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

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

Literature Number: SPRU639
March 2003

TEXAS INSTRUMENTS

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

Preface

Read This First

About This Manual

The following abbreviations are used in this document:

CPTG	Call Progress Tone (Generation)
CAS	Customer Alerting Signal
CMS	Composite Multitone Signal
DT-AS	Dual-Tone Alerting Signal
DTMF	Dual Tone Multifrequency (signaling)
MF	Multifrequency (signaling)
UMTG	Universal Multifrequency Tone Generator
XDAIS	TMS320 DSP Algorithm Standard

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)

Related Documentation

ITU-T Recommendation E.180/Q.35. Tones in national signaling systems – Operation, numbering, routing and mobile services, 1998.

ITU-T Recommendation E.180, Supplement 2. Various tones used in national networks – Telephone network and ISDN. Operation, numbering, routing and mobile service, 1994.

ITU-T Recommendation Q.23. Technical features of push-button telephone sets – International automatic and semi-automatic working, 1993.

ITU-T Recommendation Q.24. Multifrequency push-button signal reception – International automatic and semi-automatic working, 1993.

ITU-T Recommendation Q.320, Signal code for register signalling – Specifications of signalling system R1, 1993.

ITU-T Recommendation Q.322, Multifrequency signal sender – Specifications of signalling system r1, 1993.

ITU-T Recommendation Q.323, Multifrequency signal receiving equipment – Specifications of signalling system R1, 1993.

ITU-T Recommendation Q.441, Signalling code – Specifications of signalling system R2, 1993.

ITU-T Recommendation V.8. Procedures for starting sessions of data transmission over the public switched telephone network – General, 1998.

ITU-T Recommendation V.25. Automatic answering equipment and general procedures for automatic calling equipment on the general switched telephone network including procedures for disabling of echo control devices for both manually and automatically established calls – Interfaces and voiceband modems, 1996.

Public Switched Telephone Network (PSTN); Protocol over the local loop for display and related services; Terminal Equipment requirements; Part 1: Offline data transmission, ETS 300 778-1, September 1997, DE/ATA-005062-1

Public Switched Telephone Network (PSTN); Protocol over the local loop for display and related services; Terminal Equipment requirements; Part 2: Online data transmission, ETS 300 778-2, September 1997, DE/ATA-005062-2

Calling Line Identification Service, British Telecommunication plc, SIN227, Issue 03.

CCITT Recommendation V.23 (1988): “600/1 200-baud modem standardized for use in the general switched telephone network”.

EIA/TIA-464-A. Private Branch Exchange (PBX) Switching Equipment for Voiceband Application. ANSI/EIA/TIA, February, 1989.

Trademarks

TMS320™ is a trademark of Texas Instruments.

SPIRIT CORP™ is a tradmark of Spirit Corp.

All other trademarks are the property of their respective owners.

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 Universal Multifrequency Tone Generator (UMTG) Algorithms

This chapter is a brief explanation of the Universal Multifrequency Tone Generator (UMTG) and its use with the TMS320C5400 platform.

For the benefit of users who are not familiar with the TMS320 DSP Algorithm Standard (XDAIS), brief descriptions of typical XDAIS terms are provided.

Topic **Page**

1.1 Introduction

This document describes the implementation of Universal Multifrequency Tone Generator (UMTG) developed by SPIRIT Corp. for the TMS320C54xx platform and intended for integration into embedded devices for generating various telephone service tones including:

- standard CPTD tones ():
  - Busy
  - Dial
  - Ringback
  - Reorder
- Extended set of CPTD tones for a majority of countries:
  - Recall dial tone
  - Special ringback tone
  - Intercept tone
  - Call waiting tone
  - Busy verification tone
  - Executive override tone
  - Confirmation tone
- DTMF signaling
- MF-R1, MF-R2 signalling
- Caller ID CAS tone for various standards
- Modem specific tones:
  - Bell 103 answer tone
  - V.23 forward/backward mark bit
  - CED
  - CNG
  - ANS
  - ANSam, etc.

Also, UMTG can be used as a simple tone generator for custom applications since it provides low MIPS consumption (approx. 0.1 MIPS).

The generator can be configured easily to the most country specific CPTD standards.

The SPIRIT UMTG software is a fully TMS320 DSP Algorithm Standard (XDAIS) compatible, reentrant code. The UMTG interface complies with the TMS320 DSP Algorithm Standard and can be used in multitasking environments.

The TMS320 DSP Algorithm Standard (XDAIS) provides the user with object interface simulating object-oriented principles and asserts a set of programming rules intended to facilitate integration of objects into a framework.

The following documents provide further information regarding the TMS320 DSP Algorithm Standard (XDAIS):

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)

However, if the user prefers to have non-eXpressDSP-compliant interface, for example, when a framework is not eXpressDSP-oriented (it usually means that dynamic memory management is not supported), the XDAIS interface can be omitted, as it is merely a wrapper for the original interface.

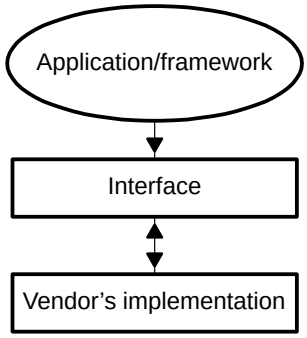
1.2 XDAIS Basics

This section instructs the user on how to develop applications/frameworks using the algorithms developed by vendors. It explains how to call modules through a fully eXpress DSP-compliant interface.

Figure 1–1 illustrates the three main layers required in an XDAIS system:

- Application/Framework layer
- Interface layer
- Vendor implementation. Refer to appendix A for a detailed illustration of the interface layer.

Figure 1–1. XDAIS System Layers



1.2.1 Application/Framework

Users should develop an application in accordance with their own design specifications. However, instance creation, deletion and memory management requires using a framework. It is recommended that the customer use the XDAIS framework provided by SPIRIT Corp. in ROM.

