Application Report
SPRA862 - January 2003

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

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

Leonid Purto *SPIRIT CORP*
Maxim Silchev

ABSTRACT

This document describes basic flex examples, located in the CST software development kit (SDK), in directory CST\Src\FlexExamples\.

The flex examples are intended to illustrate integration of your applications into the SPIRIT's Client Side Telephony (CST) Framework. The examples show how to use the CST action interface in the flex mode.

This document does not describe the CST action interface, nor does it contain any other fragments of the CST specification. The document also does not describe how to modify the examples to adjust their functionality, build the applications and download them into the DSP, reconfigure the CST Framework to work with your firmware, and so on. To get acquainted with the CST flex mode and learn CST, read [1].

For related documents please see section 3.

Go to http://www.spiritDSP.com for the latest version of the CST documentation and patch.

Contents

List of Figures

List of Tables

1 Introduction

Unlike many modem chipsets, the CST chip grants you the opportunity to develop your own applications, download them into the chip, and control the CST solution inside. Thus, there are two main modes of operation of the CST chip – chipset mode and flex mode.

In the chipset mode, only the CST software is running inside the CST chip. It can be controlled externally by assembly test (AT) commands transferred over the serial link between the CST on-chip universal asynchronous receiver/transmitter (UART) and your host. In this mode, the CST chip can be used as a standard data modem with voice features, including duplex voice transfers, as all standard functionality of the CST Software is also accessible through AT commands.

In the flex mode, your code is downloaded into the CST chip, where it runs, using the CST Software in read-only memory (ROM) as a library. This mode gives you a more flexible access to the CST software, as it is possible to interact with its components directly when your control software is contained inside the chip as well. The flex mode also allows you to build applications using only the CST chip, without the need for any host controller and processing the AT commands and responses to them.

This document describes twelve standard flex applications, located in the directory, Src\FlexExamples\.

1.1 Abbreviations

For the purposes of this API description, the following abbreviations are used:

CID	Caller ID
Chipset mode	Mode of CST chip operation when it is controlled only externally, by AT commands sent over serial link.
CPTD	Call progress tone detector
CST	Client side telephony. Also means the CST chip solution.
DTMF	Dual-tone modulated frequency signal
Flex mode	Mode of CST chip operation when it is controlled internally by your program, downloaded into internal or external memory of the CST chip.
UART	Universal asynchronous receiver/transmitter, the chip which allows data exchange over serial link
UMTD	Universal multi-tone detector
XDAIS, XDAS	TMS320 DSP Algorithm Standard (also known as eXpressDSP).

2 Flex Examples

The flex examples are intended to illustrate integration of your applications into the SPIRIT's CST Framework. The examples show how to use the CST action interface in the flex mode.

The examples are presented in order of increasing complexity. Each example can also work under DSP/BIOS™. To learn how to compile and run the examples, read the section "Flex Mode Quick Start" in [1].

2.1 Template for Flex Application – main1 (empty app).c

This is the first example. It contains an empty project and can be used as a template for flex applications. All described examples are based on this template, so the listed source code of the following examples will contain only differences from the template. The source annotation is shown below.

```
#include "..\Framework\CSTChannel.h"
#include "..\Framework\CSTAction.h"
```

```
#ifdef _CST_WITH_BIOS
#define BIOS_APPLICATION 1
#else
#define BIOS_APPLICATION 0
#endif

#if !BIOS_APPLICATION
  //Prevent improper compilation
  #define MyPeriodicThread _MyPeriodicThread
  asm("__sys_memory .usect \".sysmem\",0");
#endif //!BIOS_APPLICATION
  asm("__STACK_BEGINNING .usect \".stack\",0");
```

This C-preprocessor definition is used to build the example for the DSP BIOS environment. Make sure it is defined in the project if you want to run the application under DSP BIOS.

DSP/BIOS is a trademark of Texas Instruments.

```
static bool MyCallback (tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,int
Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {
    //////////////////
    //USER'S CODE...//
    //////////////////
  }
  return 1;
}
```

Implementation of the main callback function called from CST. Each CST action-oriented program should implement this method. Almost all data and information messages go through this callback routine.

For information about the function arguments see the section entitled "User's Callback Function to Process CST Commander Messages" in [1].

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

  //////////////////
  //USER'S CODE...//
  //////////////////
}
```

Your periodic thread function. The thread has its own internal time synchronization and will call your callback function, MyCallback().

In the case of BIOS configuration, the thread function is normally called from a software interrupt (SWI). If it is not called from a periodically posted SWI, it should be called as often as possible. Frequent calls do not lead to noticeable overhead because all time consuming tasks are executed every 10 new samples (sampling frequency is 8 kHz). If there are not 10 new samples, the function will do nothing.

Calling this function too seldom will cause failures in time-sensitive state machines, such as distributed application architecture (DAA) ring detector and samples losses.

```
void MyInitialization ()
{
  //////////////////
  //USER'S CODE...//
  //////////////////
}
```

Your initialization function.

You can insert custom initialization code here.

```
#define EVM54CST_59MHZ_MULT  4
#define EVM54CST_118MHZ_MULT 8
```

```
void main ()
{
```

Standard main function. The function code fits most of flex applications, so you do not need to change it.

```
#if !BIOS_APPLICATION
  //Processor boot init.
  CST_DSPInit ();
#else
  initBiosConst();
#endif //BIOS_APPLICATION
```

```
  CST_bssInit ();
```

CST internal data sections initialization.

`TargetBoardInit (BIOS_APPLICATION, EVM54CST_118MHZ_MULT, 2);`	Target board initialization. This function is specifically for the EVM5406 board, and you may need to change it according to your board hardware specification.
`CSTAction_Init (&Ch0,BIOS_APPLICATION, MyCallback);`	CST Framework initialization
`CSL_init();`	CSL initialization
`TargetPeriphInit (BIOS_APPLICATION, 1);`	Target board's peripheral initialization. You may need to change this according to your UART and codec installed on the board, if they are different from the TMS320C54CST on-chip UART and DAA.

`#if !BIOS_APPLICATION`
`  asm (" rsbx INTM");`
`#endif //BIOS_APPLICATION`	

`MyInitialization ();`	Perform your specific initialization.

`#if !BIOS_APPLICATION`
`  while ( 1 )`
`  {`
`  MyPeriodicThread ();`
`  }`
`#endif //BIOS_APPLICATION`
`}`
	Main loop.

2.1.1 Initialization to Run Under DSP/BIOS

The initialization in DSP/BIOS-based flex applications is different from the initialization in non-DSP/BIOS applications (see Figure 1). The differences are listed below.

- You should call a DSP/BIOS processor initialization function on boot.

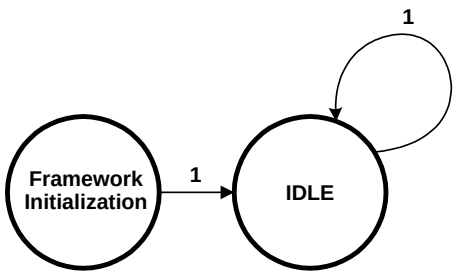
```
#if !BIOS_APPLICATION
  CST_DSPInit ();
#else
  initBiosConst();
#endif //BIOS_APPLICATION
```

- You should not enable hardware interrupts during initialization; the DSP/BIOS will do that at the right time later.

```
#if !BIOS_APPLICATION
  asm (" rsbx INTM");
#endif //BIOS_APPLICATION
```

- You should call the periodic thread function from an SWI rather than from the `main()` function. See Figure 1.

```
#if !BIOS_APPLICATION
  while ( 1 )
  {
    MyPeriodicThread ();
  }
#endif //BIOS_APPLICATION
```



NOTE: State transitions marked with “1” are unconditional transitions.

Figure 1. Empty Application

2.2 “Hello!” Application – main2 (hello).c

If using a terminal program other than CSTHost, the following settings are required:

Bits per second:	115200
Data bits:	8
Parity:	None
Stop bits:	1
Flow control:	Hardware

The idea of this example is to output the message “Hello!” to the UART (see Figure 2). To observe the results of this example application execution, you should connect the EVM5406 board to the PC with a COM cable, and run a terminal program. The source annotation is shown below.

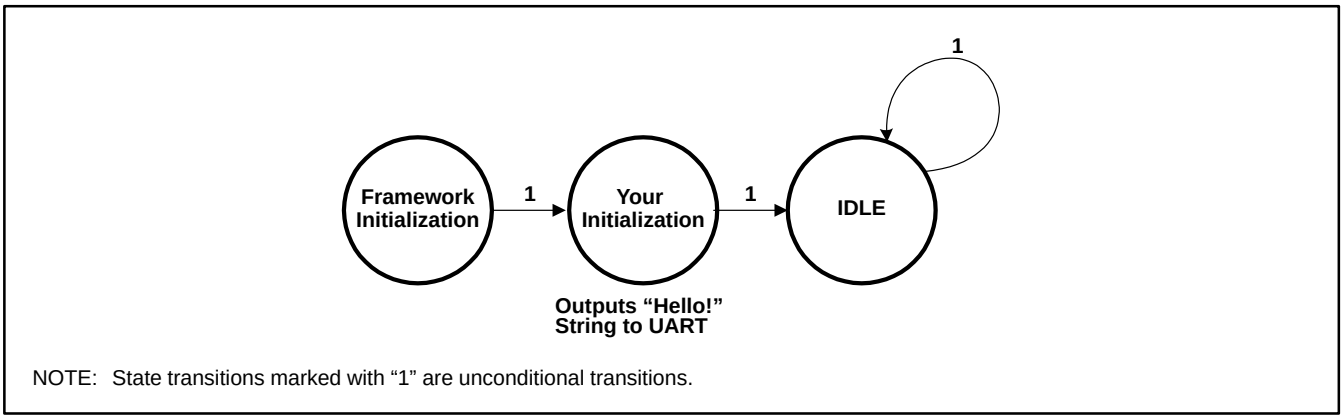
```
void MyInitialization ()
{
  /////////////////
  //USER'S CODE...//
  /////////////////

