International Telecommunication Union

# ITU-T G.191 STL-2019 Manual

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(02/2019)

International telephone connections and circuits —
Software tools for transmission systems

## ITU-T Software Tool Library 2019 User's Manual

Recommendation T G.191 STL-2019 Manual

ITU-T

International Telecommunication Union

# Recommendation ITU-T G.191 STL-2019 Manual

## ITU-T Software Tool Library 2019 User's Manual

**Foreword**



Published by the ITU and available as part of the STL distribution, available from:
http://www.itu.int/rec/T-REC-G.191/en

The STL distribution is maintained on an open-source collaboration platform:
https://www.github.com/openitu/STL

## Introduction

In July 1990, Study Group XV of the then CCITT (Comité Consultatif International Téléphonique et Télégraphique) decided to set up a group to deal with the development of common software tools to help in the development of speech coding standards. In the same period, cooperation was requested with SG XII Speech Quality Experts Group (SQEG), and a group called 'User's Group on Software Tools' (UGST) was initially established with almost 20 corresponding members. The basic means of interaction were the then incipient electronic mail (e-mail) messages, for the exchange of files and experiences — UGST was actually one of the pioneer groups in ITU collaborating via electronic means. In addition to this, there were meetings held mainly during regular Working Party XV/2 (Signal Processing) sessions, where most of the decisions were made.

As result of that very intensive work, several software tools evolved forming the '*1992 ITU-T Software Tool Library*' (STL92) which included, as its first application, the Qualification Test for a Speech Coder at 8 kbit/s. After this initial release, another release was approved by ITU-T Study Group 15 in May, 1996, and called STL96. The STL96 introduced substantive improvement and new features to the STL92. In November 2000, ITU-T Study Group 16 approved an updated version to the STL, the STL2000. In 2005, another updated version of the STL, STL2005, was accepted.

In 2009, a new version of the STL, STL2009 was developed. STL2009 corrects bugs and brings revisions (such as G.722 codec software, basic operators C-code, or reverberation tool), and adds new tools (more FIR filters, new EID tool, basic operator counters, floating-point complexity counters, and a stereo operator tool). Note that for STL2009 release, non-ASCII (American Standard Code for Information Interchange) encoded characters were substituted by ones in ASCII, for wider compiler portability. A potential bug related to memory leak in `ugst-demo.h` possibly caused by very long filenames is fixed. All files that make use of `GET_PAR_C()` macro is affected.

Approved in January 2019 under the responsibility of ITU-T Study Group 12, STL2019 incorporated new basic operators to accommodate state-of-the-art processor architectures which supports wide accumulators, SIMD (Single Instruction Multiple Data) and VLIW (Very Long Instruction Word). Thus the new operators provide support for 64-bit accumulator, complex numbers, as well as enhanced 32-bit operations and additional control code operators. The software package was reworked to make it available as a truly open-source project hosted on an open-source collaboration platform: https://www.github.com/openitu/STL.

The STL2019 build toolchain is using CMake to generate platform-dependent and tool-dependent build scripts as well as to execute regression tests for each module in the STL. STL2019 modules have been tested on Windows, MacOS and several Linux flavors.

Terms and conditions on the usage of the ITU-T STL are found in Recommendation ITU-T G.191 [19].

The remaining chapters of this document describe the principles that guided the generation of the ITU-T STL, as well as the description of its organization. The various tools are described in separate chapters. These descriptions have the following general outline:

a) technical description of the method or algorithm involved;
b) description of the algorithm implementation in this release (including prototypes, parameters, returned value, etc.); and
c) testing, applications and examples.

**Organization of the Software Library**

Each tool of the STL has been produced as a stand-alone module, such that it may be linked to a user's program, application or system. In the present version, there are several of these modules:

| | |
|---|---|
| **G.711** | The 64 kbit/s PCM algorithm with A and $\mu$ law of ITU-T Rec. G.711. |
| **G.711-PLC** | The high-quality, low complexity packet-loss concealment specified in ITU-T Rec. G.711 Appendix I. |
| **G.726** | 40, 32, 24, and 16 kbit/s ADPCM algorithm of ITU-T Rec. G.726. |
| **G.727** | 40, 32, 24, and 16 kbit/s embedded ADPCM algorithm of ITU-T Rec. G.727. |
| **G.728** | Low-delay CELP algorithm at 16 kbit/s of ITU-T Rec. G. 728. |
| **G.722** | 64, 56, and 48 kbit/s wideband speech ADPCM algorithm of ITU-T Rec. G.722. |
| **RPE-LTP** | The 13 kbit/s RPE-LTP algorithm of the full-rate GSM system (GSM Rec. 06.10). |
| **RATE-CHANGE** | An up- and down-sampling algorithm with embedded filtering:<br>– ITU-T Rec. G.712 filter for factors of 1:2, 2:1 and 1:1<br>– High-quality filter for factors 1:2, 2:1, 1:3, and 3:1<br>– IRS send-side weighting filter, for several sampling rates: 8, 16, and 48 kHz. This includes the "full-IRS" as in ITU-T Rec. P.48 as well as the "modified" IRS as in Annex D of ITU-T Rec. P.830.<br>– Modified-IRS receive-side filter is also available for 8 and 16 kHz sampled data.<br>– $\Delta_{SM}$ weighting filter for near-to-far field conversion<br>– Psophometric weighting filter of ITU-T Rec. O.41 for noise measurements<br>– ITU-T P.341 weighting filter for wideband signal (50-7000 Hz)<br>– 100-5000 Hz bandpass filter<br>– 50-14000 Hz bandpass filter (P341 extension for super-wideband signal)<br>– 20-20000 Hz bandpass filter<br>– MUSHRA anchors (1.5 kHz, 3.5 kHz, 7 kHz, 10 kHz, 12 kHz and 14 kHz low-pass filters). |
| **EID** | Error insertion algorithm, with routines for generation of bit error patterns (random or burst) as well as random and burst frame erasure, and adaptation to layered bitstream. |
| **MNRU** | The modulated noise reference unit of ITU-T Rec. P.810 (formerly ITU-T Rec. P.81). |

| | |
|---|---|
| **SVP56** | The Speech Voltmeter for measuring the active speech level (which skips over silence in a utterance) of ITU-T Rec. P.56. |
| **REVERB** | Tool to add reverberation to both mono and stereo speech and audio. |
| **TRUNCATE** | Bitstream truncation tool. |
| **FREQRESP** | Frequency response measurement tool. |
| **STEREOOP** | Stereo processing tool. |
| **BASOP** | The set of basic digital signal processing (DSP) operators that represent instructions typically available in digital signal processors *(revised in STL2019)*. |
| **UTILITIES** | Tools that have been developed to assure proper interfacing between the various tools. These tools do not relate to any ITU-T Recommendation. Included are tools for conversion between float and short data representations, between parallel and serial (bit-stream) formats, and for scaling of data. |

It should be noted that C code is available for a number of codecs as a normative part of the respective standards, e.g. ITU-T G.711.0, G.711.1, G.718, G.719, G.722.1, G.722.2, G.723.1, G.729, G.729.1, enhanced aacPlus general audio codec; ETSI GSM-HR, GSM-EFR, GSM-AMR; TIA IS-641, IS-127, IS-96A, among others. These source codes are not appropriate for inclusion in the ITU-T STL for a number of reasons: they are an integral part of the respective standards, are maintained within the scope of the respective standards development organizations (SDOs), are protected by copyrights, and are openly available. Parties interested in acquiring these source codes should contact the appropriate SDO.

### Whom to contact

In case of problems with any of the tools, please contact the ITU-T Study Group 12 secretariat at <tsbsg12@itu.int>. Please provide a precise description of the problem with proper reference to the C-code, and possible solution(s), if known.

### Acknowledgements

Several organizations which participate in ITU-T Study Groups 12, 15 and 16 have substantially contributed to the completion of this release of the ITU-T STL.

First and foremost, UGST wishes to thank CPqD/Telebrás (Brazil) for its support of the early coordination (1990-1993) of the activity and of the development of the following tools: Utilities, G.711, G.726, MNRU, and SVP56. For the first two, the work was shared with PKI (Germany), which also provided the initial version of the modules EID and RATE-CHANGE, as well as basic material that supported the initial organization of the work, together with Telenor (formerly NTA, Norway) and the DBP-Telekom (Germany).

DBP-Telekom also collaborated in providing several software tools used in the Host Laboratory for the ITU-T 8 kbit/s speech coder: modified IRS filters, adaptation of the Bellcore burst frame erasure model, and $\Delta_{SM}$ filter. UGST also wants to thank CSELT (Italy) for making available its Fortran MNRU

program, which was the starting point of the present implementation, and for the implementation of the psophometric filter.

CNET (France) provided the G.722 tool, which was greatly appreciated.

UGST kindly thanks Mr Jutta Deneger for allowing the incorporation of his implementation of the RPE-LTP algorithm in the STL.

Also, Bellcore provided several programs in Fortran and C that, while not used directly in the present version of the STL, were important in various stages of the development of the Library, especially a version of the Red Book G.721.

PTT Ukraine graciously provided the G.727 implementation, which was warmly welcomed.

COMSAT Labs (subsequently, part of Lockheed Martin Global Telecommunications, LMGT), in turn, provided essential help in funding the coordination work and the harmonization and documentation of the tools during an important consolidation period (1994-2001).

Also important was the testing work done by the Research Institute of the Deutsche Telekom (now T-Nova/DT), as well as PKI, Telebrás, AT&T (USA), and CNET.

Several parts of this manual were possible only by the contribution of several individuals: Mr Pierre Combescure (CNET) for the description of the G.722 algorithm, Mr Rudolf Hofmann (PKI), for description the Gilbert-Elliot channel implemented in the EID module, Mr Peter Kroon (AT&T) for the description of the RPE-LTP algorithm, and Mr Vijay Varma (Bellcore) for the text describing the Bellcore Burst Error Model.

Since 2003, several companies have jointly worked on the Basic Operators revision and an alternative set addition: Texas Instruments, Conexant Systems, STMicroelectronics, Hughes Software Systems, France Telecom, and VoiceAge.

Besides this work on Basic Operators, ITU-T Q7/12 and Q10/16 experts work on the addition of new tools. France Telecom and Polycom have provided essential contributions in these STL2005 works.

Special thanks to ITU-T Q7/12 rapporteurs, Mr Paolo Usai (ETSI) and Ms Catherine Quinquis (France Telecom), ITU-T Q10/16 STL work moderators (2004-2008), Mr Karim Djafarian (Texas Instruments) and Mr Stéphane Ragot (France Telecom). France Telecom also provided great support for the management of Q10/16, responsible for the up-keeping of the STL since 2002.

The following persons have contributed to the 2005 edition of this manual: Mr Karim Djafarian (Texas Instruments) to the edition of the Basic Operators chapter, Mr Claude Marro (France Telecom) to the chapter on the reverberation tool, Mr Cyril Guillaumé (on behalf of France Telecom) to chapters on the frequency response measurement tool and the bitstream truncation tool, Mr David Kapilow (AT&T) to the chapter of G.711 PLC.

For the release of STL2009, EID-EV tool and new 20 Hz to 20 kHz bandpass filter were prepared by Mr Jonas Svedberg (Ericsson). The revision of G.722 tool to introduce basic PLC options, G.192 bitstream and basic operators was performed by Mr Jonas Svedberg (Ericsson) and Mr Balazs Kovesi (France Telecom Orange). The stereo measured impulse responses for the reverberation tool were provided by Mr David Virette and Mr Claude Marro (France Telecom Orange) while the simulated fullband impulse response was provided by Mr Minjie Xie (Polycom). Thanks goes to Mr Balazs Kovesi (France Telecom Orange) for developing the program ROM evaluation counters and to Mr Tommy Vaillancourt, Mr Vaclav Eksler, and Mr Vladimir Malenovsky (VoiceAge) for introducing floating-point complexity counters. For the new anchor 12 kHz low-pass filters, there was a contribution from Mr Miao Lei (Huawei Technologies). Special thanks to Mr Hans Gierlich, ITU-T Q6/12 Rapporteur, for providing the guidelines on test signals suitable for the frequency response measurement tool. Revision of frequency response measurement tool was performed by Mr Pierre Berthet (on behalf of

France Telecom Orange) and Mr Deming Zhang (Huawei Technologies). The G.728 C-source code both in fixed- and floating-point arithmetics was kindly provided by Mr David Kapilow (AT&T). Some useful examples were added on usage of basic operators and Mr Noboru Harada (NTT) and Mr Karim Djafarian (Texas Instruments) should be thanked for this work. Mr Adrien Cormier (France Telecom Orange) reviewed STL manual chapters and tools. Mr Xu Jianfeng (Huawei) should also be thanked for his assistance in compilation of the new STL2009 tool packages. Last but not least, it should be mentioned that this release was only possible owing greatly to Ms Claude Lamblin (France Telecom Orange), ex-Q10/16 Rapporteur and WP3/16 Chair. She has devoted a lot of time revising the manual text for STL2009 release, and above all, took all the responsibility in releasing STL2005.

The new basic operators in STL1019 were kindly contributed by 3GPP SA4. The work leading to the release of STL2019, including the migration of the STL to an open-source environment; the development of a new build toolchain; the integration of new base operators; and the update of this manual, was performed by Mr Dennis Guse and Mr Ludovic Malfait (Dolby Laboratories). Other GitHub contributors to STL2019 were Mr Thomas Schlien (RWTH Aachen University), Mr Simão Ferraz de Campos Neto (ITU), and Mr Martin Adolph (ITU).

Above all, special thank goes to ITU-T SG16 Counselor Mr Simão Ferraz de Campos Neto, the "father" of the STL.

## FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications , information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1 .

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

## NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency .

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party .

## INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at http://www.itu.int/ITU-T/ipr/.

## Table of Contents

**Page**

# Recommendation ITU-T G.191 STL-2019 Manual

## ITU-T Software Tool Library 2019 User's Manual

### 1. Scope

None.

### 2. References

None.

### 3. Definitions

None.

### 4. Abbreviations and acronyms

None.

### 5. Conventions

None.

### 6. Tutorial

#### 6.1. Acronyms

Several acronyms are used in this text. The most relevant are:

| | |
|---|---|
| ANSI | American National Standards Institute. |
| BBER | Burst Bit Error Rate |
| BER | Bit Error Rate (refers to *random* bit errors) |
| BFER | Burst Frame Erasure Rate |
| DAT | Digital Audio Tape. |
| EID | Error insertion device. |
| ETSI | European Telecommunications Standards Institute. |
| FER | Frame Erasure Rate (refers to *random* frame erasures) |
| GSM | Global System for Mobile Communications. Pan-European digital-cellular system operating at a net rate of 13 kbit/s in its full-rate system. |
| IRS | Intermediate Reference System, defined in ITU-T Rec. P.48 for the so-called "full-IRS" mask, or in Annex D of ITU-T Rec. P.830 for the so-called "modified" IRS mask. |
| ITU | International Telecommunication Union. |
| ITU-T | Standardisation Sector of the International Telecommunication Union. |
| LSB | Least significant bit. |
| MIRS | Modified-IRS telephony speech weighting (in ITU-T Rec. P.830 Annex D). |

| | |
|---|---|
| MSB | Most significant bit. |
| PSTN | Public Switched Telecommunication Network. |
| R&O | Requirements and Objectives, for performance of software tools. |
| SQEG | Speech Quality Experts Group, of ITU-T Study Group 12. |
| PLC | Packet loss concealment |
| STL92 | ITU-T Software Tools Library, release 1992. |
| STL96 | ITU-T Software Tools Library, release 1996. |
| STL2000 | ITU-T Software Tools Library, release 2000. |
| STL2005 | ITU-T Software Tools Library, release 2005. |
| STL2009 | ITU-T Software Tools Library, release 2009. |
| STL2019 | ITU-T Software Tools Library, release 2019. |
| UGST | Users' Group on Software Tools, of ITU-T Study Group 16. |

## 6.2. Definition of terms

In the documentation of the ITU-T software tools, several terms are widely used and are defined below.

### 6.2.1. Overload point

The overload point within the digital domain is defined by the (normalized) amplitude value.

$$x\_over \triangleq 1.0$$

How this overload point relates to the analogue world depends on the conversion method between the analog and digital domains, and is beyond the scope of this document. All signals in this manual are relative to this overload point in the digital domain.

NOTE – This overload point does NOT depend on the quantisation method used and remains identical, regardless of whether the quantisation is done e.g. with 32, 16, 13 or 8 bits.

1) In floating-point (either single or double precision) implemntations, the representation of this value is exact. In this text, and also in the tools, this data type is called `float`.
2) In 32 bit 2's complement representation the data can be represented by multiplying the normalized value by $2^{31}$. For example, the largest possible positive value is represented by `0x7FFFFFFF`. The largest negative value is represented by `0x80000000`. In this text, and also in the tools, this data type is called `long`.
3) In 16 bit 2's complement representation the data can be represented by multiplying the normalized value by $2^{15}$. For example, the largest possible positive value is represented by `0x7FFF`. The largest negative value is represented by `0x8000`. In this text, and also in the tools, this data type is called `short`.
4) The statements above may be generalized for all wordlengths in fixed-point representation. The idea is to set the decimal point just after the MSb (sign bit).

### 6.2.2. Signal power

The power of a signal x(n) with a length of N samples is defined by

$$P = \frac{1}{N}\sum_{n=0}^{N-1} x(n)^2$$

A signal which does not contain amplitude values exceeding the overload point can have a maximum signal power of 1.0. This is the power of a DC signal with an amplitude of 1.0 or of any other signal comprising only the values $\pm 1.0$ (e.g., a square wave signal).

### 6.2.3. Signal level

The power level in decibels is defined relative to a reference power level $P_0 = 1.0$:

$$L = 10\log_{10}(P/(P_0))\,(\text{dBov})$$

The level of a signal power $P = 1.0$ is thus 0 dBov (where the characters "ov" arbitrarily mean digital **ov**erload signal level), which is chosen to be the reference level. A signal with such power level could be either (a) a sequence of maximum positive numbers $(+1)$, (b) a sequence of maximum negative numbers $(-1)$, or (c) a rectangular function exercising only the positive or negative maximum numbers $(\pm 1)$. The level of a sinewave with an amplitude (peak value) of 1.0 is therefore $L = -3.01$ dBov.

### 6.2.4. Relation between overload and maximum levels

The measurement of signal levels in the digital part of the network is normally expressed by telecommunications engineers as $y$ dBm0, i.e., the level relative to 1 mW in 600 $\Omega$. However, from the software point of view, it is more convenient to represent levels relative to the maximum power that can be stored in integer format on a computer, e.g. $z$ dBov. A conversion between both representations can be expressed as:

$$y(\text{dBm0}) = z(\text{dBov}) + C$$

For the G.711 encoding rule, a sinewave which exercises the maximum level has a power $T$max of 3.14 dBm0 for A-law, and of 3.17 dBm0 for $\mu$-law. On the other hand, the RMS level of these sinewaves would always be -3.01 dBov. Therefore, $C$ above becomes 6.15 dB for A-law and 6.18 dB for $\mu$-law. For the G.722 wideband coding algorithm, the overload point of the A/D and D/A converters should be 9 dBm0. Therefore, in that case, C becomes 12.01 dB.

The following relationships summarize the discussion:

$$\Lambda_A(\text{dBm0}) = L_{ov}(\text{dBov}) + 6.15dB(\text{A-law})$$

$$\Lambda_\mu(\text{dBm0}) = L_{ov}(\text{dBov}) + 6.18dB(\mu - \text{law})$$

$$\Lambda_{wb}(\text{dBm0})) = L_{ov}(\text{dBov}) + 12.01dB(\text{G.722})$$

### 6.2.5. Saturation

Saturation is the limitation of signal amplitudes to values equal to or smaller than the overload point:

$$y(k) = \begin{cases} -1.0, & \text{if} \quad x(k) < -1.0 \\ x(k), & \text{if} \quad -1.0 \le x(k) \le +1.0 \\ +1.0, & \text{if} \quad x(k) > +1.0 \end{cases}$$

### 6.2.6. Data representation

Unless otherwise noted all waveforms within the signal processing are assumed to have infinite precision and unlimited amplitude. The overload point is therefore the reference point only. In practice these signals may well be represented in 32 bit floating-point arithmetic or high precision integer arithmetic (24 bit for data and coefficients, 48 to 56 bit for products and accumulation). In most cases, 16 or 32 bit integer arithmetic is not precise enough.

Signals derived from 16 bit 2's complement representation (DAT, files, digital I/O interface) should be converted to this (approximately) infinite precision before processing by modules that require floating-point input. Normalization of the floating-point values to the overload point is recommended.

### 6.2.7. Data justification

Justification of data here is used without distinction to data alignment and data adjustment: where the upper or lower significant bit of an integer sample is located.

*Left-justified* data are samples whose most significant bit is located at the leftmost position of the computer storage unit used for it. Remaining low-bit positions must be set to zero.

*Right-justified* data are samples whose least significant bit is located at the rightmost position of the computer storage unit used for it. Remaining upper bits depend on the data representation: if two's complement, sign extension from sample's MSb to storage's MSb is needed; otherwise, the upper (unused) bits shall be zeroes.

As an example, suppose a 12-bit resolution, two's complement sample, to be stored for processing in a `short`. If left-justified, then a sign bit (the MSb!) is found in bit 15 (the MSb) of the `short` that stores it. On the other hand, if right-justified, the LSb will be the bit 0 of the `short`, in this case. If it is a negative number, there would be sign extension for bit 12 to 15. If it is an unsigned number, the upper 4 bits (in the example) are all zeros. Example illustrates these three cases.

**EXAMPLE — Illustration of a left- and right-justified data with 12-bit resolution. Bit types *s*, *v*, and *x* represent respectively sign bit(s), valid bits and unused bits.**

**Left-justified data**

| ***Bit number*** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Bit type* | s | v | v | v | v | v | v | v | v | v | v | v | x | x | x | x |

**Right-justified, sign-extended data**

| ***Bit number*** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Bit type* | s | s | s | s | v | v | v | v | v | v | v | v | v | v | v | v |

**Right-justified, unsigned data**

| ***Bit number*** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Bit type* | 0 | 0 | 0 | 0 | v | v | v | v | v | v | v | v | v | v | v | v |

### 6.2.8. Equivalent results

Several software tools, such as the G.711 algorithm, are defined in terms of precise fixed-point operations. Therefore, when comparing the output of one of these algorithms on different platforms, or for compilation using different C compilers, one should expect identical sample values for reference processed materials.

Other algorithms, however, may include highly intensive processing, or complex mathematical functions. Examples of these are rate change filters and floating-point arithmetic speech coders, such as the 16 kbit/s LD-CELP of ITU-T Rec. G.728. In such cases, it is expected that the processing of the same reference material on different platforms will generate almost identical results. The generated files will probably be identical for most of the samples,and for some samples they will differ by a small amount, e.g. $\pm 1$, or more rarely by $\pm 2$ or more. For the purposes of the STL, such an implementation is said to produce *equivalent results* on different platforms.

### 6.2.9. Little- and big-endian data ordering

Present computer systems agree only on the data access for byte-oriented data structures. Although computer systems exist whose bytes do not have 8-bits, the majority of the systems implement bytes as 8-bit data structures. In general, the computer architectures do not differ in the way they access the bit-order within a byte. In other words, for the vast majority of the computer systems existing today, the least significant bit occupies the lower memory position (i.e., bit 0), and the most significant bit occupies the higher memory position in the byte (i.e., bit 7). In terms of C operations, if `b` is a byte structure, then `b&0x1` returns the LSb, and `(b>>7)&0x1` returns the MSb.

Although most computer architectures agree on the definition of a byte and how its bits are accessed, they vastly differ on how multi-byte structures are accessed. Trivial examples of multi-byte structures are 16-bit `short` words or 32-bit `long` words. There are currently two access means currently implemented by different CPUs in the market, which differ on the significance of the bytes that are first read from memory positions.

On the so-called *big-endian* systems, the first byte read from a multi-byte structure is always the most significant byte. For example, if the two bytes 0x12 (low address) and 0x34 (high address) are stored in two consecutive memory addresses, then the number read and stored in the CPU accumulator would be 0x3412, or 13330 in decimal. The big-endian data organization is, for this reason, also known as *high-byte first*.

For the so-called *little-endian* systems, the first byte read from a multi-byte structure is always the least significant byte. For this reason, the little-endian data organization is also known as *low-byte first*. Using the same example as before, for the two consecutive bytes in memory `0x12` and `0x34`, the value loaded on a little-endian CPU will be `0x1234`, or 4660 in decimal.

The concept is extended to other multi-byte data structures, such as 32-bit or 64-bit integers. For example, the consecutive bytes `0x12`, `0x34`, `0x56`, and `0x78` would be loaded as the 32-bit integer `0x78563412` on the accumulator of a big-endian CPU and as `0x12345678` on the accumulator of a little-endian CPU.

**Table 4 — Example of big- and little-endian systems**

| Big-endian | | Little-endian | |
|---|---|---|---|
| Computer | Microprocessor | Computer | Microprocessor |
| Sun-3 | Motorola 680x0 | IBM-PC/compatible[a] | Intel x86 |
| Sun-4 | Sun SPARC family | DEC-Stations | MIPS RISC |
| Silicon Graphics | MIPS RISC | DEC Alpha | DEC Alpha AXP |
| IBM 370 | IBM | VAX-/VMS- | VAX CPU |
| HP 9000-700 | HPPA RISC | Microcomputers | |
| Legend:<br>CISC: Complex Instruction Set Computer<br>RISC: Reduced Instruction Set Computer | | | |
| a) Includes Windows 9x/NT/2000/XP/Vista/7, Linux and Solaris on Intel CPUs. | | | |

Table 4 indicates the data organization for several computer platforms. It should be noted that the data organization is a function of the CPU family rather than of the operating system used. For example, Solaris on Sparc platforms uses big-endian data organization, while Solaris on Intel 80x86/Pentium platforms uses little-endian data organization. Similarly, most Linux systems are little-endian (because they run on Intel 80x86/Pentium CPUs), but several other implementations are actually big-endian (e.g. PowerPC CPU used in Macintosh machines).

The segment of C code in Figure 1 can be used to determine whether a given computer system has big- or little-endian data organization.

```
#include <stdio.h>
#include <string.h>

int is_little_endian()
{
  /* Hex version of the string ABCD */
  unsigned long tmp = 0x41424344;

  /* Compare the hex version of the four characters with the ASCII version */
  /* On big-endian (or high-byte-first) systems, 0x41 ('A' in ASCII) */
  /* is stored in the first memory position, and the equivalent string */
  /* is "ABCD". On a little-endian (or low-byte-first) system, 0x41 is */
  /* stored in the last position, and the equivalent string will be */
  /* "DCBA". Function strncmp will return 0 if both strings are equal */
  /*  upto the first four characters. */
  return(strncmp("ABCD", (char *)&tmp, 4));
}

void main()
{
  printf("System is %s-endian\n", is_little_endian()? "little" : "big");
}
```

**Figure 1 — Sample code for determination of byte organization**

The approach above determines whether a platform is big- or little-endian, but it does not answer the question of what is the byte orientation in a given file. Although there is no closed-form method for such a determination, there is an empirical method that can be carefully used for speech signals (usually represented using 16-bit linear PCM words) based on two speech properties: speech signals follow a gamma distribution (hence most of the samples have small amplitude), and levels in voiced segments are usually in the -15 dBov through -40 dBov range. For files that have a byte orientation mismatching that of the computer platform, the mostly small samples of the speech signal will be measured as having large amplitude. Hence, if a high-level power is found when measuring the power of a voiced segment (typically around -4 dBov), one can assume that the file needs to be byte-swapped. It is important however to measure the level for *voiced segments*, since for silent intervals the increase in gain is not so dramatic and will not allow for a conclusion on the byte-orientation of the file.

When the change of format is necessary for `short` and `long` data, the operations in Figure 2 should be used. The conversion between big- and little-endian data representation for 16-bit data is simple and is known as *byte swapping*. The byte swapping operation can be implemented in several fashions. For example,

```
short swap_one_short(short in)
{
  return (((in>>8)&0xFF) | (in<<8));
}
```

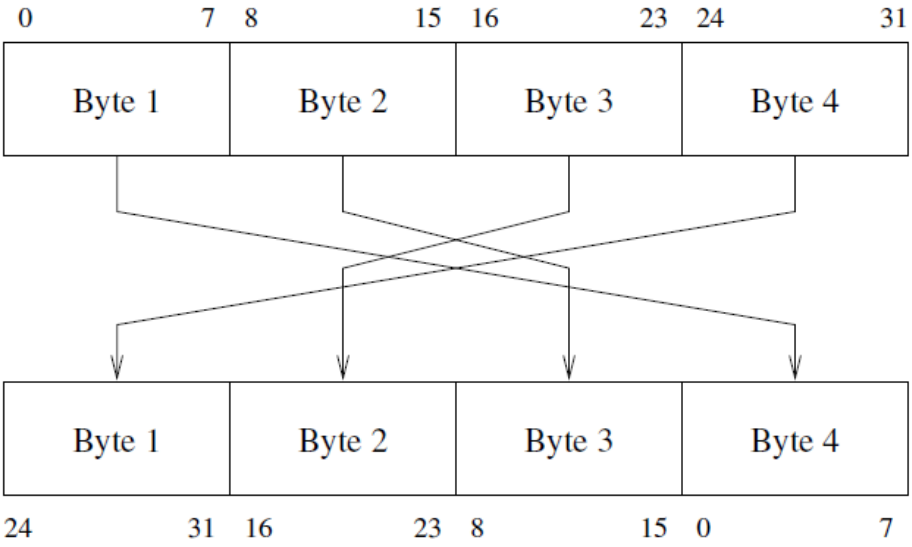



**Figure 2-a — Conversion between little− and big−endian for 32−bit data**

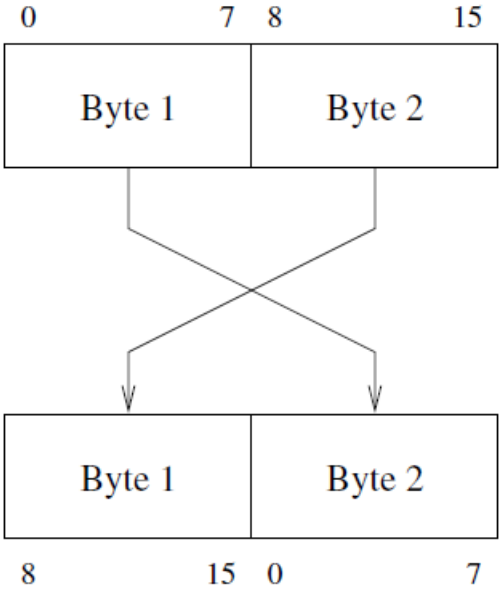



**Figure 2-b — Conversion between little− and big−endian for 16−bit data**

**Figure 2 — Conversion between big- and little-endian**

It should be noted that the simple byte-swapping above does not work properly for conversion of other multi-byte structures. For the purposes of the STL, however, 16-bit structures is the most import case. For several of the STL modules, the provided test files in general need to be byte swapped in one or another computer platform. The documentation and the "manifesto" accompaining each software tool module describe which files, if any, should be byte-swapped on certain platforms. As default, binary files organized in 16-bit words are provided in big endian format in the STL distribution.

### 6.3. Guidelines for software tool development

The software tools provided by the ITU-T User's Group on Software Tools are to be used by laboratories with different computers and A/D-D/A equipment. To make the software accessible to everybody, it should be highly portable across operating systems and allow for easy implementation in existing hardware environments.

To achieve this, some simple guidelines were followed in the development of the tools. The following are the UGST guidelines used to generate the official and beta releases of the ITU-T Software Tool Library.

a) All software should be written in ANSI C.
b) Features of the language whose representation may create side-effects should not be used (e.g. `union`).
c) All variables must be declared and the types used in the declarations must be the least platform dependent. For example, the keyword `int` must be avoided. Instead `short` should be used for 16-bit integers and `long` should be used for 32-bit integers.
d) The software should not contain any input or output that may be system dependent (e.g. open, read and write file operations). Instead, data must be passed to the modules as parameters of function calls. This will allow each laboratory to integrate the modules with their own application software without changing the modules. Interfaces to various file formats and user interaction can optionally be provided as example main programs[1] that will not be a part of the library module and should contain the least possible amount of code.
e) Well defined digital signal formats should be used and documented for each module to allow the various modules to work together.
f) The interface to the file system should be made in a standard way, but only within the example programs.
g) The source code should be properly documented, with a standard header.
h) Modularity is encouraged in the software design. All modules are self-contained, i.e. global definitions should be avoided.
i) Each module should have an attached specification document explaining the function and use of the module, the level of detail depending on its complexity.
j) The software modules shall be distributed to interested laboratories for comments and testing before they are approved and included in the ITU-T Software Tools Library, to minimize the ocurrence of bugs and to assure conformance with related ITU-T Recommendations (when applicable). Two test procedures have been devised: compliance and portability.

The *compliance procedure* (or compliance test) is to certify that a given tool module fully complies with specifications, which should be carried out by at least one organization other than the proponent organization (or by a group of organizations, each one checking a different subset of the specifications, such that all together cover all the specifications). In order to minimize the probability of systematic errors, these procedures should be defined by the verifying organization(s) without input from the tool provider(s).

The *portability verification procedure* (or portability test) is to certify that a given validated tool works on platforms other than the one(s) where they were generated and validated. In simple cases these verification procedures could be just test vectors (e.g. speech or noise files). It was also pointed out that problems may arise in Unix platforms, due to the existence of several flavors of Unix available today (this means that a verification procedure could be valid in one Unix machine, but not in other).

Portability verification procedures should be provided by the proponents and shall be run on at least two relevant operating systems (DOS, UNIX). In the past, procedures for the VMS operating system used to be required, however this operating system has become less common. For DOS, the "pure" 16-bit mode has become less common, and 16-bit emulation window under a 32-bit version of MS Windows is now prevalent. These facts affect the choice of compiler.

---

1 Also called *"demonstration programs"* in this manual.

The following is a list of compilers used to test the portability of tools in the STL, although not all tools were necessarily tested with all compilers.

| | |
|---|---|
| *HP/c89* | This is the `c89` compiler that can be purchased from HP for use in HP-UX systems. For the STL, tests with this compiler were performed with HP-UX 9.05. |
| *HP/gcc* | This is the HP-UX port of the `gcc` compiler. The specific version may differ from tool to tool. Versions used included gcc 2.7.2.2 for HP-UX 9.05 and gcc-2.95.2 for HP-UX 10.20. |
| *MSDOS/gcc* | This is the MSDOS-6.22 port of the `gcc` compiler version 2.6.3-DJGPP V1. This is a 32-bit compilation of the code, however using a 16-bit interface. Executables are not likely to run under Windows MS-DOS emulation window. Needs a run-time 32-bit extender called `go32.exe`. |
| *MSDOS/tcc* | This is the Borland Turbo C++ Version 1.00 `tcc` compiler. |
| *MSDOS/bcc* | This is Borland C++ `bcc` compiler. Versions used included 3.0 and 4.5. |
| *Solaris/gcc* | This is the `gcc` compiler version 2.95 running under Solaris 7, usually in a Sparc platform. |
| *SunOS/cc* | This is the basic `cc` C compiler bundled in the SunOS distribution. For the STL, SunOS version 4.1.3 was used. |
| *SunOS/acc* | This is the licensed `acc` C compiler sold by Sun Microsystems. For the STL, SunOS version 4.1 was used. |
| *Win32/gcc* | This is the `gcc` compiler version 2.95 running under Windows NT 4 SP 4 and with the CYGWIN Unix emulation interface. These executables need either the CYGWIN environment or the run-time library cygwin1.dll to run, and they are expected to work properly in a DOS emulation window under Windows 95/98 as well. This version will not run under native MS-DOS. |
| *Win32/cl* | This is the command-line `cl` version 12.00.8168 C compiler of the MS Visual C V.6 SP3 running under the WinNT 4 SP4 (the executables will also run in Windows 95/98/SE/Me/2000). This version will not run under native MS-DOS. |

### 6.4. Software module I/O signal representation

The idea behind the choice of the convention in this section is that all software modules within the ITU-T tool library should be independent building blocks which can easily be combined by connecting the output of one module to the input of the next module. With this characteristic, various systems may be very easily constructed. The individual software modules must have well-defined interfaces to allow such simple connections, especially at the I/O level. This convention is based on the following:

1) All modules work 'from RAM to RAM'. This means that the working modules are independent from physical I/O functions which are normally machine dependent. This approach also allows easy cascading of modules within one 'main' program.

