      CSTTimerInit(TimerToBeUsed?TIMER_DEV1:TIM-
ER_DEV0);
      SetIntVect (TimerTo-
BeUsed?IRQ_EVT_TINT1:IRQ_EVT_TINT0, &CSTTimerISR);
    }
  }
  /* Initialize the UART driver */
  ...
  /* Initialize the DAA driver */
  // Copy LIO function table
  DAADrvILIO = MyDAADrvILIO;
  // Copy scripts pointers
  memcpy (&apDAAStdRequests[0], &MyapDAAStdRequests[0],
sizeof(MyapDAAStdRequests));
  // ISR accesses DAA LIO channel object indirectly,
  // and MyDAAChanObj and MyDAAIsrWrapper() should be
already defined:

////////////////////////////////////////////////////////
/////////////
  /* Initialize the DAA hardware devices */
  My_DAA_setup (...);
  /* Open DAA channels (LIO) */
  Ch0.DAAChanHandle = DAAOpen (&MyDAAChanObj);
  if (!IsBIOSUsed)
  {
    /* Install the DAA ISR */
    SetIntVect (DAA_INTERRUPT_VECTOR_NUMBER, &MyDAAIsr-
Wrapper);
  }
  /* Initialize the high-level DAA driver state machine
(for channel 0) */
  DAACodecInit (&Ch0);
}
```

Note, when implementing the scripts and the LIO `ctrl()` function, remember that the high-level DAA driver supports a maximum of 19 registers with numbers 0 through 18. If there are more than 19 registers, see if you really need only 19 or fewer of them or may multiplex the registers.

7.7.7.4 Replacing the Peripheral Driver Functions

In all cases, when the UART or DAA driver is reloaded, the default peripheral driver (see files `EVM54CSTDrv.c`, `EVM54CSTDrv.h`) needs to be replaced. An example of the function `TargetPeriphInit()` has been given in the preceding sections describing reloading of the UART and DAA drivers.

Besides the `TargetPeriphInit()` function, the static function `EVMPeriphDriver()` will also need to be replaced, if the target board does not have the LEDs connected to the DSP in the same way as on the C54CST EVM (again, see the file `EVM54CSTDrv.c`). The companion static function `EVMPeriphProcess()` will need to be replaced as well, but this is mainly because the new peripheral driver can't access the old static function `EVMPeriphProcess()` from another module.

The `TargetBoardInit()` function is likely to be replaced as well. The reason for that may have to do with LEDs, port 0, GPIOCR register and wait state settings, which are hardware-specific.

After creating all of the replacements for the peripheral driver functions, make sure that the `main()` function of your application calls the new peripheral driver's functions, for example, `MyTargetBoardInit()` and `MyTargetPeriphInit()` functions.

A template for the peripheral driver is available in the files `DriversTemplates\MyBoardDrv.c` and `DriversTemplates\MyCSTPeriph.h`.

7.7.7.5 Multiple Channels in CST With Multiple DAA Devices

It is easy to create a multi-channel flex application that would use two DAA devices. The hardware setup consists of a standard C54CST EVM and a Texas Instruments DAA Daughter Card with a single DAA device mounted on it. The daughter card should be connected to the EVM.

The CSL library for the C54CST chip has been made supporting external DAA devices such as Si3021. So, it is possible to use the same CSL and LIO code as a driver for the external DAA as well.

A complete two-channel flex example application is available in the directory `FlexAppMultichan`. The example application uses a new CST peripheral driver to set up the two DAAs, C54CST's internal and daughter card's external. The new peripheral driver is contained in the files `EVM54CSTDrv2DAA.c` and `MyCSTPeriph.h`. The main application file is `main.c`. The files `CSTBIOS2.c`, `CSTBIOS2.h` and `CSTFlexAppBIOS2.cdb` are supplementary to compile the application for the DSP/BIOS environment.

Chapter 8

C54CST Resources: Registers Conventions, Memory, and MIPS

This chapter is a summary of important information about C54CST chip resources and their use by CST Framework and algorithms.

Topic **Page**

8.1 Overview

This chapter summarizes the most important information on the C54CST chip resources, their use by the CST Framework and algorithms. This information includes general register conventions (important for creating flex applications that will use CST), detailed memory address space layout (important when resolving memory-related problems), and memory/MIPS requirements of the CST Framework and the CST XDAIS algorithms.

8.2 General Register Conventions

In flex mode, for developer's code to be able to use the CST solution and co-exist with CST code, it is important that the developer would follow some conventions when using DSP registers.

The CST solution was developed with intent to use as little DSP registers as possible. Table 8-1 lists the registers, which are used by some of the CST modules, and their values, which are important for CST solution.

Table 8-1. DSP Registers Used by CST Solution

Registers	CST Usage and Conventions	File, Function
ST0 = 0	Boot routine, initialization of DP and ARP	Boot.s54 CSTChipsetEntry()
ST1, INTM = 1	Boot routine, disable all interrupts in the very beginning of boot function in order to initialize DSP correctly	Boot.s54 CSTChipsetEntry()

ST1,
CPL = 1
OVM = 0
SXM = 0
C16 = 0
CMPT = 0
FRCT = 0
	Boot routine, preset all these flags according to C conventions	Boot.s54
		CSTChipsetEntry()

PMST, MP/MC = 0	Boot routine, turn on "MicroComputer" mode, when internal ROM is enabled. The user must keep this bit equal 0 in order to be able to use the CST solution!	Boot.s54 CST_DSPInit()
PMST, DROM = 1	Boot routine, map internal ROM into data space in order to access CST's section ".const". The user must keep this bit equal 1 in order to be able to use the CST solution, or the user should connect external memory to data space of CST chip and copy CST's section ".const" into this external memory, because CST algorithms and framework keep their constants in ROM and need to have them visible in data memory space.	Boot.s54 CST_DSPInit()
PMST, OVLY = 1	Boot routine, maps the internal RAM into the program address space, so that CST's interrupt table would always be mapped to the program address space. The user must keep this bit equal 1 in order to be able to use the CST solution and CST interrupt processing!	Boot.s54 CST_DSPInit()

Table 8-1. DSP Registers Used by CST Solution (Continued)

Registers	CST Usage and Conventions	File, Function

PMST,
AVIS = 0
CLKOFF = 0
SMUL = 0
SST = 0
	Boot routine, preset all these flags according to C conventions and hardware requirements	Boot.s54
		CST_DSPInit()

IMR = 0	Boot routine, disable all interrupt in interrupt mask register	Boot.s54 CST_DSPInit()
SWCR = 1	Boot routine, enable wait states multiplier 2 in order to be on a safe side when operating with external memory	Boot.s54 CST_DSPInit()
SWWSR = 0x7fff	Boot routine, turn on maximum amount of wait states in order to be on a safe side when operating with external memory	Boot.s54 CST_DSPInit()
GPIOCR = 0	Boot routine, disable all general purpose I/O pins	Boot.s54 CST_DSPInit()
BSCR = 2	Boot routine, configure external bus operation mode	Boot.s54 CST_DSPInit()
PMST, bits 7-15	Interrupt initialization routine. Sets these bits to the upper address bits of `aIntEntrance` interrupt vector table	int.s54 IntInit()
DAA accessed via McBSP2	Low-level DAA driver	DAADrv54CST.c and CSL
BSCR, bit 3	EVM driver, sets DAACLK bit (bit 3) if DSP clock is 118 MHz	EVM54CSTDrv.c TargetBoardInit()
BSCR, bit 4	Low-level DAA driver, reset DAA	CSL DAA_reset()
UART registers USAR, USDR	Low-level UART driver	Uart550Drv.c, UartAutoBaud.c, CSL
GPIOSR, bits 0, 1, 2, 3, 4, 5	Low-level UART driver (hardware flow control) bit 0 – input, DTR bit 1 – input, RTS (from host point of view, CTS) bit 2 – output, CTS (from host point of view, RTS) bit 3 – output, DSR bit 4 – output, DCD bit 5 – output, RI	Uart550Drv.c, CSL
GPIOCR = 0x3C	UART driver, configure UART pins according to assignment defined in section .	Uart550Drv.c, CSL UART_FlowCtrlInit()

Table 8-1. DSP Registers Used by CST Solution (Continued)

Registers	CST Usage and Conventions	File, Function
GPIOCR = 0	EVM driver, reset all general purpose I/O pins at initialization	EVM54CSTDrv.c TargetBoardInit()
Clock PLL register CLKMD	EVM driver, set appropriate DSP clock multiplier (4 or 8)	EVM54CSTDrv.c SetDSPClockFreq()
SWCR = 0	EVM driver, reset wait state	EVM54CSTDrv.c TargetBoardInit()
SWWSR	EVM driver, set user-defined wait states for external memory access	EVM54CSTDrv.c TargetBoardInit()
I/O Port 0, bits 0-3	EVM driver, LED indication	EVM54CSTDrv.c BrdLEDToggle()
I/O Port 0, bits 6, 7	EVM driver, Flash/RAM configuration	EVM54CSTDrv.c TargetBoardInit()
Timer 0 registers	EVM driver, optionally initialize Timer 0 for CST statistics	EVM54CSTDrv.c, CSL TargetPeriphInit()
Timer 1 registers	EVM driver, optionally initialize Timer 1 for CST statistics	EVM54CSTDrv.c, CSL TargetPeriphInit()
ST1, INTM = 0	Main function in chipset or flex mode, enables interrupts	main.c main()
IMR	Device drivers enable specific interrupts	DAADrv54CST.c Uart550Drv.c, CSL

If the user's application requires that the CST solution do not access certain DSP registers, the user may redefine one or more of the CST drivers (see sections 7.7.3 and 7.7.7).

To disable use of some registers by CST, however, it may be enough to change some parameters during CST initialization.

For example, to tell CST not to use any DSP timers for MIPS measurement, it is enough to call function `TargetPeriphInit(bool IsBIOSUsed, int TimerToBeUsed)` with second parameter being negative:

TargetPeriphInit(xxx, -1);

8.3 Program and Data Address Space Memory Map

For detailed description of C54CST chip generic memory map, please, refer to *TMS320C54CST Client Side Telephony DSP* (SPRS187). This section describes mostly CST software-specific memory map distribution.

The CST solution occupies around 120 kW of TMS320C54CST's ROM. This ROM consists of 4 pages (residing from 0x6000 to 0xFFFF, 0x18000 to 0x1FFFF, 0x28000 to 0x2FFFF and 0x38000 to 0x3DFFF. Some memory in Page 0 of ROM is occupied by a core code of DSP/BIOS (from 0xB200 to 0xBB1F), and some – by the start up bootloader (from 0xBB20 to 0xBFFF). The rest of the ROM is occupied by the CST code.

TMS320C54CST has 40 kW of internal DARAM (dual access RAM), residing from 0x80 to 0x9FFF.

The external RAM is visible in data address space from 0xA000 to 0xFFFF, however the CST solution has its `.const` section in the ROM Page 0, that is why it needs to have the ROM Page 0 mapped to the data address space, from 0xC000 to 0xFFFF. For this reason the DROM bit should be set to 1.

Also, in order to have the interrupt vectors table in internal RAM and be able to use it even in Far mode of DSP, the CST solution operates with the OVLY bit equal to 1 (in this case the DSP's internal RAM is mapped to the program address space, from 0x80 to 0x5FFF).

Table 8-2 describes data address space of CST chip, and shows overview of program and data memory space.

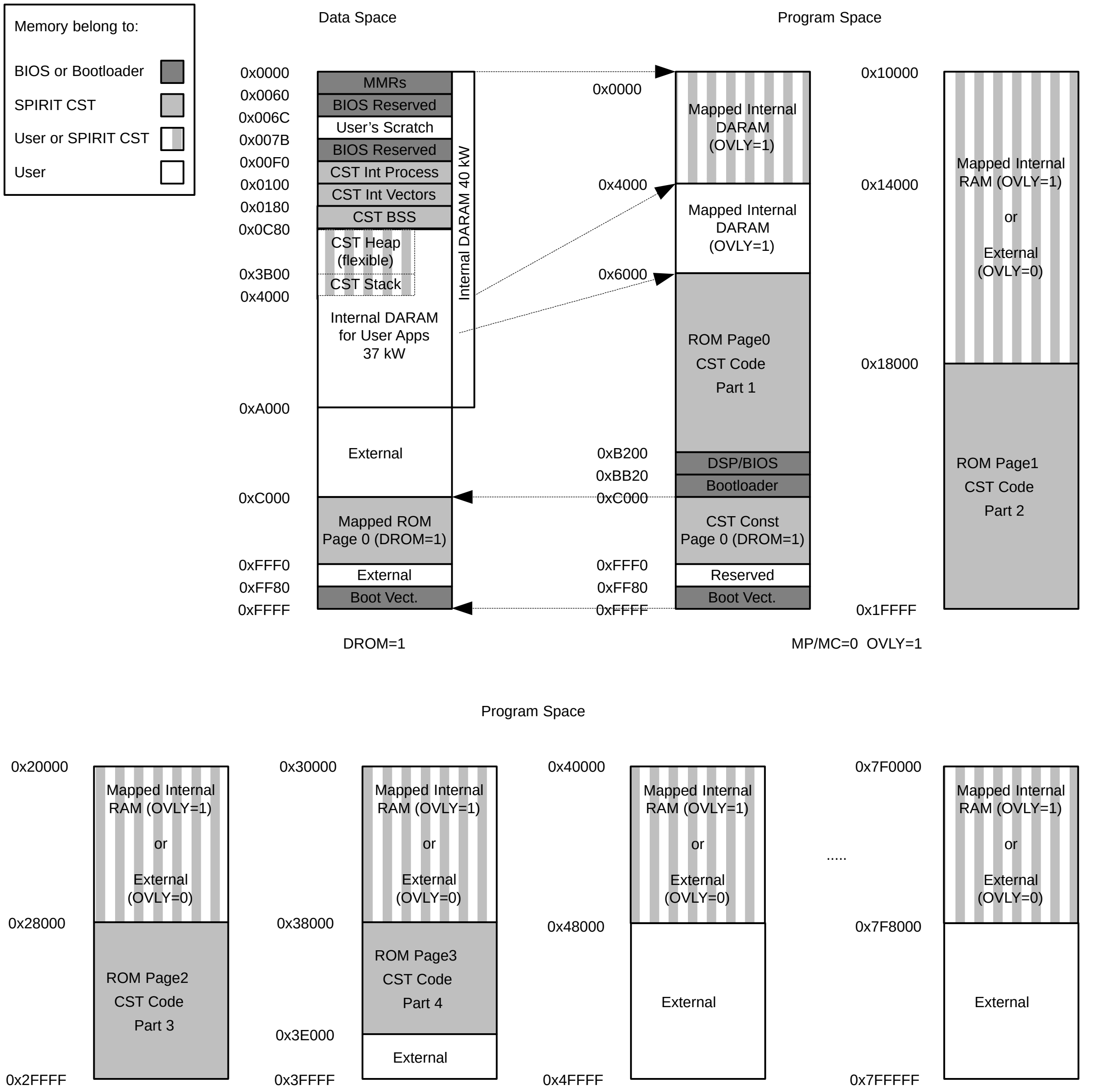
Table 8-2. CST RAM Areas Description

Area Type	Location	Sections and Explanation
Reserved for CST only. The user should not use it.	0x60 to 0x6B 0x7B to 0xEF 0xF0 to 0xFF 0x100 to 0x17F 0x180 to 0xC7F	CSTTrap, CSL and DSP/BIOS reserved area; CST Interrupt Processing CST Interrupt Vectors Table CST BSS
Reserved for CST, but the user may share both of them with CST.	0xC80 to 0x3AFF 0x3B00 to 0x3FFF	CST Heap CST Stack
		This area can be used to allocate user's data and program memory. Since OVLY==1, all internal RAM up to the address 0x5FFF is mapped to the program address space, and thus the user's program can be loaded here. If CST Stack area is moved from its original location, disable stack statistics.