The framework in its most basic form is defined as a combination of a memory management service, input/output device drivers, and a scheduler. For a framework to support/handle XDAIS algorithms, it must provide the framework functions that XDAIS algorithm interfaces expect to be present. XDAIS framework functions, also known as the ALG Interface, are prefixed with "ALG_". Below is a list of framework functions that are required:

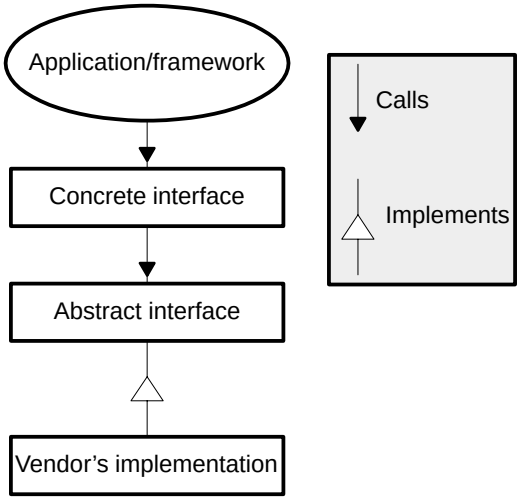
- `ALG_create` – for memory allocation/algorithm instance creation
- `ALG_delete` – for memory de-allocation/algorithm instance deletion
- `ALG_activate` – for algorithm instance activation

- ALG_deactivate – for algorithm instance de-activation
- ALG_init – for algorithm instance initialization
- ALG_exit – for algorithm instance exit operations
- ALG_control – for algorithm instance control operations

1.2.2 Interface

Figure 1–2 is a block diagram of the different XDAIS layers and how they interact with each other.

Figure 1–2. XDAIS Layers Interaction Diagram



1.2.2.1 Concrete Interface

A concrete interface is an interface between the algorithm module and the application/framework. This interface provides a generic (non-vendor specific) interface to the application. For example, the framework can call the function MODULE_apply() instead of MODULE_VENDOR_apply(). The following files make up this interface:

- Header file MODULE.h – Contains any required definitions/global variables for the interface.
- Source File MODULE.c – Contains the source code for the interface functions.

1.2.2.2 Abstract Interface

This interface, also known as the IALG Interface, defines the algorithm implementation. This interface is defined by the algorithm vendor but must comply with the XDAIS rules and guidelines. The following files make up this interface:

- Header file `iMODULE.h` – Contains table of implemented functions, also known as the IALG function table, and definition of the parameter structures and module objects.
- Source File `iMODULE.c` – Contains the default parameter structure for the algorithm.

1.2.2.3 Vendor Implementation

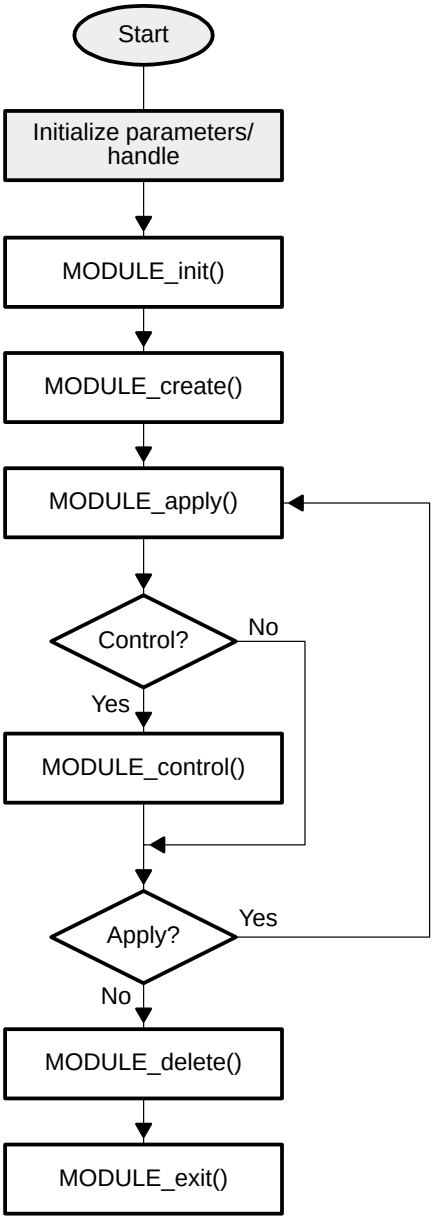
Vendor implementation refers to the set of functions implemented by the algorithm vendor to match the interface. These include the core processing functions required by the algorithm and some control-type functions required. A table is built with pointers to all of these functions, and this table is known as the function table. The function table allows the framework to invoke any of the algorithm functions through a single handle. The algorithm instance object definition is also done here. This instance object is a structure containing the function table (table of implemented functions) and pointers to instance buffers required by the algorithm.

1.2.3 Application Development

Figure 1–3 illustrates the steps used to develop an application. This flowchart illustrates the creation, use, and deletion of an algorithm. The handle to the instance object (and function table) is obtained through creation of an instance of the algorithm. It is a pointer to the instance object. Per XDAIS guidelines, software API allows direct access to the instance data buffers, but algorithms provided by SPIRIT prohibit access.

Detailed flow charts for each particular algorithm is provided by the vendor.

Figure 1–3. Module Instance Lifetime



The steps below describe the steps illustrated in Figure 1–3.

Step 1: Perform all non–XDAIS initializations and definitions. This may include creation of input and output data buffers by the framework, as well as device driver initialization.

Step 2: Define and initialize required parameters, status structures, and handle declarations.

Step 3: Invoke the `MODULE_init()` function to initialize the algorithm module. This function returns nothing. For most algorithms, this function does nothing.

Step 4: Invoke the `MODULE_create()` function, with the vendor's implementation ID for the algorithm, to create an instance of the algorithm. The `MODULE_create()` function returns a handle to the created instance. You may create as many instances as the framework can support.