2) All signals at the I/O interfaces of modules are represented in one of the following ways:
   i) in single or double precision (32 or 64 bit) floating-point representation. The normalized signal is used directly (*overload point = reference point = 1.0*)
   ii) in 32 bit 2's complement representation. The normalized signal must be multiplied by $2^{31}$ (i.e. the decimal point is just after the MSb, same as for 16 bit representation). If less than 32 bits are required, then the signal is left adjusted within the 32 bit longword and the LSbs are optionally set to 0.
   iii) in 16 bit 2's complement representation, as described in section clause 6.2.1. If less than 16 bits are required, then the signal is left adjusted (left-justified) within the 16 bit words and the LSbs are optionally set to 0. If the host machine does not provide a format with 16 bit width, then the next longer wordlength should be used with the 16 bits right adjusted.
3) Data exchange with a module shall be done directly within the calling statement (not by global variables).
4) Data exchange with a module shall be done sample-by-sample (FIR-filtering, MNRU, etc.) or frame-by-frame (block oriented speech codec, etc.), whichever is more convenient. Larger blocks may be formed (e.g. 128 samples at a time) for better efficiency, however the block size should be rather small (less than 512). The block and its length shall be variables.
5) All modules shall be constructed in a way that infinitely long signals may be processed with a reasonable amount of internal storage. As an example, the 'main' program could read a block of input data (e.g., next frame of time signal samples) from the disk, call a module or sequence of modules, write the output signal (e.g., next frame of coded parameters) back onto disk. This process is repeated for all the input data blocks of interest.
6) All modules shall have
   i) an initialization part (if necessary) and
   ii) a working part

   The initialization part may be necessary to reset internal state variables, define the mode of operation (e.g. MNRU-mode), and so on. It is called only once at the beginning or whenever a reset to an initial state is needed.

   The working part performs the processing itself. It leaves all state variables in a well-defined manner for the immediate use within the next call. One possible way to do this is to introduce a flag-variable within the call statement (e.g., named 'Initialize') which is set by the 'main' program to '1' for initialization and is set to '0' during normal operation. In this way, only one function for one module is necessary. Alternatively, a specialized initialization routine may be written, to be called before the main processing routine of the module. Only one of the approaches will be followed in the future. However, both are present in the current version of the STL.

   NOTE –

   All state variables (if any) must be initialized at execution time, not at compile or load time.
7) The RAM allocation shall in principle be split into 'static' and 'temporary' parts. 'Static' means that the contents must be saved from call to call, preferrably by means of state variables rather than truly static variables[2]. 'Temporary' means that the contents are not saved between sucessive calls of the module.
8) All modules are separated in clearly and independently defined functions, but accompanied by an example 'main' program which may also include file I/O.

[2] As a rule, state variables should not be defined as truly static ones because this may cause side-effects.

## 6.5. Tool specifications

For each tool, there are 'Requirements and Objectives' (R&Os) associated. Each of the R&Os has both a general and a specific part.

The general part includes the following[3]:

1) Portability among platforms and Operating Systems (DOS, UNIX, and VMS):
   i) compilation [GL-i];
   ii) usage of language features that may cause side-effects [GL-ii];
   iii) usage of language features that may be ambiguous among platforms [GL-iii];
   iv) usage of system dependent calls (to access resources such as files, etc. within the modules) [GL-iv];
2) Efficiency:
   i) use of CPU (i.e., execution speed);
   ii) use of I/O (intensity of access to files, etc.);
   iii) use of memory (physical/virtual);
   iv) code's coverage (verbosity versus laconism);
3) Documentation:
   i) Self-documentation (e.g., comments, variables and structure resembling ITU-T Recommendations, etc.)[GL-vii];
   ii) Separate documentation (clarity, objectivity, etc.)[GL-ix];
4) Modularity [GL-viii]
5) Fixed- versus floating-point implementations;

Following are descriptions of each of the General R&Os. Full description of the R&Os can be found in [40], Annex 4.

*General performance specification* refers to the document that specifies the tool in question, e.g. an ITU-T Recommendation or ANSI or ETSI standard.

*Portability* addresses several points related to the tool's capacity of working on several platforms: *Compilation and linkage* refers to the necessity of changes in the source code to make a tool compile without any modification in a given environment. It was identified that the operating systems of most interest are DOS and Unix (both BSD and System V). *Side-effectable features* are those that, if used in a program, when changing one parameter, may cause other(s) to be changed implicitly. *Ambiguous features* are those that, due to the flexibility left in the C language specification, are implemented in different ways for different platforms. For example, `int` in C is 32-bit wide in VAX-C and Unix workstations, but is 16-bit wide for most compilers available on MS-DOS (Turbo-C/MS-C). *System-dependent calls* are calls that are restricted to or are implementation of features of a particular platform, to make better use of that particular computer architecture.

*Efficiency* is related to how the computer's resources are used in terms of CPU, I/O and memory allocation, that may be a burden and prevent the usage in some systems, either by lack of resources or length of time needed for execution. Efficiency also includes *code's coverage*, expressing how frequently code is accessed.

*Documentation* refers to how to describe the tool. *Self-documentation* is the documentation present in the program itself to assure that the code clearly describes the algorithm implemented, to provide compilation and linkage instructions, as well as to report known bugs, etc. A *separate document* will be mandatory when no written description of the algorithm is available, or when the written documents that specify the tool are too general.

---

[3] GL*x* refers to the Guideline number *x* in clause 6.3, e.g., GLiii is the Guideline iii.

*Modularity degree* is the degree of isolation that a particular tool has. From UGST Guidelines, all tools must be modular, i.e., self-contained blocks; nonetheless, tools may make use of system resources other than memory and CPU.

*Arithmetic* is the number representation specification, either in fixed (2's complement, 1's complement, etc.) or floating point. Here, "fixed-point" shall always be understood as 2's complement representation, except where otherwise indicated.

## 7. G.711: The ITU-T 64 kbit/s log-PCM algorithm

In the early 1960's an interest was expressed in encoding the analog signals in telephone networks, mainly to reduce costs in switching and multiplexing equipments and to allow the integration of communication and computing, increasing the efficiency in operation and maintenance [51].

In 1972, the then CCITT published the Recommendation ITU-T G.711 that constitutes the principal reference as far as transmission systems are concerned [20]. The basic principle of the algorithm is to code speech using 8 bits per sample, the input voiceband signal being sampled at 8 kHz, keeping the telephony bandwidth of 300—3400 Hz. With this combination, each voice channel requires 64 kbit/s.

### 7.1. Description of the algorithm

The idea behind the digitalization of the network involved a compromise: use as far as possible the existing infrastructure; this imposes a bandwidth limitation for the bit-streams of coded signals. A rate of 64 kbit/s was found to be reasonable.

If one thinks of using the most natural quantization scheme, one will choose linear quantization. But one drawback of this approach is that the signal-to-noise ratio (SNR) varies with the amplitude of the input signals: the smaller the amplitude, the smaller the SNR. And, from the quality point of view, if a signal has a wide variance, or a variance that changes with time (as in the case of speech signals), the SNR will also change, resulting in a wide-varying quality of the system.

To avoid this problem, one can use logarithmic quantization, which will result into a more uniform quantization noise. With this in mind, several studies were carried out in late 1960's to choose a good algorithm for this purpose. This led to the definition of two transmission schemes, one using the $\mu$ law compression characteristic:

$$c(x) = x_{\max}\frac{\ln(1+\mu|x|/x_{\max})}{\ln(1+\mu)}sgn(x)$$

and the other using the A law compression characteristic:

$$c(x) = \begin{cases} \frac{A|x|}{1+\ln(A)}sgn(x) & \text{for } 0 \le \frac{|x|}{x_{\max}} \le \frac{1}{A} \\ x_{\max}\frac{1+\ln(A|x|/x_{\max})}{1+\ln(A)}sgn(x) & \text{for } \frac{1}{A} \le \frac{|x|}{x_{\max}} \le 1 \end{cases}$$

Both characteristics behave as linear for small amplitude signals (being then equivalent to a linear quantization scheme), but are truly logarithmic for large signals. In fact, for large signals the SNR is:

$$SNR_{\mu} = 6.02B + 4.77 - 20\log_{10}(\ln(1+\mu))$$

and

$$SNR_{A} = 6.02B + 4.77 - 20\log_{10}(1+\ln A)$$

where $B$ is the number of bits used for quantization.

The ITU chose the values $A = 87.56$ and $\mu = 255$ for the G.711 standard, together with 8 bits per sample, what leads the latter two equations to:

$$SNR_{\mu} = 6.02B - 9.99 = 38.17dB$$

and

$$SNR_{A} = 6.02B - 10.1 = 38.06dB$$

The G.711 standard does not specify the law as defined above, but rather uses a good linear-piecewise approximation for 8 bit samples, which has easier implementation (in hardware), as well as other properties (see [69],p.229).

This approximation uses bit 1 for sign (1 for positive, 0 for negative), bits 2—4 to indicate a segment, and bits 5—8 for level[4]. Within each segment, the quantization is linear (4 bits, or 16 levels), having 15 segments of distinct slopes for $\mu$ law, and 13 for A law.

The A law works with signals in the range from -4096 to 4096, implying in a range of 13 bits. As for the $\mu$ law, the linear signals are accepted in the range -8159 to 8159, which is represented by 14 bits. Besides this, in the dynamic range sense, A and $\mu$ law are equivalent to 12 and 13 bit linear quantization, respectively.

One detail for the A law is that the even bits are inverted. The reason for this comes from problems observed (before the standardization of the line code HDB3) in transmission systems when long sequences of zeros happen, because small amplitudes, in A law, to be coded mostly using '0' bits. With this bit-inversion, long sequences of bits '0' becomes less probable, thus improving performance.

The conversion rule for $A/\mu$ law from/to linear is described in terms of tables in G.711. A good reason for this is that there is no closed form for the compression of linear samples (although it is possible to find a closed formulae for the expansion algorithm). Hence, two implementations are possible: table look-up, and algorithmic. For in-chip (LSI) implementations, the first one may be preferred, because it is simpler to implement, at the cost of a wider chip area. For other applications, such as using Digital Signal Processors (DSPs), or software implementations, table look-up would occupy too much memory, and the algorithmic solution would be preferred.

### 7.2. Implementation

This implementation of the G.711 can be found in the module `g711.c`, with prototypes in `g711.h`.

For the reason explained before, an algorithmic approach to the G.711 was followed. For the compression routines, first the samples are converted from two's complement to signed magnitude notation[5]. So, a segment classification is done, and then the linear quantization of a certain number of bits of the input sample, that depends on the segment number (e.g., for A law, segment 1 uses a factor of 2:1, 2 a 4:1, etc.) is carried out. Finally, the sign of the sample is added. The expansion routines are even simpler: find the sign, get the mantissa and the exponent, and compute the linear sample.

One important point here is that, following UGST Guidelines, linear input samples must be left-justified `short`s. With this approach, the knowledge of the 0 dB reference for the file is simplified, and the need of having to apply different normalization factors to files if they are to be coded by A or

[4] Please note that the bit numbering in the G.711 is the reversal of the commonly used in computer languages, G.711's bit 1 corresponding to common-sense's (most significant) bit 7, and G.711's bit 8 to the normal least significant bit 0, respectively.

[5] Using the samples as two's complement in the compression algorithm is a very common error whose effects are only noticeable for small amplitude signals. Our approach agrees to the one in G.726 [21], block *compress*.

$\mu$ law is eliminated[6]. As an example, suppose that we want to process a speech file $\chi$ by the G.711 at an input level of -20 dBov for both A and $\mu$ law. Then, if the sample representation is right-justified, and a factor $f$ brings a file's level to -20 dBov for $\mu$ law, then for A law the factor will be $2.f$, due to the difference in input signal's dynamic range of both laws (4096 and 8159, respectively). On the other hand, if the samples are left-justified, the factor is only one, and the routines will only look at the 13 or 14 most significant bits of the 16-bit word, for A and $\mu$ law, respectively. In other words, the peak value for linear and $A/\mu$ law is the same, therefore one factor is sufficient.

Compliance tests to this code have been done using a ramp file having the full excursion of the dynamic range for each of the laws, and examining the compressed and expanded samples against the values expected in tables 1a, 1b, 2a, and 2b of Recommendation ITU-T G.711 (see [20]). Another test done exploits the synchronous property of the G.711 scheme. Only samples from column 7 of G.711 tables 1 and 2 were used. These values are transparent to quantization. Hence, if the coding was done properly, output samples should match exactly the original ones.

The compression functions are `alaw_compress` and `ulaw_compress`, and the expansion functions are `alaw_expand` and `ulaw_expand`. In the next part you find a summary of calls to these functions.

### 7.2.1. `alaw_compress` and `ulaw_compress`

**Syntax:**

```
#include "g711.h"
void alaw_compress (long _smpno_, short _*lin_buf_, short _*log_buf_)
void ulaw_compress (long _smpno_, short _*lin_buf_, short _*log_buf_)
```

**Prototype:** `g711.h`

**Description:**

`alaw_compress` performs A law encoding rule according to Recommendation ITU-T G.711, and `ulaw_compress` does the same for $\mu$ law. Note that input samples shall be left-justified, and that the output samples are right-justified with 8 bits.

**Variables:**

| | |
|---|---|
| *smpno* | Is the number of samples in lin_buf. |
| *lin_buf* | Is the input samples' buffer; each `short` sample shall contain linear PCM (2's complement, 16-bit wide) samples, left-justified. |
| *log_buf* | Is the output samples' buffer; each `short` sample will contain right-justified 8-bit wide valid A or $\mu$ law samples. |

**Return value:** None.

### 7.2.2. `alaw_expand` and `ulaw_expand`

**Syntax:**

```
#include "g711.h"
void alaw_expand (long _smpno_, short _*log_buf_, short _*lin_buf_)
void alaw_expand (long _smpno_, short _*log_buf_, short _*lin_buf_)
```

**Prototype:** `g711.h`

**Description:**

[6] In the case of stand-alone tools, this would mean that two copies of the same file should be available!

`alaw_expand` performs A law decoding rule according to Recommendation ITU-T G.711, and `ulaw_expand` does the same for $\mu$ law. Note that output samples will be left-justified, and that the input samples shall be right-justified with 8 bits.

**Variables:**

| | |
|---|---|
| *smpno* | Is the number of samples in log_buf. |
| *log_buf* | Is the input samples' buffer; each `short` sample shall contain right-justified 8-bit wide valid A or $\mu$ law samples. |
| *lin_buf* | Is the output samples' buffer; each `short` sample will contain linear PCM (2's complement, 16-bit wide) samples, left-justified. |

**Return value:** None.

### 7.3. Tests and portability

Portability may be checked by running the same speech file in a proven platform and in a test platform. Files processed this way should match exactly. Source and processed reference files for portability tests are provided in the STL distribution.

These routines had portability tested for VAX/VMS with VAX-C and gcc, MS-DOS with Turbo C v2.0, HPUX with gcc, and Sun-OS with Sun-C.

### 7.4. Example code

#### 7.4.1. Description of the demonstration program

One program is provided as demonstration program for the G.711 module, g711demo.c.

Program `g711demo.c` accepts input files in 16-bit linear PCM format for compression operation and produces files in the same format after the expansion operation. The compressed signal will be in 16-bit, right adjusted format, according to the logarithmic law specified by the user. Three operations are possible: linear in, linear out (*lili*) linear in, logarithmic out (*lilo*), or logarithmic in, linear out (*loli*).

#### 7.4.2. Simple example

The following C code gives an example of companding using either the A- or $\mu$-law functions available in the STL.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "g711.h"

#define BLK_LEN 256
#define QUIT(m,code) {fprintf(stderr,m); exit((int)code);}

main(argc, argv)
  int             argc;
  char           *argv[];
{
  char            law[4];

  char            FileIn[180], FileOut[180];
  short           tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
  FILE           *Fi, *Fo;
  void          (*compress)(), (*expand)(); /* pointer to a function */

  /* Get parameters for processing */
  GET_PAR_S(1, "_Law (A,u): ................... ", law);
```

```
  GET_PAR_S(2, "_Input File: ................. ", FileIn);
  GET_PAR_S(3, "_Output File: ................ ", FileOut);

  /* Opening input and output LOG-PCM files */
  Fi = fopen(FileIn, RB);
  Fo = fopen(FileOut, WB);

  /* Choose compression/expansion routinies according to the law */
  if (toupper(law[0])=='A')
  {
     compress = alaw_compress;
     expand = alaw_expand;
  }
  else if (tolower(law[0])=='u')
  {
     compress = ulaw_compress;
     expand = ulaw_expand;
  }
  else
    QUIT("Bad law chosen!\n",1);

 /* File processing */
  while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
  {
    /* Process input linear PCM samples in blocks of length BLK_LEN */
    compress(BLK_LEN, inp_buf, tmp_buf);

    /* Process log-PCM samples in blocks of length BLK_LEN */
    expand(BLK_LEN, tmp_buf, out_buf);

    /* Write PCM output word */
    fwrite(out_buf, BLK_LEN, BLK_LEN, sizeof(short), Fo);
  }

  /* Close input and output files */
  fclose(Fi);
  fclose(Fo);
  return 0;
}
```

## 8. G.711 Appendix I: A high quality low-complexity algorithm for packet loss concealment with G.711.

### 8.1. Introduction

Packet Loss Concealment (PLC) algorithms hide transmission losses in audio systems where the input signal is encoded and packetized at a transmitter, sent over a network, and received at a receiver that decodes the packet and plays out the output. G.711 Appendix I [82], approved by ITU-T in September 1999, describes a high quality, low complexity PLC algorithm designed for use with G.711.

### 8.2. Description of the algorithm

A brief description of the PLC algorithm is given. A more extensive presentation can be found in Section I.2, "Algorithm description", of G.711 Appendix I [82].

The PLC algorithm is inserted after the G.711 decoder at the receiver. The algorithm is designed to work with 10 ms frames, or 80 samples per frame at 8 KHz sampling. An external mechanism is needed to signal when packets are lost. Since speech signals are often locally stationary, the signals

recent history is used to generate a reasonable approximation to lost frames. If the losses are not too long, and do not land in a region where the signal is rapidly changing, the losses may be inaudible after concealment.

When a frame is received the decoded speech is given to the PLC algorithm. Received frames are saved in a 48.75 ms circular history buffer, and the output is delayed by 3.75 ms (30 samples).

When a packet is lost the concealment algorithm starts synthetic signal generation. First the pitch is estimated by finding the peak of the normalized autocorrelation of the most recent 20 ms of speech in the history buffer with the previous speech at taps from 5 to 15 ms. Using the pitch estimate, the most recent pitch period from the history buffer is repeated for the duration of the first lost frame (10 ms). If the pitch estimate is longer than 10 ms, only a portion of the most recent pitch period will be used in the first lost frame. A 1/4 pitch period overlap add (OLA) with a triangular window is performed at all repetition boundaries, including the transition between the last received frame and the start of the synthetic signal.

If consecutive frames are lost, the number of pitch periods used to generate the synthetic signal is increased by one pitch period at the start of the 2nd and 3rd lost frames. When the number of pitch periods is increased, the output is smoothly transitioned to the oldest used pitch period of the history signal with an additional 1/4 pitch period OLA. Increasing the number of pitch periods reduces the number of unnatural harmonic artifacts in the concealed speech for long losses. The algorithm does not distinguish between voiced and un-voiced speech and uses the same procedure for both types of speech.

At the start of the first received frame after a loss, the synthetic signal generation is continued and OLAed with the received speech. This OLA window length increases with the length of the loss. For single frame losses it is 1/4 of the estimated pitch period. 4 ms are added for each additional consecutive lost frame, up to a maximum of 10 ms.

If the loss exceeds 10 ms the synthetic signal is also linearly attenuated at the rate of 20% per frame. If the loss exceeds 60 ms the synthesized signal is set to silence.

## 8.3. Implementation

### 8.3.1. Introduction

The g711iplc directory contains an ANSI C implementation of the G.711 Appendix I PLC algorithm. The C++ version of this algorithm is in the `g711iplc\cpp_cod` directory. Sample test programs read lost frame patterns in G.192 file format and apply the PLC algorithm to audio files. The software in the `g711iplc` directory is covered by a more restrictive copyright than the STL. See the `copyrght.txt` file for details.

### 8.3.2. PLC Algorithm Implementation

A detailed line by line description of the C++ code can be found in section I.3 "Algorithm description with annotated C\+ code" of G.711 Appendix I <<G711-appendix-I>> and will not be repeated here. The public interface functions that are called by applications are covered. The C+ version is in the `g711iplc\cpp_code` directory (files `lowcfe.h` and lowcfe.cc). The ANSI C version, contained in the files `lowcfe.h` and `lowcfe.c`, is a translation of the C\+ code to C. The interface functions are the same for both versions, with the exception that the C versions of the routines take an extra argument for the data structure that is implicitly passed to C+ member functions in the class instance data. As for other STL modules, only the ANSI C version is compiled during STL2005 building.

#### 8.3.2.1. Constructor

**C++ syntax:**

```
#include "lowcfe.h"
LowcFE  _lc_; // No argument constructor
```

**C syntax:**

```
#include "lowcfe_c.h"
g711plc_construct(_LowcFE_c*_); /* explicit constructor call */
```

**Description:**

Before the PLC algorithm can be called the data structure containing the algorithm's internal storage, such as the history buffer and buffer pointers, must be initialized.

#### 8.3.2.2. Received Frames

**C++ syntax:**

```
void LowcFE::addtohistory(short _s_); /* add a frame to the history buffer */
```

**C syntax:**

```
void g711plc_addtohistory(_LowcFE_c*_, short _*s_);
```

**Description:**

Frames of speech received from the transmitter are given to the PLC algorithm with `addtohistory` function. The argument *s* points to a short array of length FRAMESZ (80 samples, or 10 ms) that is used as both an input and output. Before the call is made *s* is filled with the decoded G.711 data received from the transmitter. On return, it contains the data that is output to the listener. `addtohistory` performs several operations. It stores the input speech into the history buffer for use in generating the synthetic signal if a loss occurs. If this is the first received frame after a loss, an OLA is performed with the synthetic signal to insure a smooth transition between the signals. In addition, it delays the output so an OLA can be performed at the start of a loss.

#### 8.3.2.3. Lost Frames

**C++ syntax:**

```
void LowcFE::dofe(short _*s)_;    /* synthesize speech during loss */
```

**C syntax:**

```
void g711plc_dofe(_LowcFE_c*_, short _*s_);
```

**Description:**

If a frame is lost, the `dofe` routine is called. As with `addtohistory`, *s* is a pointer to short array of FRAMESZ samples. With `dofe`, *s* is only an output. The PLC algorithm fills *s* with the synthetic signal that conceals the missing frame.

#### 8.3.2.4. Support Functions

`error`

**Syntax:**

```
#include "error.h"
void error(char _*s_, ...);
```

**Description:**

Error handles fatal errors in the programs. The pattern string, *s*, and optional following arguments should be in the format of arguments accepted by the C library `printf` function. Error prints its argument message on stderr and then exits the program. The error function never returns.

`readplcmask_open`

**Syntax:**

```
#include "plcferio.h"
void readplcmask open(readplcmask _*r_, char _*fname_);
```

**Description:**

The `readplcmask_open` function opens a G.192 format file containing a packet loss pattern. *fname* is the file path. If successfully opened, *r* contains the state information needed for reading the patterns. `readplcmask_open` internally calls the STL eid module to determine the type of the G.192 file and select an appropriate reading function. If the open fails or an unknown pattern is detected in the file, function `error` is called and `readplcmask_open` will not return.

`readplcmask_erased`

**Syntax:**

```
#include "plcferio.h"
int readplcmask erased(readplcmask _*r_);
```

**Description:**

`readplcmask_erased` reads the next value from the opened G.192 format pattern file. It returns 1 if the frame is lost and should be concealed and 0 if the frame is ok. If the end of the G.192 file is reached, the routine seeks back to the beginning of the file and the pattern sequence is repeated. If an illegal value is found in the G.192 file, the error function is called.

`readplcmask_close`

**Syntax:**

```
#include "plcferio.h"
void readplcmask close(readplcmask _*r_)
```

**Description:**

`readplcmask_close` is used to close a G.192 file that was opened with `readplcmask_open`.

### 8.3.3. Test Program

#### 8.3.3.1. Test Program Usage

The PLC algorithm is tested using g711iplc.c. The PLC test programs take 3 file arguments:

```
g711iplc mask.g192 input.raw output.raw
```

The `mask.g192` file contains the lost frame pattern and should be in the G.192 format as specified in the software tools library. The g192, byte, and compact representations are supported. The G.192 file should contain only the frame headers words (G192_SYNC or G192_FER, see `softbit.h`), and not the data words.

A frame corresponds to 10 ms, or 80 samples. If the lost frame pattern file is shorter than the number of frames in the `input.raw` file, the program will roll-over back to the start of the pattern file. For example if the `mask.g192` file contains the binary data:

```
0x6B21 0x6B21 0x6B21 0x6B21 0x6B21,
0x6B21 0x6B21 0x6B21 0x6B21 0x6B20
```

a 10% uniform loss pattern will be applied to the whole file. Erasures will occur at 90-100 ms, 190-200 ms, 290-300 ms … in the file.

While the algorithm is designed for packets containing 10ms of speech, it can be applied to packetizations containing speech chunks that are integer multiples of 10ms. For example, for a 10% uniform loss pattern with 20ms packetization one could use:

```
    0x6B21 0x6B21 0x6B21 0x6B21 0x6B21,
    0x6B21 0x6B21 0x6B21 0x6B21 0x6B21,
    0x6B21 0x6B21 0x6B21 0x6B21 0x6B21,
    0x6B21 0x6B21 0x6B21 0x6B20 0x6B20
```

to cause erasures at 180-200ms, 380-400ms, 580-600ms, etc.

The input audio file, `input.raw`, should contain header-less 16-bit binary data, sampled at 8 KHz, in the native byte order for the machine running the test programs (big-endian on SPARC or MIPS, little-endian on Intel). The test programs do not contain the G.711 encoder or decoder. If you have a G.711 bit-stream, it must be decoded before the `g711iplc` program is run.

The output audio file, `output.raw`, also contains header-less 16-bit binary data. The PLC algorithm delays the output by 3.75 ms. The test programs compensate for this delay by not outputting the first 3.75 ms of the first packet. This way the input and output files will be time aligned if they are overlaid in an audio waveform editor. In addition, after the last full packet is input to the PLC algorithm, an extra zero filled frame is input, and the first 3.75 ms of the corresponding output frame is sent to the output file. The length of the output file will always be a multiple of the 10ms frame size. If the input file length is not an integral number of frames the last partial input frame will be discarded.

The test programs can also simulate a silence insertion algorithm instead of the PLC algorithm with the `-nolplc` option:

```
  g711iplc -noplc mask.g192 input.raw output.raw
```

Instead of calling the concealment algorithm the lost frames are simply zero filled. This is helpful if you want to use a wave editor to view the location of the missing frames.

Use the `-stats` option to print out the number and percentage of frames concealed in the processed file.

#### 8.3.3.2. Test Program Implementation

A simplified version of the C++ test program is shown next. This program does not support any options, such as `-noplc`, or compensate for the algorithm delay, but demonstrates how the components work together.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "error.h"
#include "plcferio.h"
#include "lowcfe.h"

int main(int argc, char *argv[]) {
    FILE        *fi;        /* input file */
    FILE        *fo;        /* output file */
    LowcFE      fesim;      /* PLC simulation class */
    readplcmask mask;       /* error pattern file reader */
    short       s[FRAMESZ]; /* i/o buffer */

    argc--; argv++;
    if (argc != 3)
        error("Usage: g711iplc plcpattern speechin speechout");
    readplcmask_open(&mask, argv[0]);
    if ((fi = fopen(argv[1], "rb")) == NULL)
        error("Can't open input file: %s", argv[1]);
    if ((fo = fopen(argv[2], "wb")) == NULL)
        error("Can't open output file: %s", argv[2]);
    while (fread(s, sizeof(short), FRAMESZ, fi) == FRAMESZ) {
```

```
        if (readplcmask_erased(&mask))
            fesim.dofe(s);      /* lost frame */
        else
            fesim.addtohistory(s);  /* received frame */
        fwrite(s, sizeof(short), FRAMESZ, fo);
    }
    fclose(fo);
    fclose(fi);
    readplcmask_close(&mask);
    return 0;
}
```

### 8.3.4. Loss Pattern Conversion Utility

The PLC directory includes a tool, `asc2g192`, for converting ASCII loss pattern files containing sequences of 0s and 1s into G.192 format pattern files. In ASCII loss pattern files, a "1" represents a lost frame and a "0" represents a received frame. For example, to create a 10% uniform loss pattern with each loss being 10ms, use a text editor to create a text file called `fe10.txt`:

```
0000000001
```

Then, convert it to the G.192 format for use by the g711iplc program with the following command:

```
asc2g192 fe10.txt fe10.g192
```

Similarly, to create a 10% uniform loss pattern with each loss being 20ms (2 frames for each loss), create the text file `fe10_2.txt`:

```
00000000000000000011
```

Then convert it to the G.192 format with:

```
asc2g192 fe10_2.txt fe10_2.g192
```

The `asc2g192` conversion program ignores new lines and carriage returns in the input file so the patterns can span multiple lines.

## 9. G.726: The ITU-T ADPCM algorithm at 40, 32, 24, and 16 kbit/s

In 1982, a group was established by the then CCITT Study Group XVIII to study the standardization of a speech coding technique that could reduce the 64 kbit/s rate used in digital links, as per Recommendation ITU-T G.711 (see related Chapter), by half while maintaining the same voice quality.

After considering contributions received from several organizations, there was a general feeling that the ADPCM *(Adaptive Differential Pulse Code Modulation)* technique could provide a good quality coder. This process of finalizing an algorithm took 18 months of development and objective and subjective testings, to culminate in an ITU Recommendation, published in October, 1984, and available in the Red Book series as Recommendation ITU-T G.721.

Meanwhile, problems were found with the G.721 algorithm of 1984 regarding voice-band data signals modulated using the Frequency Shift Keying (FSK) technique, and changes had to be done to the algorithm. These changes were approved in 1986 and published in the next series of Recommendations of the CCITT, the Blue Book series, superseeding the Red Book version of the G.721. This is why a note in the Blue Book G.721 warns the user that the bit stream of coded speech from this version is incompatible with the old one. Also in that Study Period (1985-1988), a need for other rates was identified, and a new Recommendation, ITU-T G.723, was approved to extend the bitrate to 24 and 40 kbit/s.

In the Study Period of 1989—1992, these two Recommendations have been joined into a single one, keeping full compatibility with the former ones, and adding a lower rate of 16 kbit/s. This

new Recommendation was named ITU-T G.726, and the former ITU-T G.721 and G.723 have been replaced.

The current version of the STL includes a G.726 implementation. In the section to follow, the operation of the G.726 algorithm is described only for the 32 kbit/s bit rate. A complete description of the G.726 algorithm can be found in [21]. Other analyses of the algorithm, besides some based on the Red Book version, can be found in several studies [89], [97], [26].

Despite the change in numbering, the ITU-T ADPCM algorithm for speech coding at 32 kbit/s, the term "G.721 algorithm" has been retained for simplicity of the text, although a more formal reference should be *"G.726 at 32 kbit/s"*.

## 9.1. Description of the 32 kbit/s ADPCM

The basic idea behind the G.721 coder is to code into 4-bit samples the input speech-band signals, sampled at 8 kHz and represented by the 8-bit of G.711 A or $\mu$ law samples. The decoder just implements the reverse procedure.

The ADPCM algorithm of the G.721 exploits the predictability of the speech signals. Therefore, an adaptive predictor is used to compute the difference signal $d(k)$ (based on the expanded input log-pcm sample $s(k)$), which is then quantized by an adaptive quantizer using 4 bits. These bits are sent to the decoder and then fed into an inverse quantizer. The difference signal is used to calculate the reconstructed signal, $s_r(k)$, which is compressed (A- or $\mu$-law) and output from the decoder ($s_d(k)$).

From this description, one could ask the following:
– If only the quantized signal is transmitted, how can the decoder reconstruct the signal?
– How can one assure estability of the predictor?
– Will this bitrate reduction degrade the voice quality?

These and others have already been considered in the design of the G.721, and many blocks of the algorithm are made to assure a good behaviour. For example, one possibility in this backward approach for adaptation is to have encoder and decoder starting from the same point, which is accomplished by reseting key variables to a known state (useful for implementation verification). Leak factors have been introduced to ensure that the algorithm will always converge, independently of the initial state. To avoid instabilities, some parameters had their range limited. To provide some insight in the building blocks of the G.721 algorithm, a short description of each of them is given [21], [97].

### 9.1.1. PCM format conversion

The input signal $s(k)$, in either A- or $\mu$-law format, must be converted into linear samples. This expansion is accomplished using the same algorithm in G.711 [20], but converting from signed magnitude to 14-bit two's complement samples.

### 9.1.2. Difference Signal Computation

This block simply calculates the difference between the (expanded) input signal and the estimated signal:

$$d(k) = s_l(k) - s_e(k)$$

### 9.1.3. Adaptive Quantizer

A 15-level, non-uniform adaptive quantizer is used to quantize the difference signal. Before the quantization, this signal is converted to a logarithmic representation[7] and scaled by a factor ($y(k)$), that is computed in the scale factor adaptation block (see below).

The output of this block is $I(k)$, and it is used twice; first, is the ADPCM coded (*quantized*) sample; second, is the input to the backward part of the G.721 algorithm, to provide information for quantization of the next samples. One relevant point to notice here is that the backward adaptation is done using the quantized sample. If one starts the decoder from this very point, one will find identical behaviour. That is why only the quantized samples are needed in the decoder (i.e., no side information).

### 9.1.4. Inverse Adaptive Quantizer

The inverse adaptive quantizer takes the signal $I(k)$ and converts it back to the linear domain, generating a quantized version of the difference signal, $d_q(k)$. This is the input to the adaptive predictor, such that the estimated signal is based on a quantized version of the difference signal, instead of on the unquantized (original) one.

### 9.1.5. Quantizer Scale Factor Adaptation

This block computes $y(k)$, the factor used in the adaptive quantizer and inverse quantizer for domain conversion. As input, this block needs $I(k)$, but also $a_l(k)$, the adaptation speed control parameter. The reason for the latter is that the scaling algorithm has two modes (*bimodal adaptation*), one fast, another slow. This has been done to accomodate signals that in nature produce difference signals with large fluctuations (e.g. speech) and small fluctuations (e.g. tones and voice-band data), respectively.

This block computes two scale factor (fast, $y_u(k)$, and slow, $y_l(k)$) based on $I(k)$, which combined using $a_l(k)$ produce $y(k)$.

### 9.1.6. Adaptation Speed Control

This block evaluates the parameter $a_l(k)$, which can be seen as a *proportion* of the speed (fast or slow) of the input signal, and is in the range [0,1]. If 0, the data are considered to be *slowly* varying; if 1, they are considered to be *fast* varying.

To accomplish this, two measures of the average magnitude of $I(k)$ are computed ($d_{ms}(k)$ and $d_{ml}(k)$). These, in conjunction with delayed tone detect and transition detect flags ($t_d(k)$ and $t_r(k)$, calculated in the Tone Transition and Detector block), are used to evaluate $a_p(k)$, whose delayed version ($a_p(k-1)$) is used in the definition of $a_l(k)$, limiting the range to [0,1][8].

An analysis of $a_p(k)$ gives insight on the nature of the signal: if around the value of 2, this means that the average magnitude of $I(k)$ is changing, or that a tone has been detected, or that it is idle channel noise; on the other side, if near 0, the average magnitude of $I(k)$ remains relatively constant.

### 9.1.7. Adaptive Predictor and Reconstructed Signal Calculator

The adaptive predictor has as its main function to compute the signal estimate based on the quantized difference signal, $d_q(k)$. It has 6 zeroes and 2 poles, structure that covers well the kind of input signals

---

[7] Remember that to multiply samples in the linear domain one may add in the logarithmic one. Using efficient log and exponentiation algorithms (as done here), this turns out to be very advantageous.

[8] This limitation delays the start of a fast to slow transition until the average magnitude of $I(k)$ remains constant for some time; acting so, premature transitions for pulsed input signals, such as switched carrier voiceband data, are avoided.

expected for the algorithm. With these coefficients, and past values of $d_q(k)$ and $s_e(k)$, the updated value for the signal estimate $s_e(k)$ is computed.

The two sets of coefficients (one for the pole section, $a_i(k), i = 1..2$, other for the zero section, $b_i(k), i = 1..6$) are updated using a simplified gradient algorithm. At this point, since a situation in which the poles cause instability may arise, the two pole coefficients $a_i$ have their ranges limited. In addition, if a transition from partial band signal is detected (signaled by $t_r(k)$), the predictor is reset (all coefficients are set to 0), remaining disabled until $t_r$ comes back to zero[9].

The reconstructed signal $s_r(k)$ is calculated using the signal estimate $s_e(k)$ and the quantized difference signal $d_q(k)$.

#### 9.1.8. Tone Transition and Detector

This block is one of the changes from the Red Book version. It was added to improve algorithm performance for signals originating from FSK modems operating in the character mode. First, it checks if the signal has partial band (e.g., a tone) by looking at the predictor coefficient $a_2(k)$, that defines the signal $t_d(k)$. Second, a transition from partial band signal indicator $t_r(k)$ is set, such that predictor coefficients can be set to 0 and the quantizer can be forced into the fast mode of operation.