Table 8-2. CST RAM Areas Description (Continued)

Area Type	**Location**	**Sections and Explanation**
Internal RAM	0x4000 to 0x9FFF	Not used by the CST software in chipset mode (update patch uses it though), available for the user
External Memory	0xA000 to 0xBFFF	User's external RAM
Map of ROM Page 0	0xC000 to 0xFFFF *.const* resides from 0xC000 to 0xFF00	CST needs this ROM page mapped to data space in order to have access to its *.const* section, which resides in ROM.

Figure 8-1. CST Solution Memory Map



In chipset mode, if update patch is not loaded, CST uses only 16 kW of internal RAM of C54CST chip (from 0x60 to 0x3FFF). After update patch is loaded, CST starts using all 40 kW of available internal RAM.

In flex mode, the User is free to use all the internal DARAM from 0xC80 to 0x9FFF, with several requirements to be met to enable correct operation of CST Framework:

- ❑ At least 0x500 words stack size should be reserved. If stack location is different from its original chipset mode location (from 0x0x3B00 to 0x3FFF), stack statistics should be disabled (this is done by default when `CSTAction_Init()` is called; can also be done by `CSTStatistics.Flags &= ~sf_STACK_MEMORY;`) to avoid unpredictable problems.

- ❑ CST memory manager has to be told what dynamic memory areas are available (again, by default it is done when `CSTAction_Init()` is called). The original size of CST's heap can be either decreased or increased, depending on the needs of the User application.

See and use flex example's cmd-file at `CST\Src\FlexApp` and flex example DSP/BIOS configuration at `CST\Src\FlexAppBIOS` as a template for flex mode memory configuration.

Since internal DARAM is mapped into program space (`OVLY==1`), from 0x80 to 0x5FFF, it is possible to place user's program space sections into it (like `.text`, `.cinit`, `.switch` and so on).

Internal DARAM from 0x6000 to 0x9FFF can be used only for data (dynamic memory, stack, `.bss`, `.const`), because this portion is not visible in program space when on-chip ROM is enabled.

To reference CST solution's objects residing in ROM, you have to include `CSTRom.s54` file into your project, which contains references to all global identifiers of CST. Additionally, it contains all global identifiers of DSP/BIOS components residing in ROM.

8.4 DSP Resource Usage for Each Algorithm and Framework

Table 8-3. CST Algorithms ROM/RAM Characteristics

Algorithm	ROM, W	CONST, W	BSS, W
V.32bis/V.32/V.22bis/V.22	17 822*	3 704*	24
V.42/V.42bis	13 412	162	40
Modem Integrator + V.14	3 129	26	81
UMTD	2 342	64	69
DTMF configuration	19	204	17
CPTD configuration	19	169	17
UMTG	1 122	219	79
DTMF configuration	14	158	10
CPTD configuration	27	67	21
Caller ID Type I and II	2159	248	32
CID Message Parser	740	595	19
G.168	2 354	0	29
G.726+G.711	2 152	502	29
G.723**	RAM 0.9K	-	8
G.729AB**	RAM 0.5K	-	2
LBR 1200**	RAM 2.3K	-	4
AGC	440	0	33
VAD	1 992	130	37
CNG	346	2	24
Common Library	1 367	74	0
CST Framework:			
AT-commands	4 755	1 892	1 036
CST Commander	1 291	289	28
CST Service	2 788	104	72
Voice Controller	2 454	8	18

Table 8-3. CST Algorithms ROM/RAM Characteristics (Continued)

Algorithm	ROM, W	CONST, W	BSS, W
DAA Driver	1311	11	11
UART Driver	1 792	1	691
DSP Driver	2 834	45	104
Memory manager	781	0	26
BIOS parts in CST	274	0	12
Misc (Periph, Int, Alg)	2 828	15	64
CST Bootloader	78	0	0
RTS	456	0	0
CST Bundle ver. 2.0	101 422	15 740	2 613

* This number also includes some portions of Fax G3 and V.29 fast connect add-ons

** This algorithm is provided as an Add-on for CST chip, and requires small additional portions of program memory in RAM

Table 8-4. CST Algorithims MIPS Characteristics

Algorithm	Configuration/Parameters	Buffer Length, 8kHz Samples	MIPS		Heap, W
			Peak	Average	
UMTD	DTMF	10	9.5	1.8	140
		100	1.6	1.5	
	CPTD	10	4.9	1.1	100
		100	0.9	0.7	
UMTG	DTMF	10	0.8	0.4	28
		100	0.1	0.1	
	CPTG	10	1.2	0.4	
		100	0.2	0.1	
Caller ID		10	3.8	1.7	44+

Table 8-4. CST Algorithims MIPS Characteristics(Continued)

Algorithm		Configuration/Parameters		Buffer Length, 8kHz Samples	MIPS		Heap, W
					Peak	Average	
				100	1.4	1.4	255*
VAD				10	11.7	1.1	372
				100	1.0	1.0	
AGC		adaptation	enabled	10	0.7	0.7	20
				100	0.6	0.6	
			disabled	10	0.5	0.5	
				100	0.4	0.4	
CNG		without filtering		10	1.6	1.6	37
				100	1.6	1.6	
		with 10 LPC		10	1.9	1.9	
				100	1.8	1.8	
G.168		127 taps		10	6.0	4.6	439
				100	5.6	5.4	
		255 taps		10	8.8	7.1	825
				100	8.3	7.4	
		511 taps		10	14.2	11.9	1591
				100	13.7	12.8	
G.711	encoder	64 kbps		80	0.9	0.8	10
	decoder			80	0.9	0.8	
G.726	encoder	u/A-law		80	6.5	5.8	10
		linear		80	5.9	5.3	
	decoder	u/A-law		80	7.4	6.6	
		linear		80	5.2	4.7	

Table 8-4. CST Algorithims MIPS Characteristics(Continued)

Algorithm		Configuration/Parameters		Buffer Length, 8kHz Samples	MIPS		Heap, W
					Peak	Average	
G.723	encoder	5.3 kbps		240	25.9	23.6	950+ 1280**
			HP filter	240	24.0	23.1	
			HP filter, VAD	240	24.3	6.7	
		6.3 kbps		240	24.2	24.0	
			HP filter	240	24.4	23.2	
			HP filter, VAD	240	24.3	6.6	
	decoder	5.3 kbps		240	1.2	1.2	
			post filter	240	2.3	1.7	
		6.3 kbps		240	1.2	1.2	
			post filter	240	2.4	2.3	
G.729	encoder	8 kbps		80	10.2	10.1	1846+ 980*
			VAD	80	10.2	4.4	
	decoder			80	2.5	2.4	
LBR 1200	encoder			320	11	-	1920+ 1200***
	decoder			320	4.5	-	
Data pump	V.32bis	14400	100 ms far echo buffer, APP on	10	14.6	13.6	2284
V.42 / V.42bis	V.42 only		heap size = 1500	100	1.9*2	1.4*2	2308

Table 8-4. CST Algorithims MIPS Characteristics(Continued)

Algorithm	Configuration/Parameters	Buffer Length, 8kHz Samples	MIPS Peak	MIPS Average	Heap, W
V.42 + V.42bis (duplex)	heap size = 1500, duplex compression with dictionary 512		n/a****	30 or more	4936
Full modem (V.32/V.42/V.42bis + modem integrator)	100 ms far echo buffer, APP on, heap size = 1500, duplex compression with dictionary 512	10	n/a****	30 or more	7460

* Allocated for parse message

** scratch

*** stack

**** n/a - MIPS value depends on the load ov V.42bis, and should be limited by the User via real-time feedback to Modem Integrator, or via running V.42bis in lower priority thread

Chapter 9

AT Command Set Descriptions

This chapter provides the user with description of AT commands, syntax, shielded codes, and results tokens

Topic **Page**

9.1 AT Command Set Description

SPIRIT CST AT command set supports a subset of standard AT-commands, allowing access to all the algorithms in CST solution. Some features of this CST AT command set, however, are proprietary and go beyond the scope of standard AT-commands, in order to give richer functionality (such as duplex voice mode, extended result tokens, control of CST hardware and software).

In chipset mode of CST chip, AT commands allow user to control the chip completely via serial link, eliminating the need for any other external interfaces with the chip for high level control. User can use standard Windows'9x drivers (for generic 14400 modem) to control CST chip and connect via it with conventional ISPs. Example of how to use AT commands in CST chip is given in CST Chipset Mode Application Note.

CST host program supplied with CST chip demonstrates how to control CST by using AT commands in chipset mode.

In flex mode, user may still want to use AT commands to control CST chip from some external device or even from inside of the chip (user's code inside CST chip can control CST software via AT commands sent to AT parser as if they came from UART). In this case, the user can use AT parser object from CST framework, or, since SPIRIT CST AT command parser is supplied with CST Software in open source code, the user can modify the code of AT parser and add or remove any functionality that they want, and then load it to CST chip in flex mode. Read more on AT parser open code in section 7.4.

9.2 AT Command Set Modes

AT commands parser operates in several modes depending on the command issued and on some other events:

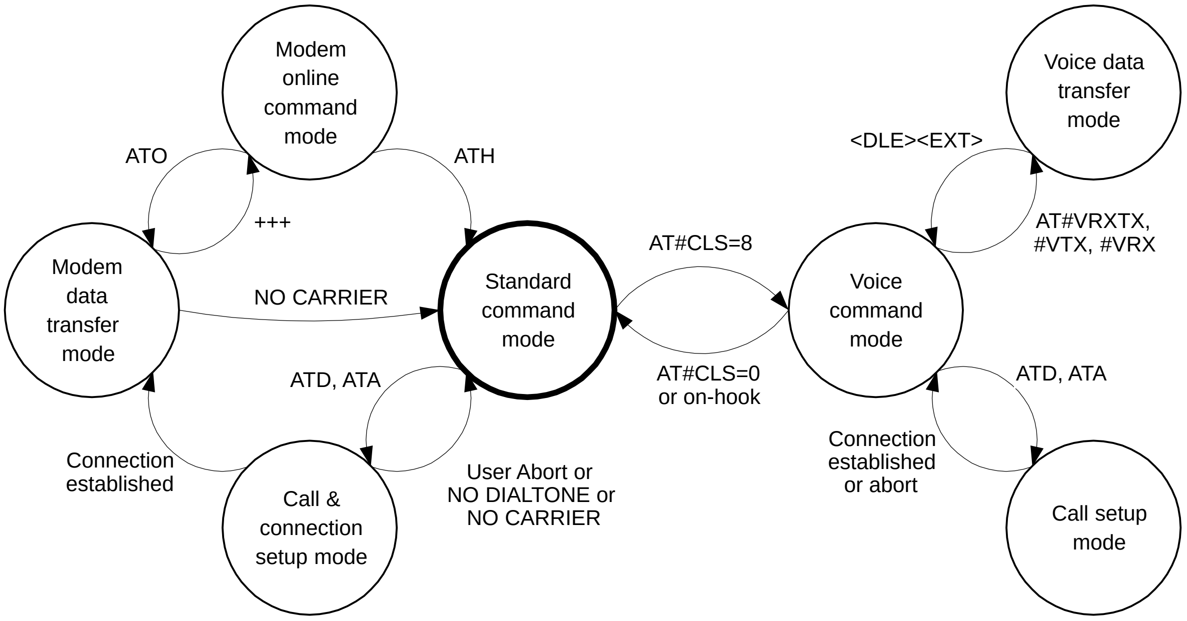
- ❑ Standard Command Mode
- ❑ Call and Connection Setup Mode
- ❑ Modem Data Transfer Mode
- ❑ Modem Online Command Mode
- ❑ Voice Command Mode
- ❑ Call Setup Mode
- ❑ Voice Data Transfer Mode

In each of the command modes, all data received from DTE (UART or user code) is passed to AT parser for processing.

In different Data Transfer modes, all data is passed to modem or to voice processing tasks.

The diagram of transactions between these modes is shown in Figure 9-1.

Figure 9-1. AT Parser State Diagram



After initialization, AT parser is in standard command mode. If ATD or ATA command is issued, it will start dialing, and will be in call and connection setup mode until connection is established, and then will move to modem data transfer mode. If, however, DTE sends any character while in call and connection setup mode, or CPTD does not detect dial tone before dialing or detects Busy signal after dialing (recognition of this signals is controlled by ATX command, see 9.4.1.23), call mode will be aborted and AT parser will switch back to standard command mode.

While in modem data transfer mode, AT parser is searching the incoming data for the escape control sequence (<Guard_Pause>+++<Guard_Pause>), which switches AT parser back to command mode (in this case called “modem online command mode”), even while modem is still connected. The user can use only a limited amount of AT commands in this mode. To return back to modem data transfer mode, ATO command should be issued. To terminate connection and return to standard command mode – ATH command.

To switch to voice command mode, DTE should issue “AT#CLS=8” command. This allows usage of all other voice mode commands. To switch back to standard command mode, DTE should issue ATH or AT#CLS=0 command.

To switch to voice data transfer mode, DTE should issue one of the following commands:

AT#VRX - To start recording samples from phone line

AT#VTX - To start transmitting samples to phone line

AT#VRXTX - To start full-duplex samples exchange

While in voice data transfer mode, AT parser processes so called “shielded” codes (codes that start with <DLE>[5] symbol, described in section) to enable transfer of some control information inside of the voice data stream (like termination command or DTMF and CPTD detectors result codes).

To return to voice command mode from voice data transfer mode, DTE should send characters <DLE><ETX>[6] (if voice data transfer mode was entered by AT#VRX command, it is enough to send any character).

The summary definition of AT parser modes is given in Table 9-1.

[5] <DLE> is a symbol from ASCII table equal to 0x10

[6] <ETX> is a symbol from ASCII table equal to 0x03

Table 9-1. Definition of AT Parser Modes

Mode Name	**Definition**
Command Mode	The DCE (in our case, CST solution) is not operating in the voice mode, the DCE is not communicating with a remote station, and the DCE is ready to accept commands. Data signals from the DTE are treated as command lines and processed by the DCE, and DCE responses are sent to the DTE. The DCE enters this mode upon power-up, and when a call is disconnected.
Voice Mode	The overall DCE mode of operation that performs voice functions by accepting special commands (voice commands), and providing voice and call discrimination event reports to the DTE.
Online Command Mode	In online command mode, the DCE is communicating with a remote station, but treats signals from the DTE as AT commands and sends responses to the DTE as AT result codes. Data received from the remote station during Online command mode is discarded until Online data mode is once again entered (by ATO command from the DTE). Data previously transmitted by the local DTE and buffered by the DCE is discarded. Online command mode may be entered from online data mode (modem data transfer mode) via special escape control sequence.
Voice Receive Mode	The DCE enters the voice receive mode upon #VRX command. In this mode, the DCE digitizes the analog signal from the line, converts the analog signal into binary data, compresses the data, and transfers it to the DTE.