  //Just print out to the UART
  UartPutString (&Ch0,"Hello!\r\n");
}
```

Your initialization.

In this example, the string “Hello!” has just been output to the UART.

The string “Hello!” should appear in the terminal program’s window after executing this example. See Figure 2.



NOTE: State transitions marked with “1” are unconditional transitions.

Figure 2. “Hello!” String Output

2.3 ON/OFF HOOK Processing – main3 (mute).c

If using a terminal program other than CSTHost, the following settings are required:

Bits per second:	115200
Data bits:	8
Parity:	None
Stop bits:	1
Flow control:	Hardware

This example demonstrates OFF/ON HOOK processing. It contains a simple state machine. The example outputs notification messages to the terminal via the UART to inform you of RING or BUSY detection.

For correct OFF/ON HOOK processing, you should add handlers for external event messages eme_PERIPH_DATA and eme_CPTD_DATA in the function MyCallBack(). The source annotation is shown below.

```
  typedef enum {
    ms_WAIT_FOR_RING,
    ms_GO_OFF_HOOK,
    ms_WAIT_FOR_BUSY,
    ms_TURN_OFF_GO_ON_HOOK
  } tMainState;
tMainState MainState=ms_WAIT_FOR_RING;
```

Defining state machine states.

State Machine State	Description
ms_WAIT_FOR_RING	Initial state
ms_GO_OFF_HOOK	Rings detected.
ms_WAIT_FOR_BUSY	Wait for busy.
ms_TURN_OFF_GO_ON_HOOK	Go on hook, and to the initial state.

```
static bool DoStandardOperation (
  tCSTStandardOperationType OperationType,
  uint8 *pData)
{
  tCSTAction Action;
  int ITemp;

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


  if ( pData )
    for(ITemp=0;
      ITemp<CST_STANDARD_OPERATION_DATA_LEN;
      ITemp++ )
      Action.Action.CSTStandardOperation.aData[
        ITemp] = pData[ITemp];

  return CSTAction (&Ch0,&Action) !=
    cmr_TRY_AGAIN;
}
```

All described flex examples use existing scripts of atomic commands to perform standard operations.

OperationType specifies the script to perform a standard operation.

This routine can be used in most of standard flex applications without any modifications.

Parameter	Description
OperationType	Selects a script
pData	Attached data depends on the operation type (e.g., phone number or custom script. Zero value of pData means absence of additional data for the operation.

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

    case eme_PERIPH_DATA:
      if ( Data==cpe_RING )
      {
        MainState=ms_GO_OFF_HOOK;
        ...
      }
      break;

    case eme_CPTD_DATA:
      if ( Data==ICPTDDET_BUSY )
      {
        MainState=ms_TURN_OFF_GO_ON_HOOK;
        ...
      }
      break;
```

External messages callback function. It receives messages from CST and processes only eme_PERIPH_DATA, eme_CPTD_DATA, and eme_AUTOTURNOFF_ALL messages.

Switch state machine's state to ms_GO_OFF_HOOK upon RING signal detection.

BUSY signal detection terminates the connection. CST will turn off everything automatically (see eme_AUTOTURNOFF_ALL) when BUSY signal is detected, so in most cases it is not necessary to have a handler for BUSY signal detection.

```
      case eme_AUTOTURNOFF_ALL:
        MainState=ms_TURN_OFF_GO_ON_HOOK;
        break;
    }
    return 1;
}
```

If you do not want CST to automatically turn off all and hang up upon busy detection, the function should return 0 instead of 1.

```
   if ( Data==atk_BUSY )
      return 0;
```

Some applications (like this) do not not need to process this message.

```
void MyInitialization ()
{
  UartPutString (&Ch0,
    ”Please, call to the phone\r\n”);
}
```

Your initialization.

Just output the string “Please, call to the phone” to the UART.

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

  switch ( MainState )
  {
    case ms_GO_OFF_HOOK:
      if (DoStandardOperation(sot_OFF_HOOK,0))
      {
        MainState=ms_WAIT_FOR_BUSY;
        ...
      }
      break;
```

When the RING message arrives, the sot_OFF_HOOK standard operation executes. The corresponding script (aOffHook) also activates the DTMF and CPTD detectors.

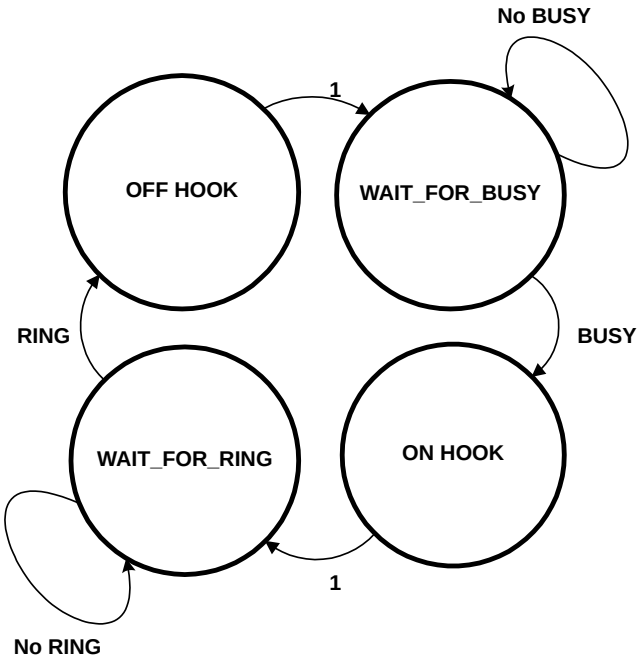
(The script aOffHook is described in CSTAtomic.c).

```
    case ms_TURN_OFF_GO_ON_HOOK:
      if (DoStandardOperation(
            sot_TURNOFF_ALL,0))
      {
        MainState=ms_WAIT_FOR_RING;
        ...
      }
      break;
  }
}
```

In case of BUSY signal detection, all active algorithms are turned off by executing the standard operation, sot_TURN_OFF_ALL. The application comes back to the initial state upon completing the script execution.

(The corresponding script aTurnOffAll is described in CSTAtomic.c).

WAIT_FOR_RING is the initial state. The state machine is waiting for RING signal detection. Upon RING detection, the application performs OFF HOOK processing and stops in the WAIT_FOR_BUSY state. The CPTD detector will start automatically. The application will be in this state until BUSY signal is detected. You will usually make your own processing here. When the BUSY signal is detected, CST performs ON HOOK processing and turns off all active algorithms, so it is not necessary for you to process BUSY messages. After ON HOOK processing, the state machine comes back to the initial state. See Figure 3.



NOTE: State transitions marked with “1” are unconditional transitions

Figure 3. OFF/ON HOOK Processing

2.4 Music Playback Control – main4 (music on hold).c

If using a terminal program other than CSTHost, the following settings are required:

Bits per second:	115200
Data bits:	8
Parity:	None
Stop bits:	1
Flow control:	Hardware

This example is based on the previous one. The difference is that it plays a number of short sounds (pseudo music) in a loop, while being off hook. Music playing is controlled by DTMF symbols you may send from a phone by pressing buttons. Table 1 shows accepted DTMF symbols and their purposes. The source annotation follows.

Table 1. Accepted DTMF Symbols for Music Playback Control

DTMF Symbol	Purpose
0	Decreases loudness
1	Increases loudness
*	Stops playback
#	Proceeds to playback

```
static void SetRegister (int InternalSReg,int
Value)
{
  tCSTAction Action;

  Action.ActionType=cat_SET_REGISTER;
  Action.Action.CSTConfigCommand.InternalSReg=
    InternalSReg;
  Action.Action.CSTConfigCommand.Value=Value;
    CSTAction (&Ch0,&Action);
}
```

This function is used to set a new S-register value. To set an S-register value, you should call this function with the S-register number and its new value.

```
static int GetRegister (int InternalSReg)
{
  tCSTAction Action;

  Action.ActionType=cat_GET_REGISTER;
  Action.Action.CSTConfigCommand.InternalSReg=
    InternalSReg;
  return (int)CSTAction (&Ch0,&Action);
}
```

This function is used to get an S-register value. To get the S-register value you should call this function with the S-register number.

```
static void SendVoiceData ()
{
    ...
  while ( DataAction.Action.CSTServiceMessage.
    DataLength<CST_MAXDATALENGTH )
  {
    DataAction.Action.CSTServiceMessage.aData[
      DataAction.Action.CSTServiceMessage.
        DataLength++]=
       pMusic[MusicSoundPos++];
    ...
  }
```

This function prepares music bitstream for playing. At first, fill the action message data field.