#### 9.1.9. Output PCM Format Conversion

This block is unique to the decoder. Its sole function is to compress the reconstructed signal $s_r(k)$, which is in linear PCM format, using A or $\mu$ law, and is a complement of the PCM format conversion block.

#### 9.1.10. Synchronous Coding Adjustment

This block is also unique to the decoder. It has been devised in order to prevent cumulative distortions occuring on synchronous tandem codings (ADPCM—PCM—ADPCM, etc., in purely digital connections, i.e., with no intermediate analog conversions), provided that:

– the transmission of the ADPCM and the intermediate PCM are error-free, and
– the ADPCM and the intermediate PCM are not disturbed by digital signal processing devices.

#### 9.1.11. Extension for linear input and output signals

An extension of the G.726 algorithm was carried out in 1994 to include, as an option, linear input and output signals. The specification for such linear interface is given in its Annex A [6].

This extension bypasses the PCM format conversion block for linear input signals, and both the Output PCM Format Conversion and the Synchronous Coding Adjustment blocks, for linear output signals. These linear versions of the input and output signals are 14-bit, 2's complement samples.

The effect of removing the PCM encoding and decoding is to decrease the coding degradation by 0.6 to 1 qdu, depending on the network configuration considered (presence or absence of a G.712 filtering).

Currently, this extension has not been incorporated in the STL.

### 9.2. ITU-T STL G.726 Implementation

The STL implementation of the G.726 algorithm can be found in module `g726.c`, with prototypes in `g726.h`.

Originally in Fortran (VAX Fortran-77), the source was translated by means of the public-domain code converter *f2c* [27]. This explain why the code makes extensive use of passage of parameters by

9 Note that when this happens, the quantizer is forced into the fast mode of adaptation.

reference, rather than by value, and why many functions, that could be implemented as macros (using the C pre-processor directive `#define`), are routines, and as well as all routines return `void`.

The problem of storing the state variables was solved by defining a structure containing all the necessary variables, defining a new type called `G726_state`. By means of this approach, several streams may be processed in parallel, provided that one structure is assigned (and that one call to the encoding/decoding routines is done) for each data stream (this can be advantageous for machines with support for parallel processing). The G726 state variable structure has the following fields (all are `short`, except *ylp*, which is `long`):

| | |
|---|---|
| *sr0* | Reconstructed signal with delay 0 |
| *sr1* | Reconstructed signal with delay 1 |
| *a1r* | Delayed 2nd-order predictor coefficient 1 |
| *a2r* | Delayed 2nd-order predictor coefficient 2 |
| *b1r* | Delayed 6th-order predictor coefficient 1 |
| *b2r* | Delayed 6th-order predictor coefficient 2 |
| *b3r* | Delayed 6th-order predictor coefficient 3 |
| *b4r* | Delayed 6th-order predictor coefficient 4 |
| *b5r* | Delayed 6th-order predictor coefficient 5 |
| *b6r* | Delayed 6th-order predictor coefficient 6 |
| *dq0* | Quantized difference signal with delay 0 |
| *dq1* | Quantized difference signal with delay 1 |
| *dq2* | Quantized difference signal with delay 2 |
| *dq3* | Quantized difference signal with delay 3 |
| *dq4* | Quantized difference signal with delay 4 |
| *dq5* | Quantized difference signal with delay 5 |
| *dmsp* | Short term average of the $F(I)$ sequence |
| *dmlp* | Long term average of the $F(I)$ sequence |
| *apr* | Triggered unlimited speed control parameter |
| *yup* | Fast quantizer scale factor |
| *tdr* | Triggered tone detector |
| *pk0* | Sign of dq+sez with delay 0 |
| *pk1* | Sign of dq+sez with delay 1 |
| *ylp* | Slow quantizer scale factor |

The encoding function is `G726_encode`, and the decoding function is `G726_decode`. There are 41 other routines that, grouped in individual calls inside the encoder and decoder, implement the algorithm. Therefore, none of these 41 routines are expected to be accessed by the user, and only the two main ones.

In the following part a summary of calls to both functions is found.

### 9.2.1. G726_encode

**Syntax:**

```
#include "g726.h"
void G726_encode (short _*inp_buf_, short _*out_buf_, long _smpno_, char
                  _*law_, short _rate_, short _reset_, G726_state _*state_)
```

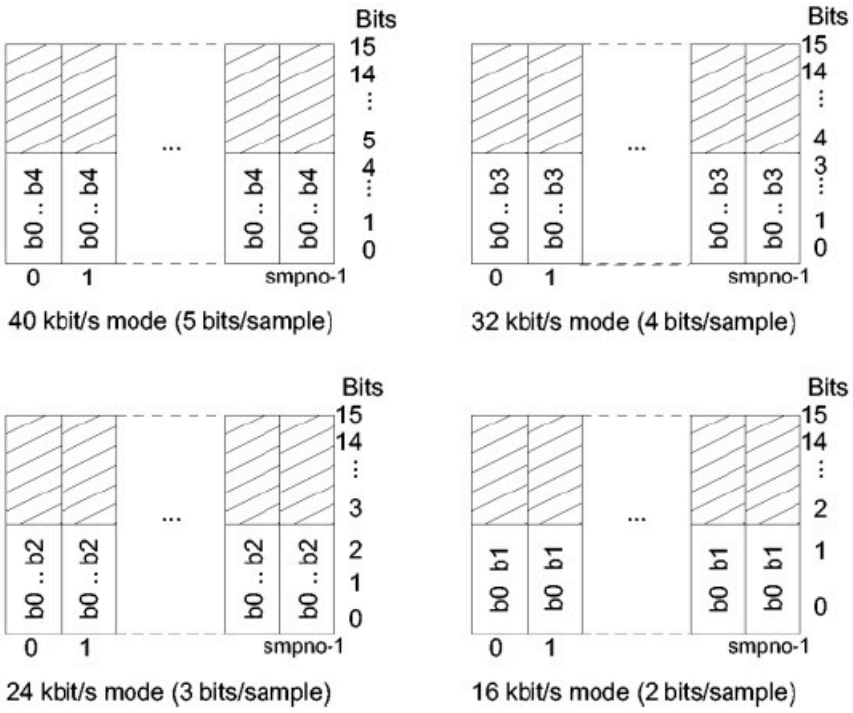



**Figure 3 — Packing of G.726-encoded signals (right-aligned, parallel format)**

**Prototype:** `g726.h`

**Description:**

Simulation of the ITU-T G.726 ADPCM encoder. Takes the A or $\mu$ law input array of shorts *inp_buf* (16 bit, right-justified, without sign extension) with *smpno* samples, and saves the encoded samples in the array of shorts *out_buf*, with the same number of samples and right-justified. An example of the sample packing for the G.726 encoded bitstream is shown in Figure 3.

The state variables are saved in the structure *state*, and the reset can be stablished by making *reset* equal to 1. The law is A if *law*==`'1'`, and mu law if *law*==`'0'`.

**Variables:**

| | |
|---|---|
| *inp_buf* | Is the input samples' buffer; each `short` sample shall contain right-justified 8-bit wide valid A or $\mu$ law samples. |
| *out_buf* | Is the output samples' buffer; each `short` sample will contain right-justified 2-, 3-, 4-, or 5-bit wide G.726 ADPCM samples, depending on the rate used. |
| *smpno* | Is the number of samples in inp_buf. |
| *law* | Is a char indicating if the law for the input samples is A (`'1'`) or $\mu$ (`'0'`). See note below. |
| *rate* | Is a short indicating the number of bits per sample to used by the algorithm: 5, 4, 3, or 2. |
| *reset* | Is the reset flag (see note below): |

– *1*: reset is to be applied in the variables;
– *0*: processing is carried out without setting state variables to the reset state.

Please note that this should normally be done only in the first call to the routine in processing a sample stream.

*state* The state variable structure; all the variables here are for internal use of the G. 726 algorithm, and should not be changed by the user. Fields of this structure are described above.

NOTE – Please note the difference between *reset* and *law*: *reset* must be either 1 (0x01) or 0 (0x00), not '1' (0x31) or '0' (0x30), while *law* is exactly the opposite.

**Return value:** None.

### 9.2.2. G726_decode

**Syntax:**

```
#include "g726.h"
void G726 decode (short _*inp_buf_, short _*out_buf_, long _smpno_, char
                  _*law_, short _rate_, short _reset_, G726_state _*state_)
```

**Prototype:** `g726.h`

**Description:**

Simulation of the ITU-T G.726 ADPCM decoder. Takes the ADPCM input array of shorts *inp_buf* (16 bit, right-justified, without sign extension) of length *smpno*, and saves the decoded samples (A or $\mu$ law) in the array of shorts *out_buf*, with the same number of samples and right-justified.

The state variables are saved in the structure *state*, and the reset can be stablished by making *reset* equal to 1. The law is A if *law*==`'1'`, and mu law if *law*==`'0'`.

**Variables:**

*inp_buf* Is the input samples' buffer; each `short` sample will contain right-justified 2-, 3-, 4-, or 5-bit wide G.726 ADPCM samples.

*out_buf* Is the output samples' buffer; each `short` sample shall contain right-justified 8-bit wide valid A or $\mu$ law samples.

*smpno* Is the number of samples in inp_buf.

*law* Is a char indicating if the law for the input samples is A (`'1'`) or $\mu$ (`'0'`). See note below.

*rate* Is a short indicating the number of bits per sample to used by the algorithm: 5, 4, 3, or 2.

*reset* Is the reset flag (see note below):
– *1*: reset is to be applied in the variables;
– *0*: processing done without setting state variables to reset state.

Please note that this should normally be done only in the first call to the routine in processing a sample stream.

*state* The state variable structure; all the variables here are for internal use of the G. 721 algorithm, and should not be changed by the user. Fields of this structure are described above.

NOTE – Please note the difference between *reset* and *law*: *reset* must be either 1 (0x01) or 0 (0x00), not '1' (0x31) or '0' (0x30), while *law* is exactly the opposite.

**Return value:** None.

### 9.3. Portability and compliance

Code testing has been done using the reset test sequences for 40, 32, 24, and 16 kbit/s provided in the G.726 test sequence diskettes (available from the ITU sales department). Other tests were also done with speech files for the 32 kbit/s mode, comparing with reference implementations, most noticeably the one from AT&T Bell Laboratories, which is the original implementation. Both test approaches generated 100% compatibility of this implementation with the G.726.[10]

The portability of the STL G.726 encoding function has been tested by feeding the routine with the reset test sequences of the G.726 test sequences diskettes (available from the ITU Secretariat). As inputs, a binary version of the files nrm.a, ovr.a, nrm.m, ovr.m have been used for the 4 bit rates; the output of `G726_encoder` was then compared with a binary version of the files rn*rr*fa.i, rv*rr*fa.i, rn*rr*fm.i, rv*rr*fm.i, *rr*=16,24,32,40, accordingly for each input sequence and rate. The encoding routine passed the test when no differences in the bit streams were found.

The portability test of the decoding function was carried out by feeding this routine with the pertinent test sequences of the G.726 Test Sequences Diskettes. As inputs, a binary version of the files rn*rr*fa.i, rv*rr*fa.i, rn*rr*fa.i, rv*rr*fa.i, rn*rr*fm.i, rv*rr*fm.i, rn*rr*fm.i, rv*rr*fm.i, and i*rr* (twice: one for A and another for $\mu$ law) have been used, *rr* being 16, 24, 32, and 40. The output of `G726_decoder` was then compared with a binary version of the files rn*rr*fa.o, rv*rr*fa.o, rn*rr*fx.o, rv*rr*fx.o, rn*rr*fm.o, rv*rr*fm.o, rn*rr*fc.o, rv*rr*fc.o, ri*rr*fa.o, ri*rr*fm.o (*rr* as above), respectively for each input sequences. All test vectors were properly processed.

These routines have been tested in VAX/VMS with VAX-C and GNU-C, in the PC with Borland C v3.0 (16-bit mode) and GNU-C (32-bit mode). In the Unix environment for Sun cc, acc, and gcc, and in HP for gcc.

### 9.4. Example code

#### 9.4.1. Description of the demonstration programs

Two programs are provided as demonstration programs for the G.726 module, `g726demo.c` and vbr-g726.c.

Program `g726demo.c` accepts input files in either 16-bit, right-justified A- or $\mu$-law format (as generated by `g711demo.c`) and encodes and/or decodes using one of the G.726 bit rates (16, 24, 32, or 40 kbit/s). Linear PCM files are not accepted by the program. Three operations are possible: logarithmic in, logarithmic out (*lolo*) logarithmic in, ADPCM out (*load*), or ADPCM in, logarithmic out (*adlo*).

Program `vbr-g726.c` can perform the same functions as `g726demo.c`, however it is capable of two additional features. It can perform in variable bit rate mode, which is switched at user-specified frame sizes (i.e. number of samples), and it can operate from 16-bit linear PCM input files. In the latter case, A-law is used to compand the linear signal prior to G.726 encoding, since G.726 Annex A [6] is not yet implemented in the STL.

---

[10] The problem with the A-law 40 kbit/s test vector `ri40fa.o` present in the STL96 has been solved in the STL2000.

### 9.4.2. Simple example

The following C code gives an example of G.726 coding and decoding using as input speech previously encoded by either the A- or $\mu$-law functions available in the STL. The output samples will be encoded using the same law of the input signal.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "g726.h"

#define BLK_LEN 256

void main(argc, argv)
  int             argc;
  char           *argv[];
{
  G726_state      encoder_state, decoder_state;
  char            law[4];
  short           bitrate, reset;
  char            FileIn[180], FileOut[180];
  short           tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
  FILE           *Fi, *Fo;

  /* Get parameters for processing */
  GET_PAR_S(1, "_Law: ..... ", law);
  GET_PAR_I(2, "_Bit-rate: ..... ", bitrate);
  GET_PAR_S(2, "_Input File: ..... ", FileIn);
  GET_PAR_S(3, "_Output File: ..... ", FileOut);

  /* Opening input and output LOG-PCM files */
  Fi = fopen(FileIn, RB);
  Fo = fopen(FileOut, WB);

 /* File processing */
  reset = 1;                    /* set reset flag as YES */
  while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
  {
    /* Process input log PCM samples in blocks of length BLK_LEN */
    G726_encode(inp_buf, tmp_buf, BLK_LEN, law, bitrate, reset, &encoder_state);

    /* Process ADPCM samples in blocks of length BLK_LEN */
    G726_decode(tmp_buf, out_buf, BLK_LEN, law, bitrate, reset, &decoder_state);

    /* Write PCM output word */
    fwrite(out_buf, BLK_LEN, sizeof(short), Fo);

    if (reset)
      reset = 0;                /* set reset flag as NOMORE */
  }

  /* Close input and output files */
  fclose(Fi);
  fclose(Fo);
}
```

# 10. G.727: The ITU-T embedded ADPCM algorithm at 40, 32, 24, and 16 kbit/s

## 10.1. Description of the Embedded ADPCM

The G.727 algorithm is specified in Recommendation ITU-T G.727 [22] with the block diagram shown in Figure 4, and will not be further described here. Additional information can be found in [26], where a thorough comparison is made between different ADPCM schemes, including G.726 and G.727. Details on the linear interface for the G.727 algorithm are found in G.727 Annex A [7].

### 10.1.1. Extension for linear input and output signals

An extension of the G.727 algorithm was carried out in 1994 to include, as an option, linear input and output signals. The specification for such linear interface is given in its Annex A [7].

This extension bypasses the PCM format conversion block for linear input signals, and both the Output PCM Format Conversion and the Synchronous Coding Adjustment blocks, for linear output signals. These linear versions of the input and output signals are 14-bit, 2's complement samples.

The effect of removing the PCM encoding and decoding is to decrease the coding degradation by 0.6 to 1 qdu, depending on the network configuration considered (presence or absence of a G.712 filtering).

Currently, this extension has not been incorporated in the STL.

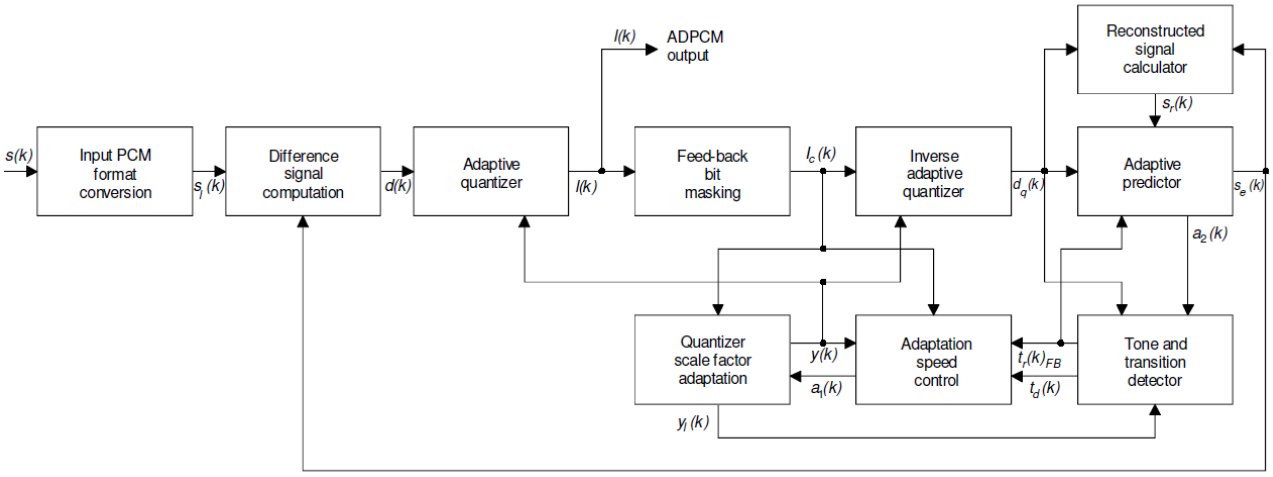



**Figure 4-a — Encoder**

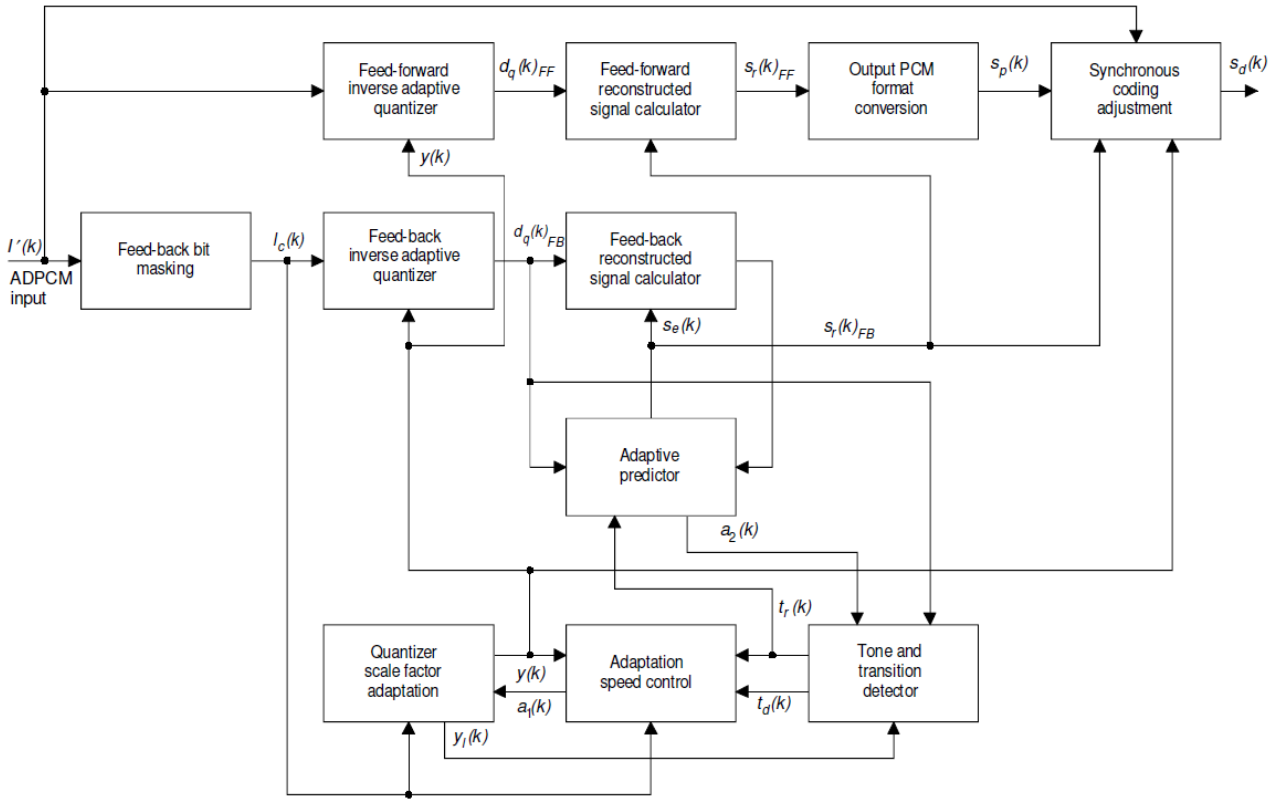



**Figure 4-b — Decoder**

**Figure 4 — G.727 encoder and decoder block diagrams**

## 10.2. ITU-T STL G.727 Implementation

The STL implementation of the G.727 algorithm can be found in module `g727.c`, with prototypes in `g727.h`.

The problem of storing the state variables was solved by defining a structure containing all the necessary variables, defining a new type called `G727_state`. As for other STL modules, the use of the state variable allows for parallel processing flows in the same executable program. The internal elements of the state variable `G727_state` should not be modified by the user, and are not described here.

The encoding function is `G727_encode`, and the decoding function is `G727_decode`. Additionally, initialization and reset of the state variable is performed by `g727_reset`. There are other internal routines which are not for access by the user, and hence are not described here. Their usage description is given below.

### 10.2.1. G727_reset

**Syntax:**

```
#include "g727.h"
void G727_reset (g727_state _*st_);
```

**Prototype:** `g727.h`

**Description:**

Reset ITU-T G.727 embedded ADPCM encoder or decoder state variable.

### 10.2.2. G727_encode

**Syntax:**

```
#include "g727.h"
void G727_encode (short _*src_, short _*dst_, short _smpno_, short _law_,
                  short _cbits_, short _ebits_, g727_state _*state_);
```

**Prototype:** `g727.h`

**Description:**

Simulation of the ITU-T G.727 embedded ADPCM encoder. Takes the A or $\mu$ law input array of shorts `src` (16 bit, right- justified, without sign extension) of length `smpno`, and saves the encoded samples in the array of shorts `dst`, with the same number of samples and right-justified. The ADPCM samples will have `cbits` core bits, and `ebits` enhancement bits.

The state variables are saved in the structure *state*, which should be initialized by `g727_reset()` before use. A-law is used if *law*==`'1'`, and $\mu$-law if *law*==`'0'`.

**Variables:**

| | |
|---|---|
| *src* | Is the input samples' buffer; each `short` sample shall contain right-justified 8-bit wide valid A or $\mu$ law samples. |
| *dst* | Buffer with right justified `short` ADPCM-encoded samples with cbits core bits and ebits enhancement bits. Unused MSbs are set to zero. |
| *smpno* | Is a `short` indicating the number of samples to encode. |
| *law* | Is a char indicating if the law for the input samples is A (`'1'`) or $\mu$ (`'0'`). |
| *cbits* | Number of core ADPCM bits. |
| *ebits* | Number of enhancement ADPCM bits. |
| *state* | The state variable structure; all the variables here are for internal use of the G.727 algorithm, and should not be changed by the user. |

**Return value:** None.

### 10.2.3. G727_decode

**Syntax:**

```
#include "g727.h"
void G727_decode (short _*src_, short _*dst_, short _smpno_, short _law_,
                  short _cbits_, short _ebits_, g727_state _*state_);
```

**Prototype:** `g727.h`

**Description:**

Simulation of the ITU-T G.727 embedded ADPCM decoder. Takes the ADPCM input array of shorts `src` (16 bit, right-justified, without sign extension) of length `smpno`, and saves the decoded samples (A or $\mu$ law) in the array of shorts `dst`, with the same number of samples and right-justified. The ADPCM samples must have `cbits` core bits, and `ebits` enhancement bits.

The state variables are saved in structure `st`, which should be initialized by `g727_reset()` before use. The law is A if *law*==`'1'`, and $\mu$ law if *law*==`'0'`.

**Variables:**

| | |
|---|---|
| *src* | Buffer with right justified `short` ADPCM-encoded samples with cbits core bits and ebits enhancement bits. Unused MSbs are zero. |

*dst* Is the input samples' buffer; each `short` sample shall contain right-justified 8-bit wide valid A or $\mu$ law samples.

*smpno* Is a `short` indicating the number of samples to encode.

*law* Is a char indicating if the law for the input samples is A (`'1'`) or $\mu$ (`'0'`).

*cbits* Number of core ADPCM bits.

*ebits* Number of enhancement ADPCM bits.

*state* The state variable structure; all the variables here are for internal use of the G.727 algorithm, and should not be changed by the user.

**Return value:** None.

### 10.3. Portability and compliance

Code testing has been done using the reset test sequences for 5, 4, 3, and 2 bits with the valid combination of core and enhancement bits. The reset test sequences can be acquired from the ITU Sales Department, and are not distributed with the STL. The testing procedure is implemented in the makefiles, which use a binary version of the test vectors. The implementation passed the compliance test when no differences were found between tested and reference test vectors. All test vectors were verified to be properly processed.

These routines have been tested in in MS-DOS with Turbo C++ v1.0 (16-bit mode) and GNU-C (go32 32-bit mode), and in Windows/32 with MS Visual C and CYGNUS/gcc. In the Unix environment, they have been tested for SunOs (cc, acc, and gcc), HP-UX (gcc), and Ultrix 4.0 (cc and gcc).

### 10.4. Example code

#### 10.4.1. Description of the demonstration program

One program is provided as demonstration program for the G.727 module, g727demo.c.

Program `g727demo.c` accepts input files in either 16-bit, right-justified A- or $\mu$-law format (as generated by `g711demo.c`) and encodes and/or decodes using the G.727 algorithm for the user-specified number of $N_c$ core bits and $N_e$ enhancement bits. The effective encoding bitrate will then be $16 \times (N_c + N_e)$ kbit/s. Linear PCM files are not accepted by the program, since G.727 Annex A [7] is not yet implemented in the STL. Three operations are possible: logarithmic in, logarithmic out (default) logarithmic in, ADPCM out (option *-enc*), or ADPCM in, logarithmic out (option *-dec*).

#### 10.4.2. Simple example

The following C code gives an example of G.727 coding and decoding using as input speech previously encoded by either the A- or $\mu$-law functions available in the STL. The output samples are encoded using the same law of the input signal.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "g727.h"

#define BLK_LEN 256

void main(argc, argv)
  int             argc;
  char           *argv[];
{
  G727_state      encoder_state, decoder_state;
  char            law;
```

```
  short           core, enh;
  char            FileIn[180], FileOut[180];
  short           tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
  FILE           *Fi, *Fo;

  /* Get parameters for processing */
  GET_PAR_C(1, "_Law: ..................... ", law);
  GET_PAR_I(2, "_Core bits: ................ ", core);
  GET_PAR_I(2, "_Enhancement bits: ......... ", enh);
  GET_PAR_S(2, "_Log-PCM Input File: ....... ", FileIn);
  GET_PAR_S(3, "_Log-PCM Output File: ...... ", FileOut);

  /* Opening input and output LOG-PCM files */
  Fi = fopen(FileIn, RB);
  Fo = fopen(FileOut, WB);

  /* Reset state variables */
  g727_reset(&encoder_state);
  g727_reset(&decoder_state);

  /* File processing */
  while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
  {
    /* Process input log PCM samples in blocks of length BLK_LEN */
    G727_encode(inp_buf, tmp_buf, BLK_LEN, law, core, enh, &encoder_state);

    /* Process ADPCM samples in blocks of length BLK_LEN */
    G727_decode(tmp_buf, out_buf, BLK_LEN, law, core, enh, &decoder_state);

    /* Write PCM output word */
    fwrite(out_buf, BLK_LEN, sizeof(short), Fo);
  }

  /* Close input and output files */
  fclose(Fi);
  fclose(Fo);
}
```

## 11. G.728: The ITU-T low-delay CELP algorithm at 16 kbit/s

NOTE – The following description is only applicable to the basic G.728 operating mode, 16 kbit/s, both in floating-point (i.e. G.728 main body [23]) and in fixed-point implementation (i.e. G.728 Annex G [8]). The packet loss concealment for the LD-CELP decoder (G.728 Annex I [9]) is also described. The other bitrate extensions (G.728 Annexes H and J) are not part of this description.

CCITT, the predecessor of ITU-T, started an effort in 1985 to define a successor to then advanced ADPCM coding at 32 kbit/s. Due to the ample spectrum of its potential applications, extremely demanding requirements were defined for its performance [28], which required quality at least the same as that of G.721 (later superseded by G.726 operating at 32 kbit/s) for one encoding, and a one-way delay under 5 ms (but preferably under 2 ms), to avoid the need of echo cancelers.

The process to identify the performance requirements and objectives (usually referred to as "Terms of Reference", or ToR) for the 16 kbit/s speech coder started in 1985 and lasted until about 1990. The process to identify suitable algorithms and their tests, after the creation of a group of experts in June 1988, started in March 1989, with two potential candidates: one from BNR (Canada) [29] and another from AT&T (USA) [30]. In 1989, BNR withdrew [31] its candidate, however another candidate appeared from the Consortium between a company called VoiceCraft (USA), University of California at Santa Barbara (USA) and Simon Fraser University (Canada), which would also soon

withdraw [32], [33], [34], [35]. This way, AT&T continued as the only candidate algorithm proponent in the CCITT standardization process.

Its algorithm, the LD-CELP *(Low-Delay Code-Excited Linear Prediction)*, underwent two testing phases (Phase 1 in 1989-1990 and Phase 2 in 1992) that were organized by volunteers from eight organizations [36]. After tests ensuring that it met all performance requirements, the LD-CELP algorithm was approved and published as ITU-T Rec. G.728 [23]. Its simplified block diagram is found in Figure 5.

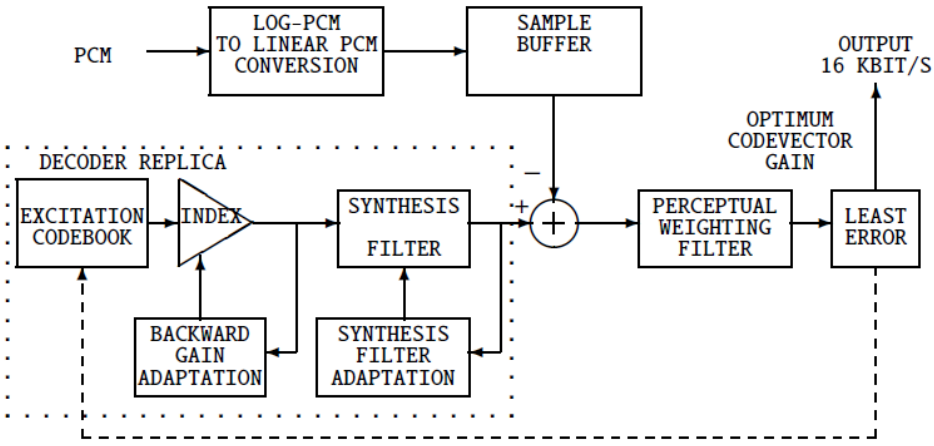



**Figure 5-a — Encoder block diagram**

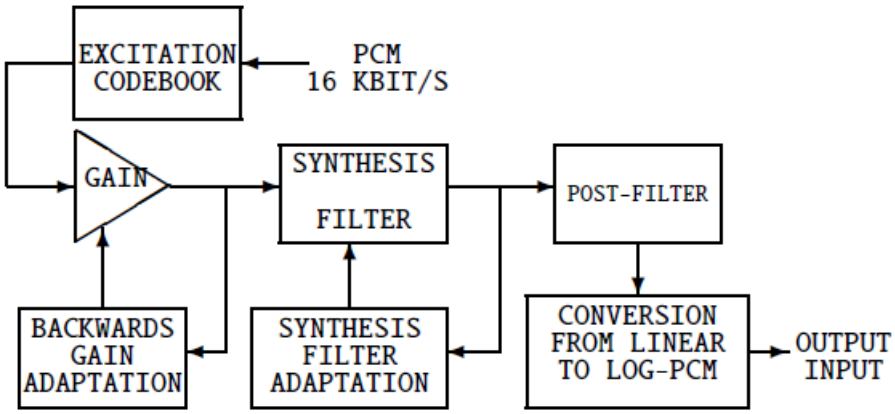



**Figure 5-b — Decoder block diagram**

**Figure 5 — Simplified LD-CELP block diagram**

### 11.1. General overview

#### 11.1.1. General characteristics

LD-CELP modified the structure present in the classical CELP-type coders [37] to meet the performance requirements for the algorithm, as well as to allow its implementation in real-time. However, it kept the basic structure of CELP coders, which is the the search in codebooks using an analysis-by-synthesis approach.

Traditionally, an analog signal is digitized into a 64 kbit/s bitstream with 8 bits per sample (ITU-T Rec. G.711), hence a 16 kbit/s bitstream would be equivalent to quantizing using 2 bits per sample. Since LD-CELP uses a basic frame of five samples, this implies that the vector quantization codebook has 1024 vectors (represented by 10-bit indices). Each codevector is the result of a 3-bit scalar gain and a shape vector of dimension 5 (represented by 7 bits). The scalar gain has one sign bit and two

magnitude bits, being symmetrical around 0. This allows doubling the range of amplitudes represented by the shape codebook without duplicating the search complexity for the optimal codevector.

The codebook was trained with the same perceptual weighting[11] implemented in the codec, what takes into account the adaptation effect in the predictor and of the excitation gain; this has a better performance than populating the codebook with Gaussian random numbers, which was the traditional approach for CELP coders [37].

After populating the codebook, indices are assigned to these values to organize the codevectors inside the codebook. To ensure a better SNR in the decoder for error-prone channels, a pseudo Gray coding was used for the indices. With this, a single error affecting the codebook index will shift it to a codeword very close to the original one, contrary to what would happen if the indices were randomly distributed.

### 11.1.2. Type of algorithm specification

The specification of an algorithm can be made in one of three methods [38]: bit-exact, bitstream, and algorithm-exact specification. The bit-exact specification implies that all variables and operations inside the algorithm have the bit length and representation precisely defined; this is the type of specification historically used for ITU speech and audio codecs, e.g. G.722, G.726, and G.729. Another approach consists in only specifying the bitstream format (or syntax) and the decoder; additionally, it is common practice to provide a reference encoder and decoder, which can usually be modified to reduce complexity or improve performance. This type of specification was used for regional cellular codec standards in Japan and USA, e.g. VSELP, for video codecs, as well as in the MPEG suite of audio and video codecs. Finally, algorithm-exact specification implies in describing in detail all parts of the algorithm, without however specifying variable length or precision. An algorithm specification made in one of these three ways can additionally be defined in terms of fixed-point arithmetic (only integer variables) or of floating-point.

The original G.728 algorithm is specified in terms of floating-point operations. Therefore, it is a non-bit-exact specification that describes precisely the operations for its implementation [38]. A fixed-point (albeit non-bit-exact) implementation of G.728 was subsequently designed to enable efficient implementation in digital circuit multiplication equipment, which maintained full interoperability with the original floating-point version[12]. An implication of this approach is the need to define procedures for more sophisticated implementation and interoperability verification than the ones used for bit-exact algorithms.

Historically, the choice for an initial implementation in floating-point was only allowed due to the availability at the time of commercially available floating-point DSPs. Even though they are common place today, back in early 1990's this was a breakthrough decision. This was further motivated by the fact that it was uncertain whether the original time schedule and performance requirements could be met if the group were to pursue a fixed-point (possibly bit-exact) implementation. In retrospect, the quality and deadlines targets could have been met, but the safe approach was to go for a floating-point specification.

---

[11] The term *perceptual* designates methods that explore the way the human hearing treats audio signals.

[12] The concept of interoperability of two implementations of an algorithm comes as a consequence of algorithm-exact specifications. They refer to the need of two different implementations to speak to each other, even though the devices are different. For example, a floating-point DSP and a fixed-point DSP, or two floating-point DSPs of different manufacturers in which number representation has different precision. The small operation differences can cause the accumulation of errors that, as time goes by, can lead the two implementations to diverge, making them fail to interoperate [38].

### 11.1.3. Delay

LD-CELP has an algorithmic delay[13] of 625 $\mu s$ (five-sample frame) and a (total) one-way delay of less than 2 ms. To obtain this delay while keeping the required quality, the designers adopted a backward predictor adaptation technique.

### 11.1.4. Backward adaptation

In the traditional CELP approach, the excitation, the LPC coefficients and gain are transmitted, while with LD-CELP only the excitation is transmitted. For this to work, the LPC analysis is made with the quantized version of the signal and the excitation gain is updated based on the gain embedded in the samples previously quantized.