Voice Transmit Mode	The DCE enters the voice transmit mode upon #VTX command. In this mode, the DCE receives the digitized data from the DTE, uncompress and converts them into analog signal, and transmits the analog signal to the line.
Voice Duplex Mode	The DCE enters the voice duplex mode upon #VRXTX command. This mode provides full duplex voice data processing. This command is a direct combination of Voice transmit mode (#VTX command) and voice receive mode (#VRX command).

9.3 AT Command Syntax

Syntax of AT commands basically complies with ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, except for the following:

Table 9-2. CST AT Commands Syntax Specifics

CST AT Commands Syntax Specifics:
Each bit of S-register can be addressed using dot delimiter
String parameters values are not used
Extended commands have at least one subparameter
Optional subparameters cannot be omitted
Space characters are filtered completely and they are ignored both in commands and parameter values
S3, S4, S5 registers cannot be equal to 0

Parameter ranges for V.250 compatible commands are within the limits required by this Recommendation.

9.3.1 General AT Commands Conventions

A command line is made up of three elements: the prefix, the body, and the termination character. AT parser is case-insensitive and accepts 7-bit symbols.

The command line prefix consists of the characters ”AT” or characters ”A/”. When command line starts with prefix “AT”, then a command or several commands should follow it, and should end with terminator character (S3 register). If the prefix ”A/” is encountered, the AT-parser immediately executes once again the body of the previous command line.

To edit command line, backspace character (S5) can be used to delete last input symbol.

The maximum length of command line is limited to 78 characters (without first “AT” characters).

The body is made up of individual commands as specified later. Space characters are ignored and may be used whenever you need.

Any control characters with ACSII codes 0 through 31, inclusive, except for the characters defined by S3 and S5 registers, are ignored by the AT-parser.

AT-parser echoes characters received from the DTE during command mode and online command mode back to the DTE, depending on the setting of the ATE command.

AT-parser considers lower-case characters to be the same as their upper-case equivalents (in other words, AT-parser is case-insensitive).

9.3.2 Types of Commands

There are two types of commands: *action* commands and *parameter* commands.

Action commands execute (invoking a particular function of the equipment), or test availability whether or not the equipment implements the action command, and, if subparameters are associated with the action, the ranges of subparameter values that are supported.

Parameter commands may ”set” (to store a value or values for later use), ”read” (to determine the current value or values stored), or ”test” (to determine whether or not the equipment implements the parameter, and the ranges of values supported).

Note: Use of Word “Action” and AT Commands

Use of the word “action” in this chapter applies only to AT commands. In the rest of the document, this word has a different meaning.

9.3.3 Basic Syntax Command Format

The format of basic syntax commands is as follows:

`<command>[<number>]`

where `<command>` is either a single character, or one of the commands listed in , or added by user. Characters used in `<command>` are from the set of alphabetic characters or one of the following characters: `‘&’, ‘#’, ‘$’`.

Parameter `<number>` may be a string of one or more characters from ”0” through ”9” representing a decimal integer value. If a command expects `<number>` and it is missing (`<command>` is immediately followed in the command line by another `<command>` or the termination character), the value ”0” is assumed. If a command does not expect a `<number>` and a number is present, an `ERROR` result code is generated. All leading zeroes in `<number>` are ignored by AT-parser.

Additional commands may follow a command (and associated parameter, if any) on the same command line without any character required for separation.

The actions of some commands can cause the remainder of the command line to be ignored (e.g. `ATA`, `ATDL`, etc.).

If the maximum number of characters that the AT-parser can accept in one line (78 characters) is exceeded, only those commands which fit into first 78 characters will be executed.

9.3.4 S-Parameters Syntax

Commands that begin with the letter ”S” constitute a special group of parameters known as ”S-parameters”. They differ from other commands in some important respects. The number following the ”S” indicates the ”parameter number” being referenced. If the number is not recognized as a valid parameter number, an `ERROR` result code is issued. If a dot (‘.’) follows the number of S-parameter, the next number is treated as referenced bit number (from 0 to 15) in this parameter to be tested, set or cleared.

Either a ”?” or ”=” character shall appear immediately following this number (or bit number correspondingly). ”?” character is used to read the current value of the indicated S-parameter; ”=” character is used to set the S-parameter to a new value.

Execution of S-parameter related command can be delayed and the result code is returned only after all associated events will be processed (e.g. DAA-driver requests, such as reading a hardware DAA register).

9.3.4.1 Set S-Parameter

Definition:

```
S<parameter_number>[.<bit_number>]=[<value>]
```

If the ”=” sign is used, the new value to be stored in S-parameter is specified in decimal following the ”=” sign. If no value is given (i.e. the end of the command line occurs or the next command follows immediately), the S-parameter specified will be set to 0. The ranges of acceptable values are given in the description of each S-parameter.

If `<parameter_number>` refers to a bit of S-register, this bit is set to 1 if <value> is not equal to 0, or cleared otherwise.

9.3.4.2 Testing S-Parameter

Definition:

`S<parameter_number>[.<bit_number>]?`

If the ”?” sign is used, the AT-parser transmits a single line of information text to the DTE. For most S-parameters, the text consists of exactly three characters, reporting the value of the S-parameter in decimal, with leading zeroes included. However, if the returned value can not be represented by 3 digits and exceeds 999, AT-parser outputs a number in decimal format without any leading zeroes.

If `<parameter_number>` refers to a bit of S-register, the bit value (0 or 1) is returned.

9.3.5 Extended Syntax Commands

Both action commands and parameter commands names typically begin with the ”+” character, but other leading characters can be used also. The first character following the `″+″` shall be an alphabetic character in the range of `″A″` through `″Z″`. This first character generally implies the application in which a command is used or the standards committee defined in ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97. The command may also include specification of a value or values. This is indicated by the appearance of <value> in the descriptions below.

Action commands may have more than one subparameter associated with them, and parameters may have more than one value. These are known as ”compound values”, and their treatment is the same in both actions and parameters.

9.3.5.1 Action Execution Command Syntax

The following syntax is used for actions that have two or more subparameters:

`+<name>[=<compound_value>]`

If the named action is supported and other relevant criteria are met (e.g. the CST-software is in the proper state), the command is executed with indicated subparameters. If <name> is not recognized, the AT-parser issues the `ERROR` result code and terminates processing of the command line. An `ERROR` is also generated if a subparameter is specified for an action that does not accept subparameters, if too many subparameters are specified, if a mandatory subparameter is not specified, if a value is specified of the wrong type, or if a value is specified that is not within the supported range.

9.3.5.2 Action Test Command Syntax

The DTE (host) may test if an action command is implemented in the CST solution by using the syntax:

```
+<name>=?
```

If the AT-parser does not recognize the indicated name, it returns an `ERROR` result code and terminates processing of the command line. If the DCE does recognize the action name, it will return an `OK` result code. If the named action accepts one or more subparameters, the AT-parser sends an information text response to the DTE, prior to the `OK` result code, specifying supported range of values for each such subparameter. The format of this information text is defined for each action command; general formats for specification of sets and ranges of numeric values are described in 9.3.5.5.

9.3.5.3 Parameter Set Command Syntax

The following syntax is used for parameters that accept a single value:

```
+<name>=[<value>]
```

The following syntax is used for parameters that accept more than one value:

```
+<name>=[<value_1>][,<value_2>]...[,<value_N>]
```

If a parameter is implemented and all values are valid according to the definition of the parameter, the specified values are stored. If <name> is not recognized, one or more values are outside the permitted range, the parser issues the `ERROR` result code and terminates processing of the command line. An `ERROR` is also generated if too many values are specified. In case of an error, all previous values of the parameter are unaffected.

9.3.5.4 Parameter Read Command Syntax

The DTE may determine the current value or values stored in a parameter by using the following syntax:

```
+<name>?
```

If a parameter is implemented, the current values stored for the parameter are sent to the DTE in an information text response. The format of this response is described in the definition of the parameter. Generally, the values will be sent in the same form in which they would be issued by the DTE in a parameter setting command; if multiple values are supported, they will generally be separated by commas, as in a parameter setting command.

9.3.5.5 Parameter Test Command Syntax

The DTE may test if a parameter is implemented in the DCE, and determine the supported values, by using the syntax:

```
+<name>=?
```

If the AT parser does not recognize the indicated name, it returns an `ERROR` result code and terminates processing of the command line. If the parser recognizes the parameter name, it returns an information text response to the DTE, followed by an `OK` result code. The information text response indicates the supported values for each such subparameter. The format of this information text is defined for each parameter; general formats for specification of sets and ranges of numeric values are described below.

In general, the format of information text returned by extended syntax commands is specified in the definition of the command.

When the action accepts a single numeric subparameter, or the parameter accepts only one numeric value, the set of supported values may be presented in the information text as an ordered list of values. The list is preceded by a left parenthesis "(", and is followed by a right parenthesis ")". If only a single value is supported, it appears between the parentheses. If more than one value is supported, then the values may be listed individually, separated by comma character, or, when a continuous range of values is supported, by the first value in the range, followed by a hyphen character "-", followed by the last value in the range.

When the action accepts more than one subparameter, or the parameter accepts more than one value, the set of supported values is presented as a list of the parenthetically-enclosed value range strings described above, separated by commas.

For example, the information text in response to testing an action that accepts three subparameters, and supports various ranges for each of them, could appear as follows:

`(0)`	Only value 0 is supported
`(0,8)`	Values 0 and 8 are supported only
`(1-3)`	Values 1 through 3 are supported
`(0),(1-3),` `(0,4-6,9,11-12)`	This indicates that the first subparameter accepts only value 0, the second accepts any value from 1 through 3 inclusive, and the third subparameter accepts any of the values `0, 4, 5, 6, 9, 11` or 12.

Value range indication is preceded by command name followed by a colon "`:`", i.e.

```
AT+VAD:(0-8000),(-32767-32767),(0-100),(0-100),(0-10)
```

9.3.6 Command Execution

9.3.6.1 Normal Execution

Upon receipt of the termination character (S3 register), the AT parser starts execution of the commands in the command line in the order received from the DTE. If the execution of a command results in an error, or a character is not recognized as a valid command, execution is terminated, the remainder of the command line is ignored, and the `ERROR` result code is issued. Otherwise, if all commands execute correctly, only the result code associated with the last command is issued; result codes for preceding commands are suppressed. If no commands appear in the command line, the OK result code is issued.

Some commands can force AT parser to ignore the remainder of command line after them (such as `ATA`, `ATD`, `ATSxx?`).

9.3.6.2 Aborting Commands

Some action commands that require time to execute may be aborted while in progress (such as dialing and establishing connection, `ATD` and `ATA`). Aborting a command is accomplished by transmission of any character from the DTE. A single character is sufficient to abort the command in progress.

When such event happens, AT parser does not echo incoming character, terminates the command in progress and returns an appropriate result code according to specification for the particular command.

9.4 AT Commands

9.4.1 General Commands

9.4.1.1 Circuit 109 (Received Line Signal Detector or DCD) Behaviour

This parameter determines how DCD line state changes depending on connection status of the modem.

Syntax `&C<value>`

Parameter(s)

Value	Description
0	DCD line always ON
1	DCD line is ON when modem is in connected state. Otherwise, it is OFF.

9.4.1.2 Circuit 108 (DTR - Data Terminal Ready) Behaviour

This parameter determines how the CST reacts when DTR circuit is changed from the ON to the OFF condition.

Syntax `&D<value>`

Parameter(s)

Value	Description
0	DTR change is ignored
1	CST switches to online command mode if it was in online data state (modem connected)
2	CST disconnects with remote modem and goes on-hook
3	CST re-initializes completely, equal to ATZ command

9.4.1.3 Set to Factory-Defined Configuration

Syntax `&F<value>`

Parameter(s)

Value	Description
0	This command does not do anything in CST. It is implemented only for compatibility.

9.4.1.4 Answer call

Answer Call. Go off-hook and start modem in answering mode. All characters following this command in the command line are ignored. This command can be aborted by sending any character from DTE.

Syntax A

9.4.1.5 Dial

Dial a number. Go off-hook, wait for dial tone to be detected, dial the specified number, if any, and then start modem in originating mode.

Syntax `D<dialing_number>`

Parameter(s) Parameter string `<dialing_number>` may contain numeric characters “0” through “9”, “A”, “B”, “C”, “D”, “#”, “*” and the following dial modifiers:

Dial Modifiers	Description
L	Redial last dialed number (see 9.4.1.6)
P	Dial in pulse mode
R	Force modem to connect in answering mode
T	Dial in tone mode
W	Wait for DIAL tone
/	Make a short pause. This pause is programmable in flex mode only.
,	Make a long pause. This pause is programmable both in chipset (register S8, see 9.4.5) and flex modes.
;	Just dial the number and go to command line mode immediately (neither modem nor voice will run after the number was dialed)
@	After the number has been dialed wait for RINGBACK signal detection and then it’s disappearance before entering Voice Mode (VCON)
!	Flash (going on hook for a short time, and then back off hook). Flash pause is programmable in flex mode only.

Dialing is aborted upon detection of any character not from this list.

Dial symbols and modifiers are case insensitive.

9.4.1.6 Dial Last Dialed Number

Dial last dialed number. All characters in the command line following this command are ignored.

Syntax `DL`

9.4.1.7 Command Echo

Select echo mode for AT-parser.

Syntax `E<value>`

Parameter(s)

Value	Description
0	Turn off echo mode for command mode of AT parser
1	Turn on echo mode for command mode of AT parser

9.4.1.8 Hook Control

Switch DAA to off hook or on hook state.

Syntax `H<value>`

Parameter(s)

Value	Description
0	Go on-hook.
1	Go off-hook. Turn on CPTD and DTMF detectors.

9.4.1.9 Request Identification Information

Request miscellaneous information.

Syntax `I<value>`

Parameter(s)

Value	Description
0	Request manufacturer info. In flex mode, the user can redefine a string returned in this case.

9.4.1.10 Monitor Speaker Loudness

Adjust speaker volume.

Syntax `L<value>`

Parameter(s)

Value	Description
0	Very low volume
1	Low volume
2	Normal volume
3	High volume

NOTICE: This command does not have any effect in CST EVM, because this EVM does not have monitor speaker. However, it is still supported to simplify monitor speaker control on User-specific platform.

9.4.1.11 Monitor Speaker Mode

Speaker On/Off. Using the speaker, you can monitor the status of each call your modem dials. This is helpful for tracking call progress.

Syntax `M<value>`

Parameter(s)

Value	Description
0	Speaker is turned off all the time
1	Speaker is turned on from going off hook till the connection is established
2	Speaker is turned on all the time
3	Speaker is turned on from the end of dialing till the connection is established

NOTICE: This command does not have any effect in CST EVM, because this EVM does not have monitor speaker. However, it is still supported to simplify monitor speaker control on User-specific platform.

9.4.1.12 Return to Online Data Mode

Return to modem data mode from modem online command mode. All data received by the modem while in modem online command mode is discarded. In response to this command, the AT parser will return the same string as when modem connects successfully, indicating current connection speed.

Syntax `O`

9.4.1.13 Select Pulse Dialing

Standard function: Select pulse-dialing mode as default.

Complimentary function: Select power-saving mode. When this command is entered, CST framework starts using IDLE 1 instruction of C54 DSP to put DSP into power-saving mode in between processing and interrupts. IDLE 1 mode stops only DSP's core, it does not stop its peripherals, however even this allows to significantly reduce power consumption in power-critical applications.