```
ActionResult=CSTAction (&Ch0,&DataAction);

if ( (ActionResult==cmr_RESULTOK)||
     (ActionResult==cmr_EXECUTING) )
  DataAction.Action.CSTServiceMessage.
    DataLength=0;
}
```

The action message will be sent to the CST action when its filling is complete.

Upon cmr_RESULTOK or cmr_EXECUTING result codes, new data can be put, instead of the old.

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

    case eme_PERIPH_DATA:
      ..
    case eme_CPTD_DATA:
      ..
    case eme_AUTOTURNOFF_ALL:
      ..
```

External messages callback function. It receives messages from CST and processes only eme_PERIPH_DATA, eme_CPTD_DATA, and eme_AUTOTURNOFF_ALL messages.

Processing of these messages is the same as in the previous examples.

```
    case eme_DTMF_DATA:
      if ( MainState==ms_TURN_OFF_GO_ON_HOOK )
        break;
      switch ( Data )
      {

        case '*':
         MainState=ms_TURNOFF_VOICE_TXDATA;
         ...
         break;

       case '#':
         if ( MainState!=ms_PLAY_MUSIC )
         {
            MainState=ms_TURNON_VOICE_TXDATA;
            ...
          }
          break;

        case '0':
          ITemp=GetRegister (srd_VOICE_GAIN);
          if ( ITemp<30 )
            ITemp++;
          SetRegister (srd_VOICE_GAIN,ITemp);
          ...
          break;

        case '1':
           ITemp=GetRegister (srd_VOICE_GAIN);
           if ( ITemp>0 )
             ITemp--;
           SetRegister (srd_VOICE_GAIN,ITemp);
           ...
           break;
      }
      break;
  }
  return 1;
}

void MyPeriodicThread ()
{
  CSTAction_Process (&Ch0);

  switch ( MainState )
  {
    case ms_GO_OFF_HOOK:
      if (DoStandardOperation(sot_OFF_HOOK,0))
      {
        MainState=ms_TURNON_VOICE_TXDATA;

        SetRegister (srd_VOICE_GAIN,0);
        SetRegister (srd_VOICE_BPS,8);
        ...
      }
      break;
```

TURN OFF protection. Process DTMF symbols in the off-hook state only.

DTMF symbol '*' detection stops music playing. It changes the state of the state machine to turn the voice TX data path off.

DTMF symbol '#' detection starts music playing. It changes the state of the state machine to turn the voice transmit (TX) data path on.

DTMF symbol '0' decreases the music's volume.

DTMF symbol '0' increases the music's volume.

OFF HOOK action (to be performed by DoStandardOperation() function) executes a standard script for going off hook.

The maximum voice volume level and voice bit rate are then set, and the application transitions to the next state.

```
    case ms_TURNON_VOICE_TXDATA:
      if (DoStandardOperation(
         sot_TURNON_VOICE_TXDATA,0))
      {
        MusicSoundPos=0;

        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;

     MainState=ms_PLAY_MUSIC;
       ...
      }
      break;
```

Activate voice TX path.

Set pointer to the beginning of the music vector.

Prepare the action message for voice TX path data.

Switch to the next state.

```
    case ms_PLAY_MUSIC:
      SendVoiceData ();
      break;
```

This is the music playback loop. Playback parameters can be adjusted by DTMF symbols (see the function MyCallBack()).

```
    case ms_TURNOFF_VOICE_TXDATA:
      if (DoStandardOperation(
            sot_TURNOFF_VOICE_DATA,0))
        MainState=ms_KEEP_SILENCE;
      break;
```

Turn the voice TX path off and switch the state machine to the SILENCE state.

The application does nothing in the SILENCE state.

```
    case ms_TURN_OFF_GO_ON_HOOK:
      if (DoStandardOperation(
            sot_TURNOFF_ALL,0))
      {
        MainState=ms_WAIT_FOR_RING;
        ...
      }
      break;
  }
}
```

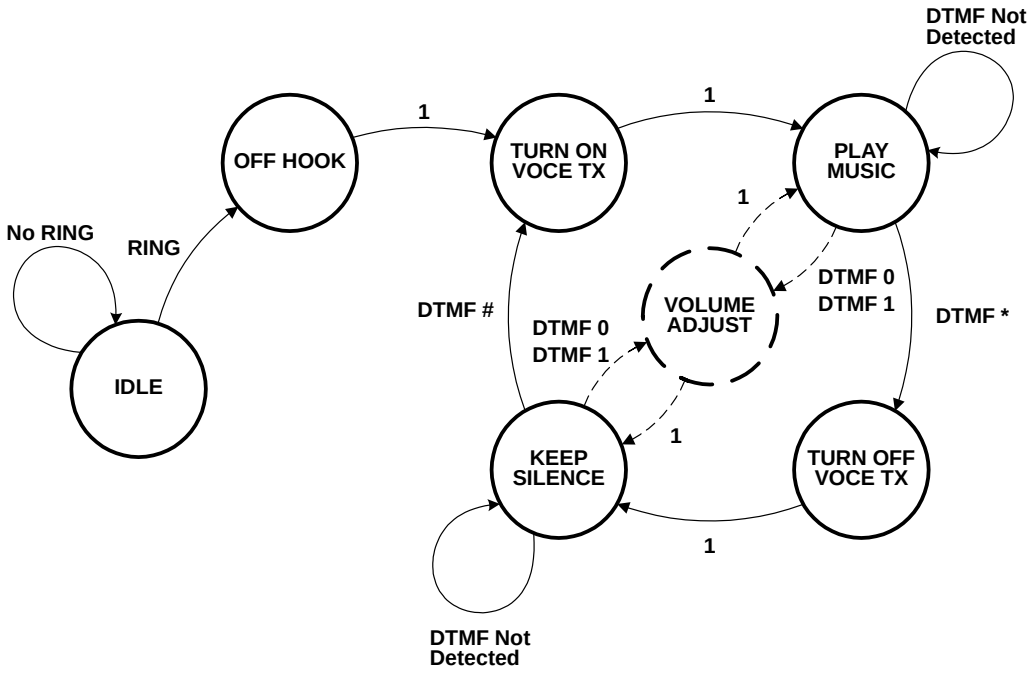
In case of BUSY signal detected, it turns off all active algorithms by executing the standard operation, sot_TURN_OFF_ALL, and then comes back to the initial state.

(The corresponding script aTurnOffAll is described in CSTAtomic.c).

Busy detection is not displayed, in order to simplify Figure 4. Busy detection is performed in all but IDLE states of flex application. The application returns to the initial (IDLE) state upon BUSY signal detection.

In the IDLE state the system is waiting for a RING signal to be detected. Upon RING detection, it performs OFF HOOK processing, which automatically starts CPT detector and switches to the next state. TURN ON VOICE TX turns the Tx voice path on and switches the state machine to the next state.

The system is in the PLAY MUSIC state now. You hear the music in this state. The DTMF detector is active and symbols “0”, “1”, “*” and “#” are accepted. Symbols “0” and “1” adjust the music’s volume – this is depicted as going to the state VOLUME ADJUST. The symbol “*” turns the voice Tx path off. In the state KEEP SILENCE, you do not hear music. The symbol “#” returns the system to the state PLAY MUSIC and you will be able to hear the music again.



NOTE: State transitions marked with “1” are unconditional transitions.

Figure 4. Music Playing Control

2.5 Recording and Playback – main5 (record & play).c

If using a terminal program other than CSTHost, the following settings are required:

Bits per second:	115200
Data bits:	8
Parity:	None
Stop bits:	1
Flow control:	Hardware

This example shows how to build a simple auto answering machine. The example can record and playback voice messages. See the source annotation below.

```
static void SendVoiceData ()
{
    ...
}
```

This function sends a message to CST action. The attached data contains recoded voice for playing. This function is analogous to the function from the previous example.

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

External messages call back function. It receives messages from CST and processes only `eme_PERIPH_DATA`, `eme_CPTD_DATA`, `eme_AUTOTURNOFF_ALL` and `eme_VOICE_DATA` messages.

```
    case eme_PERIPH_DATA:
      ..
    case eme_CPTD_DATA:
      ..
    case eme_AUTOTURNOFF_ALL:
      ..
```

Processing of these messages is the same as in the previous example.

```
    case eme_VOICE_DATA:
      if ( MainState!=ms_RECORD_MESSAGE )
        break;
      while ( Data-- )
      {
        if (pMessageRecChar>=
            pMessageBuffer+MESSAGE_BYTES-1)
        {
          MainState=ms_TURNOFF_VOICE_RXDATA;
          ...
          break;
        }
        if ( IsHighByte )
          *pMessageRecChar=*pData++<<8;
        else
        *pMessageRecChar++|=*pData++;
        IsHighByte=!IsHighByte;
      }
      break;
  }
  return 1;
}

void MyPeriodicThread ()
{
  CSTAction_Process (&Ch0);

  switch ( MainState )
  {
    case ms_GO_OFF_HOOK:
      if (DoStandardOperation(sot_OFF_HOOK,0))
      {
        MainState=ms_TURNON_VOICE_TXDATA;
      }
      break;
```

Voice message processing is performed only in the RECORD state.

Voice data is stored in the buffer until the buffer is full. Transition to the next state occurs upon buffer filling completion.

```
case ms_TURNON_VOICE_TXDATA:
  if (DoStandardOperation(
          sot_TURNON_VOICE_TXDATA,0))
  {
    MusicSoundPos=0;

    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;

    MainState=ms_PLAY_MESSAGE;
    ...
  }
  break;
```

Set pointer to the beginning of the music vector.