In the LD-CELP encoder there are three backward-adaptive structures: the LPC synthesis filter, the perceptual weighting filter and the excitation gain unit. In addition to these three structures, the decoder also has a backward adaptive post-filter.

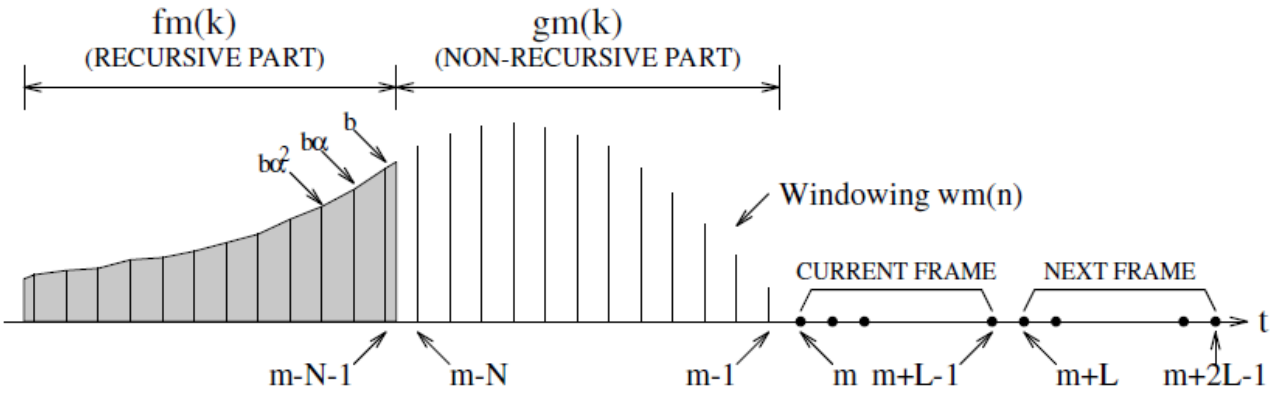



**Figure 6 — LD-CELP hybrid windowing**

### 11.1.5. Windowing used in the adaptation

In LD-CELP, except for the long-term post-filter in the decoder, all backward adaptive structures use LPC analysis. The LPC prediction coefficients are calculated using the auto-correlation method [39]. In this method, windowing is necessary to increase the prediction gain.

Normally, a Hamming window is used, but this is not appropriate for LD-CELP. The LD-CELP analysis block has only 5 samples, compared to the conventional 160 to 256 samples normally used. Hence, using a Hamming window would imply a significant overlap of windows across frames and a high computational complexity [41]. It was noticed that in the backward adaptation context of LD-CELP, the Barnwell recursive windowing technique [42] gave higher prediction gains than the Hamming windowing, in addition to a higher subjective quality for the processed speech. For this reason, the LD-CELP version tested in Phase 1 of subjective assessments used a modified adaptive windowing [43], [44]. This way, it was possible to obtain a better quality, a more balanced computational load and a lower complexity for frequent predictor updates [45].

However, aiming at a future fixed-point implementation with minimal changes compared to the floating-point version (since interoperability was a requirement), a hybrid windowing technique was

[13] The total delay introduced by a codec can be defined as the delay intrinsic to the encoder, e.g. the number of samples that it needs to buffer *before* it can start processing samples, plus the time needed by the hardware to process the samples or frame. For G.726 ADPCM, the algorithmic delay is zero, while the time needed to process the sample is non-zero but under 125 $\mu s$; this results in a total delay of one sample, or 125 $\mu s$. Obviously, the processing time is highly dependent on the implementation and cannot be determined *a priori*.

developed [41], [23], with results equivalent to those of the Barnwell version[14]. This was implemented in the version tested in Phase 2 of testings and today is present in G.728 (see Figure 6). With it, it was possible to keep the same quality (since the shape of both the hybrid and recursive windows is the same) while reducing the computational complexity between 20% and 30%. The objective was to reduce the complexity by mixing a recursive portion using the Barnwell technique with a non-recursive one, however maintaining the overall shape of the Barnwell window. The windowing was implemented such that the $m$ most recent samples are superimposed using the non-recursive portion of the window, and the samples previous to the $m$-th sample are considering in the recursive portion of the window. This window has a characteristic that non-recursive part has a sinusoidal shape and the recursive part has a decaying exponential (so as to lessen the influence of older past samples).

In terms of equations [23], [43], the hybrid window $w_m(k)$ is defined by:

$$w_m(k) = \begin{cases} f_m(k), & k < m-N \\ g_m(k), & m-N \le k < m \\ 0, & k \ge m \end{cases}$$

where

$$f_m(k) = b\alpha^{-[k-(m-N-1)]}$$

$$g_m(k) = -\sin[\,c(k-m)\,]$$

and $0 < \alpha < 1$, $0 < b < 1$ and $c$ are constants specific to each of the backward adaptive structures of the algorithm.

In the auto-correlation method, the i-th auto-correlation coefficient $R_m(i)$ of the input signal $s(n)$ is calculated as:

$$R_m(i) = \sum_{k=-\infty}^{\infty} s(k)w_m(k)s(k-i)w_m(k-i)$$

$$= \sum_{k=-\infty}^{m-1} s(k)w_m(k)s(k-i)w_m(k-i)$$

$$= r_m^{\backslash calR}(i) + r_m^{\backslash calN}(i)$$

Where $r_m^{\backslash calR}(i)$ and $r_m^{\backslash calN}(i)$ are respectively the recursive and non-recursive components of the i-th auto-correlation coefficient described by:

$$r_m^{\backslash calR}(i) = \sum_{k=-\infty}^{m-N-1} s(k)s(k-i)f_m(k)f_m(k-i)$$

and

$$r_m^{\backslash calN}(i) = \sum_{k=m-N}^{m-1} s(k)s(k-i)g_m(k)g_m(k-i)$$

Considering a predictor of order $M$ with an adaptation (update) cycle of $L$ samples, the i-th auto-correlation coefficient of the next adaptation cycle will be:

$$R_{m+L}(i) = r_{m+L}^{\backslash calR}(i) + r_{m+L}^{\backslash calN}(i)$$

with

[14] The prediction gain loss was below 0.1 dBm, a very small value.

$$r_{m+L}^{\backslash calR}(i) = \alpha^{2L} r_m^{\backslash calR}(i) + \sum_{k=m-N}^{m+L-N-1} s(k)s(k-i)f_{m+L}(k)f_{m+L}(k-i)$$

and

$$r_{m+L}^{\backslash calN}(i) = \sum_{k=m+L-N}^{m+L-1} s(k)s(k-i)g_{m+L}(k)g_{m+L}(k-i)$$

Therefore, it can be seen that $r_{m+L}^{\backslash calR}(i)$ is *recursively* calculated from its value $r_m^{\backslash calR}(i)$ from the previous adaptation cycle and that the auto-correlation coefficients always have a recursive and a non-recursive portion.

### 11.1.6. White noise correction

The use of the auto-correlation method for the calculation of the LPC coefficients has embedded in it the calculation of the inverse of the auto-correlation matrix, even though this is not explicitly made in methods such as the Levinson-Durbin method, where the prediction (and reflection) coefficients are iteratively calculated.

However, these are equivalent operations, and if the auto-correlation matrix is ill-conditioned[15], the LPC coefficients calculated by the Levinson-Durbin method can lead to an unstable filter. This effect will be even more pronounced when high-order LPC filters are used, as in this case. A simple technique to reduce the matrix ill-conditioning consists in adding white noise to the signal on which the prediction will be made, since this will fill the spectral valleys with noise and would reduce the dynamic range of the spectrum. From the computational point of view, however, it is interesting to adopt an equivalent procedure, designated in G.728 as "white noise correction".

Let's consider a voice signal $s(k)$ whose auto-correlation function is $R_s(i), i = 1..M$, where $M$ is the prediction filter order. In a synthetic notation,

$$s(k) \leftrightarrow R_s(i)$$

Similarly, we can associate to a white noise signal $n(k)$ an auto-correlation function $R_n(i)$, or:

$$n(k) \leftrightarrow R_n(i)$$

If we add signal and noise, the resulting auto-correlation function $R(i)$ is:

$$s(k) + G.n(k) \leftrightarrow R(i) = R_s(i) + G.R_n(i)$$

where G is a constant designated white noise correction factor. The SNR is given as a function of G:

$$SNR_{dB} = 20\log_{10}\left(\frac{1}{G}\right)$$

As $n(k)$ is a white noise, its auto-correlation function is given by:

$$R_n(i) = \begin{cases} 1, & \text{if } i = 0 \\ 0, & otherwise \end{cases}$$

Consequently:

$$R(i) = \begin{cases} R_s(0) + G, & \text{if } i = 0 \\ R_s(i), & otherwise \end{cases}$$

---

15 The auto-correlation matrix can be ill-conditioned because the signal representation has a finite numeric precision.

This way, it can be seen that it is sufficient to add a certain value $G$ to the coefficient $R_s(0)$ to reduce the ill-conditioning of the auto-correlation matrix. The value of $G$ is defined by the level of noise that one wants to "add" to the signal.

In LD-CELP, this technique is used to, without increasing the computational complexity, add noise at approximately 24 dB below the signal level ($G$=1/256) and lessen ill-condition of the auto-correlation matrix for the LPC synthesis filter and the perceptual weighting filter.

### 11.1.7. Bandwidth expansion

A common technique in CELP coders consists in attenuating the formant peaks through a bandwidth expansion of the LPC spectrum[16]:

$$a_i = \lambda^i \hat{a}_i$$

where $\hat{a}_i$ is the i-th coefficient calculated by the predictors and $\lambda$ is a constant that satisfies $\lambda < 1$ and $\lambda \approx 1$.

The effect of the expansion is to move the predictor poles inside the unity circle. Additionally, the impulse response of the model gets shorter, reducing the transmission error propagation inside the adaptation mechanism.

### 11.1.8. Input and output formats

Since LD-CELP is an algorithm also designed for use in the PSTN, the input and output signal representation follows the ITU-T Rec. G.711 format (either A or $\mu$ law)[17].

Since the internal algorithm operations are performed in linear PCM format, the first block in Figure 5-a consists in the expansion of the log-PCM samples into linear format; complimentary, the last block in Figure 5-b, after all decoding processing, does the compression of the linear samples into the selected log-PCM law.

## 11.2. Encoder structures

In the following, the LD-CELP encoder blocks are described, as illustrated in Figure 5-a.

### 11.2.1. LPC synthesis filter

The synthesis filter uses LPC coefficients and has order 50, and generates the quantized signal from the de-normalized excitation vector. The reason for such a high order is explained below.

A very common technique used in speech coders, in particular CELP coders, is the use of long-term prediction, or forward-adaptive pitch prediction. Conventional LPC prediction is normally based on a small number of LPC coefficients (usually, 10 to 12 in narrowband speech signal [39],pp.419—420), what does not allow to do a long term prediction that takes into account the pitch period. The use of pitch predictors is needed to make whiter the excitation signal obtained after a conventional LPC analysis, thus better exploring the signal predictability. However, since LD-CELP only transmits the excitation information, the pitch prediction would also need to be backward-adaptive, as done in [29], [34]. This technique, however, is very sensitive to transmission errors due to the high filter

16 Narrow bandwidth formants cause a chirping artifact that reduces the subjective quality of the signal. On the other hand, bandwidth expansion of the LPC synthesis filter can reduce performance with voice-band data, requiring therefore a compromise between voice and non-voice signals.

17 For the implementation verification and interoperability purposes (see G.728 Appendix I), however, the input signal format must be linear. Therefore, the $A/\mu$-law compression and expansion blocks must be bypassed.

orders, what makes the errors to propagate over a large number of samples. The use of artificial re-initializations could solve the problem, but since the algorithm was required to operate at high bit error rates, e.g., $10^{-2}$ equivalent to 160 errors per second, this would not work for the LD-CELP [43].

Together with the error propagation issue, it was noticed that the improvement introduced by the pitch predictor for female speakers was much higher than for male speakers[18] [43].

This way, with the problems of the backward-adaptive pitch predictor with transmission errors and the larger importance of the pitch prediction for female speech, it was decided to explore the predictability of the female speech using a high-order synthesis filter predictor, so that most of the pitch values for female speaker would be covered. It was then found empirically that an order of 50 for the LPC synthesis filter produced good results[19], replacing the "low order LPC predictor and pitch predictor" traditionally used in CELP coders.

### 11.2.2. Perceptual weighting filter

The general form of the perceptual weighting filter is [46]:

$$W(z) = \frac{1 - Q(z/\gamma_1)}{1 - Q(z/\gamma_2)}, 0 < \gamma_2 < \gamma_1 \leq 1.$$

where:

$$Q(z/\gamma_j) = \sum_{i=1}^{M} \gamma_j^i q_i z^{-i},\ j = 1,2.$$

and $q_i$ are the quantized LPC coefficients and $M$ is the LPC predictor order.

Its function is to model the spectral envelope of the error signal, so that it becomes similar to the spectrum of the input voice signal, this way masking the distortion which, without this weighting, could be perceived by the user. By using the weighted error signal to select the codevector, this codevector will be the one that, on the decoder, will produce the lowest quantization noise perceived by the user, increasing in the decoder the subjective quality [47].

For CELP coders, $(\gamma_1,\gamma_2 = (1.0,0.8)$ are normally used. In LD-CELP Phase 1 and 2, $(\gamma_1,\gamma_2) = (0.9,0.4)$ and $(\gamma_1,\gamma_2) = (0.9,0.6)$ were used, respectively, what resulted in a lower perceived noise level. Parameter $\gamma_2$ was changed to allow the codec to meet the three transcoding requirement, condition for which the codec had failed the performance requirement in Phase 1 [48].

The predictor order $M$ was set here to 10 to avoid artifacts[20] that appeared with the use of higher order filters (e.g. 50, as in the synthesis filter). To compensate for the low order, instead of using a sub-set

---

18 This is explained by two factors. First, since a male pitch period is much longer than for female talkers, the prediction gain is larger for the latter case. Second, backward-adaptive predictors use the *quantized* error signal as input, instead of the original error signal. The resulting quantization noise further reduces the correlation within a pitch period for male voices. Consequently, the prediction gain is even lower for male speech.

19 Additionally, the backward-adaptive LPC predictor of order 50 showed to be more resilient to transmission errors than the backward pitch predictor.

20 *Artifacts* denote here instabilities with quasi-periodic signals with long duration (more than 2 to 3 seconds), as happens with artificial voice signals (ITU-T Rec. P.50), some types of musical passages, e.g. sustained violin sound, or sustained vocalic sounds, e.g., /a/. These instabilities are reproduced as a change in the sound timbre this reducing the subjective quality.

of the main predictor coefficients (which in principle could be the same computational structure), an independent predictor is applied on the input signal (noting that the main predictor operates on the quantized samples). This is justified by two facts: since the perceptual weighting is not necessary on the decoder, nothing obliges the prediction to be made on the quantized signal; and (more importantly) the use of the original signal allows the more precise calculation of the spectral envelope.

### 11.2.3. Search of optimal excitation codevector

For each sample vector $s(n)$, a search is made to find which of the 1024 codevectors generates the lowest (perceptually weighted) average quadratic error. The selected codevector has its index transmitted to the decoder and is fed back to the adaptive part of the encoder (a replica of the decoder inside the encoder), to be used as excitation for the synthesis filter. An efficient search algorithm was selected for the LD-CELP algorithm [49], [50], which is described in detail in the Recommendation [23].

### 11.2.4. Denormalizing quantized excitation

To increase coding efficiency, the 1024 codevectors in the codebook represent typical excitation (obtained by training of this codebook from a large corpus of speech signals), whose amplitude has been normalized. Therefore the excitation vector to be used in the synthesis filter needs to be denormalized, which is done in the excitation denormalization block. The adaptation mechanism for this block is described below.

### 11.2.5. Adaptation of excitation gain

The optimal index transmitted to the decoder represents the normalized value of the excitation vector found for the input signal. Therefore, for the signal reconstruction (both in encoder and decoder), it is necessary to calculate the excitation gain value to be used for the denormalization of the excitation gain. This can be done using a fixed value pre-determined using a long-term statistical analysis of the possible gain values (similar to the fixed coefficient predictors and quantizers) of a large corpus of speech signals. A more optimal approach is to use an adaptive calculation of the gain. In LD-CELP, the calculation of the gain uses a mixed technique, which uses a fixed offset of 32 dB for the gain (empirically obtained), which is associated to a 10-th order LPC predictor using the auto-correlation method and hybrid windowing. This predictor adapts the gain around the fixed offset, in order to increase its precision.

The denormalized error signal $e(n)$ can be expressed in terms of the excitation gain $\sigma(n)$ and the normalized error $y(n)$ by:

$$e(n) = \sigma(n)\, y(n)$$

If $\sigma_y^2(n)$ and $\sigma_e^2(n)$ are respectively the mean square values of $y(n)$ and $e(n)$, then:

$$\log\sigma_e(n) = \log\sigma(n) + \log\sigma_y(n)$$

We can predict the excitation gain $\sigma(n)$ from the past values of the denormalized gain $\sigma_e$ using the predictor:

$$\log\sigma(n) = \sum_{i}^{P} p_i \log\sigma_e(n-i)$$

Here, the predictor order is $P = 10$.

It should be noted that this predictor, when combined with the last two equations, can be seen as a predictor with $P$ poles and $P$ zeros that uses $\log\sigma_y(n-i)$ as input:

$$\log\sigma(n) = \sum_{i}^{P} p_i \log\sigma(n-i) + \sum_{i}^{P} p_i \log\sigma_y(n-i)$$

The gain adaptation is backward-adaptive, as in other parts of LD-CELP. The use of the auto-correlation method for calculating the coefficients $p_i$ guarantees the stability of the filters above, what implies a filter response that falls asymptotically to zero. This limits the propagation of transmission errors, indicating a certain robustness of this block to transmission errors. This robustness brings the poles and zeros closer together, shortens the predictor impulse response and further limits the propagation of errors within the algorithm.

### 11.2.6. Adaptation of perceptual adaptation filter

To adapt the coefficients of the perceptual weighting filter the auto-correlation method is applied over the buffered (unquantized) input signal using a hybrid window, white-noise correction, and the Levinson-Durbin algorithm. The LPC coefficients are then multiplied by the factors $\gamma_1$ and $\gamma_2$, which define the degree of perceptual weighting.

### 11.2.7. Adaptation of LPC synthesis filter

The LPC predictor coefficients for the synthesis filter are adapted using the quantized input signal. Similar to the perceptual weighting filter, hybrid windowing, white noise correction and Levinson-Durbin algorithm are used (albeit with different parameters for the hybrid window and a larger order of 50 for the LP filter). After calculation of the LPC coefficients, these pass through the spectral widening and then are input to the synthesis filter.

## 11.3. Decoder structures

The decoder structure is found in Figure 5-b. The blocks that are identical to ones in the encoder will not be described again. The few blocks unique to the decoder are described in the following.

### 11.3.1. Post-filter and its adaptation

A post-filter [52] consists of a time-variant filter put at the output of a decoder with the intention to improve the subjective signal quality.

Post-filtering, a technique very common in CELP coders, was not used in the first phase of testing for two reasons. The distortion introduced by post-filtering accumulates with multiple transcodings (tandem), what can compromise speech quality[21]. Besides that, post-filtering introduce phase distortions that can make it difficult to decode properly voice-band data that relies on phase to carry information (e.g. DPSK) [53], [45].

Since the Phase 1 algorithm did not pass the three-tandem connection condition, it was modified with introduction of post-filtering for Phase 2. However, the use of traditional post-filtering would imply unacceptable distortions for the reasons mentioned above. Then instead of optimizing the post-filter for each transcoding, it was optimized for three transcodings, thus implying a smaller amount of post-filtering for each tandem step [43]. This allowed the overall distortion added by the post-filtering to remain within acceptable levels, while the subjective quality for three transcodings saw a significant improvement (a 26% increase in the MOS scores). Even for a single transcoding there was a good improvement in the subjective quality. During the Phase 2 tests, the change showed to be a good compromise, increasing the subjective quality while maintaining a good performance for voice-band data.

---

[21] The main distortion generated by post-filtering is the reduction of the bandwidth of the spectral peaks. Its accumulation causes severe distortion in the decoded speech.

The LD-CELP post-filter consists of three blocks: a long-term post-filter, a short-term post-filter and an automatic gain control (AGC).

The long-term post-filter (also called pitch post-filter), is basically a comb filter with a fundamental frequency being the reciprocal of the pitch period. This filter is described by the following equation:

$$H_l(z) = g_l(1 + bz^{-p})$$

where $p$ is the pitch period obtained by a pitch predictor and $g_l$ and $b$ are coefficients adapted from 240 buffered excitation samples. In LD-CELP, $p$ varies between 20 and 140 (57 to 400 Hz).

This way, when this post-filtering is applied to the synthesized signal, the spectrum regions around the multiples of the fundamental frequency are emphasized. This emphasizes pitch perception, thus resulting in a higher subjective quality.

The short-term post-filter, on the other hand, is used to reduce the audible coding noise by emphasizing the peaks and attenuating the valleys in the input signal spectrum. This is justified by the fact that perceptually the regions around LPC spectrum peaks, i.e. the formant regions, are more important than the regions around the valleys.

The short-term post-filter is implemented as a low-pass filter with the same number of poles and zeroes which cascades a first-order high-pass filter. The high-pass filter serves to compensate the muffling effect of the first low-pass filter. The general form of the short-term post-filter is given by:

$$H_s(z) = \frac{1 - A(z/\tilde{\gamma})}{1 - A(z/\tilde{\gamma})}(1 + \mu z^{-1}), \text{ with } 0 < \tilde{\gamma}_1 < \tilde{\gamma}_2 \leq 1$$

where:

$$A(z/\tilde{\gamma}) = \sum_{i=1}^{M} \tilde{\gamma}_j^i a_i z^{-i}, \text{ for } j = 1,2.$$

$$\mu = \tilde{\gamma}_3 k_1$$

and $\tilde{\gamma}_1$, $\tilde{\gamma}_2$ and $\tilde{\gamma}_3$ are constants, $a_i$ are the LPC coefficients, $M$ is the predictor order and $k_1$ is the first reflection coefficient.

It should be noted that the high-pass filter is implemented with an adaptive coefficient, the reflection coefficient $k_1$. Therefore, the filter characteristics might not always be that of a high-pass. Long-term speech statistics measurements at the time of development of the algorithm using about 10 minutes of speech showed that $k_1 < 0$ for major part of time (around 92%) (see Figure 7), thus guaranteeing that filter is a high-pass for the most part of the of the input signals.

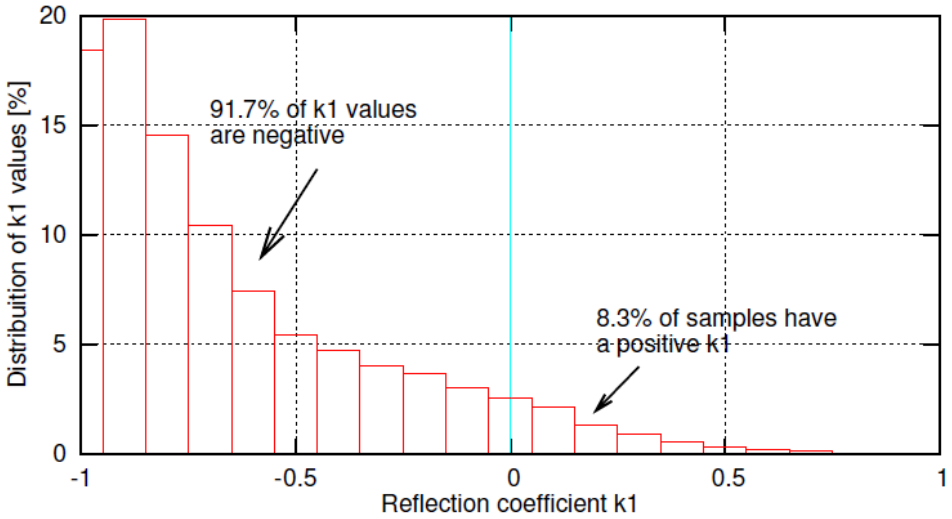



**Figure 7 — Histogram of $k_1$ values for 10 minutes of speech**

In LD-CELP, the filter order of $A(z)$ is 10 and the $a_i$ coefficients, as well as the reflection coefficient $k_1$, are obtained during the adaptation of the decoder synthesis filter. Constants $(\tilde{\gamma}_1,\tilde{\gamma}_2,\tilde{\gamma}_3)$ were empirically determined as (0.65,0.75,0.15).

Finally, the AGC is responsible for making that the power of the decoded signal after post-filtering be approximately the same as before post-filtering. The AGC is calculated by the ratio of the average signal amplitudes before and after post-filtering. This avoids too much attenuation or clipping.

### 11.4. ITU-T STL G.728 Implementation

This implementation of the G.728 algorithm is composed of source files in several directories. The floating-point version of the algorithm can be found in the directory `g728/g728float`. The fixed-point version is in `g728/g728fixed`. A third directory, `g728/testvector`, is designed to hold the test vectors for both versions of the algorithm. The contents of the testvector directory are not part of the STL, but can be obtained from G.728 Appendix I on the ITU-T web site. Similar data structures and interface functions are defined for both the floating-point and fixed-point G.728 implementations. The floating-point interface is discussed first.

#### 11.4.1. Floating-point G.728

The source code for the floating-point version of G.728 resides in the directory `g728/g728float`. All public interface function declarations and data structures needed to call the coder can be found in the header file `g728.h`. The package includes a demonstration program, `g728.c`, that shows how to call the encoder, decoder, and decoder with packet loss concealment. The demonstration program can run the coder on the G.728 test vectors.

The floating-point version of the coder can be compiled to use either double precision or single precision floating-point arithmetic. If compiled with `-DUSEDOUBLES`, `g728.h` defines the `Float` typedef as a C `double`. If compiled with `-DUSEFLOATS`, Float is a C `float`. `Float` is declared in `g728.h` as:

```
#ifdef USEDOUBLES
typedef double Float;
#endif
#ifdef USEFLOATS
typedef float  Float;
#endif
```

Single precision runs faster. Double precision runs slower but is more likely to give results that are bit-exact across different machines. When compiling for use with the test vectors double precision arithmetic should be used.

The include file, `g728.h` also defines a `Short` type:

```
typedef short  Short;
```

to hold signed 16-bit integers.

### 11.4.2. G.728 Floating-point Encoder

`g728.h` defines a C structure, `G728EncData`, to hold the encoder state variables. Its contents are described in the G.728 standard. Here we emphasize how to use it, rather than its internal members. Before calling the encoder `G728EncData` should be initialized with a call to `g728encinit`:

```
void g728encinit(
     G728EncData *e    /* encoder state, initialize */
);
```

Once initialized, frames of speech can be encoded with:

```
void g728encode(
    Short       *index,  /* output indices */
    Float       *input,  /* input speech */
    int          sz,     /* input size - must be multiple of IDIM */
    G728EncData *e       /* i/o encoder state */
);
```

The `input` speech should be an array of `Float`s in the range of -32768. to 32767. `sz` is the dimension of the `input` array, and must be a multiple of the coder frame size, IDIM. IDIM is defined to be 5 in g728.h. `index` is the output of the encoder. It is an array of codevector indices that are transmitted to the decoder. `index` has dimension sz / IDIM, as one codevector will be output for each input frame of IDIM speech samples. Only the least significant 10 bits of each index value are set. The 3 bits of gain index bits are in bits 0 to 2 (0 being the least significant bit), and the 7 bits of shape index are in bits 3 to 9. Bits 10 to 15 in each index value are unused and contain zeros. If an application desires to compactly transmit the codevectors for multiple frames, it will have to extract the lower 10 bits from each index word and pack the bits.

The floating-point G.728 software package includes utility functions for converting between `Short`s (16-bit integers) and `Float`s:

```
extern void g728_cpyr2i( /* convert Floats to Shorts,
                          * with saturation and rounding */
    Float *f,   /* input: float array */
    int   sz,   /* input: array size */
    Short *s    /* output: short array */
);
extern void g728_cpyi2r( /* convert Shorts to Floats */
    Short *s,   /* input: short array */
    int   sz,   /* input: array size */
    Float *f    /* output: float array
);
```

These are used by the test program, `g728.c`. A simple encoder loop that reads in frames of 16-bit linear PCM, encodes them with floating-point G.728, and outputs the index vectors is in the following code fragment:

```
#include "g728.h"

Short       ix;         /* output: index */
```

```
Short       s[IDIM];    /* input: 16-bit speech */
Float       f[IDIM];    /* converted to Float */
G728EncData e;          /* encode state */

g728encinit(&e);        /* initialize encoder state */
while (fread(s, sizeof(Short), IDIM, speechinf) == IDIM) {
        g728_cpyi2r(s, IDIM, f);        /* convert Short to Float */
        g728encode(&ix, f, IDIM, &e);   /* encode */
        fwrite(&ix, sizeof(Short), 1, indexf);  /* write to file */
}
```

We have omitted the details of opening the files. The code in the `(mode == M_ENC)` section of `g728.c` is a more complicated example that allows frame sizes that are multiples of IDIM, does more error checking, and also allows byte swapping of the input and output files.

### 11.4.3. G.728 Floating-point Decoder

`g728.h` defines a C structure, `G728DecData`, to hold the decoder state variables. Many of the decoder state variables replicate the state variables in the encoder. There are additional sections for variables associated with the post-filter and packet loss concealment. As with the encoder, the `G728DecData` must be initialized before it can be used:

```
void g728decinit(
    G728DecData *d    /* decoder state, initialize */
);
```

Once initialized, frames of speech can be decoded with:

```
void g728decode(
    Float       *speech, /* output: speech */
    Short       *index,  /* input: indices */
    int         sz,      /* input: size - must be multiple of IDIM */
    G728DecData *d       /* i/o: decoder state */
);
```

`speech` points to the output speech array. `sz` is the dimension of the output speech array. Like at the encoder, it must be a multiple of the coder frame size, IDIM. `index` is a pointer to the array of input codevectors. As with the encoder `index` should be dimension sz / IDIM. Five speech samples will be produced for each input codevector. The format of the index words is the same as in the encoder.

If the output speech is to be converted back to 16-bit linear PCM, the conversion routine should take care of rounding and saturation. The utility conversion routine, `g728_cpyr2i`, described above, takes care of this. Here is a code fragment that can be used to implement a floating-point decoder:

```
#include "g728.h"

Short       ix;         /* input: index */
Float       f[IDIM];    /* output: speech, Float */
Short       s[IDIM];    /* output: speech, converted to 16-bit int */
G728DecData d;          /* decode state */

g728decinit(&d);        /* initialize decoder state */
while (fread(&ix, sizeof(Short), 1, indexf) == 1) {
    g728decode(f, ix, IDIM, &d);     /* decode */
    g728_cpyr2i(f, IDIM, s);         /* convert to Short */
    fwrite(s, sizeof(Short), IDIM, speechoutf);
}
```

Again we have omitted the details of opening the files. The code in the `(mode == M_DEC)` section of `g728.c` is a more complicated example that allows frame sizes that are multiples of IDIM, does more error checking, and also allows byte swapping of the input and output arrays.

An additional interface routine, `g728setpostf`, allows the post-filter in decoder to be turned on or off. By default, the post-filter is on after initialization, but it can be disabled as required by several of the G.728 test vectors:

```
void g728setpostf(
    int          i,     /* in: 1 postfilter on, 0 off */
    G728DecData *d      /* i/o: state variables */
);
```

### 11.4.4. G.728 Floating-point Encoder/Decoder

The following program fragment shows how to run the encoder and decoder together. It combines the fragment from the encoder, with the fragment from the decoder, and uses the utility functions to convert between Floats and Shorts. The loop sequence is: read in a frame of speech, convert to Float, encode, decode, convert the output from Float to Short, and write the resulting speech to a file:

```
#include "g728.h"

Short       ix;         /* index */
Short       s[IDIM];    /* 16-bit speech vector*/
Float       f[IDIM];    /* Float speech vector */
G728EncData e;          /* encode state */
G728DecData d;          /* decode state */

g728encinit(&e);        /* initialize encoder state */
g728decinit(&d);        /* initialize decoder state */
while (fread(s, sizeof(Short), IDIM, speechinf) == IDIM) {
    g728_cpyi2r(s, IDIM, f);       /* convert Short to Float */
    g728encode(&ix, f, IDIM, &e);  /* encode */
    g728decode(f, ix, IDIM, &d);   /* decode */
    g728_cpyr2i(f, IDIM, s);       /* convert to Short */
    fwrite(s, sizeof(Short), IDIM, speechoutf);
}
```

The code that opens and closes the input and output files is omitted.

### 11.4.5. G.728 Floating-point Decoder with Packet Loss Concealment

The floating-point code supports the Packet Loss Concealment (PLC) algorithm in G.728 Annex I. This algorithm generates a synthetic speech output at the decoder and maintains the decoder internal state variables if the input codevectors are lost in transmission. This feature is implemented with two more functions. The first PLC function:

```
void g728setfesize(
    int    plc25msec,   /* input: PLC frame size, in 2.5msec */
    G728DecData *d      /* i/o: decoder state */
);
```

sets the PLC frame size, in increments of 2.5ms (20 samples). This function should be called after the G728DecData has been initialized with a call to g728decinit. If the PLC frame size argument, plc25msec, is set to 1, each lost frame will correspond to 20 samples. This variable: plc25msec should be in the range of 1 to 8, corresponding to a packet loss size of between 2.5 ms (20 samples) and 20 ms (160 samples). If the call to g728setfesize is omitted, the default PLC frame size is 10 ms (80 samples). The PLC algorithm uses state variables from the post-filter, so the post-filter must be activated when running the PLC algorithm.

The second PLC function:

```
void g728decfe(
    Float       *speech,  /* output: speech */
    int         sz,       /* input: size, in samples, PLC framesize */
```

```
    G728DecData *d        /* i/o: decoder state */
);
```

is similar to the g728decode function except it takes no input index vector. It serves two functions: it signals the decoder that the current frame is lost, and it generates the synthetic output signal for the frame. The sz argument is the number of samples in the PLC frame size. Since there are 20 speech samples in 2.5 ms of speech, the sz argument to g728decfe should be 20 times the plc25msec argument passed to g728setfesize. For frames that do not have losses, the standard g728decode function should be called.

A code fragment that shows how to call the PLC functions for a 10 ms PLC frame size (80 samples) is shown below:

```
#include "g728.h"

Short       ix[80/IDIM];/* input: index vectors */
Float       f[80];      /* output: Float array */
Short       s[80];      /* output: converted to 16-bit speech */
G728DecData d;          /* decode state */

g728decinit(&d);        /* initialize decoder state */
g728setfesize(4, &d);   /* 4 * 2.5msec frames = 10 msec */
while (fread(ix, sizeof(Short), 80/IDIM, indexf) == 80/IDIM) {
    if (ferasedin())
        g728decfe(f, 80, &d);           /* PLC decode */
    else
        g728decode(f, ix, 80, &d);     /* normal decode */
    g728_cpyr2i(f, 80, s);             /* convert to Short */
    fwrite(s, sizeof(Short), 80, speechoutf);
}
```

As in the previous section, the details on opening the files are omitted. Function ferasedin() returns 1 if the current frame should call the packet loss concealment code, and 0 if the standard decoding routine should be called. The code in the `(mode == M_PLC)` section of `g728.c` shows a more complicated version of the PLC loop that supports setting of the PLC frame size from the command line.

### 11.4.6. Fixed-point G.728

The source code for the fixed-point version of G.728 resides in the directory `g728/g728fixed`. All interface functions and data structures needed to call the coder can be found in the file `g728fp.h`. The interface mirrors the floating-point version of the codec, although the routines and data structures have different names, and the input and output speech arrays are `Short`s instead of `Float`s. The fixed-point package also includes a demonstration program, `g728fp.c`, that shows how to call the encoder, decoder, and decoder with packet loss concealment. The demonstration program can run the coder on the test vectors. Please note that the fixed-point and floating-point test vectors are different sets of files.

The fixed-point G.728 algorithm internally uses double precision floating-point arithmetic to simulate some fixed-point operations. At the time this simulation was written, this sped up the run-time and allowed the coder to run in real-time on early Pentium-based computers. With today's 64 bit integer machines and faster processors, there are likely to be better alternatives for simulating accumulator guard bits. The fixed-point simulation code should generate bit-exact output for all the test vectors.

Rather than repeating the interface presented in the floating-point section, the Table 5 lists the differences between the fixed-point and floating-point versions.