Step 5: Invoke the `MODULE_apply()` function to process some data when the framework signals that processing is required. Using this function is not obligatory and vendor can supply the user with his own set of functions to obtain necessary processing.

Step 6: If required, the `MODULE_control()` function may be invoked to read or modify the algorithm status information. This function also is optional. Vendor can provide other methods for status reporting and control.

Step 7: When all processing is done, the `MODULE_delete()` function is invoked to delete the instance from the framework. All instance memory is freed up for the framework here.

Step 8: Invoke the `MODULE_exit()` function to remove the module from the framework. For most algorithms, this function does nothing.

The integration flow of specific algorithms can be quite different from the sequence described above due to several reasons:

- Specific algorithms can work with data frames of various lengths and formats. Applications can require more robust and effective methods for error handling and reporting.

- Instead of using the `MODULE_apply()` function, SPIRIT Corp. algorithms use extended interface for data processing, thereby encapsulating data buffering within XDAIS object. This provides the user with a more reliable method of data exchange.

Chapter 2

Universal Multifrequency Tone Generator (UMTG) Integration

This chapter provides descriptions, diagrams, and examples explaining the integration of the Universal Multifrequency Tone Generator (UMTG) with frameworks.

Topic **Page**

2.1 Overview

Universal Multifrequency Tone Generator (UMTG) is designed to generate the set of Composite Multitone Signals (CMS or composite signals later).

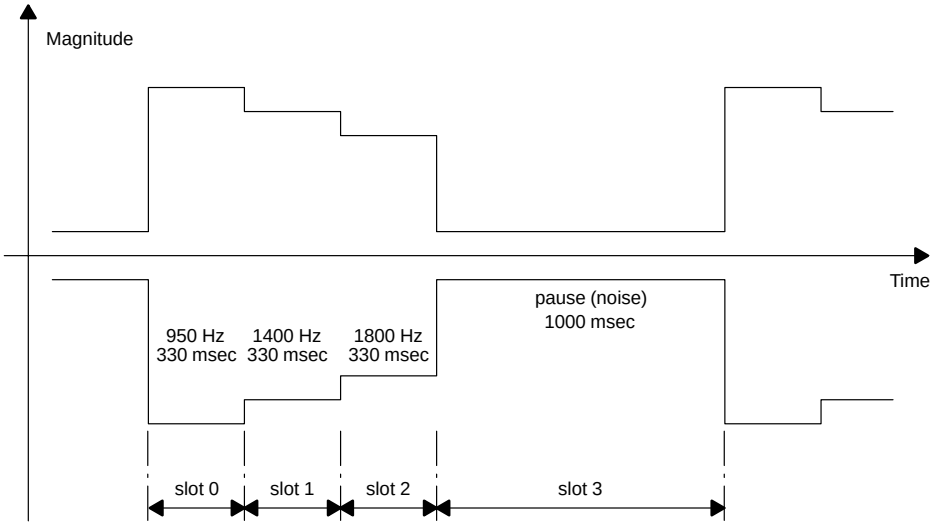
A composite signal represents a sequence of partial multitone signals that can be either successive or divided by pauses.

Each partial multitone signal can be a weighted sum of spectral components in a limited bandwidth. The sequence can be either recurrent or standalone.

This definition covers the majority of alert signals used in the telephone services.

Figure 2–1 shows a typical composite signal.

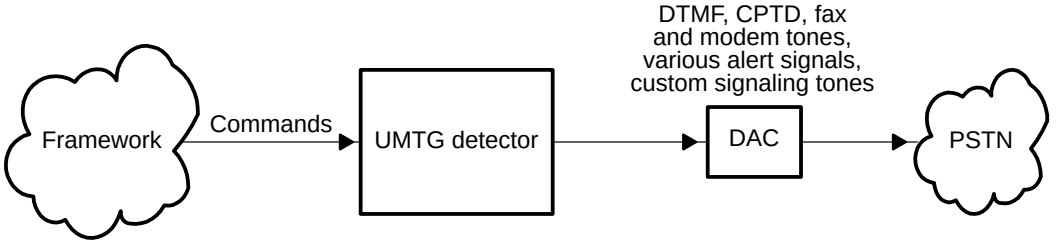
Figure 2–1. Special Information Tone Used in Most of the Countries



A large selection of options allows the user to control the generation on all layers mentioned above.

Figure 2–2 illustrates a typical UMTG integration diagram.

Figure 2–2. UMTG Integration Diagram



2.2 Integration Flow

In order to integrate the UMTG generator into a framework, the user should (see Figure 2–3):

Step 1: Create a handler that will accept messages from a number of UMTG instances

Step 2: Create a `UMTG_Params` structure and initialize it with required values.

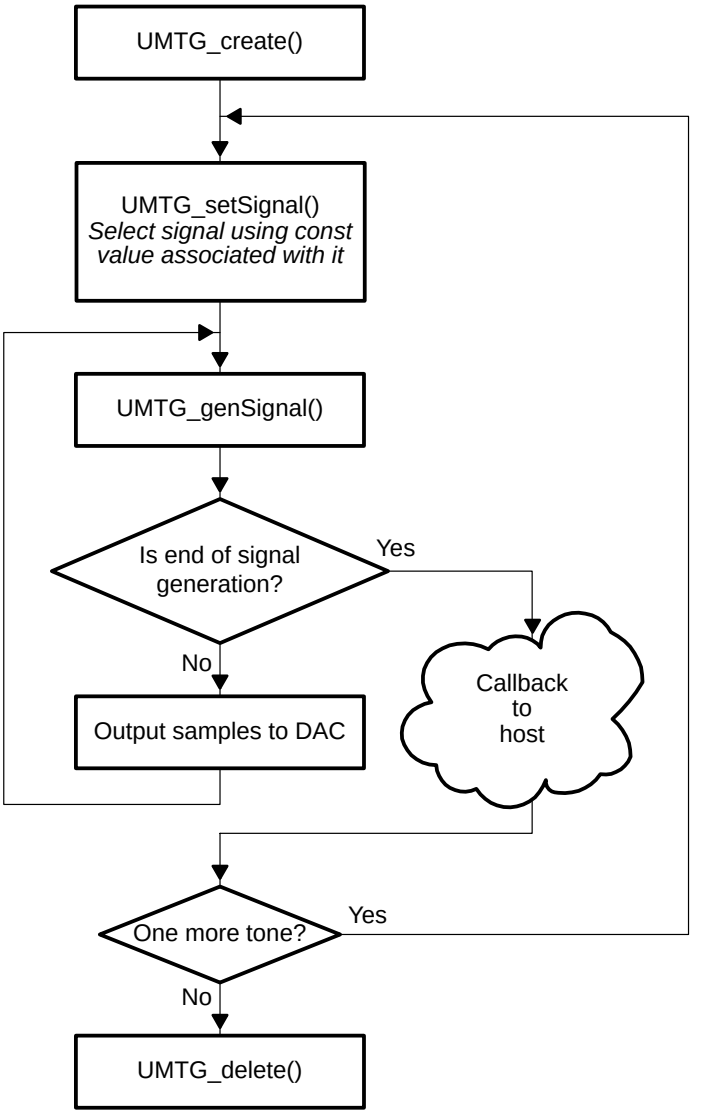
Step 3: Call `UMTG_create()` to create an instance of generator. There are no restrictions on the maximum number of generator instances created.