Prepare message for voice TX path creation.

Switch to the next state.

```
case ms_PLAY_MUSIC:
  SendVoiceData ();
  break;
```

This is the music playback loop.

```
case ms_TURNOFF_VOICE_TXDATA:
  if (DoStandardOperation(
        sot_TURNOFF_VOICE_DATA,0))
  {
    MainState=ms_TURNON_VOICE_RXDAT
      ...
  }
  break;
```

Turn the voice path off, and switch the state machine to prepare for the RECORD state.

```
case ms_TURNON_VOICE_RXDATA:
  if (DoStandardOperation(
        sot_TURNON_VOICE_RXDATA,0))
  {
    pMessageRecChar=pMessageBuffer;
    IsHighByte=1;
    MainState=ms_RECORD_MESSAGE;
    ...
  }
  break;
```

Prepare buffer for saving voice bitstream.

```
case ms_TURNOFF_VOICE_RXDATA:
  if (DoStandardOperation(
        sot_TURNOFF_VOICE_DATA,0))
    MainState=ms_TURNON_VOICE_TXDATA;
  break;
```

Switch to PLAYBACK preparation mode.

```
    case ms_TURN_OFF_GO_ON_HOOK:
      if (DoStandardOperation(
            sot_TURNOFF_ALL,0))
      {
        MainState=ms_WAIT_FOR_RING;
        ...
      }
      break;
  }
}
```

In case of BUSY signal detected, it turns off all active algorithms by executing the script corresponding to the sot_TURN_OFF_ALL standard operation, and returns to the initial state.

(The script is aTurnOffAll; it is described in CSTAtomic.c).

Busy detection is not shown in Figure 5, in order to simplify the figure. Busy detection is performed on all stages of the flex application execution. The application returns to the initial (IDLE) state in case of BUSY signal detection.

The system is waiting for a RING signal in the initial state. When RING is detected, the system performs OFF HOOK processing and turns the Rx voice path on. The system now is in the RECORD state and recording a new message. Recording is performed until the buffer is full. The voice Rx path will be turned off, and voice Tx path will be turned on. Then, the system switches back to the PLAYBACK state.

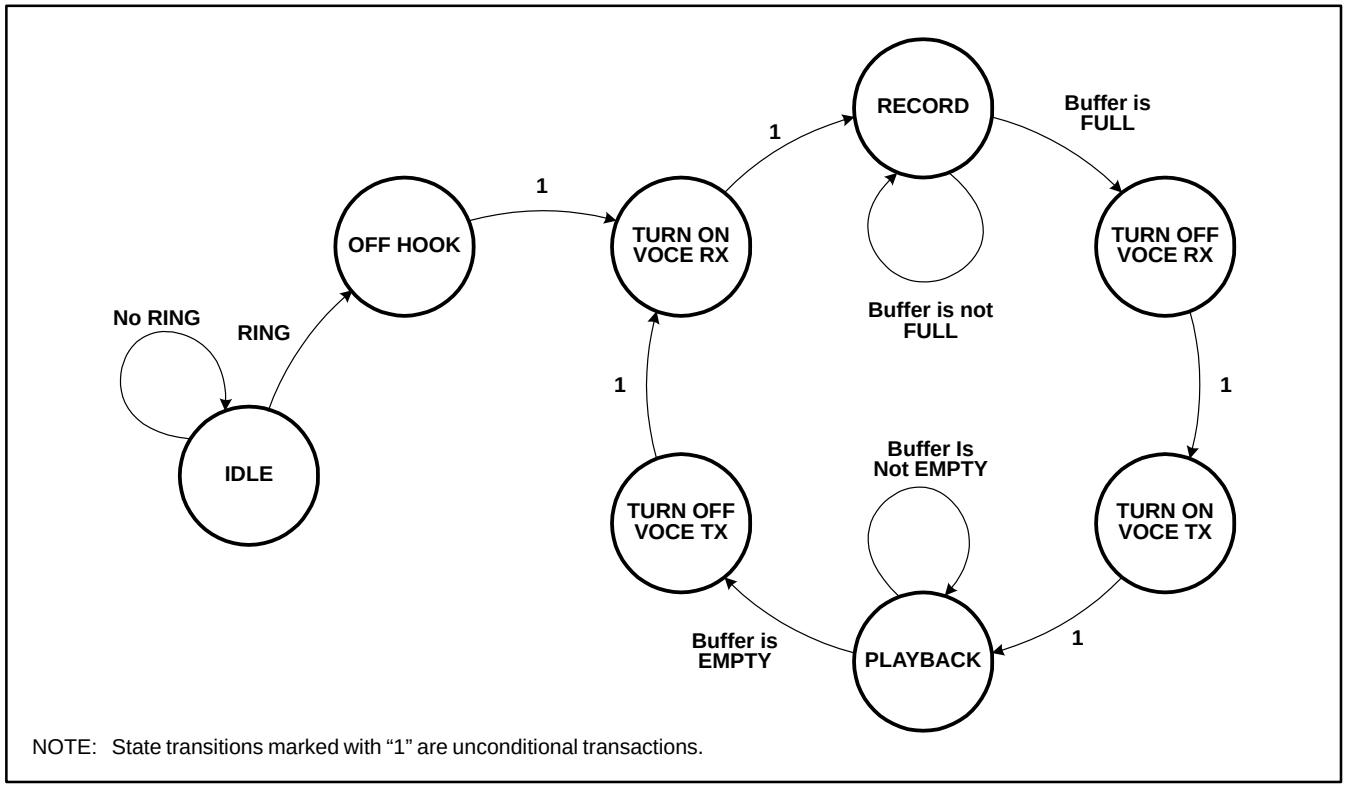


Figure 5. Recording and Playback

2.6 Configuring VAD on the Fly – main5a (+VAD manipulation).c

This flex example extends the previous one by adding the possibility of reconfiguring the value-added distributor (VAD) on the fly (changes are applied on the next iteration of the state machine cycle). Table 2 shows accepted DTMF symbols and their purpose. The source annotation follows this table.

Table 2. Accepted DTMF Symbols for Configuring VAD on the Fly

DTMF Symbol	Purpose
1	Enables VAD, 1 LPC coefficient
2	Enables VAD, 2 LPC coefficients
…	…
9	Enables VAD, 9 LPC coefficients
0	Enables VAD, 10 LPC coefficients
#	Disables VAD

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

    case eme_PERIPH_DATA:
      ..
    case eme_CPTD_DATA:
      ..
    case eme_AUTOTURNOFF_ALL:
      ..
    case eme_VOICE_DATA:
      ..

    case eme_DTMF_DATA:
      switch ( Data )
      {
        case '#':
          SetRegister (srd_VAD,0);
          ...
          break;
        default:
          LPCCount=Data-'0';
          if ((LPCCount>=0) && (LPCCount<=9))
          {
            SetRegister (srd_VAD,1);
            ...
            if ( LPCCount==0 )
              LPCCount=10;
            IVAD_PARAMS.cngOrder=LPCCount;
            ...
        }
    }
    break;
  }
  return 1;
}
```

External messages callback function (see the previous example). It receives messages from CST.

Processing of these messages is the same as in the previous examples.

Received DTMF symbols are used to change VAD parameters. The parameters are set via the S-register. They will take effect when recording a new (next) voice message, i.e., these changes do not take effect immediately.

The flow chart shown in Figure 6 is almost the same as in the previous example.

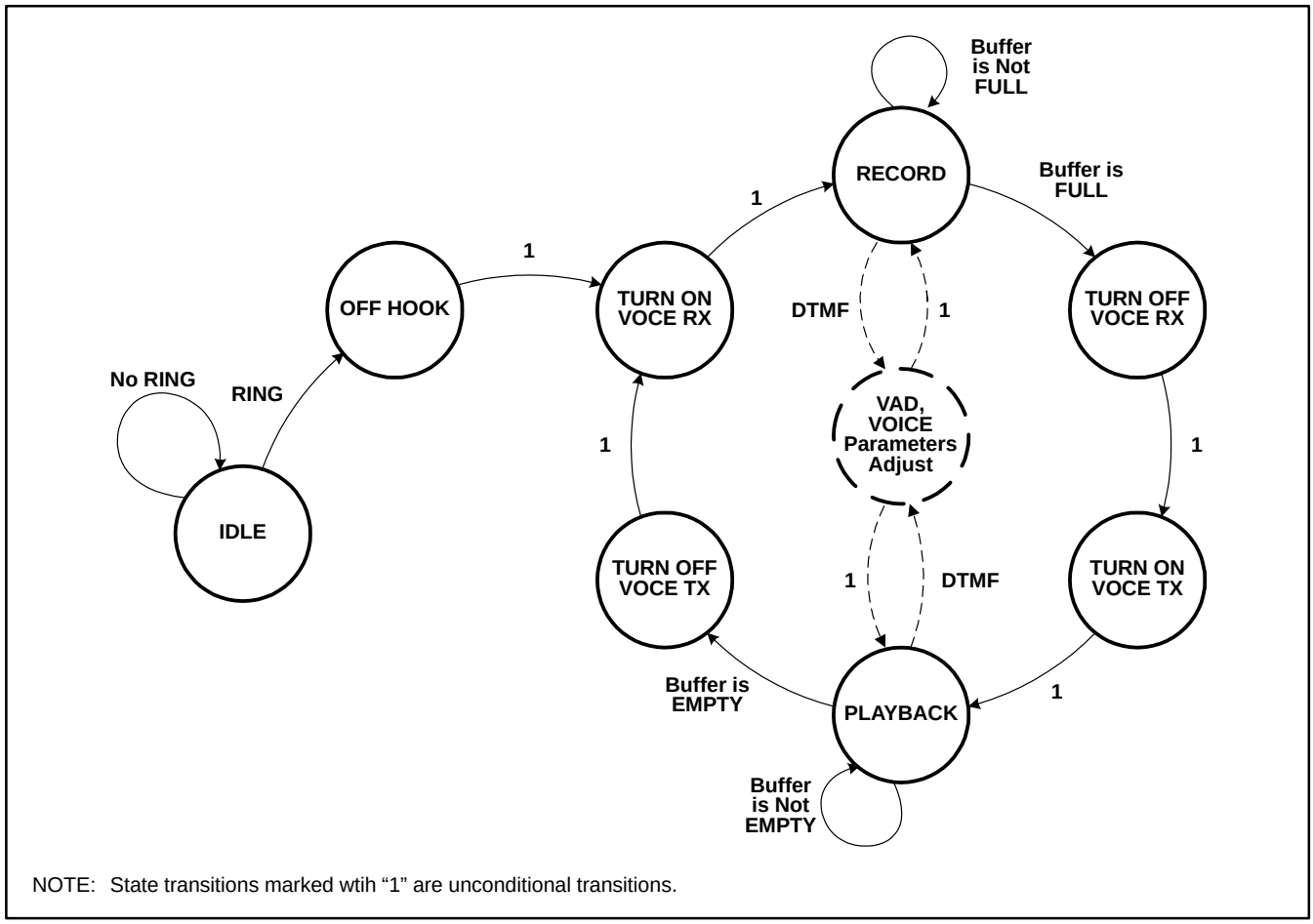


Figure 6. VAD Reconfiguration on the Fly

2.7 DTMF Echo – main6 (DTMF toy).c

This example detects 5 DTMF symbols sent by you and regenerates them back with preserving pause durations between the symbols. The source annotation is shown below.

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

External messages call back function. It receives messages from CST and processes some of them.