**Table 5 — Differences between G.728 fixed-point and floating-point versions**

| Object | Floating-Point | Fixed-Point |
|---|---|---|
| Include File | g728.h | g728fp.h |
| Encoder State | G728EncData | G728FpEncData |
| Decoder State | G728DecData | G728FpDecData |
| Encoder State Initialize | g728encinit | g728fp_encinit |
| Decoder State Initialize | g728decinit | g728fp_decinit |
| Encoder | g728encode | g728fp_encode |
| Decoder | g728decode | g728fp_decode |
| Post-filter Disable | g728setpostf | g728fp_setpostf |
| Packet Loss Size | g728setfesize | g728fp_setfesize |
| Packet Loss Concealment | g728decfe | g728fp_eraseframe |

### 11.4.7. G.728 Fixed-point Encoder/Decoder

The program fragment below shows how to run the fixed-point encoder and decoder together. Speech is input from a file, encoded, decoded, and the output speech is written back to a file:

```
#include "g728fp.h"

Short         ix;       /* index */
Short         s[IDIM];  /* 16-bit speech vector*/
G728FpEncData e;        /* encode state */
G728FpDecData d;        /* decode state */

g728fp_encinit(&e);     /* initialize encoder state */
g728fp_decinit(&d);     /* initialize decoder state */
while (fread(s, sizeof(Short), IDIM, speechinf) == IDIM) {
    g728fp_encode(&ix, s, IDIM, &e);   /* encode */
    g728fp_decode(s, IDIM, &d);        /* decode */
    fwrite(s, sizeof(Short), IDIM, speechoutf);
}
```

This code closely resembles the code fragment for the floating-point code. Note that the input and output speech arrays to the encoder and decoder are Shorts so there is no need to call the Float to Short conversion routines. Input and output speech arrays are 16-bit linear full range PCM.

### 11.4.8. G.728 Demonstration Program

Both the floating-point and fixed-point versions of the G.728 coder software provide a demonstration program, defined in `g728.c` and `g728fp.c`, respectively, that can be used for four functionalities: encode, decode, decode with packet losses, and encode and decode a file in a single pass. Scripts are also provided to run G.728 on the test vectors. The test vectors themselves are not part of the STL. The floating-point version compiles the program into a binary called `g728`. The fixed-point version compiles into a binary called `g728fp`. Since the programs provide an identical user interface, we will only discuss `g728`. In the examples below, substitute `g728fp` for `g728` to use the fixed-point version of the coder.

The demonstration programs expect speech input and output files to be header-less, and contain binary 16-bit linear full range PCM. By default it is assumed that the files are in the native byte order of the machine. The byte order of the input and output files can be overridden on the command line with the options `-little` (for little-endian files) and `-big` (for big-endian files). When the -little or -big options are given, the software first determines if the current machine is a big-endian or little-endian machine. If the machine has the same endian characteristics as the requested file, no byte swapping is performed on the input and output files. If there is a mismatch, byte swapping is performed. For example, the G.728 test vector files are little endian files. Giving the -little option on

Sun SPARC and MIPS processors (big-endian machines) will cause the input files to be byte swapped before processing, and output data streams to be byte-swapped before being written to files. On Intel processors, the -little option is a "no-op".

Bit-streams are in the format of the ITU G.728 test vectors. For every 5 input speech samples, a single 16-bit binary word is output. Only the least significant 10 bits of each 16-bit output word are used. The G.728 gain index (3 bits) resides in bits 0 to 2 (0 being the least significant bit), and the shape index (7 bits) resides in bits 3 to 9.

Input PLC mask files are ASCII files that contain '0's and '1's. A '1' implies the current frame is lost and should be concealed. A '0' implies the normal decoding routine should be called for the current frame. The PLC files may also contain newlines and return characters that will be ignored. If any other character occurs in the file, it is an error, and the program will exit with an error message. If the PLC mask file reaches the end of the file but there are still frames to decode, the PLC file will roll over and seek back to the beginning of the file. For example, a PLC mask file containing '00000000000000000001' can be used to introduce a 5% uniform loss pattern.

The demonstration program supports several other options. Options should appear in the command line before the mode of the coder:

| | |
|---|---|
| `-nopostf` | Turn off the post-filter in the decoder. This is required to run many of the test vectors. For normal operation the post-filter should be on. |
| `-stats` | PLC concealment option to print out statistics on how many frames were processed and concealed. |
| `-plcsize msec` | set the PLC concealment frame size, in milliseconds. This is how much speech will be concealed by a single lost packet. Only a limited set of values are acceptable: 2.5, 5, 7.5, 10, 12.5, 15, 17.5 and 20. The default value is 10 ms. |

### 11.4.9. G.728 Demonstration Program Modes

The program has 4 modes: encode, decode, decode with PLC, and encode and decode. To run the encoder use:

```
g728 enc speech.infile bitstream.outfile
```

To run the decoder use:

```
g728 dec bitstream.infile speech.outfile
```

To run the combined encoder and decoder use:

```
g728 encdec speech.infile bitstream.outfile speech.outfile
```

To run the decoder with Packet Loss Concealment with the default 10ms PLC frame size:

```
g728 plc bitstream.infile plcmask.infile speech.outfile
```

### 11.4.10. G.728 Demonstration Program with Test Vectors

Scripts (`testall.sh` for Linux and Unix; `testall.bat` for Windows machines) are provided for running the demonstration programs on the the G.728 test vectors. The scripts use binary compare functions to see if the encoder and decoder output files differ from the expected results. For the fixed-point code an exact match is expected. For the floating-point code an exact match is not guaranteed. However, when compiled to use double-precision arithmetic (`-DUSEDOUBLES`) we have observed an exact match on all platforms we have tested the code on. If the floating-point code is compiled with the option `-DUSEFLOATS`, differences in the test vectors should be expected. For further details on the test vector verification procedure when the outputs do not match, please refer to G.278 Appendix I.

A script (`testplc.sh`, `testplc.bat`) is also provided to run the PLC algorithm on one of the test vectors. This program is only to demonstrate how to invoke the PLC algorithm, and does not compare the output with file. It is not possible to compare the outputs to a fixed file since the PLC algorithm uses a random number generator to extract the excitation when it determines whether a lost packet frame is in an unvoiced region of speech. Depending on where the packet losses occur, the output may be different for each run on the same file.

## 12. G.722: The ITU-T 64, 56, and 48 kbit/s wideband speech coding algorithm

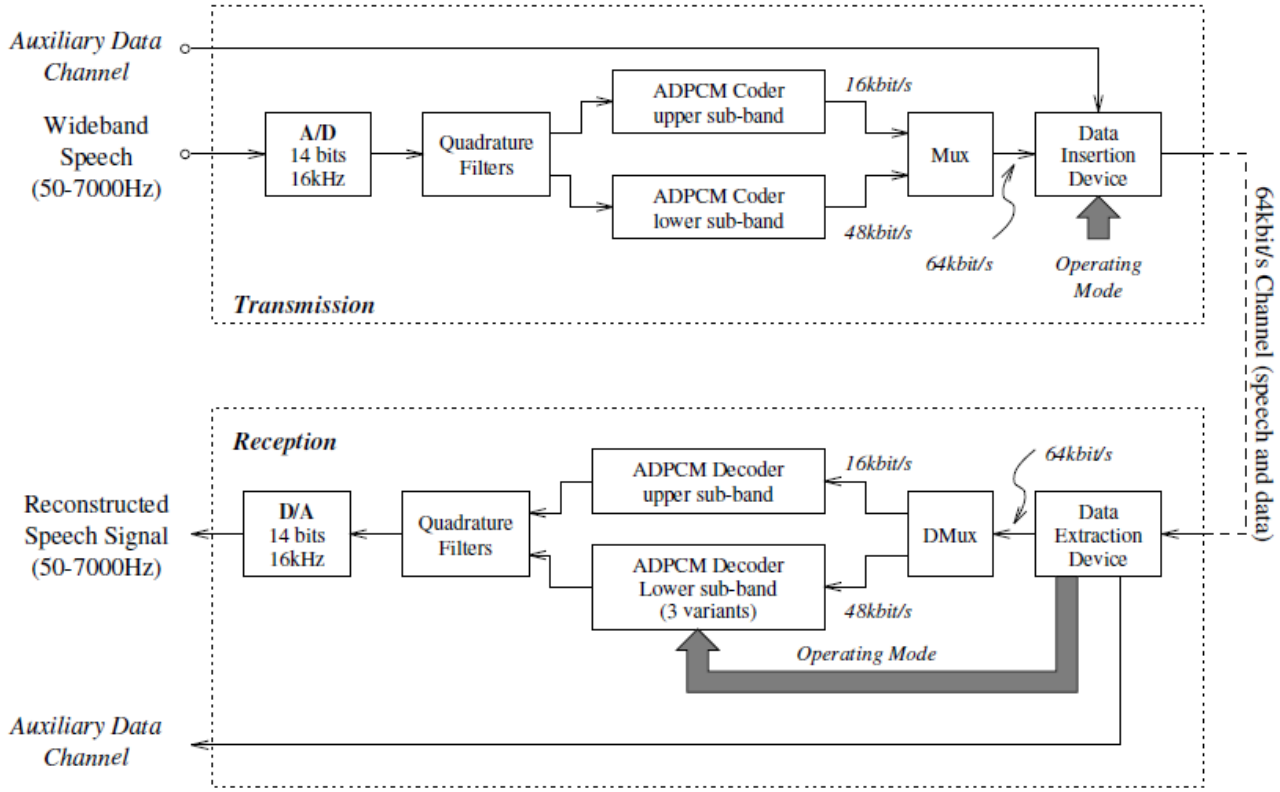
With the emergence of ISDN networks offering digital connectivity at 64 kbit/s between subscribers, the possibility was given to improve the standard telephone quality by increasing the transmitted bandwidth. A bandwidth of 50-7000 Hz corresponding to a sampling of 16 kHz was chosen because it provides a substantial improvement of the quality for applications where the speech is to be heard through high quality loudspeakers e.g. for audio or video conference services, commentary broadcasting, and high quality handsfree phones.

An expert group was created in November 1983 whose mandate was to define a standard for 7 kHz speech coding within 64 kbit/s. After many contributions received from several organisations, it has been decided to choose a coder which combined subband filtering and adaptive differential pulse-code modulation algorithms (SB-ADPCM). The final recommendation was produced in March 1986 and approved in July 1986 by the then CCITT SG XVIII as Recommendation ITU-T G.722 [4].

The full description on the implementation of the G.722 algorithm is found in [4], and network aspects related to its operation are found in [5]. Figure 8 summarizes some systemic aspects of the G.722 algorithm. Overview and notes on the development of the G.722 algorithm can be found in several papers [54], [55], [56], [57], [58], [59]. The following description of the G.722 algorithm is based on the text in [60].

In 2006, upon ETSI DECT request, packet loss concealment (PLC) procedures for G.722 were standardized to ensure a sufficient robustness over the DECT wireless interface. ITU-T G.722 at 64 kbit/s codec is the mandatory coder in ETSI New Generation DECT (NG-DECT) standards (ETSI TS 102 527-1 and -3) [61], [62]. These standards are intended for wideband audio enabled devices to be connected on VoIP networks.

In the above mentioned PLC standardization, G.722 software tool was updated. Originally, the ITU-T G.722 algorithm used its own binary bit stream format without synchronism headers; this made almost impossible to apply frame errors. In the STL 2009, G.722 codec software tool was made compliant with G.192 bitstream format and EID-XOR/G.192 style of frame and bit error application to G.722 is enabled. At the same time, STL basic operators and complexity counters were introduced. Furthermore, the standalone decoder tool now includes basic reference Packet Loss Concealment functionality.




NOTE –

– Operating modes:

| | |
|---|---|
| ***Mode 1*** | 64kbit/s for speech and 0kbit/s for auxiliary data |
| ***Mode 1-bis*** | 56kbit/s for speech and 0kbit/s for auxiliary data |
| ***Mode 2*** | 56kbit/s for speech and 8kbit/s for auxiliary data |
| ***Mode 3*** | 48kbit/s for speech and 16kbit/s for auxiliary data |
| ***Mode 3-bis*** | 48kbit/s for speech, 6.4kbit/s for auxiliary data and 1.6kbit/s for service channel framing and mode control |

– Operating modes 1-bis and 3-bis are applicable only to US national 56 kbit/s networks

– The signal in the 64kbit/s channel comprises 64, 56 or 48 kbit/s for speech and 0, 8 or 16 kbit/s for data, depending on the operating mode.

**Figure 8 — G.722 encoder and decoder block diagrams**

## 12.1. Description of the 64, 56, and 48 kbit/s G.722 algorithm

In order to improve the transmitted speech quality, the input signal has to be converted after antialiasing filtering by an analog-to-digital (A/D) converter operating at 16 kHz sampling rate and with a resolution of at least 14 uniform PCM bits. Similarly, at the receive side, a digital-to-analog (D/A) converter operating at 16 kHz sampling rate and with a resolution of at least 14 uniform PCM bits should be used. The specifications of the transmission characteristics of the audio parts suited for the G.722 algorithm are described in the Recommendation. Some flexibility of the output bit rate was implemented to allow the opening of an auxiliary data channel within the 64 kbit/s channel.

### 12.1.1. Functional description of the SB-ADPCM encoder

Figure 9 shows block diagram of the SB-ADPCM encoder which comprises the following main blocks.

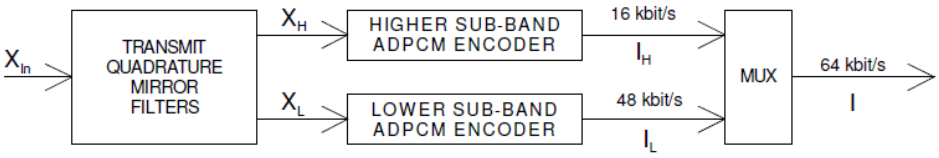



**Figure 9 — Block diagram of the SB-ADPCM encoder**

### 12.1.1.1. Transmit quadrature mirror filters

The input signal $X_{in}$ is first filtered by two quadrature mirror filters (QMF) which split the frequency band [0, 8000 Hz] into two equal subbands. The outputs $X_L$ and $X_H$ of the lower and higher subbands are downsampled at 8 kHz by the filtering procedure.

### 12.1.1.2. Lower subband ADPCM encoder

Figure 10 gives block diagram of the lower subband ADPCM encoder. To transmit the lower band, the encoder was designed to operate at 6, 5 or 4 bits per sample, corresponding to 48, 40 or 32 kbit/s, respectively. The ADPCM algorithm is very similar to the embedded ADPCM algorithm of Recommendation ITU-T G.727 [22]. It is an embedded ADPCM with 4 core bits and 2 additional bits. The embedded property was introduced to prevent degradation in speech quality when the encoder and the decoder operate during short intervals in different modes.

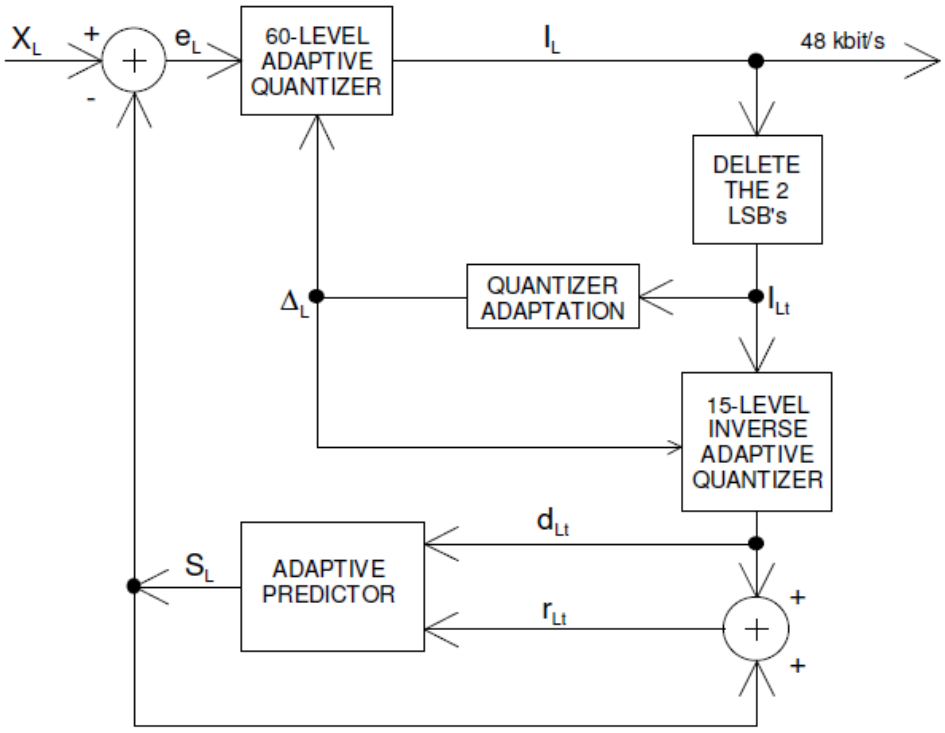



**Figure 10 — Block diagram of the lower subband ADPCM encoder**

### 12.1.1.3. Adaptive quantizer

A 60-level non-uniform adaptive quantizer is used to quantize the difference $e_L$ between the input signal $X_L$ and the estimated signal $S_L$. The output of the quantizer $I_L$ is the ADPCM codeword for the lower subband. The 4 forbidden output codewords were primarily introduced to prevent the generation of all zero codes at all modes, but have also later been used to recover the 8 kHz content used by the coder.

#### 12.1.1.4. Inverse adaptive quantizer

In the feedback loop the two least significant bits of $I_L$ are deleted to produce a 4 bit signal $I_{Lt}$ which is used for the adaptation of the quantizer scale factor and applied to a 15-level inverse adaptive quantizer to produce the quantized difference signal $d_{Lt}$.

#### 12.1.1.5. Quantizer adaptation

In order to maintain a wide dynamic range and minimize complexity, the quantizer scale factor adaptation is performed in the base 2 logarithmic domain. The log-to-linear conversion is accomplished using a lookup table. There is no adaptation of the speed control parameter as in 32 kbit/s ADPCM [21] because the encoder is designed to transmit more than voiceband data.

#### 12.1.1.6. Adaptive predictor and reconstructed signal computation

The adaptive predictor structure is similar to the one used for G.727 ADPCM standard: 2 poles and 6 zeroes. The two sets of coefficients (one for the poles and the other for the zeroes section) are updated using a simplified gradient algorithm. Stability constraints are applied to the poles in order to prevent possible unstable conditions. However, no predictor reset is applied for some specifics inputs conditions as it is done in G.726 algorithm. The reconstructed signal $r_{Lt}$ is computed by adding the quantized difference signal $d_{Lt}$ to the signal estimate $S_L$ produced by the adaptive predictor. The use of a 4-bit operation instead of a 6-bit operation in the feedback loops of the lower band ADPCM encoder and decoder allows for the insertion of data in the two least significant bits without causing mistracking in the decoder.

#### 12.1.1.7. Higher subband ADPCM encoder

Figure 11 shows block diagram of the higher subband ADPCM encoder. This encoder is designed to operate at 2 bits per sample, corresponding to a fixed bitrate of 16 kbit/s. The encoder algorithm is very similar to the lower band one but with the following main differences. The quantizer is a 4-level non-linear adaptive quantizer. The higher subband ADPCM encoder is not embedded, hence the inverse quantizer uses the 2 bits in the feedback loop.

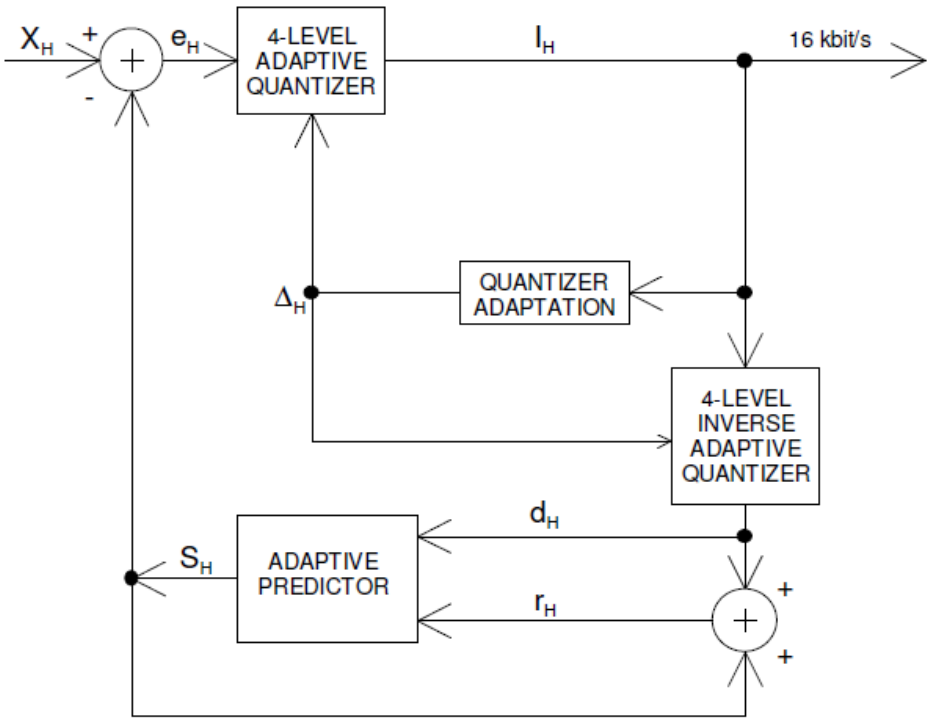



**Figure 11 — Block diagram of the higher subband ADPCM encoder**

#### 12.1.1.8. Multiplexer

The resulting codewords from the higher and lower subbands $I_H$ and $I_L$ are combined to obtain the output codeword $I$ with an octet format for transmission every 8 kHz frame resulting in 64 kbit/s. Note that the 8 kHz clock may be provided by the network as it is always done for 64 kbit/s A-law or $\mu$-law log-PCM (G.711) systems.

### 12.1.2. Functional description of the SB-ADPCM decoder

Figure 12 shows block diagram of the SB-ADPCM decoder.

#### 12.1.2.1. Demultiplexer

The demultiplexer decomposes the received 64 kbit/s octet formatted signal $I_r$ into two signals $I_{Lr}$ and $I_{Hr}$ which form the codeword inputs for the lower and higher subband ADPCM decoders, respectively.

#### 12.1.2.2. Lower subband ADPCM decoder

Figure 13 shows a block diagram of the lower subband decoder. This decoder operates in three different modes depending on the received mode indication: 64, 56 and 48 kbit/s. The block which produces the estimate signal is identical to the feedback portion of the lower subband ADPCM encoder. The reconstructed signal $r_L$ is produced by adding the signal estimate to the relevant quantized difference signals $d_{L,6}$, $d_{L,5}$ or $d_{L,4}$, which are selected according to the received indication of the mode of operation.

#### 12.1.2.3. Higher subband ADPCM decoder

This decoder (see Figure 14) is identical to the feedback portion of the higher subband ADPCM encoder described in the Section clause 12.1.1. Here, the output is reconstructed signal $r_H$.

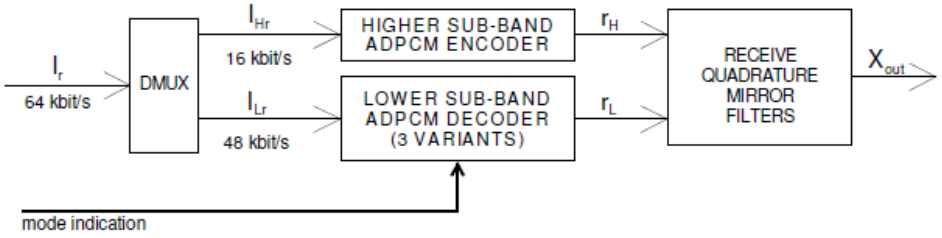



**Figure 12 — Block diagram of the SB-ADPCM decoder**

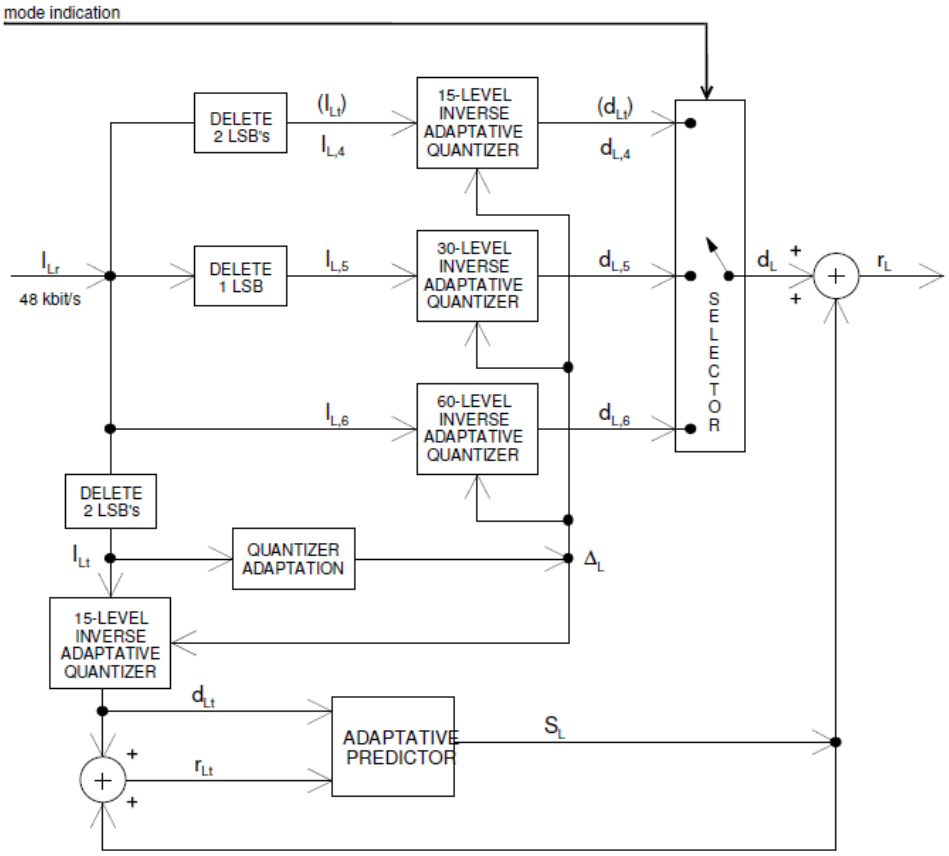



**Figure 13 — Block diagram of the lower subband ADPCM decoder**

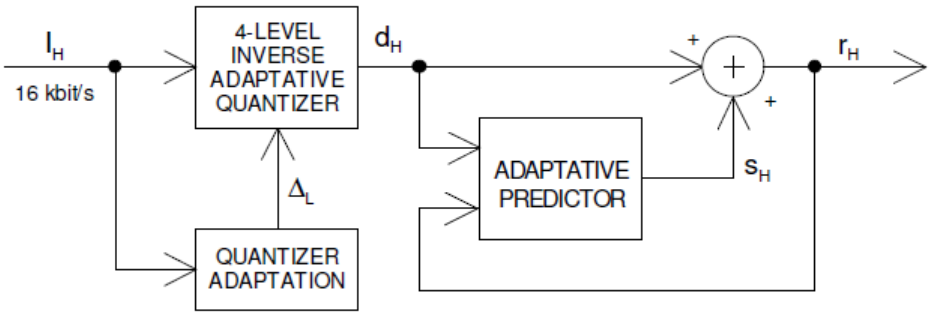



**Figure 14 — Block diagram of the higher subband ADPCM decoder**

### 12.1.2.4. Receive QMF

The receive QMF are two reconstruction filters which interpolate the ouputs of the lower and higher subband ADPCM decoders from 8 to 16 kHz ($r_H$ and $r_L$) and generate 16 kHz sampling reconstructed output $X_{out}$. Signal $X_{out}$ is converted to analog by the digital to analog converter of the receiving side.

## 12.1.3. Functional description of the basic Packet Loss Concealment functionality

In 2006, basic index domain PLC functionality (zero index and frame repeat algorithms) for detected frame errors were included in the standalone decoder tool. The numbering and description of the four basic algorithms provided can be seen in Table 6.

**Table 6 — Basic PLC algorithms supported by the G.722 standalone decoder**

| PLC algorithm number | Description of concealment action |
|---|---|
| 0 (default) | All bits in the erroneous frame are set to "1". (every 8 kHz octet index is set to 0xFF) |
| 1 | same as PLC(0), but includes a decoder reset after the frame decoding and synthesis operation. |
| 2 | The bits from the previous frame(s) are repeated. After four erroneous frames, algorithm PLC(0) is used. |
| 3 | as PLC(2), but the first (0.625 ms) bits of the first good frame after an erroneous frame are set to the value "1". |

The PLC actions in the G.722 standalone decoder are activated by incoming G.192 frames that do not have a valid G192_SYNC header tag. Note that in November 2006, ITU-T SG16 approved two new Appendices to G.722 for Packet Loss Concealment (PLC). These two Appendices (Appendices III and IV [63], [64]) provide better quality over the basic PLC functionality included in STL.

## 12.2. Standalone G.192 compatible G.722 encoder and decoder tool

### 12.2.1. G.192 bit stream format for standalone G.722 encoder and decoder

As described previously, G.722 is an embedded algorithm which supports network scaling of the bitstream in three bitrates: 48, 56 and 64 kbit/s. To support this functionality within the G.192 file format the G.722 encoder writes the 16 kbit/s embedded bits in the end of every G.192 bitstream frame.

For every 16 kHz sample, there are eight G.722 encoded bits (an octet index) to be transported in the G.192 frame. Bits b1 and b0 are the embedded bits and bits b2-b7 are the non-embedded bits. In a G.192 frame, the b2 bits are written first in chunk, followed by that of b3, until b7 bits. After all b7 bits are written, b1 bits are written and then b0 bits. By sorting bits in this manner, truncation of a G.192 type G.722 frame bitstream from 64 kbit/s to 48 or 56 kbit/s is made easy. Figure 15, Figure 16 and Figure 17 show how the G.192 frames are composed for various bit rates (64, 56, 48 kbit/s) and example frame sizes (10, 5, 20 ms). G.192 format is useful for enhanced simulation of G.722, however actual application transport formats may be different (e.g. as defined in [4] and as used in [18]).

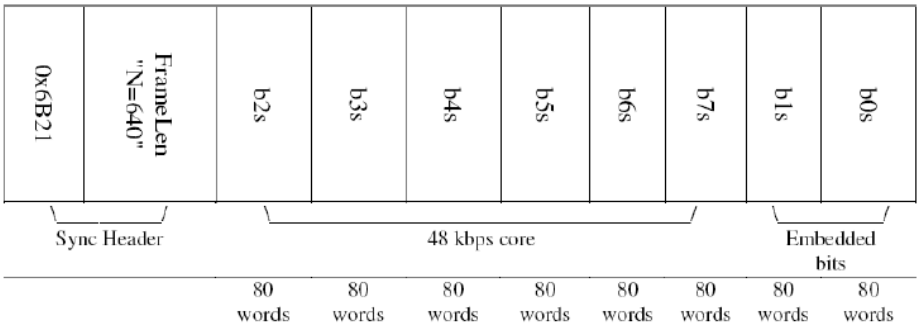



**Figure 15 — Example of a G.192 compatible G.722 encoder output frame for a 10 ms input frame size. This frame has a length of 640 bits and a G192_SYNC (0x6B21) header tag, in the G.192 file each g722 bit is stored as a 16 bit word.**

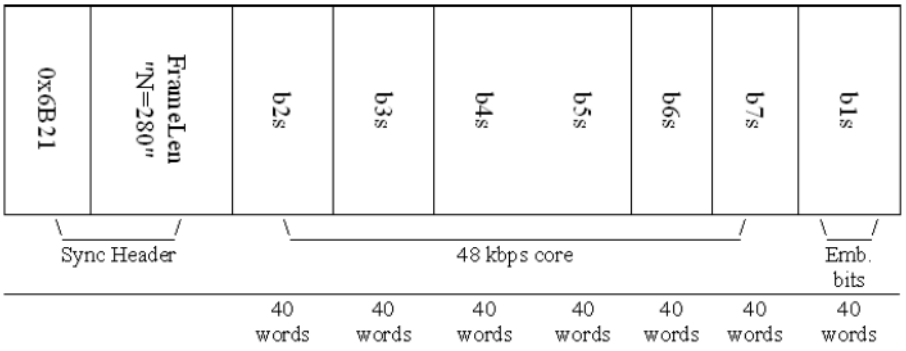



**Figure 16 — Example of a G.192 compatible G.722 56 kbit/s encoder output frame for a 5 ms input frame size. This frame has a length of 280 bits and a G192 SYNC (0x6B21) header tag.**

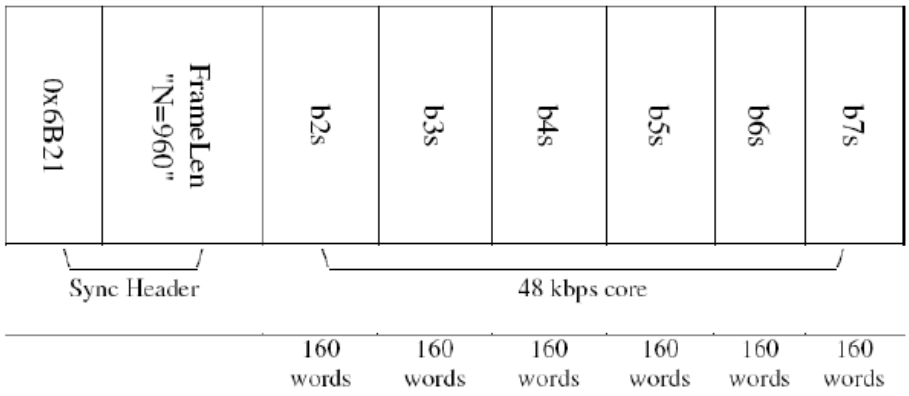



**Figure 17 — Example of G.192 compatible G.722 48 kbit/s encoder output frame for a 20 ms input frame size. This frame has a length of 960 bits and a G192 SYNC (0x6B21) header tag.**

### 12.2.2. Standalone G.722 Encoder specific operation

If the encoder can not read a complete input frame it stops its processing, this means that the final decoder output file may be up to a frame shorter than the input speech file.

## 12.3. ITU-T STL G.722 Implementation

This implementation of the G.722 algorithm is composed of several source files. The interface routines are in file `g722.c`, with prototypes in `g722.h`. The original code of the STL G.722 was provided by CNET/France and its user interface was modified to be consistent with the other software modules of the STL. The update to make G.722 tool compliant with G.192 bit stream format and include basic PLC functionality was performed by Ericsson. The basic operators and complexity counters were introduced by France Telecom.

The problem of storing the state variables was solved by defining a structure called `g722_state` which containing all the necessary state variables. By means of this approach, several streams may be processed in parallel[22], provided that one structure is assigned (and that one call to the encoding/ decoding routines is done) for each data stream (this can be advantageous for machines with support for parallel processing). The G.722 state structure has the following fields (which are all `shorts`):

*ah, al* — Second-order pole section coefficient buffer for higher and lower band, respectively

[22] This feature was not possible with the original code provided by CNET and was added in the modifications of the user interface.

| | |
|---|---|
| *bh, bl* | Seventh-order zero section coefficient buffer for higher and lower band, respectively |
| *deth, detl* | Delayed quantizer scale factor for higher and lower band, respectively |
| *dh* | Quantizer difference signal memory |
| *dlt* | Quantizer difference signal for the adaptive predictor |
| *init_qmf_rx* | Flag indicating the need to initialize the QMF filters on the reception (decoder) side |
| *init_qmf_tx* | Flag indicating the need to initialize the QMF filters on the transmission (encoder) side |
| *nbh, nbl* | Delayed logarithmic quantizer factor for higher and lower band, respectively |
| *ph, plt* | Partially reconstructed signal memory for higher and lower band, respectively |
| *qmf_rx_delayx* | Memory of past 24 received (decoded) samples |
| *qmf_tx_delayx* | Memory of past 24 transmitted (encoded) samples |
| *rh[3]* | Quantized reconstructed signal |
| *rlt[3]* | Reconstructed signal memory for the adaptive predictor |
| *sh, sl* | Predictor output value for higher and lower band, respectively |
| *sph, spl* | Pole section output signal for higher and lower band, respectively |
| *szh, szl* | Zero section output signal for higher and lower band, respectively |

The default bitstream generated by the STL G.722 encoder is a G.192 compatible 16 bit output file and it is provided using the frame size specified by the -fsize command. Bits in a G.192 bitstream frame are ordered so that the frame can be truncated at 56 and 48 kbit/s. If the number of input samples are non divisible by the frame size, the output codeword stream is truncated at the last frame boundary.

A legacy g722 octet stream can be obtained using the option -byte. The legacy bitstream generated by the STL G.722 encoder has 8 valid bits for each encoded sample, saved in right-justified `short`s. i.e., the codewords are located in the lower 8-bits of the encoded bitstream file. The MSB is always 0 for the 16 bit bitstream file. The lower 6 bits are the lower-subband encoded bits, and the upper two bits of the 8 valid bits are the upper-subband encoded bits. When the decoder is not in operation mode 1, the decoder will discard 1 or 2 of the lower bits of the lower-subband. It should be noted that, when bit errors are inserted in this bitstream and the operation mode is not mode 1, the actual bit error rate seen by the decoder may not be the one actually desired. One may consider that, in simulating a system where auxiliary data channels are used, such as modes 2 and 3, this is actually the desired behaviour, because errors hitting the auxiliary data will not affect the decoded speech quality. However, if simulation of modes 1-bis or 3-bis is intended, then the some of the errors hitting the lower 1 (mode 1-bis) or 2 bits (mode 3-bis) will not be seen by the decoder, and the overall bit error rate will actually be smaller than the desired one. There are two possible approaches to circumvent this problem:

– the use of an external program to shift the bitstream samples one or two bits (respectively for modes 1-bis or 3-bis) to the right before the bitstream serialization process for use with the STL EID module, and an external program to left-shift the bitream samples by one or two bits after

error insertion and before using the STL G.722 decoder. This solution is valid for both random and burst bit errors.