Step 4: Call `UMTG_setSignal()` to select signal for generation.

Step 5: Call `UMTG_genSignal()` until signal generation will not be finished.

Step 6: Delete the generator by using `UMTG_delete()`.

Figure 2–3. Typical Generator Integration Flow



2.3 Signal Generation

2.3.1 General

The UMTG can generate only one CMS at a time, however, only one common frequency list is defined and used for all signals in current signal series.

2.3.2 Frequency Selection

Each partial multitone signal is supplied with the options that control frequency selection for generation. No more than two frequencies can be selected for generation at one time. Table 2–1 summarizes available frequency selection options.

Table 2–1. Frequency Selection Options Summary

Name	Description
`IUMTG_FIGNORE`	Given frequency component is ignored.
`IUMTG_FGEN`	Given frequency component will be generated with amplitude `magnitude`.
`IUMTG_FGEN2`	Given frequency component will be generated with amplitude `magnitude2`.
`IUMTG_FMOD`	Given frequency component will be used as modulation frequency.

2.3.3 Signal Generation Options

Time slot flags contain fields that provide signal generation control (see Table 2–2).

Table 2–2. Signal Generation Options Summary

Name	Description
`IUMTG_GPHASEREVERSAL`	Forces UMTG to generate phase reversal signal.
`IUMTG_GAMPMOD`	Forces UMTG to generate amplitude-modulated signal. Modulation coefficient must be in `magnitude2` (`0x8000*M`, where `M<1`).

Generation always starts from the first time slot.

2.3.4 Flow Control Options

Additionally, time slot flags contain fields that provide flow control (see Table 2–3). They are used together with signal generation options.

Table 2–3. Flow Control Options Summary

Name	Description
IUMTG_FGOFIRST	Forces UMTG to start from the first time slot when this time slot will be finished.
IUMTG_FDONOTCLEARPHASE	UMTG would not reset the phase after finishing this time slot. This option can be used only with continuous signals.
IUMTG_FCALLBACK	UMTG would execute callback function after finishing this time slot.

2.4 Parameter Definition

Table 2–4. General Parameters for Generated Signal Series

```
typedef struct
{
```

Name	Type	Typical Value	Limits	Description
magnitude	XDAS_Int16	N/A	>0	Partial signal amplitude. Used by generator when option IUMTG_FGEN is set, dBm0.
magnitude2	XDAS_Int16	N/A	>0	Partial signal amplitude. Used by generator when option IUMTG_FGEN2 or IUMTG_FGEN are set, dBm0.
frqCount	XDAS_Int16	N/A	>0	Number of frequencies used in signal series.
pFrqList	const XDAS_Int16*	N/A	N/A	Frequency planner. Frequencies are in Hz.
signalsCount	XDAS_Int16	N/A	>0	Number of CMS signals can be generated.
pSignalList	const UMTG_Signal*	N/A	N/A	Data for CMS generation (see Table 2–8).
pTimeList	const XDAS_UInt16*	N/A	N/A	Tone duration planner. Durations are in tens of milliseconds.

```
}
IUMTG_GeneralSeriesParams;
```

Table 2–5. Filter Parameters

```
typedef struct
{
```

Name	Type	Typical Value	Limits	Description
coefCount	XDAS_Int16	N/A	0..16	Number of FIR filter coefficients.
pCoefs	const XDAS_Int16*	N/A	N/A	Pointer to constant array of FIR filter coefficients.

```
}
IUMTG_FilterData;
```

Note: Filter Parameter Usage

Filter is used to prevent clicks at the beginning and at the end of generated tone. The generator output will be filtered and the high frequency components will be substantially suppressed. But performance will be reduced. Set mCoefCount to zero to disable filtering.

Table 2–6. Parameters for Generated Signal Series

```
typedef struct
{
```

Name	Type	Typical Value	Limits	Description
filterData	IUMTG_FilterData	N/A	N/A	Filter parameters (see Table 2–5).
pParam	const UMTG_GeneralSeriesParams*	N/A	N/A	General series parameters described above (see Table 2–4).

```
}
IUMTG_Series;
```

Table 2–7. Common UMTG Parameters

```
typedef struct
{
```

Name	Type	Typical Value	Limits	Description
host	IUMTG_Host	N/A	N/A	Host controller (see 2.5).
seriesCount	XDAS_Int16	N/A	>0	Number of structures with data about generated signal series.
pSeries9	IUMTG_Series*	N/A	N/A	Pointer to structures containing data for signal generation.

```
}
IUMTG_Params;
```