```
    case eme_PERIPH_DATA:
      ..
    case eme_CPTD_DATA:
      ..
    case eme_AUTOTURNOFF_ALL:
      ..
```

Processing of these messages is the same as in previous examples.

```
    case eme_DTMF_DATA:
      if ( Ch0.CSTService.pDTMFGenHandle )
        break;
```

Check DTMF generator activity. Receive DTMF symbols only when in the state ms_RECEIVE_DTMF, so you will not get your own symbols during generation.

```
    if (DataAction.Action.CSTServiceMessage.
          DataLength)
    {
      DataAction.Action.CSTServiceMessage.
        aData[DataAction.Action.
          CSTServiceMessage.DataLength++]=
        Max (50,Min (4000,
          DTMFDurationInterval/8/2));
    }
```

Save the time elapsed since previous DTMF detection. Duration is measured in 8-kHz samples.

```
    DataAction.Action.CSTServiceMessage.aData[
      DataAction.Action.CSTServiceMessage.
        DataLength++]=Data;
      ...
      if (DataAction.Action.CSTServiceMessage.
          DataLength==DTMF_SYMBOLS*2-1 )
      {
        DataAction.Action.CSTServiceMessage.
          aData[DataAction.Action.
            CSTServiceMessage.DataLength++]=
              100; //default

        MainState=ms_GENERATE_DTMF;
        ...
      }

      DTMFDurationInterval=0;
      break;
```

Save detected DTMF symbol.

Switch the state machine to the DTMF generation state after you have received 5 symbols.

```
    case eme_TICK:
      DTMFDurationInterval+=Data;
      break;
  }
  return 1;
}

void MyPeriodicThread ()
{
  CSTAction_Process (&Ch0);

  switch ( MainState )
  {
    case ms_GO_OFF_HOOK:
      if (DoStandardOperation(sot_OFF_HOOK,0))
      {
        MainState=ms_RECEIVE_DTMF;
        DataAction.ActionType=
          cat_CSTSERVICE_MESSAGE;
        DataAction.Action.CSTServiceMessage.
          Task=cstst_DTMF;
        DataAction.Action.CSTServiceMessage.
          IsItTxTask=1;
```

Increment sample counter.

Perform OFF HOOK and start the CPTD and DTMF detectors.

```
        DataAction.Action.CSTServiceMessage.
          SubEvent=cse_DATA;
        DataAction.Action.CSTServiceMessage.
          DataLength=0;
        ...
      }
      break;

    case ms_GENERATE_DTMF:
      if ( SendDTMFData () )
      {
        DataAction.Action.CSTServiceMessage.
          DataLength=0;
        MainState=ms_RECEIVE_DTMF;
        ...
      }
      break;

    case ms_TURN_OFF_GO_ON_HOOK:
      if (DoStandardOperation(
            sot_TURNOFF_ALL,0))
        MainState=ms_WAIT_FOR_RING;
      break;
  }
}
```

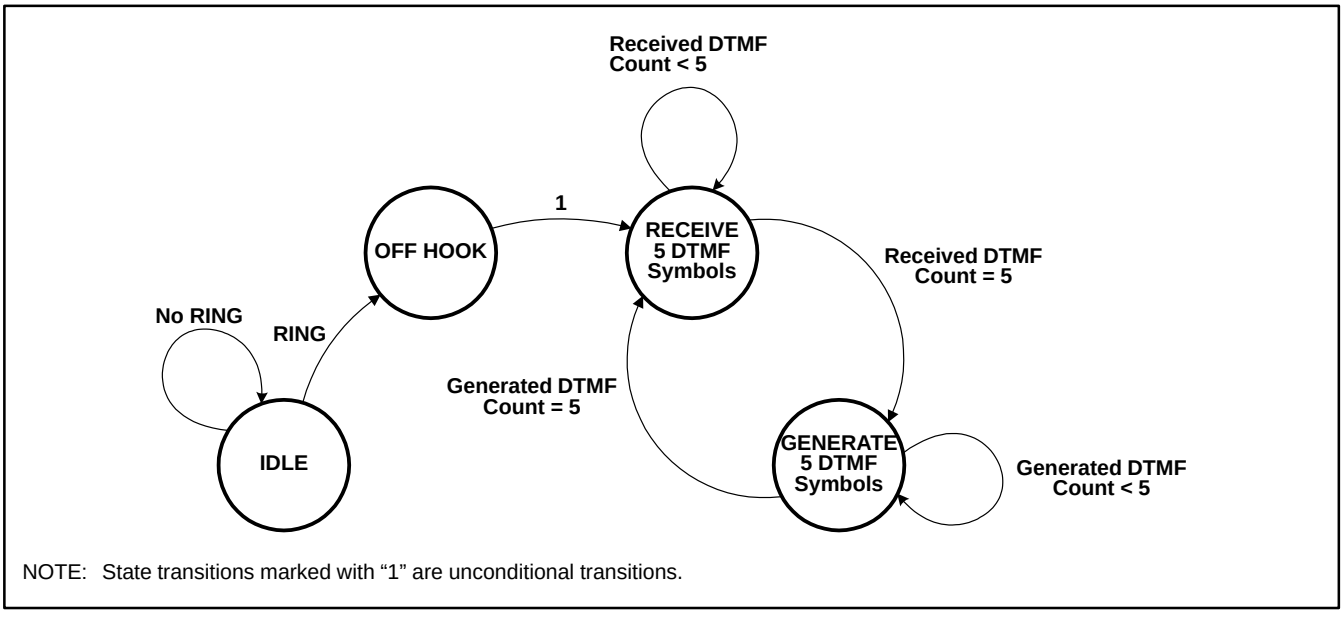
Send DTMF symbols with same time intervals as received.

DTMF generator turns on and off automatically. It depends on presence of data for the DTMF generator.

Go ON HOOK and stop everything.

Result checking for sot_TURNOFF_ALL is not necessary because this action will be accepted and executed anyway.

Busy detection is not shown in Figure 7, in order to simplify the figure. Busy detection is performed on all stages of the flex application execution. The application returns to the initial (IDLE) state in case of BUSY signal detection.



NOTE: State transitions marked with “1” are unconditional transitions.

Figure 7. DTMF Toy

2.8 DTMF/CPTG Toy – main6a (DTMF toy).c

This example application is somewhat similar to the previous one. The idea of this application is to show how to use the CPT generator and how to tune it to custom CPT signals. The application detects a DTMF symbol produced by you, and generates a CPT signal corresponding to the detected DTMF symbol (see Table 3). The CPT signals can be heard on the phone (see Figure 8). The source annotation is shown below this table.

Table 3. Accepted DTMF Symbols for DTMF/CPTG Toy

DTMF Symbol	Purpose
0	Ring signal
1	Dial signal
2	Busy signal
3	Fast busy signal
4	SIT signal
5	ANS signal

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

External messages call back function. It receives messages from CST and processes some of them.

```
    case eme_PERIPH_DATA:
      ..
    case eme_CPTD_DATA:
      ..
    case eme_AUTOTURNOFF_ALL:
      ..
```

Processing of these messages is the same as in previous examples.