– to increase proportionally the bit error rate by 1/8 (mode 1-bis) or 1/4 (mode 3-bis), to statistically compensate for errors hitting unused bits. This solution is valid only for random bit errors.

From the users' perspective, the encoding function is `g722_encode`, and the decoding function is `g722_decode`. Before using these functions, state variables for the encoder and the decoder must be initialized respectively by `g722_reset_encoder` and `g722_reset_decoder`. It should be noted that encoder and decoder need individual state variables to work properly.

In the following part a summary of calls to the three entry functions is found.

### 12.3.1. `g722_encode`

**Syntax:**

```
#include "g722.h"
long g722_encode (short _*inp_buf_, short _*g722_frame_, long _smpno_,
                  g722_state _*g722_encode_);
```

**Prototype:** `g722.h`

**Description:**

Simulation of the ITU-T G.722 64 kbit/s encoder. Takes the linear (16-bit, left-justified) input array of shorts *inp_buf* (16 bit, right-justified, without sign extension) with *smpno* samples, and saves the encoded bit-stream in the array of shorts *g722_frame*.

The state variables are saved in the structure pointed by *g722_encode*, and the reset can be established by making a call to `g722_reset_encoder`.

**Variables:**

| | |
|---|---|
| *inp_buf* | Is the input samples' buffer with *smpno* left-justified 16-bit linear `short` speech samples. |
| *g722_frame* | Is the encoded samples' buffer; each `short` sample will contain the encoded parameters as right-justified 8-bit samples. |
| *smpno* | Is a long with the number of samples to be encoded from the input buffer *inp_buf*. |
| *g722_encode* | A pointer to the state variable structure; all the variables here are for internal use of the G.722 algorithm, and should not be changed by the user. Fields of this structure are described above. |

**Return value:**

Returns the number of speech samples encoded.

### 12.3.2. `g722_decode`

**Syntax:**

```
#include "g722.h"
short g722_decode (short _*g722_frame_, short _*out_buf_, int _mode_, long
                   _smpno_, g722_state _*g722_decoder,_ );
```

**Prototype:** `g722.h`

**Description:**

Simulation of the ITU-T 64 kbit/s G.722 decoder. Reconstructs a linear (16-bit, left-justified) array of shorts *inp_buf* (16 bit, right-justified, without sign extension) with *smpno* samples from the encoded bit-stream in the array of shorts *g722_frame*. Include a basic Packet Loss Concealment functionality.

The state variables are saved in the structure pointed by *g722_decoder*, and the reset can be established by making a call to *g722_reset_decoder*.

**Variables:**

| | |
|---|---|
| *g722_frame* | Is the encoded samples' buffer; each `short` sample will contain the encoded parameters as right-justified 8-bit samples. |
| *out_buf* | Is the output samples' buffer with `smpno` left-justified 16-bit linear `short` speech samples. |
| *mode* | Is an `int` which indicates the operation mode for the G.722 decoder. If equal to 1, the decoder will operate at 64 kbit/s. If equal to 2, the decoder will operate at 56 kbit/s, discarding the least significant bit of the lower-band ADPCM. If equal to 3, the decoder will discard the two least significant bits of the lower band ADPCM, being equivalent to the 48 kbit/s operation of the G.722 algorithm. It should be noted that, for this implementation of the G.722 algorithm, mode 1-bis is identical to mode 2, and mode 3-bis is identical to mode 3. |
| *smpno* | Is a long with the number of samples in the input encoded sample buffer *g722_frame* to be decoded. |
| *g722_decoder* | A pointer to the state variable structure; all the variables here are for internal use of the G.722 algorithm, and should not be changed by the user. Fields of this structure are described above. |

**Return value:**

Returns the number of speech samples encoded.

### 12.3.3. `g722_reset_encoder`

**Syntax:**

```
#include "g722.h"
void g722_reset_encoder (g722_state _*g722_encoder_);
```

**Prototype:** `g722.h`

**Description:**

Initializes the state variables for the G.722 encoder or decoder. Coder and decoder require each a different state variable.

**Variables:**

| | |
|---|---|
| *g722_encoder* | A pointer to the G.722 encoder state variable structure which is to be initialized. |

**Return value:** None.

### 12.3.4. `g722_reset_decoder`

**Syntax:**

```
#include "g722.h"
void g722 reset decoder (g722_state _*g722_decoder_);
```

**Prototype:** `g722.h`

**Description:**

Initializes the state variables for the G.722 decoder. Coder and decoder require each a different state variable.

**Variables:**

| | |
|---|---|
| *g722_decoder* | A pointer to the G.722 decoder state variable structure which is to be initialized. |

**Return value:** None.

### 12.4. Portability and compliance

The portability test for these routines has been performed using the test sequences designed by the ITU-T for the G.722 algorithm[23]. It should be noted that the G.722 test sequences are not designed to test the QMF filters, but only to exercise the upper and lower band encoder and decoder ADPCM algorithms. Therefore, testing of the codec with the test sequences was done with a special set of test programs that used the core G.722 upper- and lower-band ADPCM coding and decoding functions. All test sequences were correctly processed.

This module has been compiled and tested on PC platform with Cygwin (CYGWIN_NT-5.0), using gcc(3.3.3), and with Microsoft Visual Studio 8.

Please note that the 16 bit oriented G.192 files require correct octet swapping of inputs and outputs on big/little-endian machines.

### 12.5. Encoder(encg722) tool command line options

```
Usage:
encg722  [-options] InpFile OutFile
where:
InpFile     is the name of the speech file to be processed;
OutFile     is the name with the processed bitstream;

options:
-fsize  #   Number of 16 kHz input samples per frame (must be an even number).
            Default is 160 samples (16 kHz) (10 ms)
-mode   #   Operating mode (1,2,3) (or rate 64, 56, 48 in kbit/s).
            Default is mode 1 (= 64 kbit/s)
-frames #   number of frames to process
            (values -1 or 0 processes the whole file )
-byte       Provide encoder output data in legacy octet format.
            (default is g192).
-h/-help    print help message
```

### 12.6. Decoder (decg722) tool command line options

```
Usage:

decg722 [-options] InpFile OutFile
```

[23] The G.722 test sequences are freely downloadable from http://www.itu.int/rec/T-REC-G.722-198703-I!AppII/en.

```
where:
InpFile     is the name of the bit stream input file;
OutFile     is the name of the file with synthesized speech;

options:

-mode #     is the operation mode for the G.722 decoder.
            Default is mode 1/64 kbit/s.
            (others are mode 2/56 kbit/s and mode 3/48 kbit/s)

-fsize #    Number of samples per frame
            Default is 160, 16 kHz samples (or 10ms)
            (NB! must be the same as on the encoder side.)

-frames #   Number of frames to process

-plc #      Packet Loss Concealment algorithm number
            (0 = zero index insertion, no decoder state reset)
            (1 = zero index insertion, decoder state reset)
            (2 = previous frame repetition, no decoder state reset)
            (3 = previous frame repetition, a few zero_indeces in first good
             frame, no decoder state reset)

-byte       Use legacy nonG192 G.722 format (byte oriented) without
            frame/synch headers.

-h/-help    print help message
```

## 12.7. Example code

### 12.7.1. Description of the demonstration programs

One demonstration program is provided for the G.722 module, g722demo.c. In addition, two programs are provided in the distribution when compliance testing of the encoder and decoder is necessary, `tstcg722.c` and tstdg722.c[24].

Program `g722demo.c` accepts 16-bit, linear PCM samples sampled at 16 kHz as encoder input. The decoder also produces files in the same format. The bitstream signals out of the encoder are always organized in 16-bit, right-justified words that use the lower 8 bits (i.e., 64 kbit/s). According to the user-specified mode, the decoder will decode the G.722-encoded bitstream using 64, 56, or 48 kbit/s (i.e. full 8 bits, discard 1 bit of the lower band, or discard 2 bits of the lower band). It should be noted that the demonstration programs `g722demo.c`, `tstcg722.c` and `tstdg722.c` only produce G.722 legacy bitstream. To produce bitstream compliant with G.192, please use `encg722.c` and decg722.c. Similarly, basic PLC functionnality is not supported the demonstration programs g722demo.cand tstdg722.c. Please use `decg722.c` for basic PLC functionnality.

### 12.7.2. Simple example

The following C code gives an example of G.722 coding and decoding using as input wideband speech which is encoded and decoded at either 64, 56, or 48 kbit/s, according to the user-specified parameter *mode*.

```
#include <stdio.h>
#include "ugstdemo.h"
```

[24] . The demonstration program `g722demo.c` cannot be used for compliance verification because the test vectors for G.722 do not foresee processing through the quadrature mirror filters.

```
#include "g722.h"
#define BLK_LEN 256

void main(argc, argv)
  int             argc;
  char           *argv[];
{
  g722_state      encoder_state, decoder_state;
  int             mode;
  char            FileIn[180], FileOut[180];
  short           smpno, tmp_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
  FILE           *Fi, *Fo;

  /* Get parameters for processing */
  GET_PAR_S(1, "_Input File: ..................", FileIn);
  GET_PAR_S(2, "_Output File: .................", FileOut);
  GET_PAR_I(3, "_Mode: ........................", mode);

  /* Initialize state structures */
  g722_reset_encoder(&encoder_state);
  g722_reset_decoder(&decoder_state);

  /* Opening input and output 16-bit linear PCM speech files */
  Fi = fopen(FileIn, RB);
  Fo = fopen(FileOut, WB);

 /* File processing */
  while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
  {
    /* Encode input samples in blocks of length BLK_LEN */
    smpno = g722_encode(inp_buf, tmp_buf, BLK_LEN, &encoder_state);

    /* Decode G.722-coded samples in blocks of length BLK_LEN */
    smpno = g722_decode(tmp_buf, out_buf, mode, smpno, &decoder_state);

    /* Write 16-bit linear PCM output decoded samples */
    fwrite(out_buf, smpno, sizeof(short), Fo);
  }

  /* Close input and output files */
  fclose(Fi); fclose(Fo);
}
```

### 12.7.3. Example operation of encoder (encg722)

```
foreach mode ( 1 2 3 ) # (corresponds to 64,56,48 kbit/s)
   foreach fsize ( 160 320 640) # (corresponds to 10,20,40 ms)
      encg722  -mode $mode -fsize $fsize inpsp.smp outp.g722.$fsize.$mode.g192
   end #fsize
end #mode
```

### 12.7.4. Example operation of decoder (decg722)

```
foreach mode ( 1 2 3 ) # (corresponds to 64,56,48 kbit/s)
   foreach fsize ( 160 320 640) # (corresponds to 10,20,40 ms)
      decg722 -plc 0 -fsize $fsize outp.g722.$fsize.$mode.g192  outsp.smp
      mv outpsp.smp outpsp.g722.$fsize.$mode.plc.0.smp
   end #fsize
end #mode
```

## 13. RPE-LTP: The full-rate GSM codec

In 1988, the Groupe Special Mobile of the Conference Europeéne des Postes et Telecommunications (CEPT) approved the first generation of a pan-European digital cellular radio system operating at a net rate of 13 kbit/s[25]. Its speech coding algorithm, the RPE-LTP (Regular Pulse Excitation, Long Term Predictor) was a compromise solution of the two best coders at that stage. The full-rate GSM system started operation in the beginning of 1992 in some European countries and its expansion is expected in a mid-term. This coder, despite not being an ITU-T standard, is relevant for standardization studies when scenarios involving tandeming conditions between the PSTN and the European cellular system need to be studied.

The current version of the STL includes a RPE-LTP implementation based on a freely available implementation originally produced at the Technical Institue of the University of Berlin, for a Unix environment. This code has been adapted, corrected to work on several platforms, and tested with the recommended test vectors, all properly processed.

Details on the algorithm can be found in several references [65], [66], [67], besides the Recommendation itself [68].

### 13.1. Description of the 13 kbit/s RPE-LTP algorithm

The RPE-LTP is a frame based coder, encoding 20 ms frames of input data at a time. The encoder converts each 160 sample frame (8 kHz sampling rate, 13 bits uniform PCM format) into a bitstream frame of 260 bits. The decoder uses the 260 bitstream bits to generate a frame of 160 reconstructed speech samples.

#### 13.1.1. RPE-LTP Encoder

A simplified block diagram of the RPE-LTP encoder [68] is shown in Figure 18.

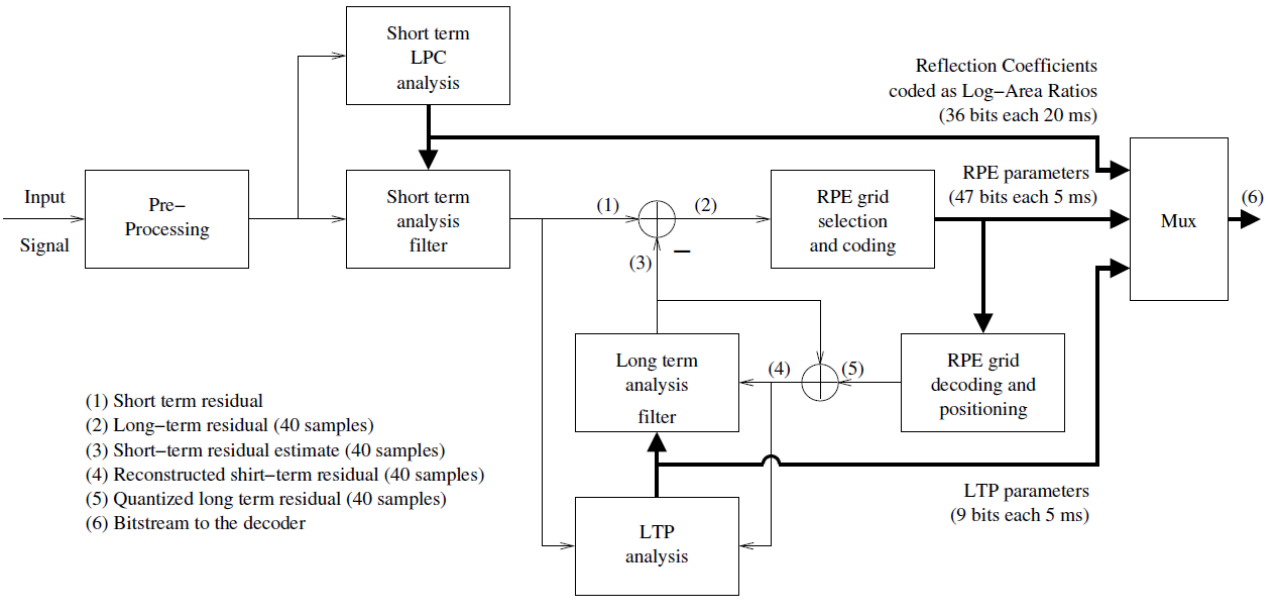



**Figure 18 — Simplified block diagram of the RPE-LTP encoder**

[25] The GSM standard developed initially under the responsability of the CEPT was later transferred to the European Standardisation Telecommunications Institute (ETSI), and the acronym GSM had its meaning changed to Global System for Mobile Communications. Currently, the GSM specifications are being maintained by the Third Generation Partnership Project, 3GPP (www.3gpp.org).

The input speech frame, consisting of 160 uniform 13 bits PCM signal samples, is first pre-processed to produce an offset-free signal, which is then subjected to a first-order pre-emphasis filter. The 160 samples obtained are then analyzed to determine the coefficients for the short-term analysis filter (LPC analysis). Using these coefficients for the filtering of the same 160 samples produce the 160 samples of the short-term residual signal. The filter parameters are represented as reflection coefficients which are transformed to log-area ratios (LARs) before transmission.

For the following operations, the speech frame is divided into 4 sub-blocks consisting each of 40 samples. Before the processing of each sub-block, the parameters of the long-term analysis filter, the LTP lag and the LTP gain, are estimated and updated in the LTP analysis block. Estimation and update is performed on the basis of the signal in the current sub-block and a stored sequence of the 120 previously reconstructed short-term residual samples.

A block of 40 long-term residual signal samples is obtained by subtracting 40 estimates of the short-term residual from the short-term residual signal itself. The resulting block is fed to the Regular Pulse Excitation (RPE) analysis which performs the basic compression function.

As a result of the RPE-analysis, the block of 40 input long-term residual samples is represented by one of 4 candidate sub-sequences of 13 pulses each. The subsequence selected is identified by the RPE grid position. The 13 RPE pulses are encoded using Adaptive Pulse Code Modulation (APCM) with estimation of the sub-block amplitude which is transmitted to the decoder as side information. The RPE parameters are also fed to a local RPE decoding and reconstruction module which produces a block of 40 samples of the quantized version of the long-term residual signal. By adding these 40 quantized samples of the long-term residual to the previously obtained block of short-term residual signal estimates, a reconstructed version of the current short-term residual signal is obtained. The block of reconstructed short-term residual signal samples is then fed to the long-term analysis filter which produces the new block of 40 short-term residual signal estimates to be used for the next sub-block thereby completing the feedback loop.

### 13.1.2. RPE-LTP Decoder

The simplified block diagram of the RPE-LTP decoder [68] is shown in Figure 19.

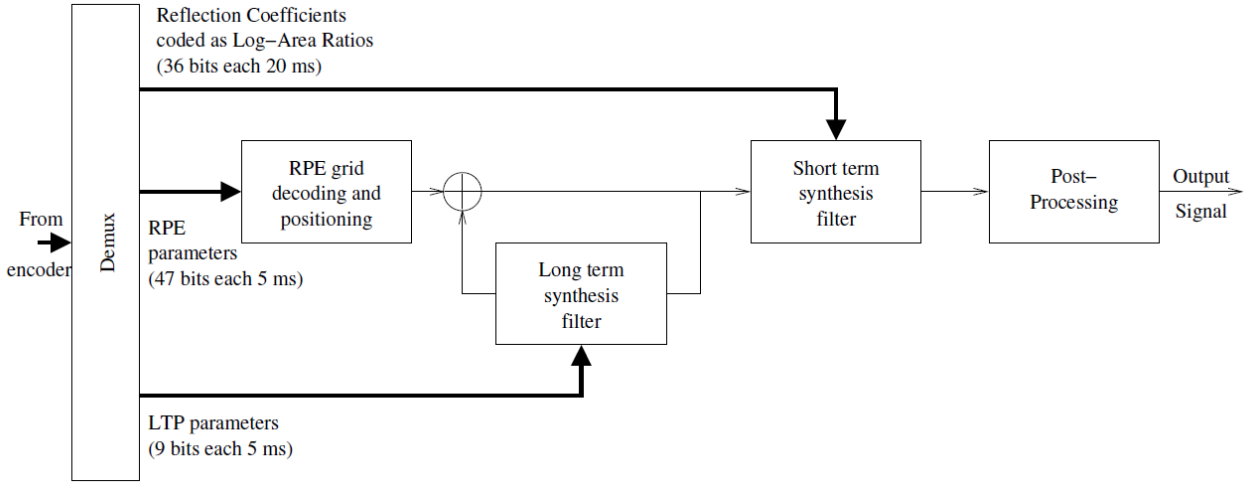



**Figure 19 — Simplified block diagram of the RPE-LTP decoder**

The decoder includes the same structure as the feed-back loop of the encoder. In error-free transmission, the output of this stage will be the reconstructed short-term residual samples. These samples are then applied to the short-term synthesis filter followed by the de-emphasis filter resulting in the reconstructed speech signal samples.

## 13.2. Implementation

This implementation of the RPE-LTP algorithm is composed of several source files. The interface routines are in `rpeltp.c`, with prototypes in `rpeltp.h`.

Originally written to be a device driver in Unix (known as *toast*), its interface was adapted to the specifications of the ITU-T STL, and modified to operate correctly in a variety of platforms, like VAX, IBM PC compatibles, and Unix workstations (Sun and HP).

The problem of storing the state variables was solved by defining a structure containing all the necessary variables, defining a new type called `gsm`, which is a pointer to a structure. By means of this approach, several streams may be processed in parallel, provided that one structure is assigned (and that one call to the encoding/decoding routines is done) for each data stream (this can be advantageous for machines with support for parallel processing). The RPE-LTP state structure has the following fields (all except *L_z2* and *mp* are `short`, which are `long` and `int`, respectively):

| | |
|---|---|
| *dp0* | Memory of 280 past samples |
| *z1* | DC-offset removal filter memory |
| *L_z2* | DC-offset removal filter parameter. |
| *mp* | Preemphasis |
| *u* | Eighth-order short term LPC analysis coefficients |
| *LARpp* | Log Area Ratio array |
| *j* | Index |
| *nrp* | Long-term synthesis parameter |
| *v* | Ninth order short-term synthesis vector |
| *msr* | Post-processing parameter |
| *verbose* | Flag used only if compiled with NDEBUG==0 |
| *fast* | Enables fast but inaccurate computation. Does not properly process the test sequences with this mode turned on. |

Table 7 presents the RPE-LTP encoder output parameters in order of occurence, with parameters defined in [68]. It should be noted that the bitstream file generated by the STL implementation of the RPE-LTP algorithm uses an unpacked format, as other codecs in the STL. Therefore, each of the 76 parameters indicated in Table 7 occupy an unsigned, right-adjusted 16-bit word. Unlikely to the G.711 and G.726 algorithms, however, the number of significant bits per bitstream parameter is not the same for all the parameters, as can be seen from the table. An important implication is that the STL bit error insertion routines cannot be applied directly to the bitstream generated by the STL RPE-LTP encoder. This limitation is not a function of the EID module itself, but of the serialization and parallelization (S/P) routines `serialize_*` and `parallize_*` implemented in the Utility module, which are able only to handle bitstreams that have the same number of valid bits per sample. Solution to this problem still needs to be implemented in the STL. It should be noted however that, since the full-rate GSM channel coding is not implemented in the STL, bit error insertion directly in the unprotected RPE-LTP bitstream will generally not be used. Should the user need bit error insertion in the unprotected RPE-LTP bitstream, there are two possible solutions:

– it will be necessary to pack the bits for each parameter in such a way that, as seen by the S/P routines, each sample in the packed bitstream will have a constant number of valid bits per bitstream sample. Since there are 260 ($4 \times 5 \times 13$) bits for each frame, possible combinations are packed bitstreams with 65 16-bit words, of which the lower 4 bits are meaningful, or with 20

16-bit words, of which the lower 13 bits are meaningful. The former is preferred, despite the longer files generated.

– the user may modify the demonstration program to generate or accept (depending on whether it is an encoding or decoding operation) a serial bitstream format, as understood by the EID module, instead of a parallel bitstream format.

**Table 7 — RPE-LTP bitstream format for each 20 ms speech frame.**

| Parameter | Parameter number | Number of bits |
|---|---|---|
| LAR1 | 1 | 6 |
| LAR2 | 2 | 6 |
| LAR3 | 3 | 5 |
| LAR4 | 4 | 5 |
| LAR5 | 5 | 4 |
| LAR6 | 6 | 4 |
| LAR7 | 7 | 3 |
| LAR8 | 8 | 3 |
| **Sub-frame No. 1** | | |
| LTP lag | 9 | 7 |
| LTP gain | 10 | 2 |
| RPE grid position | 11 | 2 |
| Block amplitude | 12 | 6 |
| RPE-pulse no. 1 | 13 | 3 |
| … | … | … |
| RPE-pulse no. 13 | 25 | 3 |
| **Sub-frame No. 2** | | |
| LTP lag | 26 | 7 |
| LTP gain | 27 | 2 |
| RPE grid position | 28 | 2 |
| Block amplitude | 29 | 6 |
| RPE-pulse no. 1 | 30 | 3 |
| … | … | … |
| RPE-pulse no. 13 | 42 | 3 |
| **Sub-frame No. 3** | | |
| LTP lag | 43 | 7 |
| LTP gain | 44 | 2 |
| RPE grid position | 45 | 2 |
| Block amplitude | 46 | 6 |
| RPE-pulse no. 1 | 47 | 3 |
| … | … | … |
| RPE-pulse no. 13 | 59 | 3 |
| **Sub-frame No. 4** | | |
| LTP lag | 60 | 7 |
| LTP gain | 61 | 2 |
| RPE grid position | 62 | 2 |
| Block amplitude | 63 | 6 |
| RPE-pulse no. 1 | 64 | 3 |
| … | … | … |
| RPE-pulse no. 13 | 76 | 3 |

From the users' perspective, the encoding function is `rpeltp_encode`, and the decoding function is `rpeltp_decode`. Before using these functions, the state variable for either the encoder or the decoder must be initialized by `rpeltp_init`. It should be noted that encoder and decoder need individual state

variables to work properly. After all the processing is performed, the memory allocated for the state variables can be freed by calling `rpeltp_delete`. The following sub-sections describe these four entry functions for the STL RPE-LTP module.

### 13.2.1. `rpeltp_encode`

**Syntax:**

```
#include "rpeltp.h"
void rpeltp_encode (gsm _rpe_state_, short _*inp_buf_, short _*rpe_frame_);
```

**Prototype:** `rpeltp.h`

**Description:**

Simulation of the GSM full-rate RPE-LTP encoder. The 16-bit, left-justified linear-PCM input array of `short` samples *inp_buf* are processed by the RPE-LTP encoder and the encoded bit-stream is returned in the right-justified array of `short` samples *rpe_frame*, with one sample for each encoded parameter. The input frame has 160 samples and the encoded frame has 76 samples.

The state variables are saved in the structure pointed by *rpe_state*, previously initialized by a call to `rpeltp_init()`. The reset can be stablished by making a call to `rpeltp_init()`.

**Variables:**

| | |
|---|---|
| *rpe_state* | A pointer to the state variable structure. All the variables here are for internal use of the RPE-LTP algorithm and should not be changed by the user. Fields of this structure are described above. |
| *inp_buf* | Is the linear-PCM input sample buffer which must have 160 left-justified 16-bit linear-PCM `short` samples. Only the 13 MSb are used. |
| *rpe_frame* | Is the encoded sample buffer. Each `short` sample will contain the encoded parameters as right-justified samples. The actual number of significant bits per sample will depend on each parameter. |

**Return value:** None.

### 13.2.2. `rpeltp_decode`

**Syntax:**

```
#include "rpeltp.h"
void rpeltp_decode (gsm _rpe_state_, short _*rpe_frame_, short _*out_buf_);
```

**Prototype:** `rpeltp.h`

**Description:**

Simulation of the GSM full-rate RPE-LTP decoder. The encoded bit-stream in the input array of right-justified `short` samples *rpe_frame* is used to reconstruct a block of the speech signal using the RPE-LTP decoder. The reconstructed speech block is returned in the 16-bit, left-justified linear-PCM output array of `short` samples *inp_buf*. The input frame has 76 samples and the decoded frame has 160 samples.

The state variables are saved in the structure pointed by *rpe_state*, previously initialized by a call to `rpeltp_init()`. The reset can be established by calling `rpeltp_init()`.

**Variables:**

| | |
|---|---|
| *rpe_state* | A pointer to the state variable structure. All the variables here are for internal use of the RPE-LTP algorithm and should not be changed by the user. Fields of this structure are described above. |

*rpe_frame* Is the encoded sample buffer, which must have 76 right-justified `short` samples. The actual number of bits per sample will depend on each parameter.

*out_buf* Is the output samples buffer, which will contain 160 left-justified, 13-bit linear-PCM `short` samples. The three LSbs are set to zero.

**Return value:** None.

#### 13.2.3. `rpeltp_init`

**Syntax:**

```
#include "rpeltp.h"
gsm rpeltp_init (void);
```

**Prototype:** `rpeltp.h`

**Description:**

Initializes the state variables for the RPE-LTP encoder or decoder. Combined coder and decoder operation requires a different state variable for the encoding and the decodeing part.

**Variables:** None.

**Return value:**

A pointer to an initialized state variable structure defined by the type `gsm`. Returns NULL in case of failure.

#### 13.2.4. `rpeltp_delete`

**Syntax:**

```
#include "rpeltp.h"
void rpeltp_init (gsm rpe_state);
```

**Prototype:** `rpeltp.h`

**Description:**

Releases memory allocated to a state variable previously initialized by rpeltp_init().

**Variables:**

*rpe_state* A pointer to a previously initialized RPE-LTP state variable structure.

**Return value:**

None.

### 13.3. Portability and compliance

The portability test for these routines has been done using the test sequences designed by the GSM for the RPE-LTP (available from ETSI), which were also used to verify the compliance of the encoding and decoding function to the full-rate GSM voice codec Recommendation [68], Annex C.

This routine has been tested in VAX/VMS with VAX-C and gcc, in the PC with Borland C v3.0 (16-bit mode) and gcc (32-bit mode). In the Unix environment in a Sun workstation with cc, acc, and gcc, and in HP with gcc. In all tested cases, 100% of the test sequences passed when the following symbols were defined at compilation time: `SASR`, `USE_FLOAT_MUL` and `NDEBUG`. The symbol *FAST* must not be defined for perfomance compliant with the GSM 06.10 Recommendation, while `USE_FLOAT_MUL` **must** be defined at compilation time. The symbol `NeedFunctionPrototypes` must be undefined for pre-ANSI-C compilers (e.g. SunOS cc compiler).

## 13.4. Example code

### 13.4.1. Description of the demonstration program

One program is provided as demonstration program for the RPE-LTP module, rpedemo.c.

Program `rpedemo.c` accepts input files in either 16-bit linear PCM format, 16-bit, right-justified A-law format, or 16-bit, right-justified $\mu$-law format for the encoding operation. The output of the decoder can also be in any of these formats, but it will have the same format as the encoding operation if encoding and decoding is performed in a single pass (default). If the encoding and decoding operations are performed in separate steps, the format of the output signal does not need to match the format of the original linear PCM signal. The encoder output and decoder input are signals in 16-bit, right-justified samples, as described before in Sections clause 13.2.1 and clause 13.2.2. Three operations are possible: encode and decode in a single pass (default), encode-only (option *-enc*), or decode-only (option *-dec*).

### 13.4.2. Simple example

The following C code gives an example of RPE-LTP coding and decoding using as input 13-bit, linear-PCM speech samples, which are encoded and decoded at 13 kbit/s.

```
#include <stdio.h>
#include "ugstdemo.h"
#include "rpeltp.h"

#define BLK_LEN 160

int main(argc, argv)
  int       argc;
  char     *argv[];
{
  gsm       encoder_state, decoder_state;

  char      FileIn[180], FileOut[180];
  short     bs_buf[BLK_LEN], inp_buf[BLK_LEN], out_buf[BLK_LEN];
  FILE     *Fi, *Fo;

  /* Get parameters for processing */
  GET_PAR_S(1, "_Input File: ................. ", FileIn);
  GET_PAR_S(2, "_Output File: ................ ", FileOut);

  /* Initialize state structures */
  encoder_state = rpeltp_init();
  decoder_state = rpeltp_init();

  /* Opening input and output LOG-PCM files */
  Fi = fopen(FileIn, RB);
  Fo = fopen(FileOut, WB);

 /* File processing */
  reset = 1;                    /* set reset flag as YES */
  while (fread(inp_buf, BLK_LEN, sizeof(short), Fi) == BLK_LEN)
  {
    /* Encode input linear PCM samples */
    rpeltp_encode(encoder_state, inp_buf, bs_buf, BLK_LEN);

    /* Decode samples */
    rpeltp_decode(decoder_state, bs_buf, out_buf);

    /* Write decoded samples */
```

```
    fwrite(out_buf, BLK_LEN, sizeof(short), Fo);

    if (reset)
      reset = 0;                /* set reset flag as NOMORE */
  }

  /* Free memory */
  rpe_delete(decoder_state);
  rpe_delete(encoder_state);

  /* Close input and output files */
  fclose(Fi);
  fclose(Fo);
  return 0;
}
```

## 14. RATE-CHANGE: Up- and down-sampling module

In certain applications involving digitized speech, such as subjective evaluation of speech processed by digital algorithms, it may be preferable to use sampling higher than the typical rate used for the algorithms under test. This is desirable because simpler analog filters with less phase distortion can be built. Another advantage is that upper frequency components of the signal are not lost. It also allows for the convenient shaping of the input signal, such as IRS, $\Delta_{SM}$, and psophometric weightings. Consequently there is a need to adapt the sampling rate of the digitized signal to that of the processing algorithm. For telephony applications, the typical sampling rate is 8000 Hz with a signal bandwidth in general of 300-3400 Hz, and for wideband speech applications, a bandwidth of 50-7000 Hz is desired with sampling rate of 16000 Hz. During the 2005-2008 ITU-T study period, greater audio bandwidth were considered and superwideband and fullband audio codecs were developed. Their typical samplings rates are respectively 32000 Hz with a signal bandwidth of 50-14000 Hz, and 48000 Hz with a signal bandwidth of 20-20000 Hz. Therefore, sampling rates above 8000 Hz and 16000 Hz are desirable, respectively. In several experiments [70] the sampling rate was 16 kHz. In others (see [71] and [72]), 48 kHz and 32 kHz were utilized. Hence the need for a software tool to carry out filtering and sampling rate change. Next, the rate change and spectral weighting routines implemented in the ITU-T STL are presented.

### 14.1. Description of the Algorithm

Signal processing theory describes the basic arrangement for decimation of signals; first the signal is low-pass filtered to limit its bandwidth in order to avoid aliasing when the rate is lowered and, second, to decimate the samples, i.e., to drop out samples from the input signal, such that the desired output rate is obtained. For example, if a rate reduction from 48 kHz to 8 kHz is desired, a decimation factor of 6:1 is necessary. This is equivalent to say that, after limiting the bandwidth of the digitized speech to 4 kHz, 5 out 6 samples are skipped, or alternatively, only 1 out of 6 samples will be kept (or saved) from the signal.

The up-sampling of signals requires that each of the input samples be followed by a number of zero samples, such that the desired output rate is achieved; after this, an interpolation operation of these zero samples is performed to obtain a continuous-envelope signal. For example, up-sampling data from 8 kHz to 16 kHz requires interleaving each sample of the input signal with a zero sample followed by interpolation of the signal. This interpolation can be carried out by means of a polynomial, which is equivalent to a filtering operation.

The type of filtering required is determined by the application intended for the signals. For the tools needed in this version of the STL, three different groups of characteristics were defined:

- **High-quality**: Change in rate without changing the frequency response of the input signal. This is accomplished with a flat, linear phase, low-pass or bandpass FIR filter.
- **Spectral weighting**: Spectral weighting without rate change is necessary for some applications. For narrow-band speech, available are the IRS weighting specified in ITU-T Rec. P.48, the so-called "modified" IRS (annex D of ITU-T Rec. P.830), the far-to-near-field conversion $\Delta_{SM}$ weighting, and the psophometric noise weighting of ITU-T Rec. O.41. For wideband signals, the mask for wideband handsets, as defined in ITU-T Rec. P.341, is also available. For super-wideband signals, the mask for super-wideband videoconferencing terminals has been derived as an extension of ITU-T Rec. P.341.
- **PCM quality**: Change in rate accompanied with modification of the frequency response of the input signal according to the mask specified in Recommendation ITU-T G.712. This is accomplished with a non-linear phase low-pass IIR filter.

### 14.1.1. High-quality

The response of the filters in this type of rate change must minimize phase and amplitude distortion. For example, for decimation from 48 kHz to 16 kHz, the filter must be flat up to about 8 kHz (except for the transition, or cut-off, region), with a linear phase. In other cases, it may be desirable to remove the DC component and hum noise (50—60 Hz AC line noise) from the signal without additional phase distortion to the upper region of the spectrum.