By default, this mode is off.

Since this command can be considered as redundant (ATDP is used much more often), such combination of functions should not cause much inconvenience.

Syntax `P`

9.4.1.14 Result Code Suppression

Syntax `Q<value>`

Parameter(s)

Value	Description
0	Enable result code indication (default).
1	Suppress result codes.

9.4.1.15 Command Line Termination Character

Syntax

`S3=<value>`	Set parameter
`S3?`	Read parameter

Parameter(s)

Acceptable Limits	Description
1...127	Command line termination character value (13 by default).

9.4.1.16 Response Formatting Character

Syntax

`S4=<value>`	Set parameter
`S4?`	Read parameter

Parameter(s)

Acceptable Limits	Description
1…127	Response formatting character value (10 by default).

9.4.1.17 Command Line Editing Character

Select Backspace character.

Syntax

`S5=<value>`	Set parameter
`S5?`	Read parameter

Parameter(s)

Acceptable Limits	Description
1…127	Backspace character value (8 by default).

9.4.1.18 Pause Before Dialing

This parameter specifies the amount of time (seconds) that the modem waits after going off hook and performing any other actions, such as dialing or answering to remote modem.

Syntax

`S6=<value>`	Set parameter
`S6?`	Read parameter

Parameter(s)

Acceptable Limits	Description
2…10	Number of seconds to wait before next action.

9.4.1.19 Comma Dial Modifier Time

This parameter specifies the amount of time (seconds) that the modem pauses during dialing when a ”,” (comma) dial modifier is encountered in a dial string (see 9.4.1.5).

Syntax

`S8=<value>`	Set parameter
`S8?`	Read parameter

Parameter(s)

Acceptable Limits	Description
1…255	Number of seconds to wait.

9.4.1.20 S-Registers Set or Test

Read or write to S-register. The description of available S-registers is given in 9.4.5.

Syntax

`S<reg_number>[.<bit_number>] =<value>`	Write S-register. When `<bit_number>` is not specified, S-register value is completely replaced by `<value>`, otherwise only selected bit is written. In the later case, any value that is not equal to 0 is treated as a command to set this bit to 1.
`S<reg_number>[.<bit_ number>]?`	Read S-register. When `<bit_number>` is not specified, the whole S-register value is printed; otherwise only selected bit value is printed.

Parameter(s)

Name	Acceptable Limits	Description
reg_number	0…199	S-register number
bit_number	0…15	Bit index of S-register

If `<reg_number>` is not any of the S-registers defined in CST AT parser (see Table 9-9), any value can be written into it, but the value of such register will always read as 0.

9.4.1.21 Select Tone Dialing

Standard function: Select DTMF tone dialing mode as default.

Complimentary function: Turn off power-saving mode. When this command is entered, CST framework stops using IDLE 1 instruction of DSP to put DSP into power-saving mode in between processing and interrupts. IDLE 1 mode stops only DSP’s core, it does not stop its peripherals, however even such behavior may be undesirable in some applications. This is why this command allows disabling usage of IDLE.
By default, this mode is on.

Since this command can be considered as redundant (ATDT is used much more often), such combination of functions should not cause much inconvenience.

Syntax `T`

9.4.1.22 DCE Response Format

The setting of this parameter determines the contents of the header and trailer transmitted with result codes and information responses. It also determines whether result codes are transmitted in a numeric form or an alphabetic (verbose) form. The text portion of information responses is not affected by this setting.

Syntax `V<value>`

Parameter(s)

Value	**Description**
0	Select numeric responses
1	Select verbose responses

9.4.1.23 Result Code Selection and Call Progress Monitoring Control

The setting of this parameter determines whether or not the DCE transmits particular result codes to the DTE. It also controls recognition busy and dial tone when going off hook.

However, this setting has no effect on the operation of the W dial modifier (see 9.4.1.5), which always checks for dial tone regardless of this setting.

Syntax `X<value>`

Parameter(s)

Value	CONNECT Response	Dial tone Detection	Busy Tone Detection
0	CONNECT	disabled	disabled
1	CONNECT <text>	disabled	disabled
2	CONNECT <text>	enabled	disabled
3	CONNECT <text>	disabled	enabled
4	CONNECT <text>	enabled	enabled

9.4.1.24 Reset To Default Configuration

Go on hook and reboot CST solution. All characters following the command are ignored.

Syntax `Z`

Note: This command is not implemented in release 1. It should be implemented in CST Release 2.

9.4.1.25 Print Brief S-Registers Summary

Print S-registers summary.

Syntax `$`

9.4.1.26 Print Brief AT Command Summary

Print AT commands summary. Both internal and user defined AT-commands are printed.

Syntax `$H`

9.4.1.27 Print Current Settings Summary

Print AT commands-related settings summary. Both internal and user defined AT-commands current status is printed.

Syntax `&V`

9.4.1.28 Switch Channel

This command does nothing by default.
User may use this parameter in user-specific multichannel application.

Syntax `#CHAN<number>`

`<number>` channel number to switch to.

9.4.1.29 Flex Application Load on The Fly

Loads flex application via UART.

Control can be passed to loaded program immediately after load, if entry point is non-zero, or can be returned to CST's AT-parser. This command can also be used also for modifying some of the variables in the internal memory on the fly, for example, for loading additional user-specific CPTD settings.

The format of the loaded image is the same as used by bootloader (see *TMS320C54CST Bootloader Technical Reference* (SPRA853)).

Syntax `#DATA`

9.4.1.30 Mode Selection

AT parser mode selection (see section 9.2 for details).

Syntax

`CLS=<mode>`	Set mode of CST system. See 9.2 for details.
`CLS?`	Retrieve current mode.
`CLS=?`	Test available modes. Parser returns `(0,8)` information response.

Parameter(s)

Value	Description
0	Standard command mode
8	Voice command mode

9.4.1.31 Country selection

Select CPT detector configuration. CST has 4 country configurations, 2 of them are already defined (user can change them), other reserved for the user. See section 7.6.3.1 for details.

Syntax

`+CNTRY=<number>`	Set country configuration for CPTD.
`+CNTRY?`	Retrieve current country configuration.
`+CNTRY=?`	Test available configuration. Parser returns `(0,3)` information response.

Parameter(s)

Value	Description
0	Default configuration. CST will detect CPT signals compliant with Q.35 recommendation.
1	CST will detect CPT signals complied with Q.35 recommendation and some signals which do not fin in Q.35 recommendation (i.e. Singapore busy tone, Italian dial tone, etc).
2	Empty. Can be defined to select user-specific configuration.
3	Empty. Can be defined to select user-specific configuration.

9.4.2 Caller ID Related Commands

CST client side caller ID component conforms to many standards of different countries, and as a result, it has many parameters, which have to be tuned to help CID operate correctly in a specific region.

Detailed definition and explanation of these parameters is given in “CID User’s Guide”. This chapter gives only brief description of AT commands, which tune some of these parameters.

9.4.2.1 TE-ACK Signal Settings

Select TE-ACK generator parameters such as duration, type and level.

Syntax

`+ATEACK=<Duration,Dtmf,Level>`	See Table 9-3 for details.
`+ATEACK?`	Tests actual parameter values.
`+ATEACK=?`	Tests available parameter values.

Parameter(s)

Table 9-3. TE-ACK Signal Settings

Parameter	Acceptable Limits	Description
Duration	65…90	Duration of TE-ACK signal, msec.
Dtmf	65…68	These values represent DTMF symbol ’A’,’B’,’C’ and ’D’ to be generated.
ToneLevel	0…16384	Generated signal level per tone (Q15.0 format) - 32768 corresponds to the full-scale sine-wave.

9.4.2.2 DT-AS Signal Settings

Select DT-AS detector parameters.

Syntax

`+ADTAS=<Duration,Twist, ToneLevel,SpuriousLevel>`	See Table 9-4 for details.
`+ADTAS?`	Tests actual parameter values.
`+ADTAS=?`	Tests available parameter values.

Parameter(s)

Table 9-4. DT-AS Detector Parameters

Parameter	Acceptable Limits	Description
Duration	50…100	Minimum acceptable duration (in msec).
Twist	8192…32767	Maximum acceptable tones twist (Q15.0 format).
ToneLevel	0…32767	Detector sensitivity. This parameter controls minimum signal level for tone to be accepted by detector (Q15.0 format).
SpuriousLevel	4096…16384	Acceptable relative spurious level (Q15.0 format). Greater values enhance tone recognition but make worse talk-off performance.

9.4.2.3 FSK Demodulator Settings

Select FSK demodulator settings that control message recognition.

Syntax

`+AFSK=<ToneLevel SpuriousLevel>`	See Table 9-5 for details.
`+AFSK?`	Tests actual parameter values.
`+AFSK=?`	Tests available parameter values.

Parameter(s)

Table 9-5. FSK Demodulator Settings

Parameter	Acceptable Limits	Description
ToneLevel	100…32767	Detector sensitivity. This parameter controls minimum signal level for tone to be accepted by detector (Q15.0 format).
SpuriousLevel	8192…16384	Acceptable relative spurious level (Q15.0 format). Greater values enhance tone recognition but make worse talk-off performance.

9.4.2.4 Caller ID Output Select

Select the form of output the caller ID information response

Syntax `#CID<value>`

Parameter(s)

Value	Description
0	Turn off caller ID

1	Caller ID is turned on after the first ring and outputs received data (caller ID information) in formatted representation. For example:
	`RING`
	`Caller ID info:`
	`Calling Line Indentity: 9117843`
	`Calling Party Name: TAGANSKAYA`

2	Caller ID is turned on after the first ring and outputs received data (caller ID information) in unformatted representation. Unformatted representation means that data, received by CID's FSK demodulator, is not parsed by CID message parser, but is output to terminal in ASCII hex format, byte by byte. For example:
	`RING`
	`Caller ID info:`
	`02073931313738343307 0F2020202020202020202020202020202 020`

9.4.3 Modem Related Commands

9.4.3.1 Data Compression

This extended-format multi-parameter command controls the V.42bis data compression component. It accepts four numeric subparameters.

Syntax

`+DS=<direction>, <negotiation>, <max_dict>, <max_string>`	See Table 9-6 for details.
`+DS?`	Tests actual parameter values.
`+DS=?`	Tests available parameter values.

Parameter(s)

Table 9-6. Data Compression Subparameters

Parameter	Acceptable Limits	Description
direction	0…3	Desired directions of operation for the data compression function: 0 – compression disabled in both directions 1 – compression is enabled for transmit direction only 2 – compression is enabled for receive direction only 3 – compression is enabled for both directions This parameter is also controlled by `%C` command.
negotiation	0	Specifies whether or not the modem should continue to operate if the desired result is not obtained. 0 means that modem does not disconnect when remote party does not negotiate V.42bis parameters.
max_dict	512, 1024, 2048, 4096	Maximum number of dictionary entries. Greatly affects memory usage by V.42bis object and compression ratio. With greater dictionary size V.42bis yields better compression, but consumes more memory. In chipset mode, or in flex mode when system has no additional memory, this parameter should be set to 512.
max_string	6…32	Maximum string length to be negotiated (V.42bis P2 parameter). Typically, it has to be set to 32.

9.4.3.2 Break Handling in Error Control Operation

This parameter is used to control the manner of handling the breaks (long times when no symbols arrive) by V.42 error correction component. This command is used just for V.250 compatibility and in reality the user can not change this parameter.

Syntax

`+EB=<break_selection>,`
`<timed>,`
`<default_length>`
	Always equal to 0,0,0. This means that breaks are completely ignored.

+EB?	Always returns 0,0,0.
+EB=?	Tests available parameter values.

Parameter(s)

See section 6.5.2 of ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, for details.

9.4.3.3 32-Bit Frame Check Sequence

This parameter controls the use of the 32-bit FCS option in V.42.

Syntax

`+EFCS=<value>`	Enable/disable usage of 32-bit FCS
`+EFCS?`	Returns current setting.
`+EFCS=?`	Tests available parameter values.

Parameter(s)

Value	Description
0	Use only 16-bit FCS
1	Use 32-bit FCS if possible (default)

See section 6.5.4 of ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, for details.

9.4.3.4 Error Control Reporting

V.250 recommendation uses this parameter to control the transfer of intermediate result code of kind “:+ER:xxx” before final result code (e.g. CONNECT) is transmitted.

Syntax

`+ER=<value>`	Always 0. This means that CST does not support intermediate result codes at all.
`+ER?`	Always return 0.
`+ER=?`	Tests available parameter values.

Parameter(s) See section 6.5.5 of ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, for details.

9.4.3.5 Error Control Selection

This command controls the manner of V.42 protocol operation in the protocol establishment phase. Modem Integrator in CST solution uses only the most generic options of V.42 object for establishing connection[7].

Syntax

`+ES=<orig_rqst>, <orig_fbk>, <ans_fbk>`	Always equal to 3,0,2. These values mean the following: 3 - use V.42 with detection phase only; 0 - use V.42 or and fall into V.14 mode if connection can not be established; 2 - both V.42 or V.14 are verified when modem is answerer
+ES?	Always returns 3,0,2
+ES=?	Tests available parameter values.

Parameter(s) See section 6.5.1 of ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, for details.

9.4.3.6 Selective Reject

Since the V.42 component version supplied in CST solution does not support selective reject, this command is used just for V.250 compatibility.

Syntax

+ESR=	Always 0. No selective reject is supported.
+ESR?	Always returns 0.
+ESR=?	Lists the supported range of values.

Parameter(s) See section 6.5.3 of ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, for details.

[7] Using custom connection procedure defined in section 2.5.2 of V.42 User Guide can extend connection capabilities at severe link conditions

9.4.3.7 Window Size

This parameters controls the maximum number of information frames that V.42 object may have unacknowledged simultaneously. To provide compatibility with ITU-T standard, the user should set this parameter to 15. However, real window size can be less than declared value, and be defined dynamically according to the actual heap size (see 9.4.3.9 +EHEAP command).

Syntax

`+EWIND=<value1>,<value2>`	See Table 9-7 for details
`+EWIND?`	Returns actual parameter values.
`+EWIND=?`	Lists the supported range of values.

Parameter(s)

Table 9-7. V.42 Window Size Subparameters

Parameter	Acceptable Limits	Description
value1	1…15	Transmit window size.
value2	0	Always is set to 0. It means that receive window size is equal to transmit windows size.

Also, see section 6.5.7 of ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, for details.

9.4.3.8 Frame Length

This parameter controls the maximum number of bytes in information field of an information frame transmitted by V.42. Typical value is 128. In very noisy environment (e.g. when BER>10-4) this parameter can be reduced, but AT parser only allows to set it as low as 32.

Syntax

`+EFRAM=<value1>,<value2>`	See Table 9-8 for details
`+EFRAM?`	Returns actual parameter values.
`+EFRAM=?`	Lists the supported range of values.

Parameter(s)

Table 9-8. V.42 Frame Length Subparameters

Parameter	Acceptable Limits	Description
value1	32…128	Transmit frame length.
value2	0	Always is set to 0. It means that receive frame length is equal to transmit frame length.