```
    case eme_DTMF_DATA:
      if (!Ch0.CSTService.pCPTDGenHandle)
      {
        MainState=ms_TURN_OFF_GO_ON_HOOK;
        UartPutString (&Ch0,”Abnormal ”
        ”termination...\r\n”);
        break;
      }

      UMTG_setSignal
        (Ch0.CSTService.pCPTDGenHandle,
        Data+0x1000);

      UartPutString (&Ch0,”DTMF symbol ”
                     ”detected...\r\n”);
      break;
```

Abort if CPT generator has not been created.

Set signal for generation according to the received DTMF symbol.

```
  }
  return 1;
}

void MyPeriodicThread ()
{
  CSTAction_Process (&Ch0);

  switch ( MainState )
  {

    case ms_GO_OFF_HOOK:
      if (DoStandardOperation (sot_OFF_HOOK,0))
      {
        MainState = ms_GEN_CREATE;

        DataAction.ActionType=
          cat_CSTSERVICE_MESSAGE;
        DataAction.Action.CSTServiceMessage.
          Task=cstst_CPTD;
        DataAction.Action.CSTServiceMessage.
          IsItTxTask=1;
        DataAction.Action.CSTServiceMessage.
          SubEvent=cse_ON;
        DataAction.Action.CSTServiceMessage.
          DataLength=0; // empty

        UartPutString (&Ch0,"Press DTMF "
                       "button\r\n");
      }
      break;
```

Perform OFF HOOK and start CPT generator.

```
    case ms_GEN_CREATE:
      if ( SendGenData () )
      {
        MainState=ms_RECEIVE_DTMF;
      }
      break;
```

Send message with CPT generator signal.

```
    case ms_TURN_OFF_GO_ON_HOOK:
    if (DoStandardOperation (sot_TURNOFF_ALL,
        0))
      MainState=ms_WAIT_FOR_RING;
    break;
  }
}
```

Go ON HOOK and stop everything.

Result checking for sot_TURNOFF_ALL is not necessary because this action will be accepted and executed anyway.

```
void MyUserOperation (tCSTChannel*pChannel,
       int16*pInput, int16*pOutput,
       int AmountOf8KHzSamples)
{
  tCSTService *pCSTService =
    &pChannel->CSTService;

  if(pChannel->CSTService.pCPTDGenHandle)
    UMTG_genSignal(pCSTService->pCPTDGenHandle,
      pOutput, INPUT_OUTPUT_LENGTH);
```

New UserOperation() function. This is used to put samples generated by the CPT generator to the output buffer.

The function overrides the original UserOperation() function.

```
  CSTAction_UserOperation(pChannel, pInput,
    pOutput, AmountOf8KHzSamples);
}
```

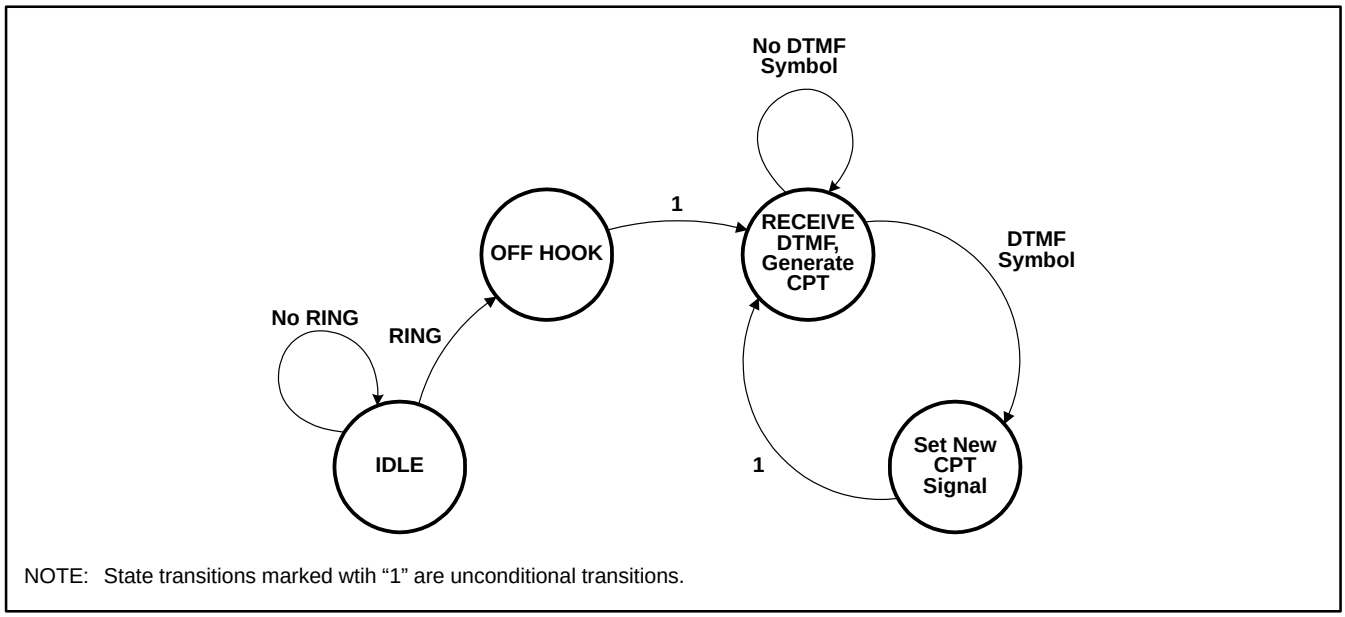
```
void MyInitialization ()
{
  UartPutString (&Ch0,"DTMF Toy\r\n");
  UartPutString (&Ch0,"Please, call to the"
                   " phone\r\n");

  IUMTG_CPTDParams.pSeries[0].
    pParam->pSignals=
      (IUMTG_Signal*)&signals_My;
  IUMTG_CPTDParams.pSeries[0].
    pParam->signalsCount=
      sizeof(signals_My)/sizeof(IUMTG_Signal);

  CSTFxns.pCSTUserOperation = MyUserOperation;
}
```

Assign a new CPT signal table to the CPT generator.

Override the original UserOperation() function.



NOTE: State transitions marked wtih “1” are unconditional transitions.

Figure 8. DTMF/CPTG Toy

2.9 Intensive Modem Tx – main7 (modem tx spam).c

This example begins the series of applications that deal with the build-in modem. This demo sends pseudo-random data after establishing a modem connection (see Figure 9). The source annotation is shown below.

```
char GetRand ()
{
  ...
}
```

Simple random number generator function.

```
void MakeNewString ()
{
  ...
}
```

Fills a string of length STRING_LEN with one of the following: random data, report message or farewell message.

```
static void SendModemData ()
{
  tCSTMessageResult ActionResult;
  char CurChar;
```

This function sends modem data to CST action layer It will be redirected to CST service layer.

```
  while ( DataAction.Action.CSTServiceMessage.
    DataLength<CST_MAXDATALENGTH )
  {
    CurChar=aString[CurStrPos++];

    if ( !aString[CurStrPos] )
      MakeNewString ();
    if ( !CurChar )
      break;

    DataAction.Action.CSTServiceMessage.Data[
      DataAction.Action.CSTServiceMessage.
        DataLength++]=CurChar;

    if ( (ByteCounter++ & 0x3FF)==0 )
      NeedToPrintAmountOfBytes=1;
  }
```

Fill the action message with random data until the message is full.

```
  if ( DataAction.Action.CSTServiceMessage.
         DataLength )
    ActionResult=CSTAction (&Ch0,&DataAction);
```

Try to send the action message to the CST action layer. It will be forwarded to the CST service layer.

```
  if ( (ActionResult==cmr_RESULTOK) ||
       (ActionResult==cmr_EXECUTING) )
    DataAction.Action.CSTServiceMessage.
      DataLength=0;
}
```

Reset the buffer length when data have been accepted. Otherwise, the action message will be pushed again later.

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

    case eme_PERIPH_DATA:
      ..
    case eme_AUTOTURNOFF_ALL:
      ..
```

Processing of these messages is the same as in previous examples.

```
    case eme_MODEM_CONNECT:
      ...
      break;
```

This message informs you about successful modem connection.

```
    case eme_MODEM_DISCONNECT:
      MainState=ms_WAIT_FOR_RING;
      break;
```

This message informs you about modem disconnection.

```
    case eme_TICK:
      if ( MainState==ms_MODEM_TX )
      {
        Timer+=Data;
        if ( Timer>WORK_TIME )
          HasTimeExpired=1;
        if ( Timer>WORK_TIME+1000L )
        {
          ...
          MainState=ms_TURN_OFF_GO_ON_HOOK;
        }
      }
      break;
  }
  return 1;
}

void MyPeriodicThread ()
{
  CSTAction_Process (&Ch0);

  switch ( MainState )
```

Increment timer. Switches the state machine state when time is expired.

```
    case ms_MODEM_ANS:
      if (DoStandardOperation (
            sot_TURNON_MODEM_ANS,0) )
      {
        MainState=ms_MODEM_TX;
        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;
        Timer=
          ByteCounter=
            NeedToPrintAmountOfBytes=
              HasTimeExpired=
                HasSaidBye=0;
        MakeNewString ();
      }
      break;
```

Now you are expecting modem connection. CPTD will be activated.

Most of message fields are set here. The cat_CSTSERVICE_MESSAGE key is used to inject a portion of data to the modem algorithm through the CST service layer. The action message cannot be allocated in stack because CST may not be able to take the whole message at once. In such a case, the message should be kept in memory for awhile, to be sent again.

```
    case ms_MODEM_TX:
      SendModemData ();

      if (HasSaidBye &&
         !DataAction.Action.CSTServiceMessage.
           DataLength)
        MainState=ms_TURN_OFF_GO_ON_HOOK;
      break;
```

Try to push data into the modem.