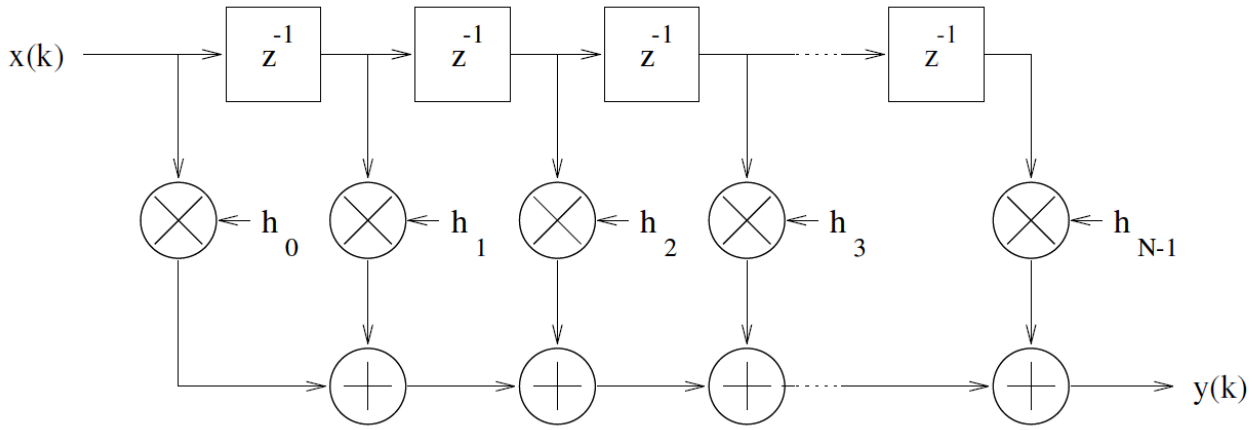



**Figure 20 — FIR filter block diagram.**

One way to do this is to use a linear phase finite impulse response (FIR) digital filter, as in Figure 20. The input and output characteristic is defined by:

$$y(k) = \sum_{i=0}^{N-1} h(i) \cdot x(k-i)$$

Linear phase is guaranteed if the filter is symmetric, i.e.:

$$h(k) = h(N-1-k), \text{ for } k = 0..N-1$$

### 14.1.2. Narrowband weighting

#### 14.1.2.1. IRS weighting

The IRS weighting corresponds to a bandpass filtering characteristic whose mask can be found in Recommendation ITU-T P.48 [12]. The send and receive spectral shapes of the IRS weighting were obtained in a round-robin series of measurements made on a number of analog telephones in the early 1970's [73]. From these measurements, the average send and receive frequency-response characteristics were derived. However, for the loudness balance purposes for which the IRS was designed, it was also necessary to include a 300-3400 Hz bandpass filter, known as the SRAEN

*(Système de Référence pour la détermination de l'Affablissement équivalent pour la Netteté; Reference System for determining Articulation Ratings)* filter [10], [1]. The values of send and receive sensitivity currently given in ITU-T Rec. P.48 (columns 2 and 3 in Table 8) are therefore composed of the average send and receive responses for a number of telephones, as well as the response of the SRAEN filter (see column 4 of Table 8).

Because the P.48 IRS weighting used to be considered to model an average narrow-band telephone handset deployed in the PSTN, the IRS weighting has been chosen to simulate speech signals obtained from a regular handset. Examples of standardization efforts using the P.48 weighting characteristic are the Recommendations ITU-T G.711, G.721, and G.728. This weighting, as defined in P.48, is sometimes called "full-IRS" weighting.

While the weighting characteristic in P.48 was considered to model connections over analog transmission facilities in the past (although it is not clear why the SRAEN filter should be included in both the send and receive paths), it is no longer representative of connections over modern digital facilities. In particular, the low frequency roll-off gives rise to unnecessary quality degradation. For the purpose of low bit-rate coder evaluation, especially where the coder is located in the telephone handset, a better characteristic can be obtained by modifying the P.48 full-IRS response to remove the SRAEN filter as shown in columns 5 and 6 of Table 8. These values are specified in Annex D of ITU-T Recommendation P.830 [17] and define the so-called "modified" IRS weighting. The modified IRS has been used in the development of Recommendations ITU-T G.723.1 and G.729.

**Table 8 — Send and receive amplitude frequency characteristics for the IRS response as in ITU-T Rec.P.48, the SRAEN filter, and the modified IRS (P.48 IRS with SRAEN filter insertion loss removed).**

| Frequency (hz) | P.48 irs | | Sraen | Modified irs | |
|---|---|---|---|---|---|
| | Send (dbPa/V) | Receive (dbPa/V) | Filter (dB) | Send (dbPa/V) | Receive (dbPa/V) |
| 100 | -45.8 | -27.2 | 14.1 | -31.7 | -13.4 |
| 125 | -36.1 | -18.8 | 11.4 | -24.7 | -7.4 |
| 160 | -25.6 | -10.8 | 8.4 | -17.2 | -2.4 |
| 200 | -19.2 | -2.7 | 5.9 | -13.3 | 3.2 |
| 250 | -14.3 | 2.7 | 4.0 | -10.3 | 6.7 |
| 300 | -11.3 | 6.4 | 2.8 | -8.5 | 9.2 |
| 315 | -10.8 | 7.2 | 2.5 | -8.3 | 9.7 |
| 400 | -8.4 | 9.9 | 1.4 | -7.0 | 11.3 |
| 500 | -6.9 | 11.3 | 0.6 | -6.3 | 11.9 |
| 600 | -6.3 | 11.8 | 0.3 | -6.0 | 12.1 |
| 630 | -6.1 | 11.9 | 0.2 | -5.9 | 12.1 |
| 800 | -4.9 | 12.3 | 0.0 | -4.9 | 12.3 |
| 1000 | -3.7 | 12.6 | 0.0 | -3.7 | 12.6 |
| 1250 | -2.3 | 12.5 | 0.0 | -2.3 | 12.5 |
| 1600 | -0.6 | 13.0 | 0.1 | -0.5 | 13.1 |
| 2000 | 0.3 | 13.1 | -0.2 | 0.1 | 12.9 |
| 2500 | 1.8 | 13.1 | -0.5 | 1.3 | 12.6 |
| 3000 | 1.5 | 12.5 | 0.5 | 2.0 | 13.0 |
| 3150 | 1.8 | 12.6 | 0.3 | 2.1 | 12.9 |
| 3500 | -7.3 | 3.9 | 7.0 | -0.3 | 10.9 |
| 4000 | -37.2 | -31.6 | 33.7 | -3.5 | 2.1 |
| 5000 | -52.2 | -54.9 | 43.2 | -9.0 | -11.7 |
| 6300 | -73.6 | -67.5 | | -23[a] | |

| Frequency (hz) | P.48 irs | | Sraen | Modified irs | |
|---|---|---|---|---|---|
| 8000 | -90.0 | -90.0 | | -40[a] | |
| [a] Values estimated from the modified IRS implemented in the STL. | | | | | |

The most important part of either the full or the modified IRS weighting is the transmission (or send) characteristic. The receive characteristic is less important because listening is in general done using handsets conforming to P.48 (which eliminates the need for filtering by the software, since it is done by the telephone terminal). In addition, the receive characteristic is relatively flat. Some studies also show that the use of headphones instead of handsets does not result in significantly different results while yielding lesser listener fatigue [74], [75]. Nevertheless, for cases where the receive-side MIRS filter is to be applied, a FIR implementation of this filter is available for 8000 Hz and 16000 Hz sampling rates.

An unspecified point in both P.48 and modified IRS is the phase response of the filter. There have been discussions within UGST on this topic and the conclusion was that, since the phase response is unspecified, it should be kept as generic as possible, what is better accomplished by keeping the phase linear[26]. If a certain non-linear phase characteristic is desired by the user, this can be implemented by cascading an all-pass filter with the desired phase response with one of the available FIR IRS implementations. Therefore, the IRS filters are implemented as FIR filters, as depicted in Figure 20.

Recommendation ITU-T P.48 presents the nominal values for the amplitude response in column 2 of its Table 1 (here reproduced in column 2 of Table 8) and then the upper and lower tolerances listed in its Table 2. For the STL approach, it was decided to design IRS filters whose characteristic would deviate no more than 0.5 dB from the average values in P.48 (see in Figure 31 the agreement of the nominal values, represented by dots, and the measured frequency response for the original P.48 IRS characteristic, represented by the continuous curve in the figure).

#### 14.1.2.2. Other weighting

A filter that simulates the input response characteristic of certain mobile terminals was incorporated in the STL for data sampled at 16 kHz. Figure 29 and Figure 30 display the respective frequency and impulse responses for the filter.

### 14.1.3. Wideband weighting

#### 14.1.3.1. P.341 weighting

While the IRS filter is applicable to telephony bandwidth (or narrowband) speech, for wideband speech the specification for the send and receive sides is given in Recommendation ITU-T P.341 [11]. The mask specified in P.341 is rather wide, and an implementation of the send-side mask agreed on by the experts has been incorporated in the STL.

#### 14.1.3.2. Other weightings

In the process to select a wideband codec at 32 and 24 kbit/s, a 50 Hz --5 kHz bandpass filter was developed and incorporated in the STL. Figure 43 and Figure 44 display the respective frequency and impulse responses for the filter.

A 100 Hz—5 kHz filter was also designed for tests of another wideband codec. Figure 45and Figure 46 show the respective frequency and impulse response of this filter.

26 In spite of that, a non-linear phase IIR IRS filter is provided in the IIR module as an example of a cascade-form IIR filter implementation.

### 14.1.4. Greater wideband weightings

#### 14.1.4.1. P.341 extension weighting

For super-wideband signals, the mask for super-wideband videoconferencing terminals is based on the ITU-T Rec. P.341. The sensitivity/frequency characteristics of the P.341 filter were extended to a larger band [50 Hz - 14 kHz] with a sampling frequency of 32 kHz. The corresponding 50 Hz-14 kHz filter was developed and incorporated in the STL. Figure 41 and Figure 42 display the respective frequency and impulse responses for the filter.

#### 14.1.4.2. Other weightings

To generate anchors typically used in BS.1534 [24] "MUlti Stimulus test with Hidden Reference and Anchor (MUSHRA)" subjective tests, seven low-pass filters are also provided in the STL. Those anchors are generated by low-pass filters with cut-off frequencies 1.5, 3.5, 7, 10, 12, 14 and 20 kHz at sampling frequency of 48 kHz. Their frequency responses are shown, respectively, in Figure 49, Figure 51, Figure 53, Figure 55, Figure 57, Figure 59, and Figure 61. Their impulse responses are also given in Figure 50, Figure 52, Figure 54, Figure 56, Figure 58, Figure 60, and Figure 62. In addition a bandpass filter [20 Hz - 20 kHz) operating at sampling frequency of 48 kHz was also designed. Its frequency response is shown in Figure 47 and its impulse responses in Figure 48.

### 14.1.5. Noise weighting

Two weighting filters are available in this version of the STL, the psophometric and the $\Delta_{SM}$ weighting filters.

The psophometric weighting curve defined by Recommendation ITU-T O.41 is used for measuring the noise level in telephone circuits, accounting for the subjective perception of noise. The psophometric noise measure (given in dBmp) is related to the North-American C-message weighting curve (given in dBrnC), using to the following:

$$dBmp = dBrnC - 90.0dB$$

The other type of signal weighting filter is the $\Delta_{SM}$, used for converting acoustic signals recorded in the far field using an omnidirectional microphone to the near-field equivalent of that signal if it were in the background of a telephone user. Owing to the directionality of the human mouth, head and torso, the high frequencies will mainly be radiated in the frontal direction, while the diffuse field will represent a spatial integration of the radiation in all directions [76]. Hence, the $\Delta_{SM}$ filter is deployed for weighting acoustic noises (babble, vehicular, etc.) before electrical summation with clean speech files, in order to simulate speech corrupted by background noise. It is useful in subjective listening tests where precise control of the actual SNR is necessary.

Both these filters have been implemented as FIR filters. The psophometric filter has been designed for speech sampled at 8 kHz, and the $\Delta_{SM}$ filter for speech sampled at 16 kHz. It should be noted that these filters, like the IRS filters, are also frequency-specific and, unlike the low-pass high-quality FIR filters described before, cannot be used for arbitrary rate ratio conversion.

### 14.1.6. PCM Quality

There are applications requiring the simulation of the response of filters found in the A/D and D/A interfaces of current transmission systems, which are in general PCM systems satisfying Recommendation ITU-T G.711. The filters associated with G.711 are specified in Recommendation ITU-T G.712 [3]. The main characteristic of these filters is the low out-of-band rejection of 25 dB.

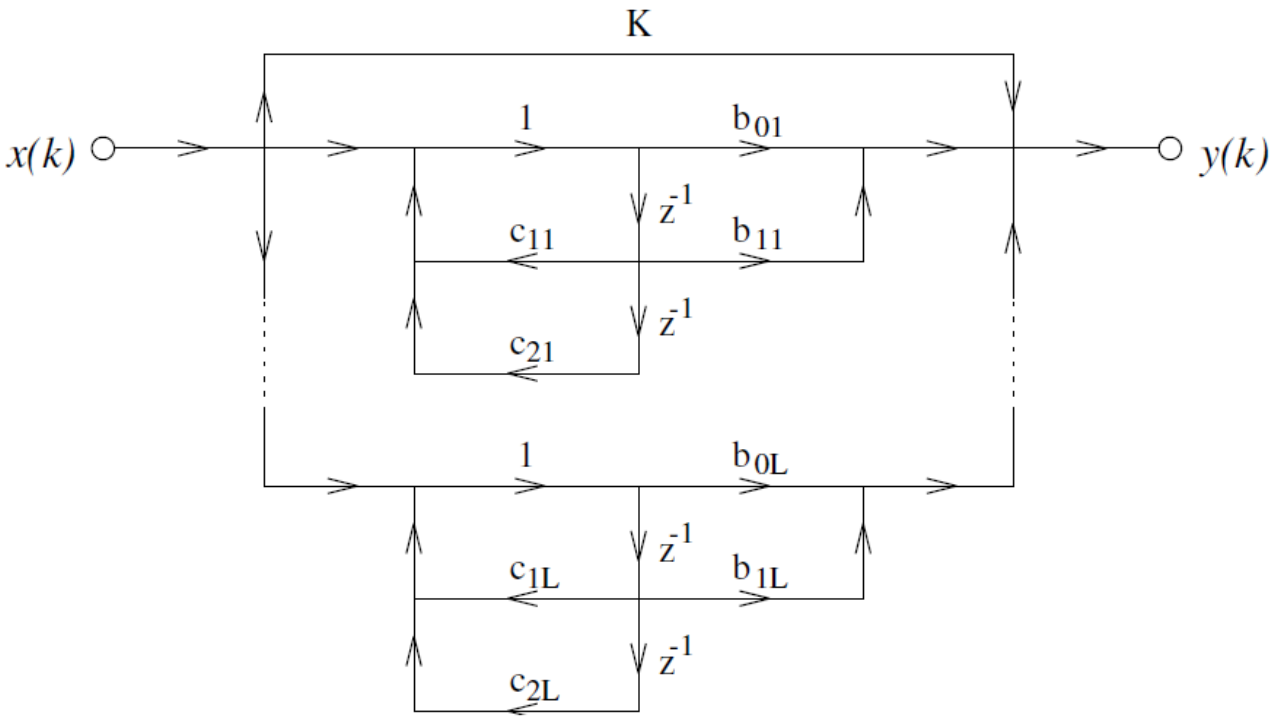



**Figure 21 — Parallel-form IIR filter block diagram.**

In this context it is also necessary to simulate the convertion back to and forth the analog domain, e.g. to simulate multiple transcodings which are called *asynchronous trancodings*[27]. One way to simulate asynchronous transcodings is by means of a non-linear phase filter (non-constant group delay), which is most efficiently implemented using IIR filters.

Infinite impulse response (IIR) filters used in this tool are of the parallel form (see Figure 21), described by the equation:

$$H_I^p(z) = K + \sum_{l=1}^{L} \frac{b_{0l} + b_{1l}z^{-1}}{\backslash 1 + c_{1l}z^{-1} + c_{2l}z^{-2} \backslash}$$

and of the cascade form (see Figure 22), described by the equation:

$$H_I^c(z) = \prod_{l=1}^{N} \frac{b_{0l} + b_{1l}z^{-1} + b_{2l}z^{-2}}{1 + c_{1l}z^{-1} + c_{2l}z^{-2}}$$

[27] As the name indicates, there is no synchronization between sampling instants of the two digital systems, i.e., re-sampling in the succeeding A/D is not synchronous to the clock in the preceding D/A converter.

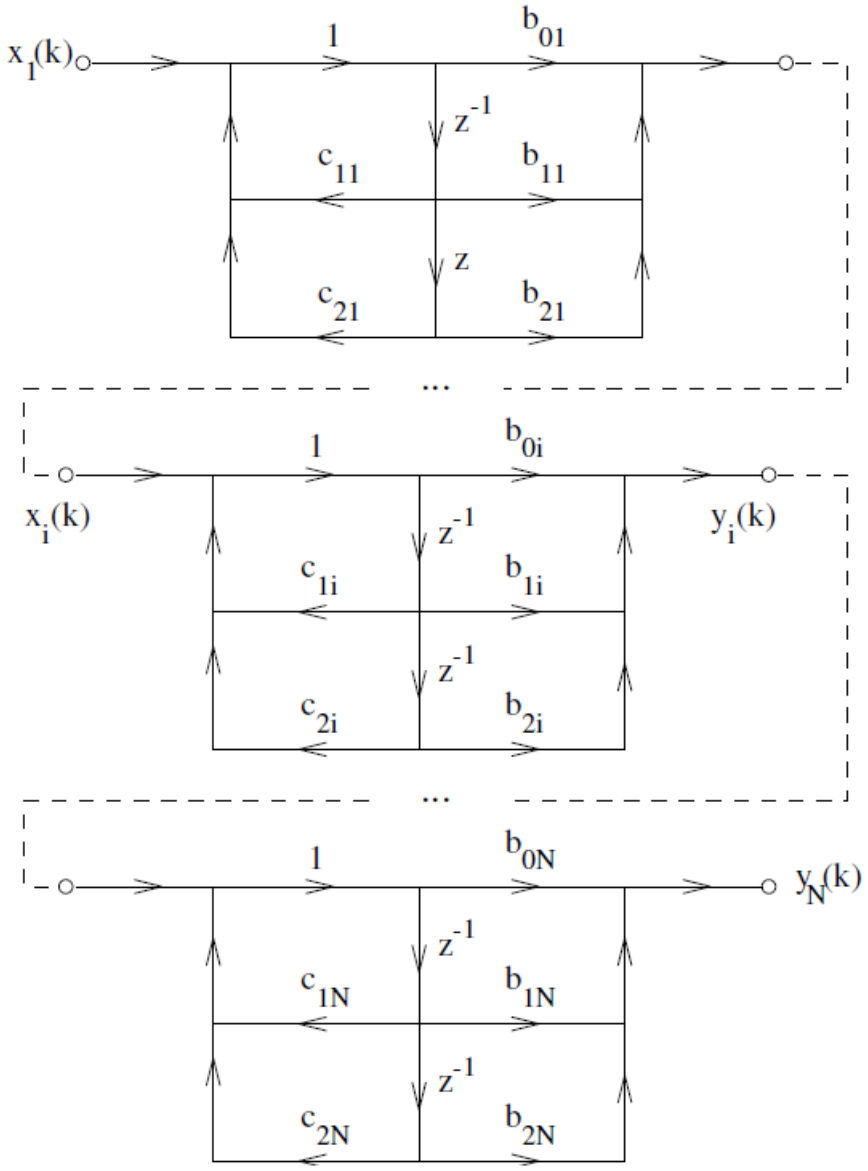



**Figure 22 — Cascade-form IIR filter block diagram.**

## 14.2. Implementation

The rate change algorithm is organized in two modules, FIR and IIR, with prototypes respectively in `firflt.h` and `iirflt.h`. It evolved from a version initially developed as part of the ETSI Half-rate GSM codec Host Laboratory exercise [77]. The rate-change functionality was incorporated in the STL92 in two main files, `hqflt.c` and `pcmflt.c`. To make these routines more flexible, the following modifications were included:

FIR — the FIR module was divided into a library source file (`fir-lib.c`) containing the basic filtering and initialization functions, as well as into source files for each kind of filter: `fir-flat.c` for high-quality low-pass and bandpass filters, `fir-irs.c` for the classical and modified IRS filters, and so on;

IIR — the IIR module was divided into a library file (`iir-lib.c`) containing basic filtering and initialization functions, as well as into source files for each kind of filter: `iir-g712.c` for G.712 filtering using the parallel-form filters, `iir-flat.c` for flat bandpass 1:3 and 3:1 asynchronization filtering using a cascade-form filter, and so on.

Files `fir-*.c` of the FIR module contain all the routines implementing FIR filters, i.e., the high-quality filters and IRS, $\Delta_{SM}$ and psophometric weighing filters. Files `iir-*.c` of the IIR module implement the IIR filters, i.e., the parallel-form PCM filter and the cascade-form 3:1 asynchronization filter.

Some of these filters have been implemented using 24-bit coefficients, thus allowing real-time, bit-exact hardware implementation of these routines. It may be noted that, for these filters in the STL, the calculations are performed in floating-point arithmetics by converting the coefficients from the range $-2^{23}..2^{23}-1$ to $-1..+1$, which is not needed in real time hardware with fixed-point DSPs.

Frequency response and impulse response plots are provided for the STL filters in the forthcoming sections. It should be noted, however, that the impulse responses shown have been computed from the 16-bit quantized impulse responses of the filters, as generated by the demonstration programs, while the frequency responses were calculated as described in section clause 14.3. It should be noted that the apparent asymmetry in some impulse responses happens because an integer number of samples are generated, and linear interpolation is used to draw the figure. If the impulse responses were derived directly from the filter coefficients, the plot would be symmetric.

NOTE – When the same filter type is used by several independent speech materials (e.g. several speech files) within the same execution of an application program, the user must remember that the filters have memory. Hence, wrong results can be obtained if a given number of initial samples are not discarded. See section clause 14.4 for an example, where the first 512 samples are skipped when calculating the power level of the output tone.

### 14.2.1. FIR module

The frequency responses of the implemented high-quality low-pass filters are shown in Figure 23 and Figure 24 (for rate-change factors 2 and 3, respectively), while the telephone bandwidth bandpass filter is given in Figure 25 (only a rate-change factor of 2 is available). The impulse responses of these filters are given in Figure 26, Figure 27, and Figure 28, respectively for the up-sampling filters (factors 2 and 3), for the down-sampling filters (factors 2 and 3), and for the bandpass filter.

The transmit-side IRS filter has been implemented for the "regular" and modified flavors. The regular transmit-side P.48 IRS filter amplitude responses are shown in Figure 31 (the available sampling rates are 8 and 16 kHz). The transmit-side modified IRS filter is available for sampling at 16 kHz and 48 kHz, and their frequency responses are shown in Figure 33. The impulse response of these transmit-side IRS filters are in figures Figure 32 and Figure 34 for the regular and modified IRS filters, respectively. The receive-side modified IRS filter has also been implemented and the frequency responses for 8 kHz and 16 kHz sampling rate are found in Figure 35. The impulse responses of the receive-side modified IRS filters are shown in Figure 36.

The frequency response of the STL psophometric filter is given in Figure 37, and that of the $\Delta_{SM}$ filter in Figure 38.

For wideband signals, three weighting filters are available. The transmit-side ITU-T P.341 filter amplitude response is shown in Figure 39, and its impulse response is shown in Figure 40. Alternatively to the P.341 filter, the frequency and impulse responses of the two bandpass filters, 50 Hz-5 kHz bandpass filter and 100 Hz -5 kHz, are shown, respectively, in Figure 43 and Figure 44, and in Figure 45 and Figure 46.

For super wideband signals, four weighting filters are available. The 50 Hz-14 kHz bandpass filter, extension of ITU-T P.341 filter, is presented in Figure 41 (frequency response), and Figure 42 (impulse response). Alternatively to this P.341 filter extension, the frequency responses of the three MUSHRA anchors filters, LP3.5, LP7 and LP10 filters, are shown Figure 51, Figure 53 and Figure 55, respectively. Their impulse responses are shown in Figure 52, Figure 54 and Figure 56, respectively.

The high-quality filters were implemented for rate-change factors of 2 and 3. The IRS filters, band-limiting filters and MUSHRA anchors have been designed for specific *sampling rates* (e.g. 8 and 48 kHz). It should be noted that, while the high-quality filters are independent of the rate, these filters are not, because their masks are specified in terms of Hz, rather than normalized frequencies. This means that to carry out a high-quality up-sampling from 8 to 16 kHz, and from 16 to 32 kHz, the same routines are called, while for IRS, band-limiting or MUSHRA anchors, there is no rate-change routine from 16 to 32 kHz.

Since the digital filters have memory, state variables are needed. In this version of the STL, a type `SCD_FIR` is defined, containing the past sample memory, as well as filter coefficients and other control variables. Its fields, whose values shall never be changed by the user, are as follows:

| | |
|---|---|
| *lenh0* | Number of FIR coefficients |
| *dwn_up* | Down-sampling factor |
| *k0* | Start index in next segment (needed in segment-wise filtering) |
| *h0* | Pointer to array with FIR coefficients |
| *T* | Pointer to delay line |
| *hswitch* | Switch to FIR-kernel: up- or down-sampling |

The relevant routines for each module are described in the next sections.[28]

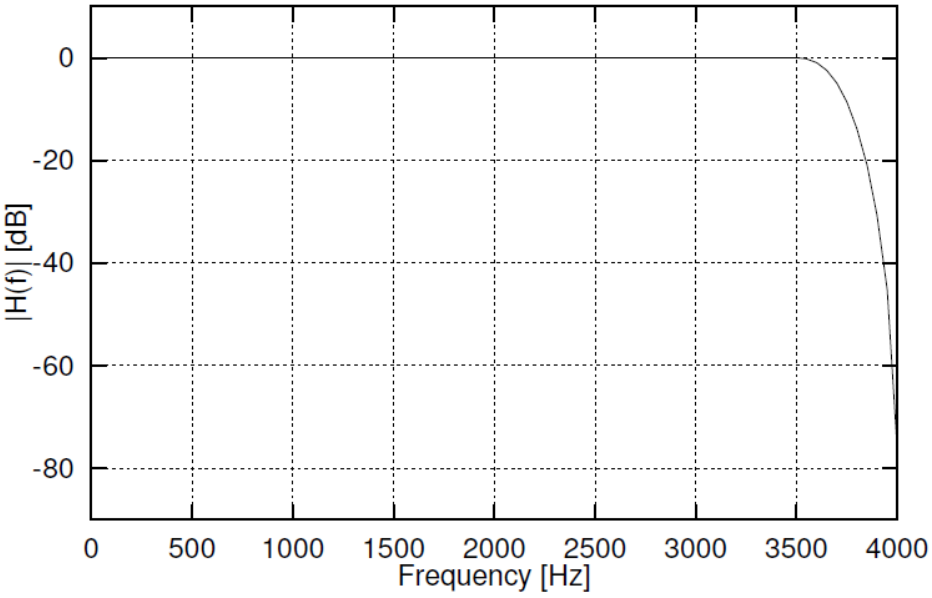



**Figure 23-a — High-quality filter for up-sampling**

[28] It should be noted that in the source code files there are local (privately-defined) functions which are not intended to be directly accessed by the user and therefore are not described here.