Also, see section 6.5.8 of ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97, for details.

9.4.3.9 V.42 Heap Select

Memory heap size available for V.42 is selected according to the system needs. Greater heap provides more robust performance for the outgoing traffic (lower delays and better outgoing throughput).

Syntax

`+EHEAP=<size>`	Heap size in words. Should be in range of 1000…4000.
`+EHEAP?`	Tests actual parameter values.
`+EHEAP=?`	Lists the supported range of values.

9.4.3.10 V.42 or Buffered V.14 Select

Select data-link layer protocol.

Syntax `\N<value>`

Parameter(s)

Value	Description
0	V.42 error correction is disabled. V.14 buffer mode is used instead
1	V.42 operation is enabled

9.4.3.11 V.42bis Compression Mode

Select V.42bis compression mode. When V.42 protocol is not in use, this setting does not have any effect, because V.42bis compression can only be used by V.42 protocol.

This option greatly affects memory size required to run the modem.

Syntax `%C<value>`

Parameter(s)

Value	Description
0	Compression is disabled in both directions
1	Compress transmitted data only
2	Decompress received data only
3	Compress/decompress in both direction

9.4.3.12 Round Trip Delay Settings

Maximal telephone line connection round trip delay. This parameter is used only by V.32bis/V.32 modem to allocate memory for far echo bulk delay buffer, in order to be able to suppress far echo from the incoming signal (see section 7.6.1.3 for details).

Syntax

`+ARTD=<delay>`	Delay in milliseconds. Should be in range of 20…2000 ms.
`+ARTD?`	Tests actual parameter value.
`+ARTD=?`	Lists the supported range of values.

Parameter(s) <delay> should be in range 20…2000 ms.

9.4.3.13 Modem Output Gain

Sets the power of modem output signal in dB. This value is written to S-register 28, so this parameter can also be controlled via S-registers.

Syntax `%L<value>`

Parameter(s) <value> should be in range 0…17 and represents output modem gain in dB (with negative sign).

9.4.3.14 Maximum Modem Speed

Select maximum modem speed.

Syntax `B<value>`

Parameter(s)

Value	Description	
0, 1	Automodem	(14400 bps Max)
2	1200 bps	(V.22bis/V.22)
3	2400 bps	(V.22bis)
4	4800 bps	(V.32bis/V.32)
5	7200 bps	(V.32bis/V.32)
6	9600 bps	(V.32bis/V.32)
7	12000 bps	(V.32bis)
8	14400 bps	(V.32bis)

9.4.3.15 Fast Connect

Controls the fast connect capability (see section 7.6.1.3 for details).

Syntax `#F<value>`

Parameter(s)

Value	Description
0	Fast connect mode is disabled.
1	Fast connect mode is enabled.

9.4.3.16 Modem Automatic Speed-up

Controls modem automatic speed-up capability.

Syntax `#ASPDUP<value>`

Parameter(s)

Value	Description
0	Speed-up mode is disabled. Even if conditions on the line allow to operate at higher bit rate, modem will not try to increase it.
1	Speed-up mode is enabled. If conditions on the line allow to operate at higher bit rate, modem will try to increase it via rate renegotiation.

9.4.3.17 Modem Slowdown

Controls modem automatic slow-down capability.

Syntax `#ASLWDN<value>`

Parameter(s)

Value	Description
0	Slow-down mode is disabled. Even if conditions on the line are poor to operate at current bit rate, modem will not automatically reduce the rate.
1	Slow-down mode is enabled. If conditions on the line are poor to operate at current bit rate, modem will automatically reduce the rate via rate renegotiation.

9.4.4 Voice Mode Commands

9.4.4.1 Select Echo Canceller Mode

Controls Line Echo Canceller

Syntax `#VEC<value>`

Parameter(s)

Value	Description
0	Turn off Echo Canceller
1	Turn on Echo Canceller without NLP (Non-Linear Processor)
2	Turn on Echo Canceller with NLP

9.4.4.2 Set Output Voice Signal Attenuation

Set output voice signal attenuation, decreasing output signal power in voice mode. The parameter denotes negative value in dB.

Syntax `+VGT<value>`

Parameter(s) <value> should be in the range from 0 to 30, denoting attenuation from 0 to -30 dB. By default, equal to 0.

9.4.4.3 Compression Method Selection

Voice Data Bit-Rate Selection

Syntax

`#VBS=<value>`	Set voice data bit rate
`#VBS?`	Prints currently selected bit rate
`#VBS=?`	Prints all available bit rate settings

Parameter(s)

Value	Description
2	Set 2 bits per 8000 Hz sample rate (G.726 ADPCM, 16 kbps)
3	Set 3 bits per 8000 Hz sample rate (G.726 ADPCM, 24 kbps)
4	Set 4 bits per 8000 Hz sample rate (G.726 ADPCM, 32 kbps)
5	Set 5 bits per 8000 Hz sample rate (G.726 ADPCM, 40 kbps)
8	Set 8 bits per 8000 Hz sample rate (PCM, μ-law, 64 kbps)

9.4.4.4 Voice Receive Mode

Turn on Voice Receive Mode. All additional characters in the command line are ignored. AT parser indicates completion of this command by issuing result code CONNECT. Immediately after this result code, CST will start sending voice data bitstream to DTE according to the selected bit rate. Different events are reported to DTE using special shielded codes (see section 9.5). The end of the bitstream (when DTE decides to turn this mode off) is marked by <DLE><ETX> shielded code sequence.

To turn this mode off and come back to Voice Command Mode, DTE should send any character.

Syntax `#VRX`

9.4.4.5 Voice Transmit Mode

Turn on Voice Transmit Mode. All additional characters in the command line are ignored. AT parser indicates completion of this command by issuing result code CONNECT. After this result code, DTE may start sending voice data bit stream according to selected bit rate. DCE reports different events to DTE using special shielded codes (see section 9.5).

To turn this mode off, DTE should send <DLE><ETX> sequence.

Syntax `#VTX`

9.4.4.6 Voice Duplex Mode

Turn on Voice Duplex Mode, when DCE receives and transmits voice data simultaneously. All additional characters in the command line are ignored. Modem indicates completion of this command by issuing result code CONNECT. Immediately after this result code, CST will start sending voice data bitstream to DTE according to the selected bit rate, and DTE can start sending voice data bitstream to CST. Different events are reported to DTE using special shielded codes (see section 9.5).

To turn this mode off, DTE should send <DLE><ETX> sequence.

Syntax `#VRXTX`

9.4.4.7 AGC Parameters

Set AGC reference signal level.

Syntax

`+AGC=<value>`	Set AGC reference signal level.
`+AGC?`	Prints current reference signal level.
`+AGC=?`	Prints acceptable range of reference signal level.

Parameter(s) <value> should be in the range from 0 to 8000.

9.4.4.8 VAD Parameters

Set VAD reference signal level.

Syntax

`+VAD=<lowAmp>, <quality>, <noiseSmooth>, <speechSmooth>, <lpcOrder>`	Set VAD algorithm creation parameters.
`+VAD?`	Prints currently selected bit rate
`+VAD=?`	Lists the supported range of values.

Parameter(s)

Parameter	Acceptable Limits	Default Value	Description
lowAmp	0..8000	80	Minimum noise amplitude. Signals with lower amplitude will be treated as noise.
quality	-32767.. 32767	-19662	Used for VAD energetic thresholds adjustment. The lower this number is, the higher is probability that noise will be recognized as speech, and vice versa. In other words, this parameters allows to tune VAD for different Speech-to-Noise ratios or for different applications. If, for example, VAD is used together with vocoder, this parameter should be low (around –19000), in order to make sure that even noise-like speech segments will be detected as speech. On the other hand, if VAD is used to control AGC, this parameter can be chosen higher (above -10000), in order to make sure that only those segments that really represent speech are detected.
noiseSmooth	0..100	0	Number of hangover frames after noise segment.
speechSmooth	0..100	0	Number of hangover frames after speech segment.
lpcOrder	0..10	10	Number of LPC coefficients for CNG.

9.4.5 S-Registers

The S-registers defined in CST AT parser are described in Table 9-9. Some of them are standard, some are proprietary. S-registers in parenthesis are implemented only for compatibility and are not used inside CST solution.

Table 9-9. S-Registers Defined in CST-Solution

Register Number	Related AT-cmd	Description
(S0)	-	Automatic Answer; does not affect CST behavior
S3	-	Command Line Termination Character <CR> By default, equal to 13.
S4	-	Response Formatting Character <LF> By default, equal to 10.
S5	-	Command Line Editing Character, backspace <BS> By default, equal to 8.
S6	-	Pause Before Blind Dialing, in seconds In CST this register contains the duration of the delay inserted after going off-hook and before any other action. By default, equal to 1 sec.

Table 9–9. S-Registers Defined in CST-Solution (Continued)

Register Number	Related AT-cmd	Description
S7	–	Connection Completion Timeout, in seconds If a modem can't establish a connection for the period of this timeout. CST will stop connecting and will go on hook. By default, equal to 60 sec.
S8	–	Comma Dial Modifier pause duration, in seconds Dialing string may contain the comma character, which sustains a pause in dialing for the specified amount of seconds. By default, equal to 2 sec.
(S10)	–	Automatic Disconnect Delay; does not affect CST behavior
S11	–	DTMF tone/space duration, msec. The duration of a DTMF tone and the pause between the DTMF tones. By default, equal to 80.
S12	–	Guard pause before and after '+++' (escape sequence) in 1/8th of msec Escape sequence is guarded with 2 periods of inactivity, when DTE should not send anything to DCE. If these periods exist before and after '+++' sequence, the AT Parser will consider the incoming sequence as Escape Sequence, and will switch to the Modem Online Command Mode. By default, equal to 8000 (1 sec).
S26	\N	Boolean flag enabling V.42 mode (when disabled, V.14 mode is used). See section 9.4.3.10 for details. By default, equal to 1.
S27	%C	V.42bis compression selection. Bit 0 enables V.42bis compressor, bit 1 enables V.42bis decompressor. See section 9.4.3.11 for details. By default, equal to 3.
S28	%L	Modem output signal attenuation in decibels (0..17 dB), treated as negative value. See section 9.4.3.13 for details. By default, equal to 9.
S29	#F	Enables the fast connect mode. See section 9.4.3.15 for details. By default, equal to 0.
S30	+VGT	Output voice signal attenuation in decibels (0..30 dB), treated as negative value. See section 9.4.4.2 for details. By default, equal to 0.
S31	–	Common input signal attenuation in decibels (0..30 dB), treated as negative value. Used only in Voice mode. See section 9.4.4.2 for details. By default, equal to 0.
S37	B	The maximum desired modem rate. See section 9.4.3.14 for details. 0,1 – Automodem; 2 – V.22 1200; … 8 – V.32bis 14400. By default, equal to 0
S38	–	An extra pause before V.42 session completion, in seconds. The modem waits this amount of time before V.42 connection is terminated, in order to flush data from internal buffers. By default, equal to 2.
S40	T and P	Default dialing mode: 0 – tone mode, 1 – pulse mode; Used when dialing string does not contain explicit dialing mode modifier. See sections 9.4.1.13 and 9.4.1.21 for details. By default, equal to 0.

Table 9–9. S-Registers Defined in CST-Solution (Continued)

Register Number	Related AT-cmd	Description
S41	#VEC	Line echo canceller mode: 0 – EC off; 1 – EC on without NLP; 2 – EC on with NLP. By default, equal to 1.
S42	#VBS	Voice Bit Per Second rate. Can be 2, 3, 4, 5 or 8, which corresponds to rates 16, 24, 32 and 40 kbps for G.726 and 64 kbps for G.711. See section 9.4.4.3 for details. By default, equal to 8.
S43	#CID	Caller ID mode. Selects: 1 – formatted CID, 2 – unformatted CID information printing; 0 – disables it. See section 9.4.2.4 for details. By default, equal to 1.
S44	–	Enables VAD in voice mode. By default, equal to 1.
S45	–	Enables AGC in voice mode. By default, equal to 1.
S46	–	Shield code value. By default, equal to 0x10 (<DLE>).
S47	–	Enable Caller ID report even if data have been received with incorrect CRC. By default, equal to 0.
S50	E	Boolean flag to enable AT Parser echo. See section 9.4.1.7 for details.
S51	–	Enables automatic adjustment of the UART baud rate. By default, equal to 1 (auto baud enabled).
S60	–	Statistics enable flags. Bit 0 enables MIPS measurement, bit 1 enables heap free size measurement, and bit 2 enables stack free size measurement. By default, equal to 7 (all flags enabled).
S61	–	Contains a number of currently active (created) xDAIS algorithms. Read only.
S62	–	Contains free heap size in words. Read only.
S63	–	Contains free stack size in words. Read only.
S64	–	Peak MIPS tracked since last reset (averaged on 4 msec block). Writing to this register a zero value resets it.
S65	–	Contains average value of input signal power, in dBm. Read only.
S70	#CHAN	Active channel number. Not used.
S101 … S119	–	Directly mapped to DAA internal Registers 1 through 19. Read section 9.4.6 for details.

9.4.6 S-Registers Controlling DAA

CST chip has an on-chip DAA, which sometimes needs to be tuned to a specific country telephone network standard. In order to be able to do this even in chipset mode, CST AT S registers `S101` through `S119` are directly mapped to the DAA internal registers 1 through 19.

Reading these S registers allows reading the contents of the DAA registers. Writing to these S registers allows writing to the DAA registers.

Since S registers in CST AT parser support bit-wise syntax for accessing them, the user can easily read or modify any bit of the DAA registers.

Short summary of DAA registers is given in Table 9-10. This table is taken from *TMS320C54CST Client Side Telephony DSP* (SPRS187) with some modifications. Refer to it for explanation of the abbreviation used in Table 9-10.

Table 9-10. DAA Registers Summary

SReg	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
101	Control 1	SR						DL	SB
102	Control 2					AL		HBE	RXE
105	DAA Control 1		RDTN	RDTP		ONHM	RDT		OH
106	DAA Control 2				PDL	PDN			
109	Sample Rate Ctrl							SRC[2..0]	
111	Chip A Revision						REVA[3:0]		
112	Line Side Status	CLE	FDT				LCS[3:0]		
113	Chip B Revision		CBID		REVB[3:0]			ARXB	ATXB
114	Line Side Validation						CHK	CIP	SAFE
115	TX/RX Gain Ctrl	TXM		ATX[2:0]		RXM		ARX[2:0]	
116	International Ctrl1	OFF/ SQL2	OHS	ACT		DCT[1:0]		RZ	RT
117	International Ctrl2		MCAL	CALD	LIM	OPE	BTE	ROV	BTD
118	International Ctrl3	FULL	DIAL	FJM	VOL	FLVM	MODE	RFWE	SQLH
119	International Ctrl4			LVCS[4:0]			OVL	DOD	OPD

International settings of DAA for different countries are given in Table 9-11.