Turn off all when requested.

```
    case ms_TURN_OFF_GO_ON_HOOK:
      if ( DoStandardOperation (
             sot_SOFT_TURNOFF_ALL,0) )
      {
        MainState=ms_WAIT_FOR_RING;
        ...
      }
      break;
  }
}
```

Perform standard operation to turn off all and go on hook. Set the state machine to the initial state.

```
#define USE_ASYMMETRIC_V42BIS 0

#if USE_ASYMMETRIC_V42BIS
#include "..\MODINT\DMController.h"
#endif //USE_ASYMMETRIC_V42BIS
```

You do not have to use data compression in both directions; therefore, the spared memory can be spent for bigger V.42bis dictionary size in one direction. For this particular application, only Tx throughput matters, and thus, Rx compression can be disabled.

NOTE: Not all modems work correctly with asymmetric V.42bis.

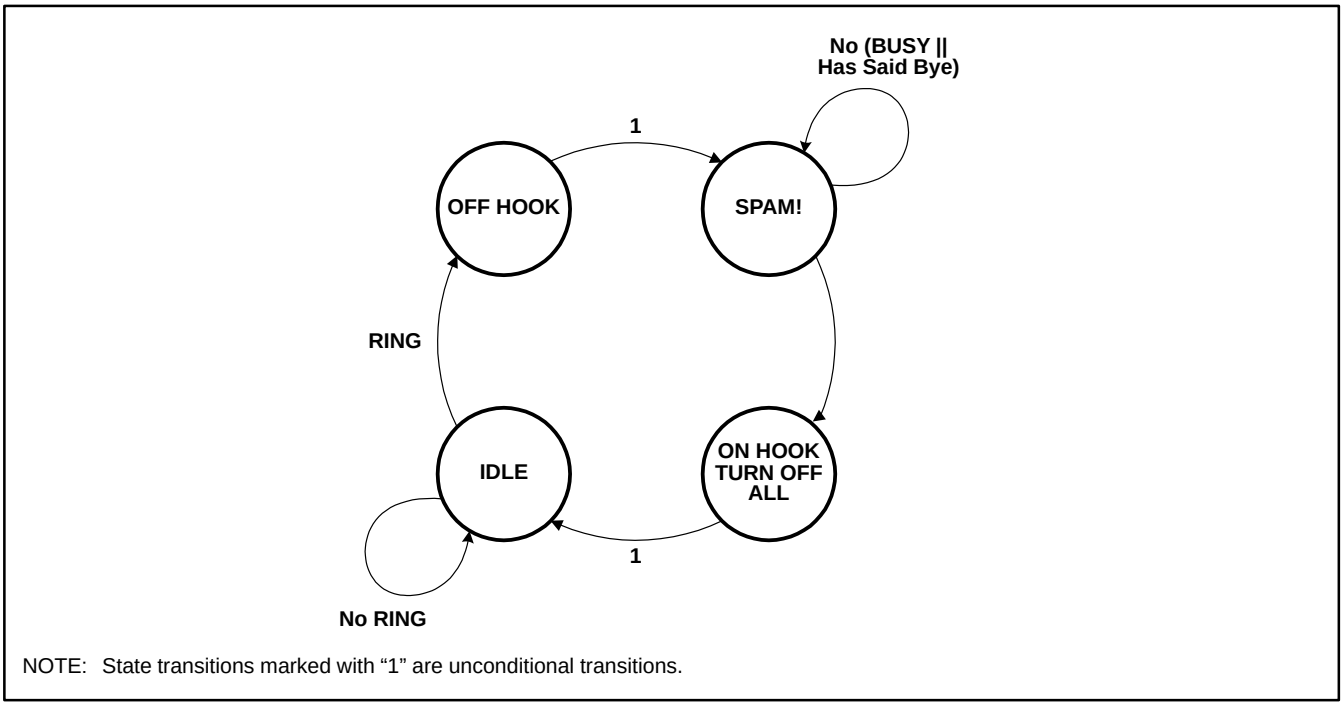
```
void MyInitialization ()
{
  ...
#if USE_ASYMMETRIC_V42BIS
  Ch0.CSTCommanderSettings.IsV42bis=1;

  V42Params.V42bisParams.dictionarySize=1024;
#endif //USE_ASYMMETRIC_V42BIS
}
```

To disable RX compression, the S-register, srd_V42BIS, should be set to 0x01 instead of 0x03.
Lines Ch0.CSTCommanderSettings.IsV42bis=1
and SetRegister (srd_V42BIS,1) have the same result.

To double V.42bis dictionary size, the initialization parameter `dictionarySize` should be set to 1024 instead of 512.

See S-register assignment in the file `CSTSReg.c`.



NOTE: State transitions marked with "1" are unconditional transitions.

Figure 9. Modem Tx Spam

2.10 Modem Terminal – main8 (modem terminal).c

This example establishes modem connection with a remote modem, whose phone number is received from the serial port. The dial symbols are shown in Table 4, and the source annotation is shown below this table. After connection, it works like a simple terminal processor. See Figure 10.

Table 4. Dial Symbols

Dial Symbols	Description
0–9	Phone number digits
ABCD	Digits that can be dialed only in the tone mode
P	Pulse-mode dialing modifier
T	Tone-mode dialing modifier
,	Long pause
/	Short pause
W	Dial-tone waiting
R	Response/answer mode
“Enter” (0x0D)	“Enter” (end of the phone number)

NOTE: The UART is not a part of the CST action interface. Therefore, in the flex mode, the serial port can be freely used.

```
static void SendModemData (char UARTChar)
{
  tCSTMessageResult ActionResult;

  if ( (DataAction.Action.CSTServiceMessage.
          DataLength<
        CST_MAXDATALENGTH) )
    DataAction.Action.CSTServiceMessage.Data[
      DataAction.Action.CSTServiceMessage.
        DataLength++]=UARTChar;

  ActionResult=CSTAction (&Ch0,&DataAction);

  if ( (ActionResult==cmr_RESULTOK) ||
       (ActionResult==cmr_EXECUTING) )
    DataAction.Action.CSTServiceMessage.
      DataLength=0;
}
```

This function sends modem data to the CST action layer (it will be redirected to the CST service layer).

Put new data byte into the action message until the message is full.

Try to push the data message into the modem.

Verify whether the message has been sent successfully or not.

```
static bool MyCallback (tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,int
  Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {
    case eme_MODEM_CONNECT:
      ...
      break;
```

External messages callback function. It receives messages from CST and processes some of them.

This message informs you about successful modem connection.

```
    case eme_AUTOTURNOFF_ALL:
      ..
    case eme_MODEM_DISCONNECT:
      UartReset (Ch0.UartRxChanHandle, NULL);
      MainState=ms_GET_NUMBER;
      aPhoneNumber[0]=0;
      break;
```

Processing of these messages is the same as in previous examples.
Discard the UART RX buffer, and return to the initial state.

```
    case eme_MODEM_DATA:
      UartWrite (Ch0.UartTxChanHandle,
      (unsigned char*)pData, Data);
      break;

  }
  return 1;
}

void MyPeriodicThread ()
{
  int UARTChars,ITemp;
  char UARTChar;

  CSTAction_Process (&Ch0);
  switch ( MainState )
    case ms_GET_NUMBER:
      UARTChars=
        UartReadAvail (Ch0.UartRxChanHandle);

      while ( UARTChars-- )
      {
        //Read a char
        UartRead (Ch0.UartRxChanHandle,
          (unsigned char*)&UARTChar, 1);
        {
          static const char *pDialSymbols=
            "0123456789ABCD"
            "PT,/W"
            "R"
            "\xD";

          if (!strchr(pDialSymbols,UARTChar))
            continue;

          UartWrite (Ch0.UartTxChanHandle,
            (unsigned char*)&UARTChar, 1);

          if ( UARTChar==0x0D )
          {
            MainState=ms_MODEM_CALL;
            ...
            break;
          }
```

Send data from modem to UART.

CAUTION:
The UART TX buffer can overflow. See the next example to learn how to do this more accurately.

Get number of unread characters in UART RX buffer and read them.

Validate the received character. Check whether it belongs to the set. Discard the char if it does not belongs to the set; otherwise, send it back to the host.

NOTE: Validation is not necessary because it is already done in the CST Commander (see aDialSymbols in CSTCommander.c).