Table 2–8. CMS Generation Parameters

```
typedef struct
{
```

Name	Type	Typical Value	Limits	Description
signalID	XDAS_Int16	N/A	>0	CMS signal identifier. Each signal has to be provided with unique identifier. This value shall be used by a host to choose CMS signal for generation.
count	XDAS_Int16	N/A	>0	Number of time slots.
pTimeSlots	const UMTG_TimeSlot*	N/A	N/A	Pointer to time slots belong to CMS signal.

```
}
UMTG_Signal;
```

Table 2–9. UMTG Time Slots

```
typedef struct
{
```

Name	Type	Typical Value	Limits	Description
aFSO	XDAS_UInt16[2] (bitfield, see section 2.4.1)	N/A	N/A	Frequency selector options that control the generation of this CMS. Format of bit fields is defined in Table 2–10 and Table 2–11.
flags	XDAS_Int16 (bitfield, see section 2.4.2)	N/A	N/A	Flags that control actions associated with this time slot (see Table 2–12 and Table 2–14). And time slot duration index.

```
}
UMTG_TimeSlot;
```

2.4.1 Frequency Selection Options

These flags control the selection of frequencies for generation in the current time slot. This member consists of eight 4-bit fields, each corresponding to an appropriate frequency in the common frequency list (see Table 2–10 and Table 2–11).

Table 2–10. Bit Field Positions in the Frequency Selection Options

Bit Numbers	Word Number	Description
0-3	0	options for frequency with index 0
4…7	0	options for frequency with index 1
8…11	0	options for frequency with index 2
12…15	0	options for frequency with index 3
0-3	1	options for frequency with index 4
4…7	1	options for frequency with index 5
8…11	1	options for frequency with index 6
12…15	1	options for frequency with index 7

Table 2–11. Frequency Selection Options

```
typedef enum
{
```

Name	Value	Description
IUMTG_FIGNORE	0x00	Given frequency component is ignored
IUMTG_FGEN	0x01	Given frequency component will be generated with amplitude magnitude.
IUMTG_FGEN2	0x02	Given frequency component will be generated with amplitude magnitude2.
IUMTG_FMOD	0x03	Given frequency component will be used as modulation frequency.

```
}
IUMTG_FSOptions;
```

2.4.2 Time Slot Flags

This member consists of two parts. The higher 8 bits of mFlags contains flags, which controls signal generation (see Table 2–12) and flow control (see Table 2–13). The lower 8 bits contain the index of time slot duration in pTimeList array (see Table 2–4).

Time slot duration index = (flags & 0x0ff).

In case of IUMTG_GPHASEREVERSAL option, time between phase reversals is get from array pointed by pTimeList, item index is calculated using formula:

Index of duration between phase reversals = (flags & 0x0ff)+1.

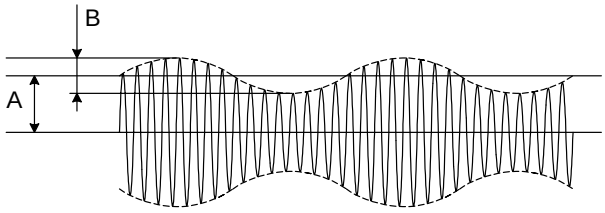
2.4.2.1 Signal Generation Options

Signal generation options control state machine operation and generation process.

Generation goes slot-by-slot until the entire signal will be generated.

Execution is always started from the first time slot.

Table 2–12. Signal Generation Options

Name	Value	Description
IUMTG_GPHASEREVERSAL	0x0100	Forces UMTG to generates phase reversal signal. Time between phase reversals is get from array pointed by `pTimeList` with index calculated using following formula `((flags & 0x0ff)+1)`.
IUMTG_GAMPMOD	0x0200	Forces UMTG to generate amplitude-modulated signal. Modulation coefficient is in `mMagnitude2` (`magnitude2 = 0x8000*M`, where `M = B/A; A = magnitude`). 

2.4.2.2 Flow Control Options

Additionally, time slot flags contain fields that provide flow control (see Table 2–13). They are used together with signal generation options.

Table 2–13. Flow Control Options

Name	Value	Description
IUMTG_FGOFIRST	0x1000	Forces UMTG to start from the first time slot when this time slot will be finished.
IUMTG_FDONOTCLEARPHASE	0x2000	UMTG would not reset the phase after finishing this time slot. This option can be used only with continuous signals.
IUMTG_FCALLBACK	0x4000	UMTG would execute callback function after finishing this time slot.

2.5 Host Interface

UMTG generator informs the host about the end of signal generation by calling the callback function. The host is attached to the generator on object creation (see section 2.2). The generator executes callback function only when signal generation is finished and option `IUMTG_FCALLBACK` is set for last time slot.

The second parameter in callback function is UMTG's object handle, so you can set next signal for generation or/and do something else.

Table 2–14. Host Controller

```
typedef struct
{
```

Name	Type	Typical Value	Limits	Description
`pInstance`	`Void*`	N/A	N/A	Internal host instance handle. It is always used as the first parameter in the function defined below.
`pfnHandler`	`XDAS_Void (*) (Void*, IUMTG_Handle)`	N/A	N/A	Host callback function to be invoked when generation of last time slot is finished. UMTG uses field `mpInstance` defined above as a first parameter for this function. Second parameter is UMTG handle, so you can set next signal for generation or do anything else.

```
}
UMTG_Host;
```