Table 9-11. Country Specific DAA Register Settings

	S-Register 116					117	118
Country / **AT Bit Reference**	**S116.6 OHS**	**S116.5 ACT**	**S116.2,3 DCT[1:0]**	**S116.1 RZ**	**S116.0 RT**	**S117.4 LIM**	**S118.4 VOL**
Australia	1	1	01	0	0	0	0
Bulgaria	0	0 or 1	10	0	0	0	0
China	0	0	01	0	0	0	0
CTR21	0	0 or 1	11	0	0	1	0
Czech Republic	0	1	10	0	0	0	0
FCC	0	0	10	0	0	0	0
Hungary	0	0	10	0	0	0	0
Japan	0	0	01	0	0	0	0
Malaysia	0	0	01	0	0	0	0
New Zealand	0	1	10	0	0	0	0
Philippines	0	0	01	0	0	0	1
Poland	0	0	10	1	1	0	0
Singapore	0	0	10	0	0	0	0
Slovakia	0	0 or 1	10	0	0	0	0
Slovenia	0	1	10	0	0	0	0
South Africa	1	1	10	1	0	0	0
South Korea	0	0	01	1	0	0	0

Notes: 1) CTR21 includes the following countries: Austria, Belgium, Denmark, Finland, France, Germany, Greece, Iceland, Ireland, Italy, Luxembourg, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the United Kingdom.

2) This table is copied from *TMS320C54CST Client Side Telephony DSP* (SPRS187) with some modifications.

Country-specific settings for DAA can be configured via CST host, read chapter 10.2.2.

Detailed description of DAA's registers is given in *TMS320C54CST Client Side Telephony DSP* (SPRS187) (for C54CST's DAA) and *Si3044 User Guide. 3.3 V ENHANCED GLOBAL DIRECT ACCESS ARRANGEMENT*, ©Silicon Laboratories, 2000 (for Si3021 DAA).

9.5 Shielded Codes in Voice Mode

While in voice mode, AT parser processes so called “shielded” codes (codes that start with <DLE> symbol) to enable transfer of some control information inside of the voice stream (like termination command or DTMF and CPTD detectors result codes).

The codes used in the description below have the following numerical values:

- ETX - 0x03
- DLE - 0x10

The following codes are sent from DTE to AT parser while in voice data transfer mode:

Table 9-12. CST AT Parser Voice Mode Shielded Codes Sent From DTE

Shielded Code Sent to AT Parser	Meaning
<DLE><ETX>	Exit voice data transfer mode after finishing playing all data in the playback buffer. AT parser returns VCON result code.

The following codes are sent from AT parser to DTE while in voice data transfer and voice command mode:

Table 9-13. CST AT Parser Voice Mode Shielded Codes Sent to DTE

Shielded Code Sent to DTE	Meaning
<DLE>0-9,*,#,A-D	DTMF digits were detected by DTMF detector. Reported only once for each continuous tone.
<DLE>b	Busy tone was detected by CPT detector. Reported only once.
<DLE>c<String><CR><LF>	Report a string, terminated with new line characters – <CR><LF> Used to report decoded CID information and RING result code during voice data mode.
<DLE>d	Dial tone was detected by CPT detector. Reported only once.
<DLE>e	End of tone that was earlier detected by CPTD (this way DTE is informed when, for example, ringback tone finishes).
<DLE>n	Used by voice controller to transmit noise LPC instead of vocoder data, when VAD detects absence of voice activity
<DLE>r	Ringback tone was detected by CPTD detector. Reported only once.

Table 9-13. CST AT Parser Voice Mode Shielded Codes Sent to DTE (Continued)

Shielded Code Sent to DTE	Meaning
<DLE><ETX>	Denotes end of Voice Received Data stream. Occurs only after DTE issues a command to abort Voice Receive Mode.

If a voice data stream contains a character with the same code as <DLE> symbol has, this character is converted to two <DLE> characters on sending side, and is converted back to one character on receiving side.

9.6 AT Result Tokens

AT parser outputs the following result tokens:

Table 9-14. CST AT Parser Result Tokens

Token Name	Result Code	Description
OK	0	Command is accepted
CONNECT	1	Modems have connected
RING	2	Ring is detected
NO CARRIER	3	Connection with remote modem was lost
ERROR	4	Wrong AT command or parameter
NO DIALTONE	6	Dial tone was not detected within specified timeout (10 sec by default)
BUSY	7	Busy signal was detected while trying to connect
NO ANSWER	8	Modem is not responding
CONNECT Protocol/Rate	46	Reports protocol and rate at which modems have connected
NO CARRIER	4096 (0x1000)	Connection establishment was aborted by the user
VCON	4097 (0x1001)	Voice connect – output when voice command mode is entered and call setup is completed successfully.
DIALTONE	4098 (0x1002)	Dial signal was detected
RINGBACK	4099 (0x1003)	Ringback signal was detected
NO MEMORY	4105 (0x1009)	There is not enough memory to create one of the algorithms, required for AT-command execution
Formatted/ Unformatted CID data	-	A CID signal was received (this may happen at any time after the first ring and till the end of voice connection, i.e. till the AT parser goes on-hook).

RING token and CID data are unsolicited results, in other words they can be output at any time, not in response to some user's AT command.

9.7 AT Commands Summary

This section gives a brief overview of all AT commands, split into two tables:

- ❑ standard V.250-compatible commands (Table 9-15)
- ❑ CST-proprietary commands (Table 9-16)

Table 9-15. Summary of Standard V.250 Commands Supported by CST

Name	Description	Type[1]	Syn-tax[2]	Availability Options[3]	Category	Section to reference[4]	Ref.
&C	Circuit 109 (Received line signal detector) Behavior	P	B	A	General	6.2.8	9.4.1.1
&D	Circuit 108 (Data terminal ready) Behavior	P	B	A	General	6.2.9	9.4.1.2
&F	Set to Factory-Defined Configuration	A	B	A	General	6.1.2	9.4.1.3
A	Answer	A	B0	CV	General	6.3.5	9.4.1.4
D	Dial	A	B	CV	General	6.3.1	9.4.1.5
DL	Dial Last Dialed Number	A	B0	CV	General	6.3.1	9.4.1.6
E	Command Echo	P	B	A	General	6.2.4	9.4.1.7
H	Hook Control	A	B	A	General	6.3.6	9.4.1.8
I	Request Identification Information	A	B	A	General	6.1.3	9.4.1.9
L	Monitor Speaker Loudness	P	B	A	General	6.3.13	9.4.1.10
M	Monitor Speaker Mode	P	B	A	General	6.3.14	9.4.1.11
O	Return to Online Data Mode	A	B0	O	General	6.3.7	9.4.1.12
P	Select Pulse Dialing	P	B0	A	General	6.3.3	9.4.1.13
Q	Result Code Suppression	P	B	A	General	6.2.5	9.4.1.14
S	S-Registers Set or Test	P or A	S		General		9.4.1.20
T	Select Tone Dialing	P	B0	A	General	6.3.2	9.4.1.21

Notes:
1) Types of commands:
   A - action commands, P - parameter selection
2) Syntax of commands:
   B - basic (9.3.3); B0 – basic, without parameters (9.3.3);
   S - S-register specific (9.3.4); E - extended (9.3.5)
3) Availability options:
   V - Voice Mode; C - Command Mode;
   O - Online Command Mode; A - all modes.
4) ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97

Table 9-15. Summary of Standard V.250 Commands Supported by CST (Continued)

Name	Description	Type[1]	Syn-tax[2]	Availability Options[3]	Category	Section to reference[4]	Ref.
V	DCE Response Format	P	B	A	General	6.2.6	9.4.1.22
X	Result Code Selection and Call Progress Monitoring Control	P	B	A	General	6.2.7	9.4.1.23
Z	Reset To Default Configuration	A	B0	A	General	6.1.1	9.4.1.24
+DS	Data Compression	P	E	A	Modem	6.6.1	9.4.3.1
+EB	Break Handling in Error Control Operation	P	E	A	Modem	6.5.2	9.4.3.2
+EFCS	32-bit Frame Check Sequence	P	E	A	Modem	6.5.4	9.4.3.3
+EFRAM	V.42 Frame Size	P	E	A	Modem	6.5.8	9.4.3.8
+ER	Error Control Reporting	P	E	A	Modem	6.5.5	9.4.3.4
+ES	Error Control Selection	P	E	A	Modem	6.5.1	9.4.3.5
+ESR	Selective Reject	P	E	A	Modem	6.5.3	9.4.3.6
+EWIND	V.42 Window Size	P	E	A	Modem	6.5.7	9.4.3.7

Notes:
1) Types of commands:
   A - action commands, P - parameter selection
2) Syntax of commands:
   B - basic (9.3.3); B0 – basic, without parameters (9.3.3);
   S - S-register specific (9.3.4); E - extended (9.3.5)
3) Availability options:
   V - Voice Mode; C - Command Mode;
   O - Online Command Mode; A - all modes.
4) ITU-T Recommendation V.250: *Serial asynchronous automatic dialing and control*, 07/97

Table 9-16. Summary of CST-Solution Proprietary AT Commands

Name	**Description**	**Type[1]**	**Syntax[2]**	**Availability Options[3]**	**Category**	**Ref.**
$	S-Register Summary	A	B0	A	General	9.4.1.25
$H	AT Command Summary	A	B0	A	General	9.4.1.26
&V	Current Settings Summary	A	B0	A	General	9.4.1.27
#CHAN	Current channel	A	B0	A	General	9.4.1.28
#DATA	Flex application load	A	B0	A	General	9.4.1.29
#CLS	Mode selection	A	E	A	General	9.4.1.30
+CNTRY	Country selection	A	E	A	General	9.4.1.31
+EHEAP	V.42 Heap Select	P	E	A	Modem	9.4.3.9
\N	V.42 or Buffered V.14 Select	P	B	A	Modem	9.4.3.10
%C	V.42bis Compression Mode	P	B	A	Modem	9.4.3.11
+ARTD	Round Trip Delay Settings	A	E	A	Modem	9.4.3.12
%L	Modem Output Gain	P	B	A	Modem	9.4.3.13
B	Maximum Modem Speed	A	B	A	Modem	9.4.3.14
#F	Fast Connect Control	P	B	A	Modem	9.4.3.15
+ASPDUP	Speedup Control	P	E	A	Modem	9.4.3.16
+ASLWDN	Slowdown Control	P	E	A	Modem	9.4.3.17
#CID	Caller ID Output Select	P	B	A	Caller ID	9.4.2.4
+ATEACK	TE-ACK Signal Settings	P	E	A	Caller ID	9.4.2.1
+ADTAS	DT-AS Signal Settings	P	E	A	Caller ID	9.4.2.2
+AFSK	FSK Demodulator Settings	P	E	A	Caller ID	9.4.2.3

Notes:
1) Types of commands:
A - action commands, P - parameter selection
2) Syntax of commands:
B - basic (9.3.3); B0 – basic, without parameters (9.3.3);
S - S-register specific (9.3.4); E - extended (9.3.5)
3) Availability options:
V - Voice Mode; C - Command Mode;
O - Online Command Mode; A - all modes.

Table 9-16. Summary of CST-Solution Proprietary AT Commands (Continued)

Name	Description	Type[1]	Syntax[2]	Availability Options[3]	Category	Ref.
#VEC	Echo Canceller Mode	P	B	V	Voice	9.4.4.1
+VGT	Set Voice Loop Gain	P	B	V	Voice	9.4.4.2
#VBS	Compression Method Selection	P	E	V	Voice	9.4.4.3
#VRX	Voice Receive Mode	A	B0	V	Voice	9.4.4.4
#VTX	Voice Transmit Mode	A	B0	V	Voice	9.4.4.5
#VRXTX	Voice Duplex Mode	A	B0	V	Voice	9.4.4.6
+AGC	AGC Parameters	P	E	A	Voice	9.4.4.7
+VAD	VAD Parameters	P	E	A	Voice	9.4.4.8

Notes:
1) Types of commands:
   A - action commands, P - parameter selection
2) Syntax of commands:
   B - basic (9.3.3); B0 – basic, without parameters (9.3.3);
   S - S-register specific (9.3.4); E - extended (9.3.5)
3) Availability options:
   V - Voice Mode; C - Command Mode;
   O - Online Command Mode; A - all modes.

Table 9-17. Summary of Commands by Categories

Category	Commands
General	&C, &D, &F, &V, A, D, DL, E, H, I, L, M, O, P, Q, S, T, V, X, Z, $, $H, #CLS, #CHAN, #DATA, +CNTRY S0, S3, S4, S5, S6, S8, S11, S12, S31, S40, S46, S50, S51, S60, S61, S62, S63, S64, S65, S70
Modem	+DS, +EB, +EFCS, +ER, +ES, +ESR, +EWIND, +EFRAM, +ARTD, +EHEAP, +ASPDUP, +ASLWDN, %L, \N, %C, B, #F S7, S10, S26, S27, S28, S29, S37, S38
Voice	#VEC, +VGT, #VBS, #VRX, #VTX, #VRXTX, +AGC, +VAD S30, S41, S42, S44, S45, S47
Caller ID	+ATEACK, +ADTAS, +AFSK, #CID S43, S47

Chapter 10

CST Host Utility

In order to better control CST Chip over serial link, SPIRIT has developed a special PC application - CST host. It can be used as terminal in data mode, as play and record utility in voice mode, and also it simplifies setting the CST chip settings.

To start CST Host, run `CST\CSTHost\CSTHost.exe`.

Topic **Page**

10.1 Minimum System Requirements

CST host requires the following parameters to be met on PC on which it is running:

Table 10-1. CST Host Parameter Requirements

Parameter	**Minimum Requirement**
Processor	Intel Pentium®, 300 MHz
Hard drive free space	3 Mb
RAM	16 Mb
Video	VGA 800*600
Communication adapter	Standard COM-port

10.2 CST Host Settings

Note: Running applications or demo examples on EVM boards

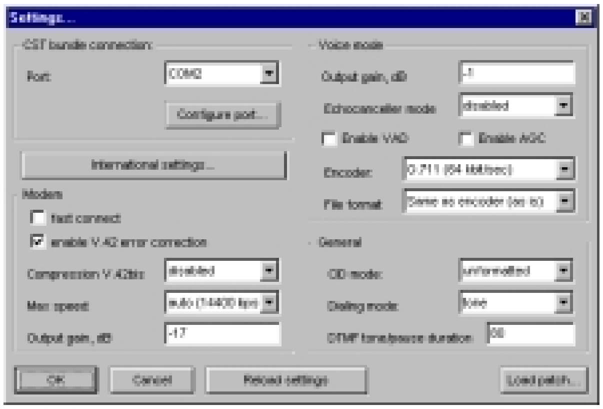
Before running any application or demo example on EVM board, it is important to correctly tune the COM port, DAA settings and CST solution settings. All these settings are stored in CST host initialization file.
DAA and CST solution settings have to be reloaded every time EVM board is powered on or CST chip is reset.

CST host settings dialog divided on following sections:

- ❑ COM port settings – selects and configures COM port.

  International settings – configures DAA driver for the standards of your country.
- ❑ Modem – allows changing of some modem parameters.
- ❑ Voice mode – selects encoder, input/output file format, echo canceller mode, switches ON/OFF VAD and AGC.
- ❑ General – selects representing type of CID information, and DTMF tone/pause durations.

Figure 10-1. CST Host Settings Dialog



CST bundle connection:
Port: COM2

Modem
☐ fast connect
☑ enable V.42 error correction
Compression V.42bis: disabled
Max speed: auto (14400 bps)
Output gain, dB: -17

Voice mode
Output gain, dB: -1
Echocanceller mode: disabled
☐ Enable VAD ☐ Enable AGC
Encoder: G.711 (64 kbit/sec)
File format: Same as encoder (as is)

General
CID mode: unformatted
Dialing mode: tone
DTMF tone/pause duration: 80

10.2.1 COM Port Settings

To configure COM port in CST host, open terminal window at `File->CST Terminal`, and press `Settings` button. Choose COM port to which EVM is connected, and press `Configure Port` button. Set the port for 115200 bps, 8 bits of data, 1 stop bit, no parity, Hardware flow-control, and press `OK`.