Dial the number if the “Carriage Return” (0x0D) symbol has been received from the UART.

```
          ITemp=strlen (aPhoneNumber);
          if ( ITemp<
            CST_STANDARD_OPERATION_DATA_LEN )
          {
            aPhoneNumber[ITemp++]=UARTChar;
            aPhoneNumber[ITemp]=0;
          }
        }
      }
      break;
    case ms_MODEM_CALL:
      if ( DoStandardOperation (
             sot_TURNON_MODEM_CALL_X,
            (uint8*)aPhoneNumber) )
    {
      MainState=ms_TERMINAL_CONNECT;

      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;
    }
    break;

    case ms_TERMINAL_CONNECT:
      UARTChars=
        UartReadAvail (Ch0.UartRxChanHandle);

      while ( UARTChars-- )
      {
        UartRead (Ch0.UartRxChanHandle,
          (unsigned char*)&UARTChar, 1);

        SendModemData (UARTChar);
      }
      break;
  }
}
```

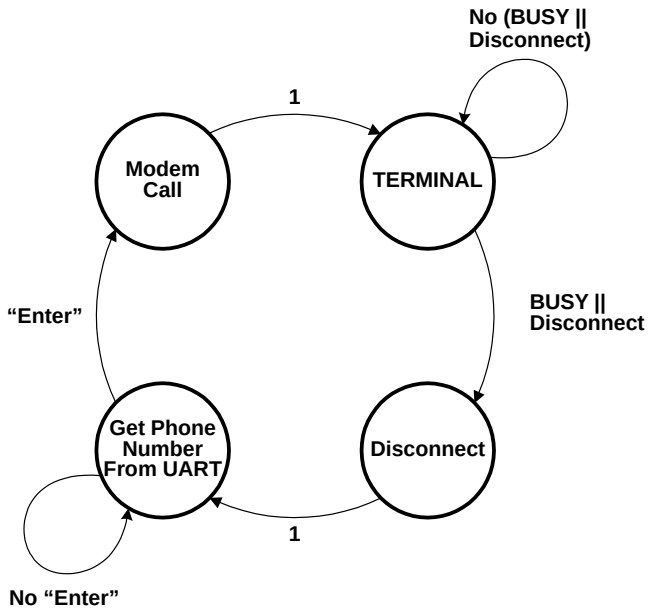
Add the character to the phone-number string.

Run the script, performing a modem call to the specified number. The CPTD and DTMF detectors will be activated.

Perform modem data action initialization.

Warning: It is recommended to limit quantity of characters to be read and send per time frame (for example, see the function `CSTUserOperation()` in the file `ATParser.c`) or use a low priority thread. Otherwise, we may get off real-time.

Get the characters from the UART Rx buffer and send them to the modem.



NOTE: State transitions marked wtih "1" are unconditional transitions.

Figure 10. Modem Terminal

2.11 Modem Throughput – main9 (modem throughput).c

This example can be used to measure the modem throughput. This application receives the incoming modem data and echoes it back. Current throughput is measured and reported on the regular basis, with a period of one second. The source annotation is shown below, and a flow chart of this example is shown in Figure 11.

NOTE: To prevent data loss due to possible difference between Rx and Tx throughputs, a separate modem callback function is used. Initially, the modem callback function is set by the CST action layer to common CST action callback, which forces you to accept all data even if you are not able to process it.

```
static void SendModemData (uint8 *pData,int
  Count)
{
  tCSTMessageResult ActionResult;
  int Result=Count;
```

This function sends modem data to the CST action layer. It will be redirected to the CST service layer).

```
  while ((DataAction.Action.CSTServiceMessage.
            DataLength<CST_MAXDATALENGTH) &&
           Count)
  {
    DataAction.Action.CSTServiceMessage.aData[
    DataAction.Action.CSTServiceMessage.
      DataLength++]=*pData++;
    Count--;
    ByteCounter++;
  }
```

Put new byte into the action message until the message is full.

```
  ActionResult=CSTAction (&Ch0,&DataAction);
```

Trying to push the data message into the modem.

```
  if ( (ActionResult==cmr_RESULTOK) ||
       (ActionResult==cmr_EXECUTING) )
    DataAction.Action.CSTServiceMessage.
      DataLength=0;
```

Verify whether the message has been sent successfully or not.

```
  return Result–Count;
}
```

Return the number of bytes put to the message.

```
static bool MyCallback (tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,int
  Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {
    case eme_PERIPH_DATA:
      if ( Data==cpe_RING )
      {
        MainState=ms_MODEM_ANS;
        ...
      }
      break;
```

External messages callback function. It receives messages from the CST and processes some of them.

RING detected, start answering.

```
    case eme_MODEM_CONNECT:
      ...
      break;
```

This message informs you about successful modem connection.

```
    case eme_AUTOTURNOFF_ALL:
      ..
```

CST automatically turns off all active algorithms and hangs up upon busy detection.

To disable this behavior, return 0 instead of 1.

```
    case eme_MODEM_DISCONNECT:
      UartReset (Ch0.UartRxChanHandle, NULL);
      MainState=ms_GET_NUMBER;
      aPhoneNumber[0]=0;
      break;
```

Discard UART Rx buffer data and return to the initial state.

```
    case eme_TICK:
      if ( MainState!=ms_MODEM_ECHO_LOOP )
        break;

      Timer+=Data;
      if ( Timer>8000 )
      {
        char aIToA[20];

        UartPutString (&Ch0,
          ”Current throughput is ”);
        UartPutString(&Ch0,
          IToA(ByteCounter,aIToA));
        UartPutString (&Ch0,” byte/sec\r\n”);

        Timer–=8000;
        ByteCounter=0;
      }
      break;
```

TICK counting is performed only in ms_MODEM_ECHO_LOOP state.

Print current throughput (byte per sec) every second.

```
  }
  return 1;
}


void MyPeriodicThread ()
{
  CSTAction_Process (&Ch0);

  switch ( MainState )
    case ms_MODEM_ANS:
      if ( DoStandardOperation(
             sot_TURNON_MODEM_ANS,0) )
      {
        MainState=ms_MODEM_ECHO_LOOP;

        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;
        Timer=
          ByteCounter=0;
      }
      break;
```

Now you are expecting modem connection. The CPT detector will be activated.

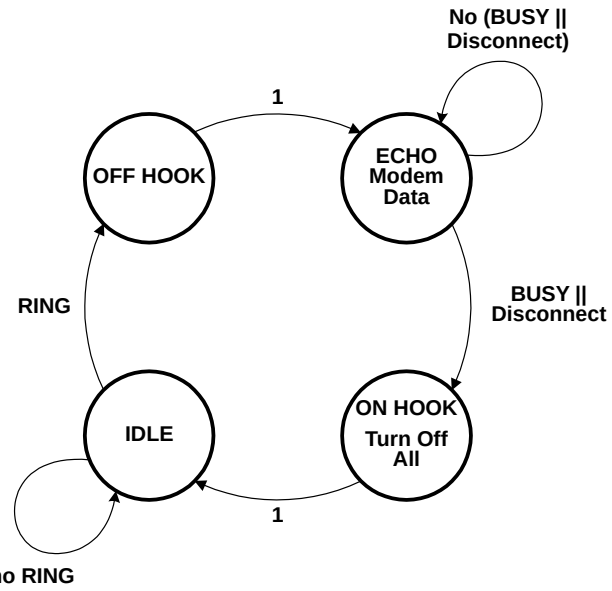
Modem data action initialization.

```
    case ms_MODEM_ECHO_LOOP:
      SendModemData (0,0);
      break;
  }
}
```

No data will be put into message, just attempt to resend any unsent messages to the modem.

```
void MyInitialization ()
{
  DMController_setTransferDataFunc(
    MyGetDataFromModem);
  ...
}
```

Register custom modem callback function instead of the default CST action's routine.



NOTE: State transitions marked with “1” are unconditional transitions.

Figure 11. Modem Throughput

2.12 CID Data Receiving – main10 (Caller ID).c

This example illustrates how to obtain and print out Caller-ID information. The program waits for an incoming ring, and then displays the Caller-ID information. The source annotation is shown below, and the flow chart can be seen in Figure 12.

```
static bool MyCallback (tCSTChannel* pChannel,
  tCSTExternalMsgEvent CSTExternalMsgEvent,int
  Data,int16 *pData)
{
  switch ( CSTExternalMsgEvent )
  {
    case eme_PERIPH_DATA:
      if ( Data==cpe_RING )
        UartPutString (&Ch0,
          ”Ring detected\r\n”);
      break;

    case eme_CID_DATA:
      if ( Data==CID_NOERROR )
      {
        UartPutString (&Ch0,
          ”Caller ID information”
          “ completed\r\n”);
```

External messages call back function. It receives messages from CST and processes some of them.

Just inform about RING detection.

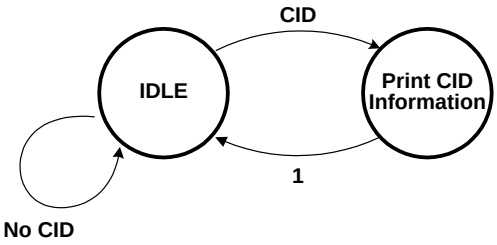
CID_NOERROR result means that CID processing has successfully finished.

```
        if (CIDParseGetString(
              Ch0.CSTService.
                pCIDPresentationMessage,
              aLabelStr, aContextStr,
              CID_VALIDCALLINGLINEIDENTITY))
        {
          UartPutString (&Ch0,”The number: ”);
          UartPutString (&Ch0,aContextStr);
          UartPutString (&Ch0,”\r\n”);
        }
        else
          UartPutString (&Ch0,
            ”The number is not”
            “ recognized\r\n”);
      }
      break;
  }
  return 1;
}
```

Decoded Caller-ID data is not attached to the eme_CID_DATA event message. You may select the type and form of CLI data you need and get it directly from the Caller-ID Parser. Here you are interested in the calling line identity (identifies the origin of the call) in decoded ASCII format.

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

My periodic thread performs only CST action processing.



NOTE: State transitions marked wtih “1” are unconditional transitions.

Figure 12. CID Data Receiving

3 References

1. *Client Side Telephony (CST) Chip Software User’s Guide* (SPRU040).
2. CST Hands-on Lab (download from http://www.spiritDSP.com).

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265