2.6 Examples

2.6.1 Typical CPTD Settings for USA's PSTN

2.6.1.1 Header File

```
#ifndef UMTG_CPTD_USA_H___

#define UMTG_CPTD_USA_H___ 1
```

2.6.1.2 Source File

```
#include "iumtg.h"

// Frequency planner. Frequencies are in Hz.

static const XDAS_Int16 UMTG_CPTDfrq_USA[] =
{  350, 440, 480, 620

};

// Duration array, durations are in tens of msec

XDAS_UInt16 UMTG_CPTDdur_USA[] =

{    20, 30, 50, 100, 200, 400, 1000

};//  0   1   2    3    4    5     6

static const IUMTG_TimeSlot RING_slots_USA[] =

{ { {0x0110, 0x0000}, 0x04 }, // 2000 msec
  { {0x0000, 0x0000}, 0x05|IUMTG_FGOFIRST|IUMTG_FCALLBACK }, // 4000 msec

};

static const IUMTG_TimeSlot DIAL_slots_USA[] =

{ { {0x0011, 0x0000}, 0x06|IUMTG_FGOFIRST|IUMTG_FDONOTCLEARPHASE|IUMTG_FCALLBACK
}, // 10000 msec

};

static const IUMTG_TimeSlot BUSY_slots_USA[] =

{ { {0x1100, 0x0000}, 0x02 }, // 500 msec

  { {0x0000, 0x0000}, 0x02|IUMTG_FGOFIRST|IUMTG_FCALLBACK }, // 500 msec

};

static const IUMTG_TimeSlot FASTBUSY_slots_USA[] =

{ { {0x1100, 0x0000}, 0x01 }, // 300 msec

  { {0x0000, 0x0000}, 0x01|IUMTG_FGOFIRST|IUMTG_FCALLBACK }, // 300 msec

};

static const IUMTG_Signal signals_USA [] =

{

  ADD_UMTG_SIGNAL(0x2001, RING_slots_USA),

  ADD_UMTG_SIGNAL(0x2002, DIAL_slots_USA),
```

```
  ADD_UMTG_SIGNAL(0x2003, BUSY_slots_USA),
  ADD_UMTG_SIGNAL(0x2004, FASTBUSY_slots_USA),
};
IUMTG_GeneralSeriesParams CPTD_USA_constSerie =
{
  8000,                                        // magnitude;
     0,                                        // magnitude2;
  sizeof(UMTG_CPTDfrq_USA)/sizeof(XDAS_Int16), // frqCount;
  (XDAS_Int16*)UMTG_CPTDfrq_USA,               // pFrqList;
  sizeof(signals_USA)/sizeof(IUMTG_Signal),    // signalsCount;
  (IUMTG_Signal*)signals_USA,                  // pSignals;
  UMTG_CPTDdur_USA                             // pTime
};
```

2.6.2 Typical DTMF Settings

2.6.2.1 Header File

```
#ifndef UMTG_DTMF_H___
#define UMTG_DTMF_H___ 1
#include ”iumtg.h”
extern const IUMTG_GeneralSeriesParams DTMF_constSerie;
#endif /* UMTG_DTMF_H___ 1 */
```

2.6.2.2 Source File

```
#include ”iumtg.h”
#define TONE_FLAGS                     (0x00)
#define PAUSE_FLAGS   IUMTG_FCALLBACK| (0x01)
static const XDAS_Int16 UMTG_DTMFfrq[] =
{  697,  770,  852,  941,
  1209, 1336, 1477, 1633
};
// now you can easy change DTMF tone
// and pause durations on the fly
XDAS_UInt16 UMTG_DTMFdur[] = { 8, 8 };
static const IUMTG_TimeSlot DTMF_1[] =
{ { {0x0001, 0x0002}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
```

```
};
static const IUMTG_TimeSlot DTMF_2[] =
{ { {0x0001, 0x0020}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_3[] =
{ { {0x0001, 0x0200}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_4[] =
{ { {0x0010, 0x0002}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_5[] =
{ { {0x0010, 0x0020}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_6[] =
{ { {0x0010, 0x0200}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_7[] =
{ { {0x0100, 0x0002}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_8[] =
{ { {0x0100, 0x0020}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_9[] =
{ { {0x0100, 0x0200}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_0[] =
{ { {0x1000, 0x0020}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
```

```
};
static const IUMTG_TimeSlot DTMF_A[] =
{ { {0x0001, 0x2000}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_B[] =
{ { {0x0010, 0x2000}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_C[] =
{ { {0x0100, 0x2000}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_D[] =
{ { {0x1000, 0x2000}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_STAR[] =
{ { {0x1000, 0x0002}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_TimeSlot DTMF_GRID[] =
{ { {0x1000, 0x0200}, TONE_FLAGS  },
  { {0x0000, 0x0000}, PAUSE_FLAGS }
};
static const IUMTG_Signal DTMF_signals [] =
{
  ADD_UMTG_SIGNAL(0x4001, DTMF_1),
  ADD_UMTG_SIGNAL(0x4002, DTMF_2),
  ADD_UMTG_SIGNAL(0x4003, DTMF_3),
  ADD_UMTG_SIGNAL(0x4004, DTMF_4),
  ADD_UMTG_SIGNAL(0x4005, DTMF_5),
  ADD_UMTG_SIGNAL(0x4006, DTMF_6),
  ADD_UMTG_SIGNAL(0x4007, DTMF_7),
  ADD_UMTG_SIGNAL(0x4008, DTMF_8),
  ADD_UMTG_SIGNAL(0x4009, DTMF_9),
```

```
  ADD_UMTG_SIGNAL(0x4000, DTMF_0),
  ADD_UMTG_SIGNAL(0x400A, DTMF_A),
  ADD_UMTG_SIGNAL(0x400B, DTMF_B),
  ADD_UMTG_SIGNAL(0x400C, DTMF_C),
  ADD_UMTG_SIGNAL(0x400D, DTMF_D),
  ADD_UMTG_SIGNAL(0x400E, DTMF_STAR),
  ADD_UMTG_SIGNAL(0x400F, DTMF_GRID),
};
IUMTG_GeneralSeriesParams DTMF_constSerie =
{
   2000,                                        //magnitude;
   2000,                                        //magnitude2;
  sizeof(UMTG_DTMFfrq)/sizeof(XDAS_Int16),      //frqCount;
  (XDAS_Int16*)UMTG_DTMFfrq,                    //pFrqList;
  sizeof(DTMF_signals)/sizeof(IUMTG_Signal),    //signalsCount;
  (IUMTG_Signal*)DTMF_signals,                  //pSignals;
  UMTG_DTMFdur                                  //pTime
};
```