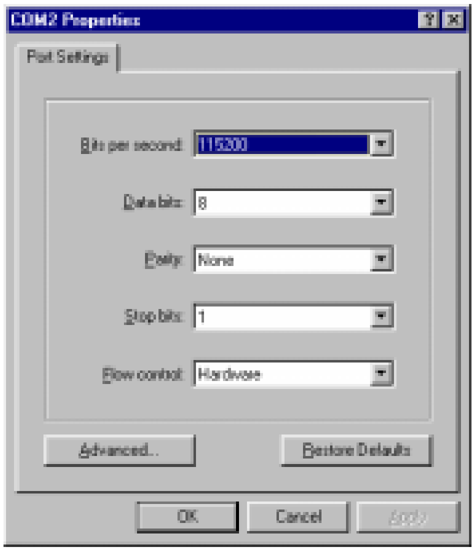
Then press `OK` again on "Settings…" window.

Try to type AT<*ENTER*> to check if the COM was configures correctly and the EVM can receive and send data over it. If everything is correct, you should see echo of the command that you enter, and then OK response:

```
AT
OK
```

CST chip UART driver has a limited capability of autobaud detection, so if for some reason the port speed was selected other than 115200, the user must help CST chip to synchronize to the new baud rate. In order to do this, keep typing several continuous "AT" commands without <Enter> until you see the correct echo.

Figure 10-2. COM Port Settings Dialog



Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow control: Hardware

10.2.2 DAA International Settings

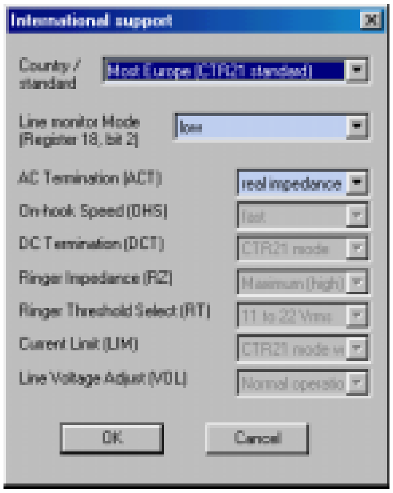
It is important to tune DAA properly for the standards of your country. This can be done via special dialog in CST host, which is invoked by pressing `Settings`, `DAA International settings`. The window shown in Figure 10-3 should appear.

This dialog allows you to choose one of the presets for different countries and Line monitor Mode. The presets are described in Table 9-11. There is also a “User-defined” preset, which allows tuning different DAA parameters manually. By default, it uses settings from the previously selected country preset.

All settings in this dialog box are saved in the initialization file of CST host.

Transfer of these settings to CST chip is done via special S-registers, using AT commands. Every time these settings are modified, EVM board is powered on or CST chip is reset, DAA settings have to be reloaded to CST chip.

Figure 10-3. DAA Settings Dialog



Country / standard	Most Europe (CTR21 standard)
Line monitor Mode (Register 18, bit 2)	low
AC Termination (ACT)	real impedance
On-hook Speed (OHS)	fast
DC Termination (DCT)	CTR21 mode
Ringer Impedance (RZ)	Maximum (high)
Ringer Threshold Select (RT)	11 to 22 Vrms
Current Limit (LIM)	CTR21 mode w
Line Voltage Adjust (VOL)	Normal operatio

Line monitor mode controls bit 2 of DAA register 18 (S register 118). If it selected as “high”, this bit is set to 1, otherwise – to 0. This bit influences DAA functionality in conjunction with several other bits, detailed description of this is given in *TMS320C54CST Client Side Telephony DSP. Data Manual. Texas Instruments* (SPRS187). For the most part, this bit is used to control on-hook current line monitor – high or low.

A short description of DAA's AT S-registers is given in section 9.4.5

of this document. Detailed description of DAA's registers is given in *TMS320C54CST Client Side Telephony DSP. Data Manual. Texas Instruments* (SPRS187) (for C54CST's DAA) and *Si3044 User Guide: 3.3 V ENHANCED GLOBAL DIRECT ACCESS ARRANGEMENT*, Silicon Laboratories, 2000 (for Si3021 DAA).

10.2.3 Miscellaneous Settings

There are also different CST solution settings, which can be modified using CST host and loaded to CST chip via AT commands. Most of the settings are self-explanatory and their detailed description can be found in sections describing corresponding AT commands.

Just as with DAA settings, CST solution settings have to be reloaded to CST chip every time these settings are modified, EVM board is powered on or CST chip is reset.

The only setting, which does not influence CST chip, but only influences CST host is “`File Format`” in voice mode settings. This option controls whether CST host reads/writes voice files without any processing (“`Same as encoder (as is)`” mode), or it reads/writes files in WAVe format (16 bit, 8 kHz mono). In case of WAVe file format CST host encodes/decodes the voice according to the selected “`Encoder`” mode before sending it to and after receiving it from CST chip. Read more on file format in section 10.3.1.

The “`Same as encoder (as is)`” mode should be used when files to be sent to CST chip in voice mode and files recorded from CST chip should be in encoded mode (G.726 bitstream or G.711 bytes).

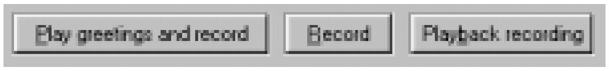
The “`Wave file (16bit 8kHz)`” mode should be used when files to be sent to CST chip in voice mode and files recorded from CST chip should be in WAVe 16-bit 8kHz file format independently of the selected encoder type for CST chip.

The names of the files to be played (read from PC and sent to CST chip) and to be recorded (received from CST chip and save to PC) are set in the corresponding fields “`Greeting message`” and “`File to be recorded`” in main CST Terminal window. Before running a script, CST host makes sure that these files can be opened for read/write.

10.3 Voice Playback and Record

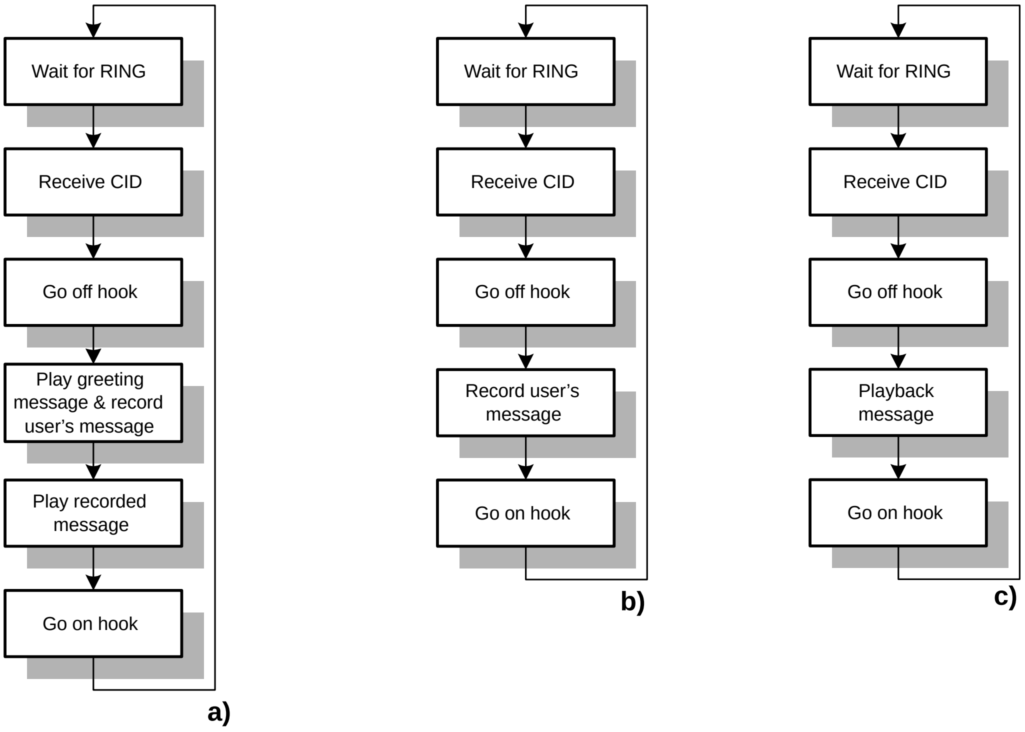
CST host has a simple scripting engine inside, with 3 simple scenarios to demonstrate voice mode functionality. To run one of these scripts, CST host has buttons shown on the figure below.

Figure 10-4. Voice Play/Record Buttons



Pictures below are describing the processing flow.

Figure 10-5. CST Host Processing Flow



It also has 2 input fields to set the names of files to be played and to be recorded, and playback/record duration field.

"`File Format`" setting in `Settings` dialog (read more on that in section 10.2.3) determines the type of the data in files.

The duration field determines how long the file will be played (if its length is greater than the specified duration), or how long it will be recorded (unless the caller hangs up and BUSY detected, which may stop recording earlier).

To start a script, press one of the script buttons.

In the beginning of any script CST host will infinitely wait for RING result code from CST chip.

Upon detection of this event, scripting engine will issue several AT commands to make CST chip go off hook and configure it to the voice mode with the specified settings.

After this, "`Record`" script will configure CST chip to send data from phone line to PC, and will save this data to a file; "`Playback recording`" script will configure CST chip to output data from PC to the phone line, and will send data from a file to the chip; "`Play greeting and record`" script will configure CST chip to simultaneously play and record data to/from phone line, and will send data from a file and save data to a file at the same time. In latter case, after "`Play greeting and record`" script finishes playing the greeting, it will play out the recorded message.

The user can terminate script execution at any time by unpressing the corresponding script button. Execution will also be terminated upon some unsolicited result code, time-out in communication with CST chip or reception of BUSY signal.

10.3.1 CST Host Audio File Format

CST host stores voice data in files in two ways. When "`Same as encoder (as is)`" mode is selected, it saves bytes that are received from COM-port "`as is`" just removing DLE formatting which do not belong to voice bitstream. <DLE><DLE> symbols are not removed. Files stored in this mode take less storage memory than in `“raw PCM”` mode.

When "`Wave file (16bit 8kHz)`" mode is selected, CST host performs on-the-fly conversion of incoming packed stream to 16-bit WAVe format at 8 kHz sample rate. CST host also performs outgoing bitstream packing for specified codec. Conventional audio editors can easily browse these files.

CST host does not automatically recognize file format when opening a file, so files stored in one format cannot be played out correctly in another format.

10.3.2 Application Sequence “Playback Greeting and Record”

The scenario of the *“Playback greeting and record”* CST host session is as follows
(also shown briefly in Figure 10-5) a):

1) CST chip is on-hook, waiting for the ring. CST host is waiting for the ”`RING`” result code from the CST chip.

2) Once ring is detected, CST chip reports ”`RING`” result to the host and starts waiting for CID signal. If CID Mark Bit is detected (1100 or 1200 Hz), CID proceeds with detection and reception of CID message, and then CST chip outputs the result of CID reception to the CSH host terminal.

3) On reception of “`RING`” result code, host increments the counter of the incoming rings, and if this counter exceeds the amount of rings which CST host application has to wait before going off hook, it issues a series of AT commands, causing CST chip to switch into voice mode and go off hook.

4) After going off hook, CST host initiates playback of the greeting message located in the file on PC, sending it as G.726 compressed bitstream to CST chip via serial link.

5) While message is played, starts recording a message from the telephone line to a file on PC. PCM samples from the telephone line are compressed in CST chip using G.726 ADPCM codec, and are passed to PC as compressed bitstream.
As the message is being recorded, CST chip normalizes its amplitude (using AGC), scans for DTMF and CPTD tones (in order to pass information about them via shielded codes to host) and detects periods of silence in the speech (using VAD). If silence is detected, CST chip stops sending compressed bitstream to PC, but instead starts sending much shorter silence frames, which contain information about Noise Spectral Envelope to PC.
Since echo canceller is turned on, the messages played back to the line are going to be suppressed in the recorded message.

6) When playback duration exceeded or file ended CST host stops recording and starts playing of recorded message.

7) In case of BUSY tone is detected, CST host stops recording the message and saves it in a file. It causes CST chip to go on hook, and goes to step 1 of this scenario. This issues on all stages of processing.

Chapter 11

Product Installation Procedure

This chapter provides brief instructions on installation of the CST SDK, setup of the CST host to communicate with the C54CST EVM, and the setup of Windows™ to communicate with the C54CST as a generic modem.

Topic **Page**

11.1 Installing CST SDK

To install CST Documentation, CST open code, CST Flex mode examples, CST host application and other auxiliary software, run the file `CSTInstall.exe` (it is a self-extracting archive). Specify the destination folder to extract the files to (it can be any folder, but it is recommended to specify your TI folder, usually `c:\ti\C5400`, so that all CST files would reside in `c:\ti\C5400\CST`), and press “Extract”. The extracted files will be located in the directories as described in section 11.2.

Note: Important notes for Code Composer Studio version 2.1 users

Notes for Code Composer Studio version 2.1 users
(for correct flex application compilation):

1) After extraction is done, you need to update the Chip Support Library (CSL) as follows:
   Follow the instructions in the `Readme.txt` file provided in the `CST\CCS_Patch` folder

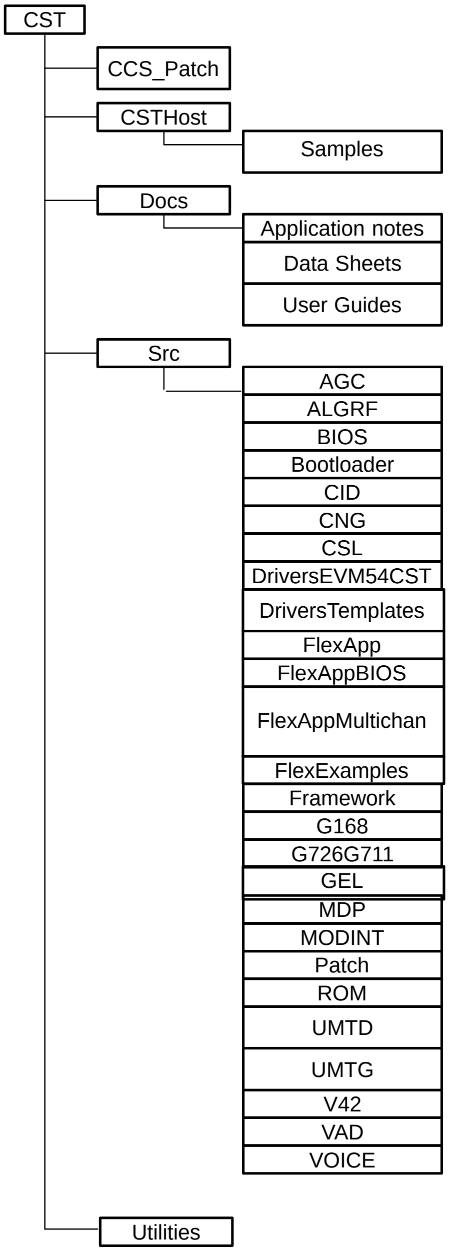
2) If an error message is displayed during compilation, saying that some header file (such as `xdas.h` or `std.h`) could not be opened, the correct paths need to be added in the CCS options of flex application project, as follows:
   Open ”Project->Build Options” and select the ”Preprocessor” category. In the ”Include Search Path” window correct the following paths to point to your specific TI folder location:
   ”`C:\ti\c5400\xdais\include;`
   `C:\ti\c5400\xdais\src\api`”
   It is a bug in CCS 2.1xx, and it is fixed in CCS 2.2.