2.6.3 Modified Answer Tone ANSam

2.6.3.1 Header File

```
#ifndef UMTG_ANSAM_H___
#define UMTG_ANSAM_H___ 1
#include ”iumtg.h”
extern const IUMTG_GeneralSeriesParams ANSam_constSerie;
#endif /* UMTG_ANSAM_H___ 1 */
```

2.6.3.2 Source Signal

```
#include ”iumtg.h”
static const XDAS_Int16 UMTG_frq[] =
{ 15, 210
};
static       XDAS_UInt16 UMTG_dur[] =
{ 100, 45
};
static const IUMTG_TimeSlot ANSam_slots[] =
```

```
{
  {{0x0013, 0x0000}, IUMTG_GAMPMOD|IUMTG_GPHASEREVERSAL|IUMTG_FDONOTCLEAR-
PHASE|IUMTG_FGOFIRST|IUMTG_FCALLBACK }
};
static const IUMTG_Signal signals [] =
{
  ADD_UMTG_SIGNAL(0x4444, ANSam_slots),
};
const IUMTG_GeneralSeriesParams ANSam_constSerie =
{
   3000,                                  //magnitude;
  (XDAS_Int16)(0x8000*0.4),               //magnitude2;
  sizeof(UMTG_frq)/sizeof(XDAS_Int16),    //frqCount;
  UMTG_frq,                               //pFrqList;
  sizeof(signals)/sizeof(IUMTG_Signal),   //signalsCount;
  signals,                                //pSignals;
  UMTG_dur                                //pTimeList;
};
```

2.6.4 UMTG_Params Structure

2.6.4.1 Header File

```
#ifndef UMTG_SIGNALS_H___
#define UMTG_SIGNALS_H___ 1
#include ”iumtg.h”
extern IUMTG_Params umtg_params;
#endif /* UMTG_SIGNALS_H___ 1 */
```

2.6.4.2 Source Signal

```
#include ”iumtg_gen.h”
#include ”umtg_DTMF.h”
#include ”umtg_ANSam.h”
static const XDAS_Int16 DTMF_coefs[] =
  { –1638,  –522,  454,  –2522,
    –4559,  2530, 13211, 13211,
     2530, –4559, –2522,   454,
     –522, –1638,  –707 };
```

```
IUMTG_Series umtg_series[] =
{
  {{sizeof(DTMF_coefs)/sizeof(int16),DTMF_coefs}, &DTMF_constSerie},
  {{0, 0}, &ANSam_constSerie}
};
IUMTG_Params umtg_params =
{
  sizeof(IUMTG_Params),
  {0, 0},
  sizeof(umtg_series)/sizeof(IUMTG_Series),
  umtg_series
};
```

Chapter 3

Universal Multifrequency Tone Generator (UMTG) API Descriptions

This chapter provides the user with a clear understanding of Universal Multifrequency Tone Generator (UMTG) algorithms and their implementation with the TMS320 DSP Algorithm Standard interface (XDAIS).

Topic **Page**

3.1 Standard Interface Stuctures

In this section, Standard interface structures for the UMTG are described.

Table 3–1 lists the UMTG Generator Real-time Status parameters.

3.1.1 Instance Creation Parameters

Description This structure defines the creation parameters for the algorithm. A default parameter structure is defined in "`iUMTG.c`".

Structure Definition Use structure `IUMTG_Params` (see Table 2–7) to provide each instance with parameters.

Type `IUMTG_Params` is defined in "`iUMTG.h`".

3.1.2 Status Structure

Description This structure defines the status parameters for the algorithm. Generator status structure is used for control purposes. Status can be received by function `UMTG_getStatus().`

Structure Definition

Table 3–1. UMTG Generator Real-Time Status Parameters

```
typedef struct IUMTG_Status
{
```

Status Type	Status Name	Description
XDAS_Int16	mID	Identification number of signal being generated.
XDAS_Int16	mSlotIndex	Index of time slot being generated.
XDAS_Int16	mSamples	Number of samples written to output buffer.

```
}
IUMTG_Status;
```

Type `IUMTG_Status` defined in "`iUMTG.h`".

3.2 Standard Interface Functions

The following functions are all required when using the UMTG algorithm.

Table 3–2 summarizes standard interface functions of UMTG Generator API.

`UMTG_apply()` and `UMTG_control()` are optional, but neither are supported by Spirit Corp.

Table 3–2. UMTG Generator Standard Interface Functions

Functions	Description	See Page...
`UMTG_init`	Algorithm initialization	3-3
`UMTG_exit`	Algorithm deletion	3-3
`UMTG_create`	Instance creation	3-4
`UMTG_delete`	Instance deletion	3-4

3.2.1 Algorithm Initialization

UMTG_init *Calls the framework initialization function to initialize an algorithm*

Description This function calls the framework initialization function, `ALG_init()`, to initialize the algorithm. For UMTG generator, this function does nothing. It can be skipped and removed from the target code according to *Achieving Zero Overhead With the TMS320 DSP Algorithm Standard IALG Interface* (SPRA716).

Function Prototype `void UMTG_init()`

Arguments none

Return Value none

3.2.2 Algorithm Deletion

UMTG_exit *Calls the framework exit function to remove an algorithm instance*

Description This function calls the framework exit function, `ALG_exit()`, to remove an instance of the algorithm. For UMTG generator, this function does nothing. It can be skipped and removed from the target code according to *Achieving Zero Overhead With the TMS320 DSP Algorithm Standard IALG Interface* (SPRA716).

Function Prototype `void UMTG_exit()`

Arguments none

Return Value none

3.2.3 Instance Creation

UMTG_create *Calls the framework create function to create an instance object*

Description In order to create a new UMTG generator object, `UMTG_create` function should be called. This function calls the framework create function, `ALG_create()`, to create the instance object and perform memory allocation tasks. Global structure `UMTG_SPCORP_IUMTG` contains `UMTG` virtual table supplied by SPIRIT Corp.

Function Prototype

```
UMTG_Handle UMTG_create
      (const IUMTG_Fxns *fxns,
       const UMTG_Params *prms);
```

Arguments

`IUMTG_Fxns*`	Pointer to vendor's functions (Implementation ID). Use reference to `UMTG_SPCORP_IUMTG` virtual table.
`UMTG_Params*`	Pointer to parameter structure. Use `NULL` pointer to load default parameters.

Return Value `UMTG_Handle` is defined in file "`UMTG.h`". This is a pointer to the created instance.

3.2.4 Instance Deletion

UMTG_delete *Calls the framework delete function to delete an instance object*

Description This function calls the framework delete function, `ALG_delete()`, to delete the instance object and perform memory de-allocation tasks.

Function Prototype

```
void UMTG_delete (UMTG_Handle handle)
```

Arguments

`UMTG_Handle`	Instance's handle obtained from `UMTG_create()`.

Return Value none

3.3 Vendor-Specific Interface Functions

In this section, functions in the SPIRIT's algorithm implementation and interface (extended IALG methods) are described.

The whole interface is placed in header files `iUMTG.h`, `UMTG.h`, `UMTG_spcorp.h`.

UMTG object can be allocated statically. `UMTG_static.h` header contains types and function definitions for static UMTG version.

Table 3–3 summarizes function names for XDAIS and static UMTG versions.

Table 3–3. Generator-Specific Interface Functions

Function		Description	See Page...
XDAIS Version	**Static Version**		
`UMTG_setSignal`	`UMTG_setSignalStatic`	Performs signal selection for generation	3-5
`UMTG_genSignal`	`UMTG_genSignalStatic`	Generates samples	3-6
`UMTG_getStatus`	`UMTG_getStatusStatic`	Returns current generator status	3-7
	`UMTG_initStatic`	Initializes static UMTG object	3-7

3.3.1 Set Signal for Generation

UMTG_setSignal *Performs search for signals with the same identifier*

Description Performs a search for signals with the same identifier and prepares them for generation. If signal with specified identification number `signalID` not exist, silence will be generated.

Function Prototype XDAS version:

```
XDAS_Void UMTG_setSignal
      (UMTG_Handle handle,
       const XDAS_Int16 mSignalID)
```

Static version:

```
XDAS_Void UMTG_setSignalStatic
      (UMTG_HandleStatic handle,
       const XDAS_Int16 mSignalID)
```

Arguments

`handle`	Pointer to UMTG instance
`mSignalID`	Signal identifier

Return Value none

Restrictions none

3.3.2 Process Generation

UMTG_genSignal *Generates count output samples for DAC*

Description Generates `count` output samples for DAC.

Function Prototype XDAS version:

```
IUMTG_Status UMTG_genSignal
      (UMTG_Handle handle,
       const XDAS_Int16 in[],
       XDAS_Int16 count)
```

Static version:

```
IUMTG_Status UMTG_genSignalStatic
      (UMTG_HandleStatic handle,
       const XDAS_Int16 in[],
       XDAS_Int16 count)
```

Arguments

`handle`	Pointer to UMTG instance
`in`	Array for output samples
`count`	Number of samples to be generated

Return Value Returns `IUMTG_Status`.

Restrictions none

3.3.3 Get Actual Generator Status

UMTG_getStatus *Returns the current generator status*

Description Returns current generator status. Just copies internal state variables into status structure.

Function Prototype XDAS version:

```
Void UMTG_getStatus
      (UMTG_Handle handle,
       IUMTG_Status* pStatus)
```

Static version:

```
Void UMTG_getStatusStatic
      (UMTG_HandleStatic handle,
       IUMTG_Status* pStatus)
```

Arguments

`handle`	Pointer to UMTG instance
`pStatus`	Pointer to the status structure to be read

Return Value Actual generator status (see Table 3–1).

Restrictions none

3.3.4 Initialize Statically Allocated UMTG

UMTG_initStatic *Initializes statically allocated UMTG objects*

Description Initializes statically allocated UMTG object. Amount of memory needed for UMTG object depend on parameters and can be calculated by formula:

`MEM_SIZE` = (0x26 + <amount of series in parameters> * 0x13) [words]

Statically allocated memory block should be aligned to double word boundary.

Function Prototype

```
UMTG_HandleStatic UMTG_initStatic
      (void* mem,
       UMTG_Params* parameters)
```

Arguments

`mem`	Pointer to memory block for UMTG object
`parameters`	Pointer to parameters structure

Return Value Handle of statically allocated UMTG object.

Appendix A

Test Environment

Note: Test Environment Location

This chapter describes test environment for the UMTG object.

For TMS320C54CST device, test environment for standalone UMTG object is located in the Software Development Kit (SDK) in `Src\FlexExamples\StandaloneXDAS\UMTG`.

Topic **Page**

A.1 Description of Directory Tree

The SDK package includes the test project “test.pjt” and corresponding reference test vectors. The user is free to modify this code as needed, without submissions to SPIRIT Corp.

Table A–1. Test Files for UMTG

File	Description
main.c	Test file
FileC5x.c	File input/output functions
..\ROM\CSTRom.s54	ROM entry address
Test.cmd	Linker command file
Vectors\output.pcm	Reference output test vectors

A.1.1 Test Project

To build and run a project, the following steps must be performed:

Step 1: Open the project: `Project\Open`

Step 2: Build all necessary files: `Project\Rebuild All`

Step 3: Initialize the DSP: `Debug\Reset CPU`

Step 4: Load the output-file: `File\Load program`

Step 5: Run the executable: `Debug\Run`

Once the program finishes testing, the file *Output.pcm* will be written in the current directory. Compare this file with the reference vector contained in the directory *Vectors*.

Note: Test Duration

Since the standard file I/O for EVM is very slow, testing may take several minutes. Test duration does not indicate the real algorithm's throughput.

Index

A

C

D

E

F

G

H

I

M

P

S

T

U

X