To create CST Flex mode applications, the user needs Code Composer Studio from Texas Instruments. Its installation is described in corresponding CCS documents, and is beyond the scope of this document.

11.2 Description of Product and Document Directory Tree

CST documentation and software has the following directory tree:

Figure 11-1. CST Documentation and Software Directory Tree



CST	CST root folder
CCS_Patch	Update files for the Code Composer Studio 2.1, contain CSL library update for CST chip
CSTHost	CST Host terminal application CSTHost.exe
Samples	Voice files to be played or recorded. Standard CST greeting at different bit rates.
Docs	CST Documentation
Application notes	Application notes, both for Chipset and Flex modes
Data Sheets	Data Sheets for each algotithm: short description, RAM/ROM/MIPS requirements; CST Chip hardware data sheet
User Guides	User’s Guides for each algorithm, for CST in general and for CST Framework
Src	CST Interface and Open Source Code
AGC	Automatic Gain Control interface C and H files
ALGRF	Reference Framework 3 XDAIS Algorithm instantiation code
BIOS	Files needed to run CST under DSP/BIOS
Bootloader	CST Bootloader assembly source code
CID	Caller ID interface C and H files, Wrapper and Parser open code
CNG	Comfort Noise Generator C and H files
CSL	Chip Support Library, C and H files specific for CST device
DriversEVM54CST	C54CST EVM Drivers for UART, DAA and peripheral drives
DriversTemplates	Template Files for new UART, DAA and peripheral drivers (includes a compatible project)
FlexApp	CST Flex application project template (not using DSP/BIOS)
FlexAppBIOS	CST Flex application project template, DSP/BIOS-based
FlexAppMultichan	A multivhannel CST Flex application (both projects: for DSP/BIOS and w/o). Shows how to create 2-channel modem application with additional external DAA
FlexExamples	CST Flex mode examples - C files
Framework	CST Framework sources
G168	Echo Canceller interface C and H files
G726G711	Waveform codec interface C and H files
GEL	Code Composer GEL file, needed to run a Flex Application
MDP	Modem Data Pump (v.42bis/V.32bis) interface C and H files
MODINT	Modem Integrator interface files, Modem Controller sources
Patch	CST Chipset mode Patch (patches some modem functions)
ROM	CST ROM reference files (global reference, memory map)
UMTD	Universal Multi-Tone Detector interface C and H files, and configuration files to implement DTMF and CPT Detector
UMTG	Universal Multi-Tone Generator interface C and H files, and configuration files to implement DTMF and CPT Generators
V42	V.42 and V.42bis interface C and H files
VAD	Voice Activity Detector interface C and H files
VOICE	Voice Controller sources (voice stream framer, etc.)
Utilities	Several usefull utilities for CST

11.3 Setting up CST Host

To control CST chip in Chipset mode via serial link, the user needs CST host application or some generic terminal, for example HyperTerminal® or Procomm Plus®. CST host application is extracted to the local hard drive along with other CST files and it does not need any special installation procedure. Just run `CST\CSTHost\CSTHost.exe`.

A COM port which is used to communicate with CST chip need to be setup with the following settings:

115200 bps, 8 bits of data, 1 stop bit, no parity, Hardware flow-control

Read section 10.2.1 to learn how to setup CST host to these settings. Setting up a generic terminal should be similar to this process.

If you also have EVM C54CST board, read the whole section 3.3 on how to install and configure the hardware (see *TMS320C54CST Evaluation Module. Technical Reference.* Spectrum Digital, Inc.).

11.4 Installing Modem Drivers for CST Chips in Windows™

CST chip can also be controlled by a Windows™ modem driver as a generic modem. In order to register CST chip in Windows, open **Control Panel**, then **Modems**, and add a modem from a list (without Windows auto-detection). Choose **Standard Modem Types**, **Standard 14400 bps Modem,** and select appropriate COM port, to which CST chip is connected. This will create a modem with the name something like “Standard 14400 bps Modem”. Then open properties window for this modem, and change COM port maximum speed to 57600. Also, make sure that hardware flow control (RTS/CST) is enabled, in Connection**/**Advanced.

Note: Notice: TMS320C54CST chip and UART capabilities

UART in TMS320C54CST chip does not have hardware auto-baud capabilities, and since the COM port speed selected by default after chip reset is 115200, it may take a while before software auto-baud implemented in CST UART driver detects a new speed. For this reason, in some cases it may be more convenient or even necessary to create a modem driver in Windows not as standard 14400 bps modem, but as standard 28800 bps modem, because in this case maximum COM port speed in Windows modem driver can be set to 115200, thus eliminating the problem of auto-baud detection delay at start up.

Chapter 12

Chipset Mode Testing and Troubleshooting

This chapter describes several test procedures that the user can perform to make sure that CST chip operates correctly in a specific hardware environment, and gives specific step on how to troubleshot some of the common problems.

Make sure to configure EVM, Host's COM port and CST chip DAA correctly before performing these tests (read sections 3.3 and 10.2 on how to configure them correctly).

You can also test CST chip and functionality of the CST software by running examples described in *Client Side Telephony (CST) Chipset Mode* (SPRA859).

Topic **Page**

12.1 Testing UART

The following UART functionality can be tested:

1) Data transfer

   This general functionality can be easily tested using AT parser.

   AT parser should echo the input commands, and should output responses correctly. For example, you can type:
   `AT`
   Response: `OK`
   `AT$`
   Response: <…S-registers Help…>

2) Autobaud

   Make sure that echo is enabled in AT parser (enter `ATE1` command). While AT parser is in command mode, change COM port speed from 115200 to a lower speed, from 57600 to 19200. Type several repetitive sequences “`ATATAT`”. When you see correct echo as “`ATATAT`”, it means the software auto-baud in CST chip has recognized the serial port speed correctly.

3) Hardware Flow Control

   Connect to remote modem using V.22bis protocol, with error correction enabled and software compression disabled. Type:
   `ATB3`
   `AT\N1%C0`
   `ATDTxxxx` *(where xxxx is the number of the remote modem)*
   After modems connect, send a big piece of data to the CST modem (you can do this by pasting some big text file from the clipboard). On remote end you should see that all this data is received identical to the original data. If some blocks of consecutive data are lost, it may be because hardware flow control is not working in serial connection between host and CST chip.

   Hardware flow control operation can also be observed using LED DS5 (section 3.5). If this LED starts blinking when big piece of data is being transferred from host to DSP, it means flow control operated correctly.

4) Intensive duplex data transfer

   This functionality can be tested in several ways:

   a) PCM Test: Run *Play Greeting and Record* script in voice host (see section 10.3) with PCM codec enabled (G.726 disabled) and hear the quality of the recorded sound, which will be played after the greeting (you should speak something while listening to greeting message).
      This script tests UART in duplex mode, with over 80 kilobits sent in both directions per second.
      Although the load is intensive in this test, it also has a drawback: if some bytes are missed in serial link, it is almost inaudible for the tester.

   b) G.726 40 kbps test: Since for G.726 codec correctness of bitstream plays much greater significance (especially when the bit-rate is 5 or 3 bits per sample), it is also recommended to test intensive transfer in this mode.
      Run *Play Greeting and Record* script in voice host (see section 10.3) with G.726 40 kbps codec enabled and hear the quality of the recorded sound, which will be played after the greeting (you should speak something while listening to greeting message).
      This script tests UART in duplex mode, with over 50 kilobits sent in both directions per second.

If some of these tests fail, check the following:

1) CST's UART is connected by modem cable to host COM port, and all the circuits in the cable are connected.

2) Host's COM port is correctly configured for data rate and flow control.

3) Try to do the same tests with a standard conventional external modem. If they fail too, something is wrong in the cable or COM port settings.

12.2 Testing DAA

The following DAA functionality can be tested:

1) Sensing ring signal

   Make sure that CST chip is in on-hook state (type `ATH` command). Dial the number of telephone line to which CST chip is connected.

   RING result codes should appear on the terminal connected to CST chip.

2) Going off hook

   Make sure that CST chip is in on-hook state (type `ATH` command). Dial the number of telephone line to which CST chip is connected.

   When RING result code appears on the terminal, type `ATH1` command. If CST chip was able to go off hook, ringback tone should stop in the phone from which you are calling, and you should hear silence.

3) Going on hook

   Do the previous test. Then, while in off hook mode, type `ATH` to go on hook. If CST chip was able to go on hook, you should hear busy tone or Disconnected message in the phone from which you are calling.

4) Enabling caller ID path in on-hook state

   Make sure that CST chip is in on-hook state (type `ATH` command) and that caller ID is enabled (type `AT#CID1`). Also, make sure that the phone line connected to CST chip is analog and is subscribed to caller ID service.

   Dial the number of telephone line to which CST chip is connected.

   After first RING, CST framework enables CID path while staying on hook so that CID could receive the information from the telephone station. If CID received the information from the station and printed it on the terminal, it means that CID path was enabled successfully.

5) Sampling signal from phone line (A/D conversion)

   This functionality can be tested in several ways:

   a) CST chip recognizes DTMF tones in voice mode. Type:
      `AT#CLS=8` *(call to CST chip after this)*
      `ATH1`
      `AT#VTX`
      Start pushing buttons on the phone from which you are calling, and CST should output detected DTMF digits to the terminal as shielded codes.

b) Run *Record* script in voice host (see section 10.3) with PCM codec enabled (G.726 disabled) and hear the quality of the recorded sound using the standard sound editor (recorded file will be in 8 kHz, MONO, 8-bit, µ-law format).

c) Connect to remote modem using V.22bis protocol via good quality telephone connection, with disabled error correction (V.42). Type:
`ATB3`
`AT\N0`
`ATDTxxxx` *(where xxxx is the number of the remote modem)*
If modem does not connect, or if after connection the modem starts outputting some strange symbols to the terminal, this may be because the signal from phone line is not sampled correctly (usually it may be overamplification or some additive noise).

d) Connect to remote modem using V.32bis protocol via good quality telephone connection, with disabled error correction (V.42). Type:
`ATB0`
`AT\N0`
`ATDTxxxx` *(where xxxx is the number of the remote modem)*
If modem does not connect at 14400 (connects at lower rate), or if after connection the modem starts outputting some strange symbols to the terminal, this may be because the signal from phone line is not sampled correctly (usually it may be overamplification or some additive noise). However, since the quality of reception in V.32bis also depends on how well modem suppresses its own echo signal, the reason of the problem may also be in how signal is output (there may be some non-linear distortions in echo path).

6) Outputting signal to phone line (D/A conversion)

This functionality can be tested in several ways:

a) CST chip generates DTMF tones. Type:
`ATH`
`ATDTxxx` *(where xxx is the number to dial)*
If the number is dialed correctly, the signal from DAA is output to phone line more or less correctly (this is a rough test).

b) Run *Playback Recording* script in Voice Host (see section 10.3) with PCM codec enabled (G.726 disabled) and hear the quality of the played file.

c) Connect to remote modem using V.22bis protocol via good quality telephone connection, with disabled error correction (V.42). Type:
`ATB3`
`AT\N0`
`ATDTxxxx` *(where xxxx is the number of the remote modem)*
If modem does not connect, or if after connection the remote modem starts outputting some strange symbols to the terminal, this may be because CST's modem does not transmit the signal correctly (usually it may be because of non-linear distortions in output DAA path, such as saturation).

d) Connect to remote modem using V.32bis protocol via good quality telephone connection, with disabled error correction (V.42). Type:
`ATB0`
`AT\N0`
`ATDTxxxx` *(where xxxx is the number of the remote modem)*
If modem does not connect at 14400 (connects at lower rate), or if after connection any of the modems (CST's or remote) starts outputting some strange symbols to the terminal, this may be because CST's modem does not transmit the signal correctly (usually it may be because of non-linear distortions in output DAA path, such as saturation). The quality of reception in V.32bis also depends on how well modem suppresses its own echo signal. If the signal is output with some distortions, modem's echo canceller may not be able to suppress it well enough to hear the remote modem signal well.

If some of these tests fail, check the following:

1) Make sure that DAA settings that are loaded via S-registers, are correct for your country or your telephone station. If you are not sure what the correct settings are, try changing the settings and perform the test again, until you find the settings at which all the tests pass.

2) Make sure that external DAA's analog circuitry is correct (if you develop your own hardware) and the signal is not overamplified or saturated.

3) Make sure that DSP is clocked at 14.7456 MHz precisely.

12.3 Troubleshooting Procedures

Note: Troubleshooting and Testing

In case of any problem, please, first try to perform elementary tests described above, in order to localize the problem better.

Table 12-1. Troubleshooting Procedures

No	Problem	Explanation and Solution
1	Modem does not connect, but outputs “NO CARRIER” message almost immediately after dialing	This happens when modem integrator can not create one of the modem objects because of the shortage of dynamic memory. This most often may happen if V.42bis is enabled and dynamic memory is occupied with something else already. **Disable V.42bis, and, if it does not help, V.42, or free dynamic memory or increase dynamic memory size.**
2	Creation of xDAIS object fails even though it seems that there is enough dynamic memory	Two possible reasons for this: 1) Specific XDAIS object may require alignment for its dynamic data, which in addition to the requested size exceeds available memory 2) Dynamic memory is severely fragmented and there is no continuos memory space enough to allocate that specific XDAIS object. **In the first case the user may try to create all the XDAIS objects, which are needed simultaneously, in a different order, and this may help Memory Manager to allocate dynamic memory for them more efficiently.** **In the second case, to avoid such situation, the user should try to delete XDAIS objects in the reverse order than they were created, because CST Memory Manager does not have a mechanism to defragment memory. To defragment memory in a running system, the user has to delete all objects.**

Table 12-1. Troubleshooting Procedures (Continued)

No	Problem	Explanation and Solution
3	CST chip does not respond to any AT commands send via serial port, and does not echo any symbols back	There may be several reasons for that: 1) COM port on host is not configures correctly. **Configure COM port to 115200 bps**, **8 bits -1stop bit – no parity, hardware flow control** 2) Serial cable is not appropriate. **Use only modem serial cable**. 3) CST Solution is not running inside CST chip. **In chipset mode, make sure that INT1 pin is connected to logic 0 and reset the chip. If using EVM, make sure to configure it according to section 3.3**. **In Flex mode, make sure that CST solution is initialized correctly and** `CSTServiceProcess()` **is called with 5 ms period or less. Also, make sure that DAA driver initialized DAA correctly and it receives 8 kHz samples from it.**
4	When playing or recording voice data, some clicks or noise can be heard periodically	This usually happens when host application does not provide continuous operation with serial port. This, in turn, may happen because host application is not able to run in real-time on PC. **Close all resource consuming applications on PC.**
5	CST chip does not physically go off hook after ATH1 command	DAA does not create loop current big enough for telephone station to realize that there is an active load. **Change DAA International settings to fit the specifics of your local telephone station. For example, change line monitor mode or DC Termination. Read more about these settings in section 10.2.2.**

Index