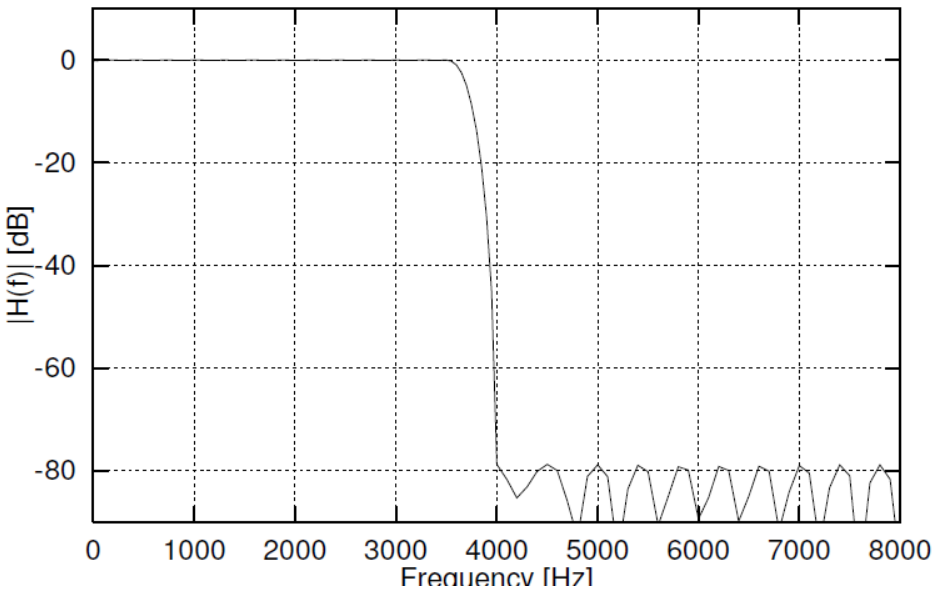



**Figure 23-b — High-quality filter for down-sampling**

**Figure 23 — High-quality filter responses for a factor of 2 and sampling rates of 8000 and 16000 Hz**

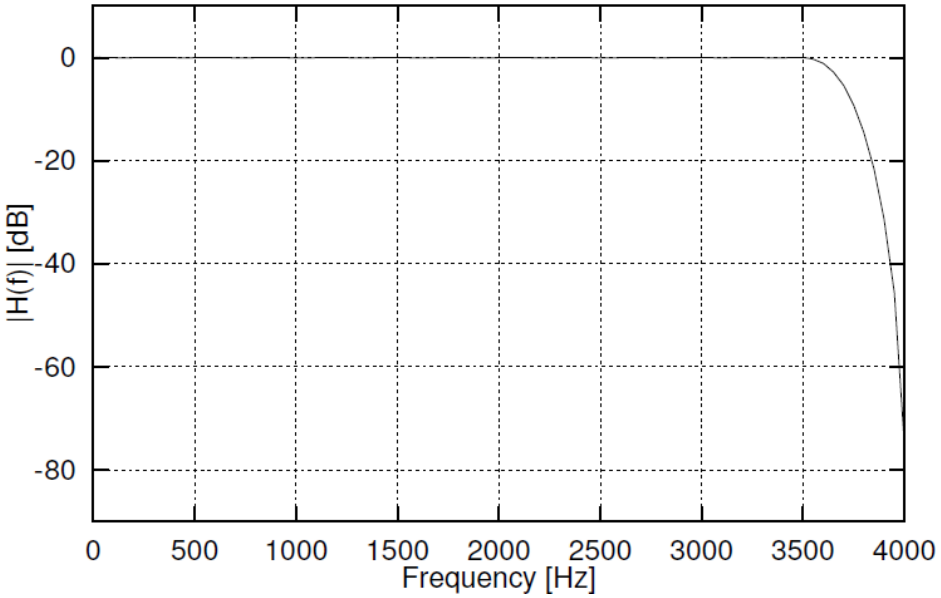



**Figure 24-a — High-quality filter for up-sampling**

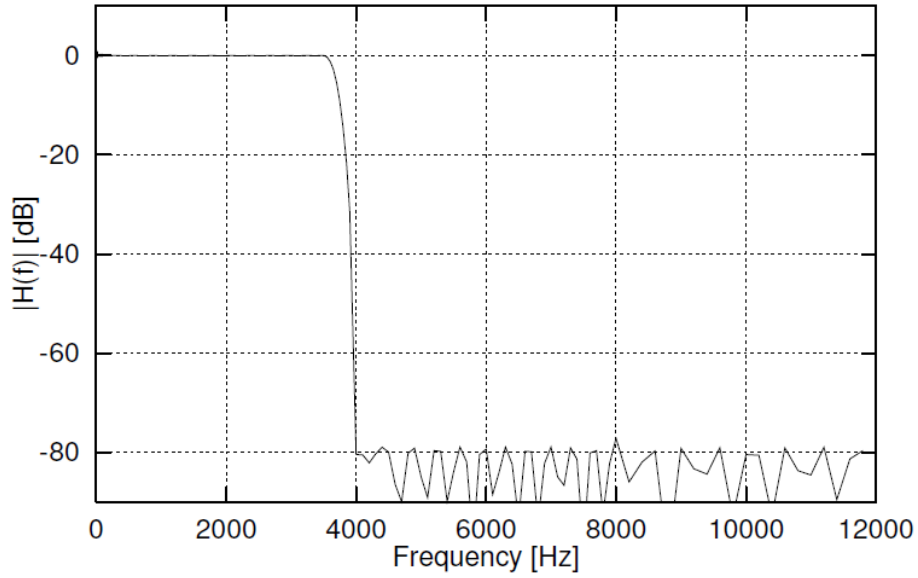



**Figure 24-b — High-quality filter for down-sampling**

**Figure 24 — High-quality filter responses for a factor of 3 and sampling rates of 8000 and 24000 Hz**

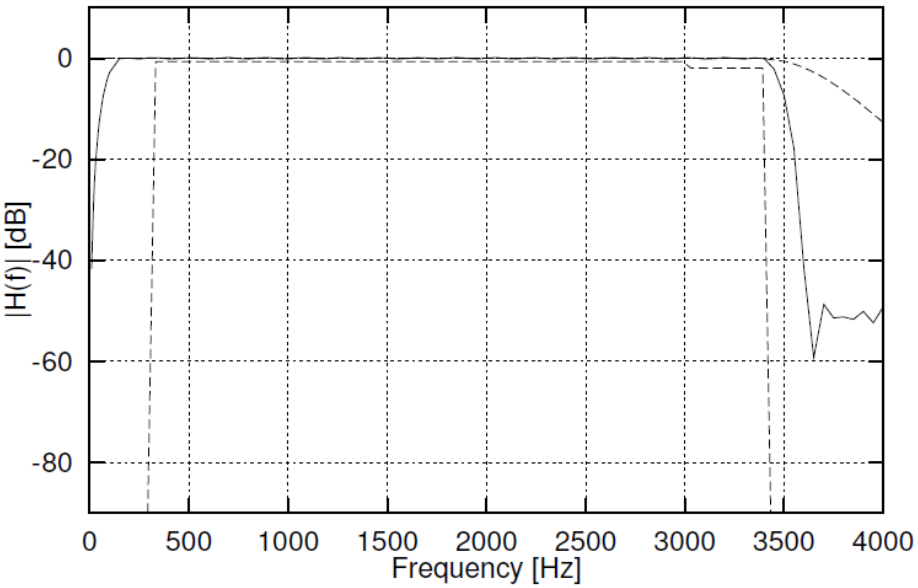



**Figure 25-a — High-quality bandpass for up-sampling (factor 1:2)**

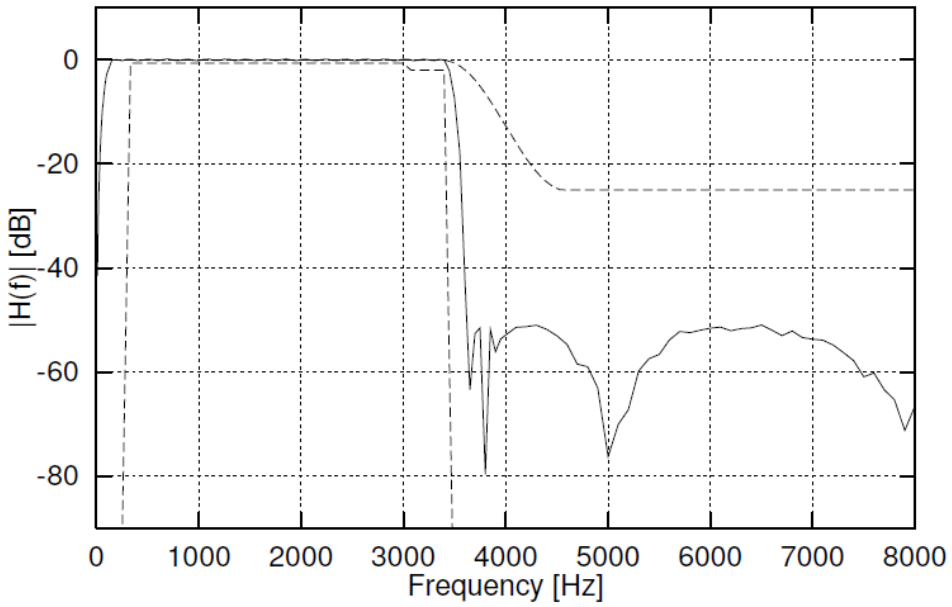



**Figure 25-b — High-quality bandpass for 2:1 down-sampling or for 1:1 filtering**

**Figure 25 — High-quality bandpass filter responses.**
**Mask shown is that of the G.712 filter, for reference**

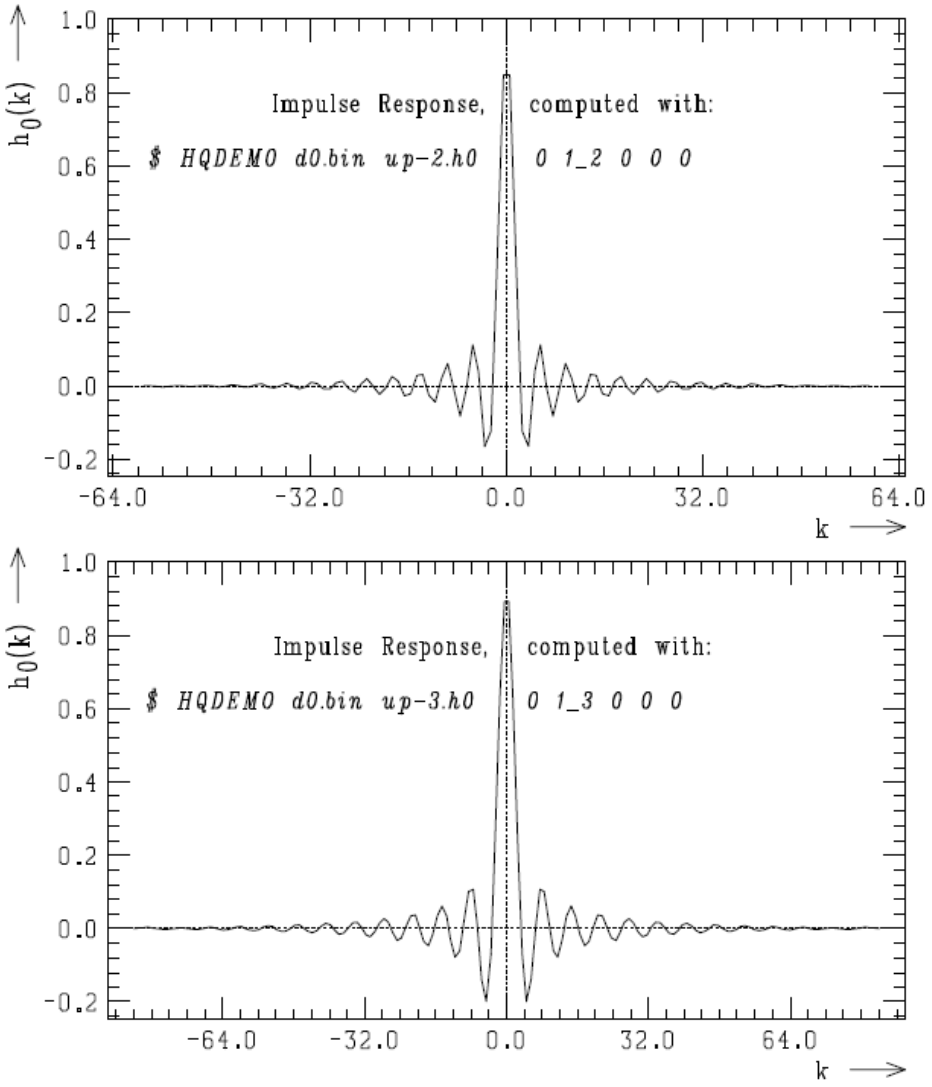



**Figure 26 — Impulse response for high-quality up-sampling filters (top, factor of 2; bottom, factor of 3)**

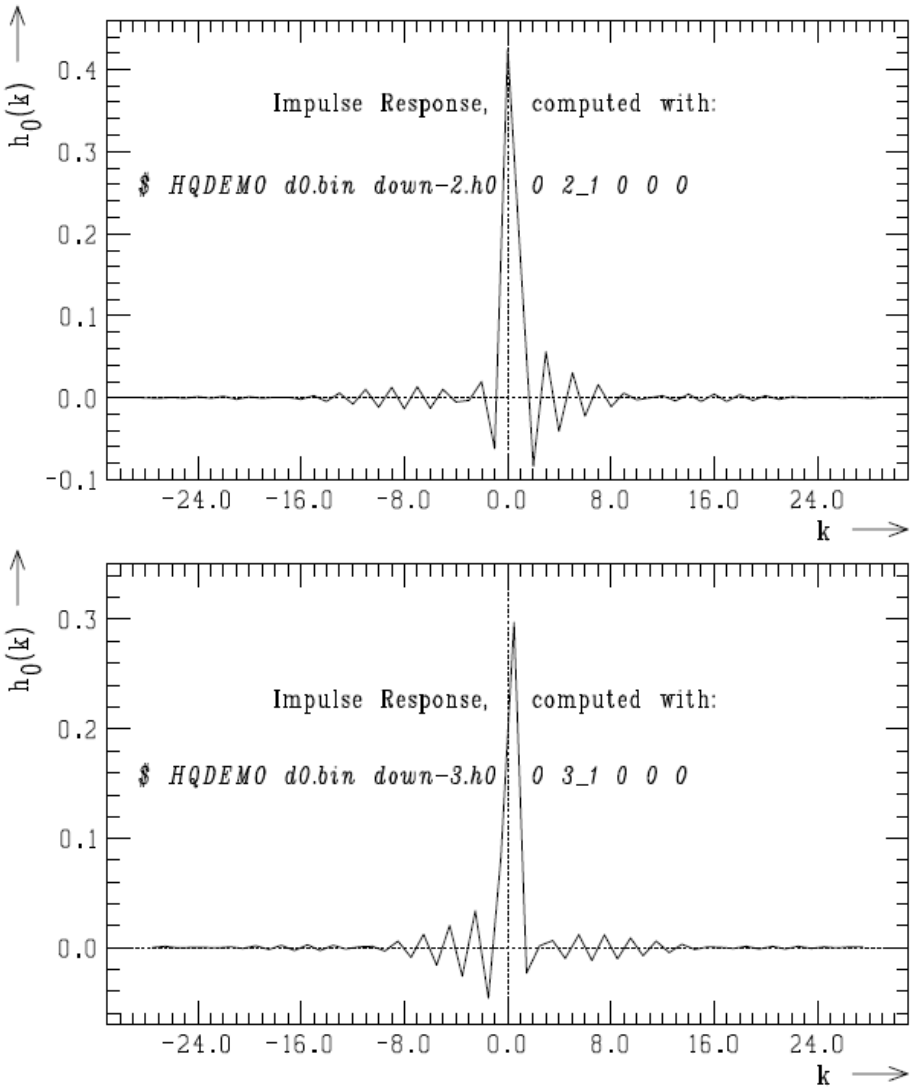



**Figure 27 — Impulse response for high-quality down-sampling filters (top, factor of 2; bottom, factor of 3)**

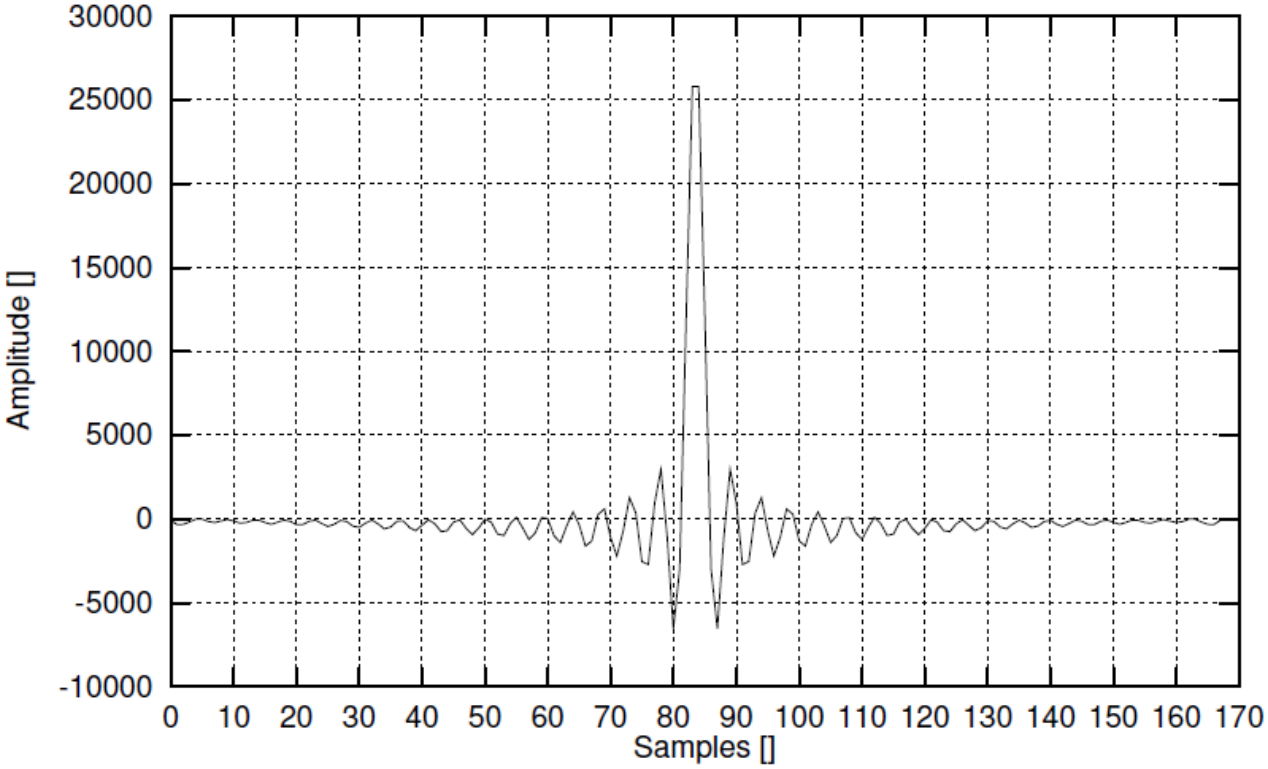



**Figure 28-a — High-quality bandpass for up-sampling (factor 1:2)**

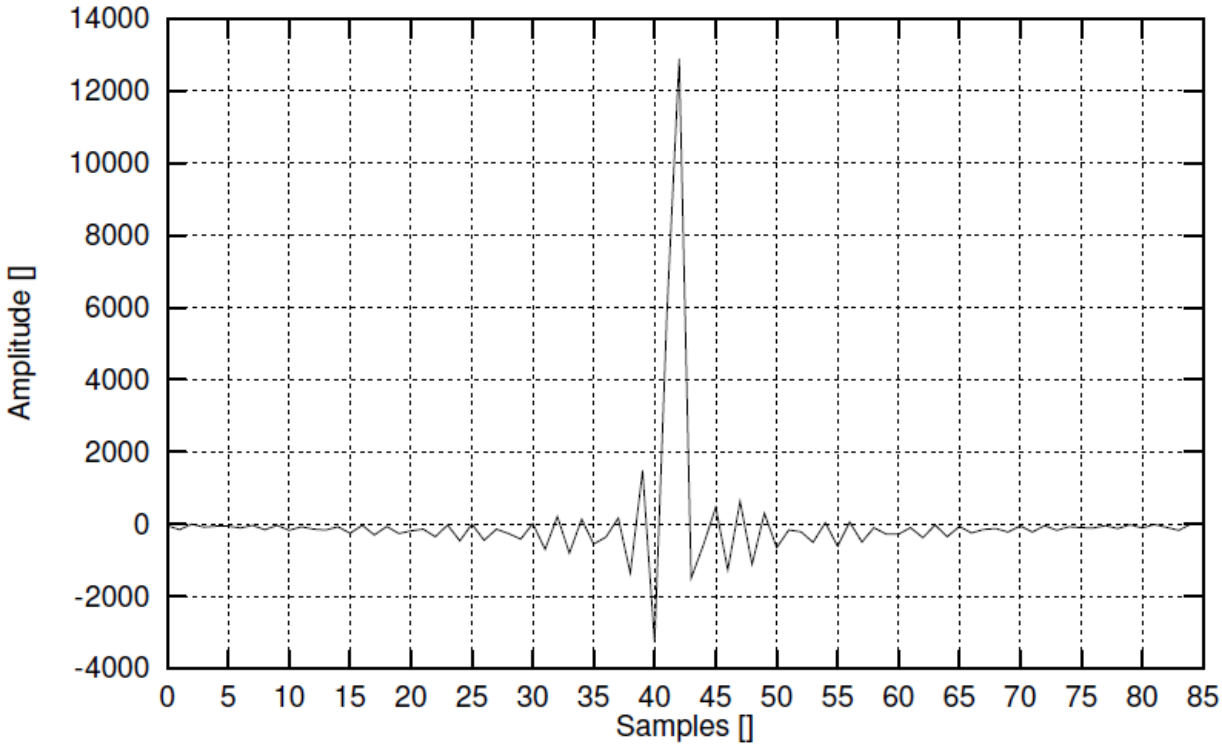



**Figure 28-b — High-quality bandpass for down-sampling (factor 2:1)**

**Figure 28 — Impulse response for high-quality bandpass filter (factors 2:1 and 1:1). Top is up-sampling by a factor of 1:2, and bottom is down-sampling by a factor of 2:1**

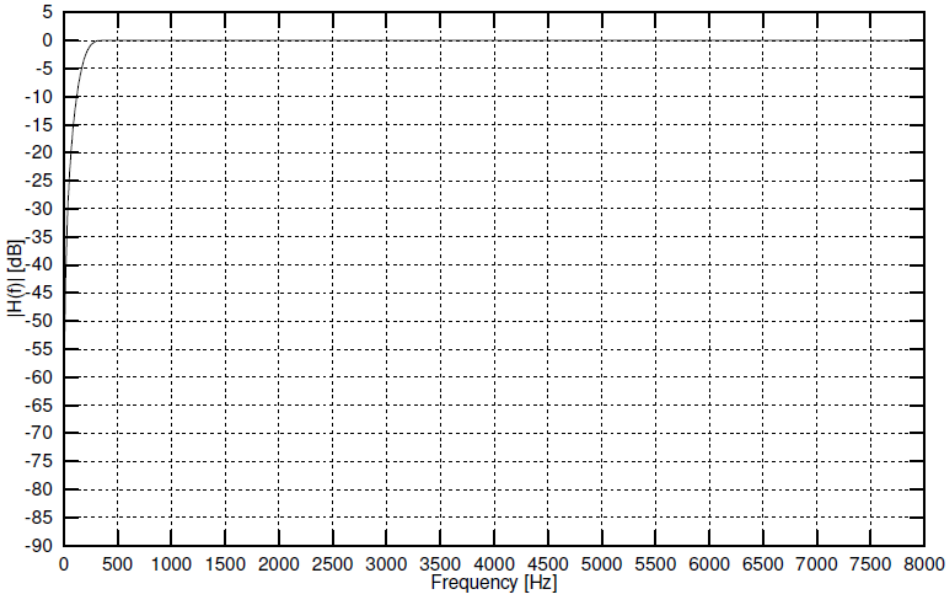



**Figure 29 — STL mobile station input (MSIN) frequency response for data sampled at 16 kHz (factor 1:1)**

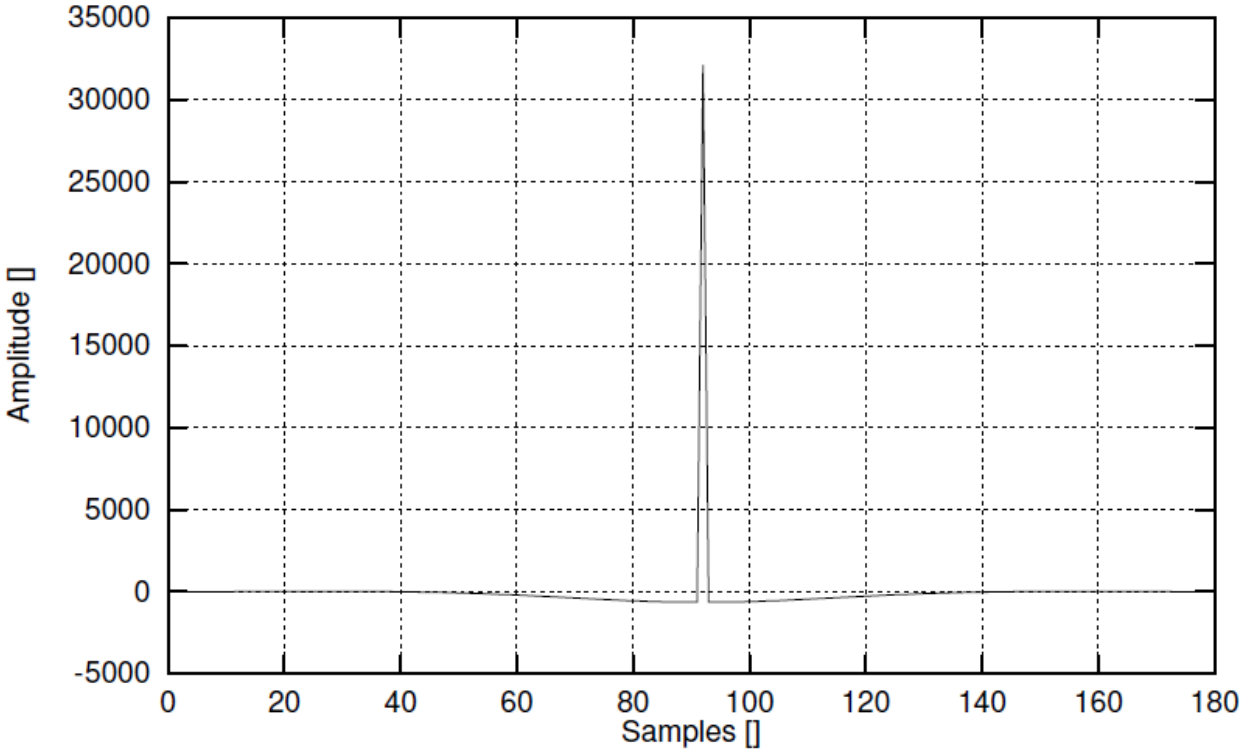



**Figure 30 — STL MSIN send-side filter impulse response**

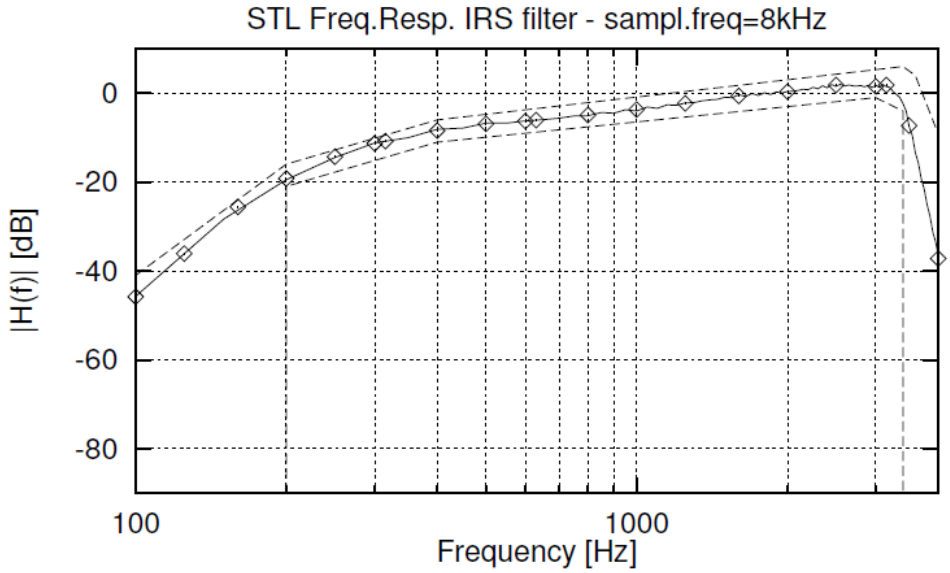



**Figure 31-a — Transmission-side IRS for input samples at 8 kHz**

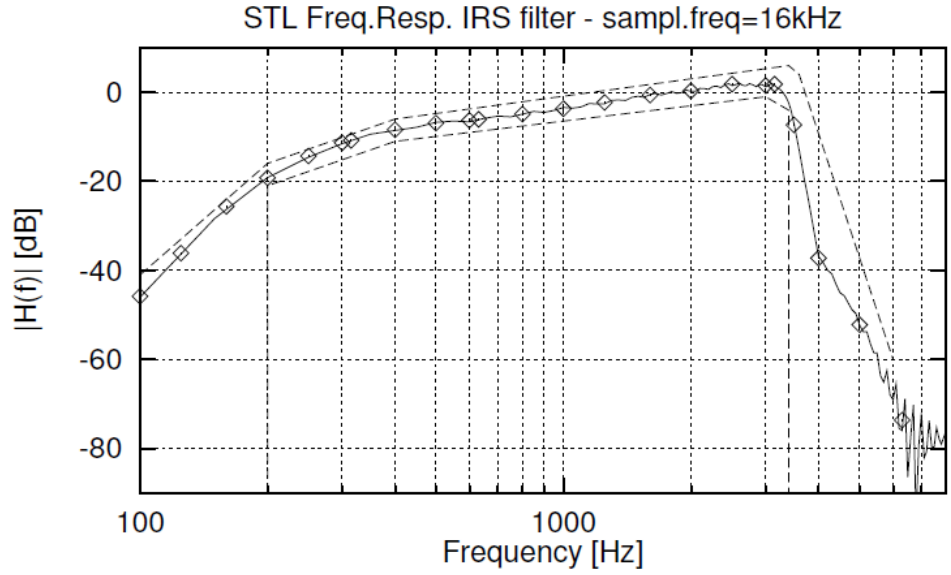



**Figure 31-b — Transmission-side IRS for input samples at 16 kHz**

**Figure 31 — Transmission-side IRS filter responses. The diamonds represent the nominal values of the "full" IRS characteristic and the interrupted line represent the mask of the "full" IRS, as shown in figure 2 of ITU-T Rec. P.48**

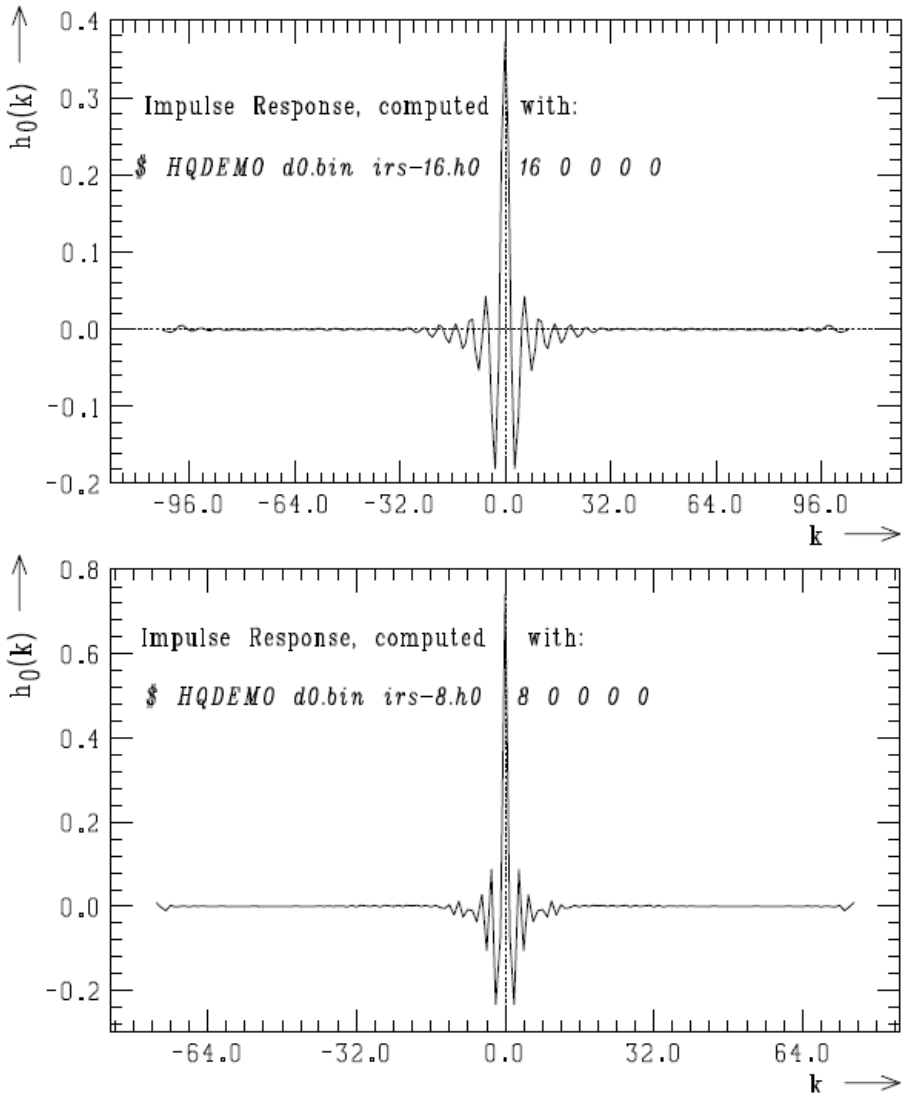



**Figure 32 — Impulse response of transmission-side IRS filters at 16 and 8 kHz**

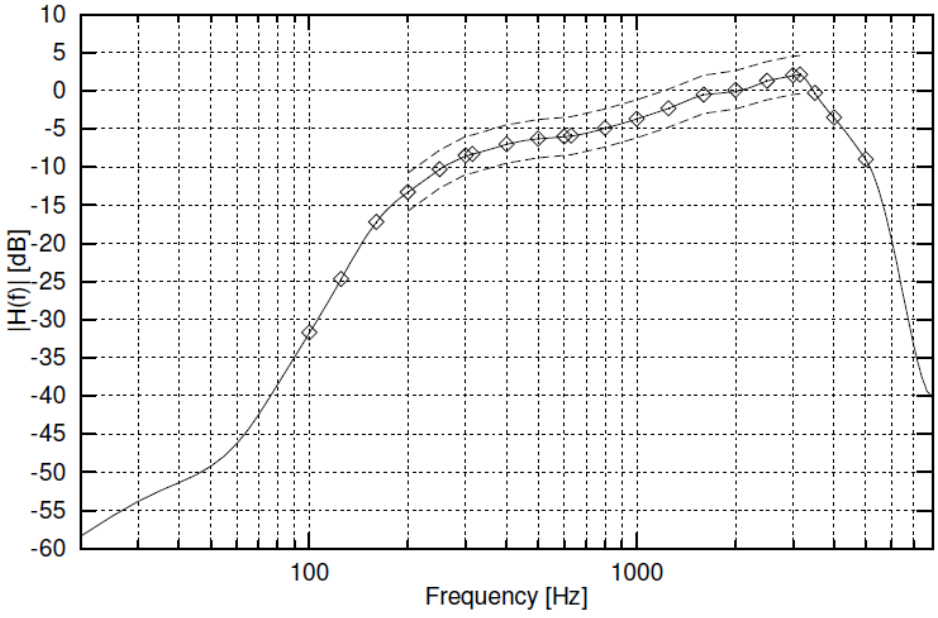



**Figure 33-a — Transmission-side modified IRS for input samples at 16 kHz**

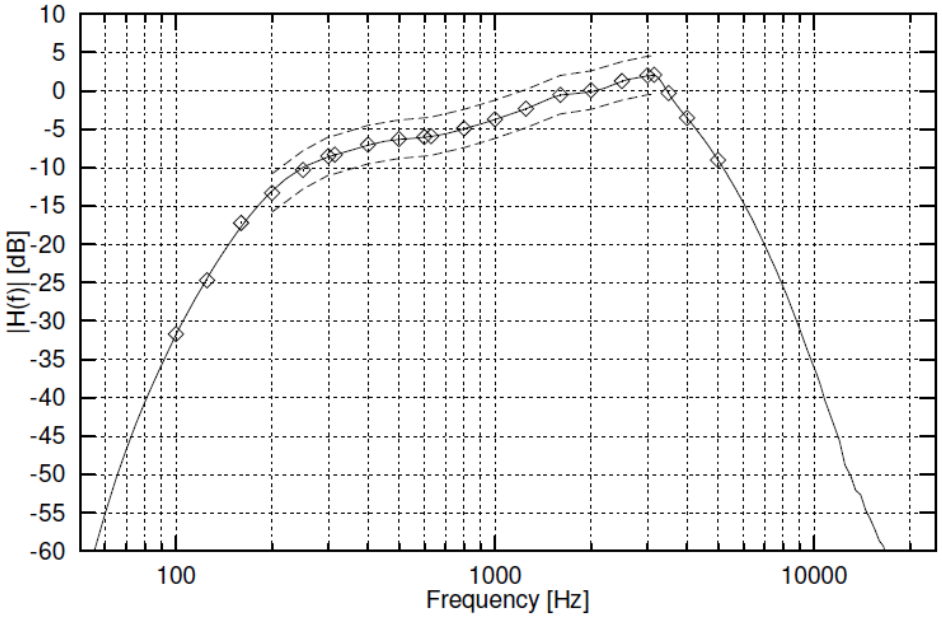



**Figure 33-b — Transmission-side modified IRS for input samples at 48 kHz**

**Figure 33 — Transmission-side modified IR filter responses.**
**The interrupted line represents the mask of the "full" IRS**

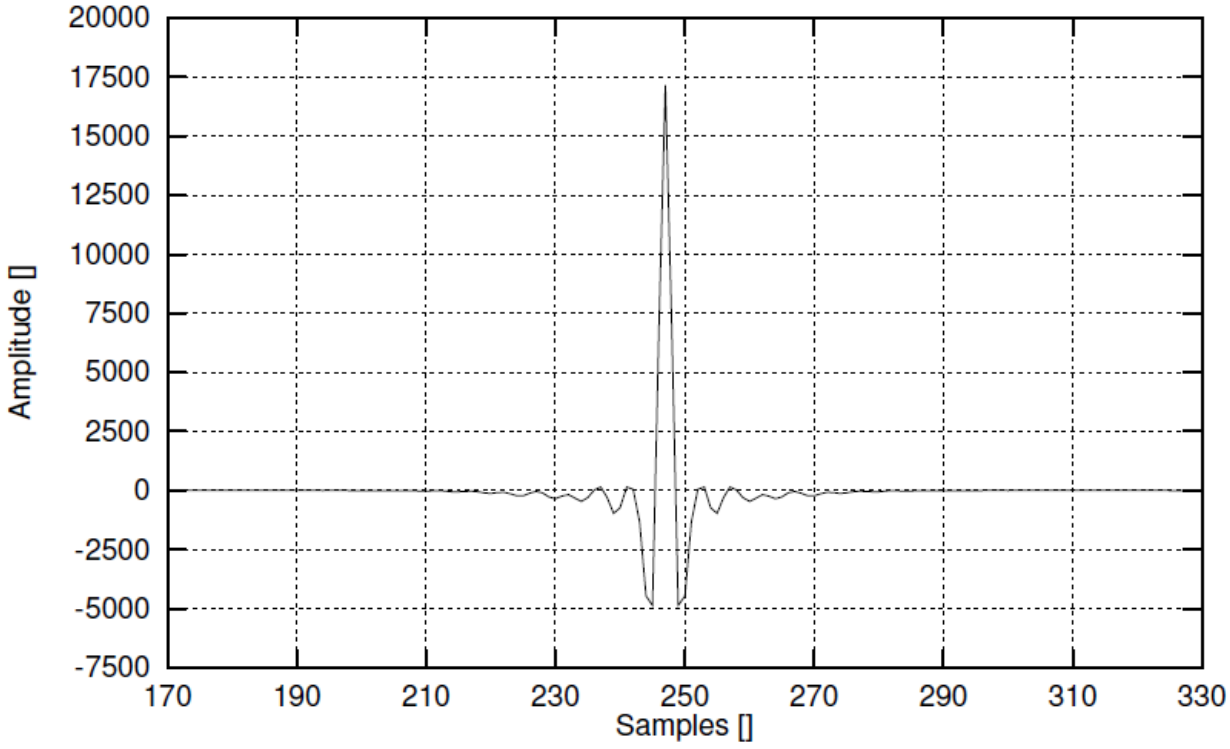



**Figure 34-a — Transmission-side modified IRS for input samples at 16 kHz**

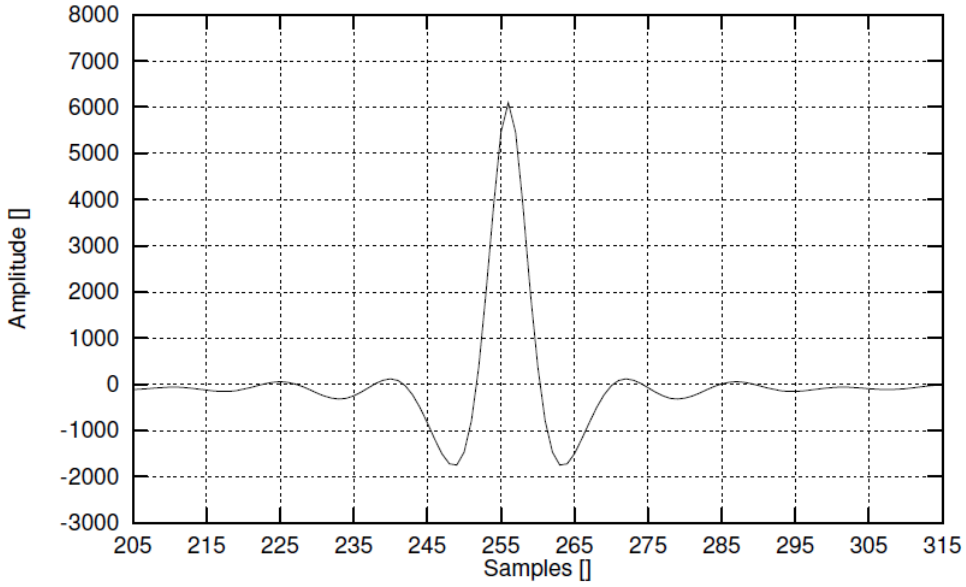



**Figure 34-b — Transmission-side modified IRS for input samples at 48 kHz**

**Figure 34 — Impulse response of transmission-side modified IRS filters at 16 kHz and 48 kHz**

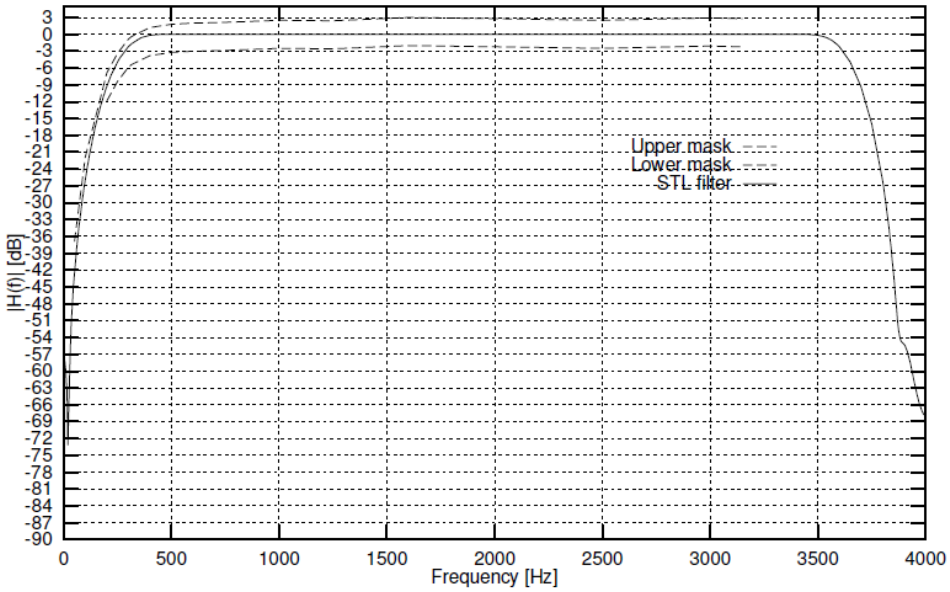



**Figure 35-a — Receive-side modified IRS for input samples at 8 kHz**

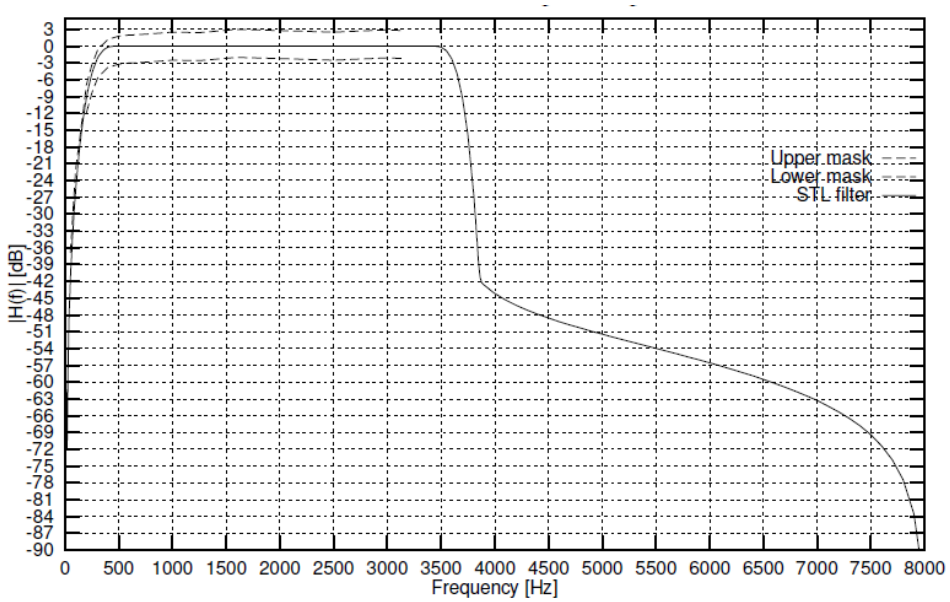



**Figure 35-b — Receive-side modified IRS for input samples at 16 kHz**

**Figure 35 — Receive-side modified IRS filter responses. The diamonds represent the nominal values of the modified IRS characteristic and the interrupted line represent the mask of the "full" IRS**

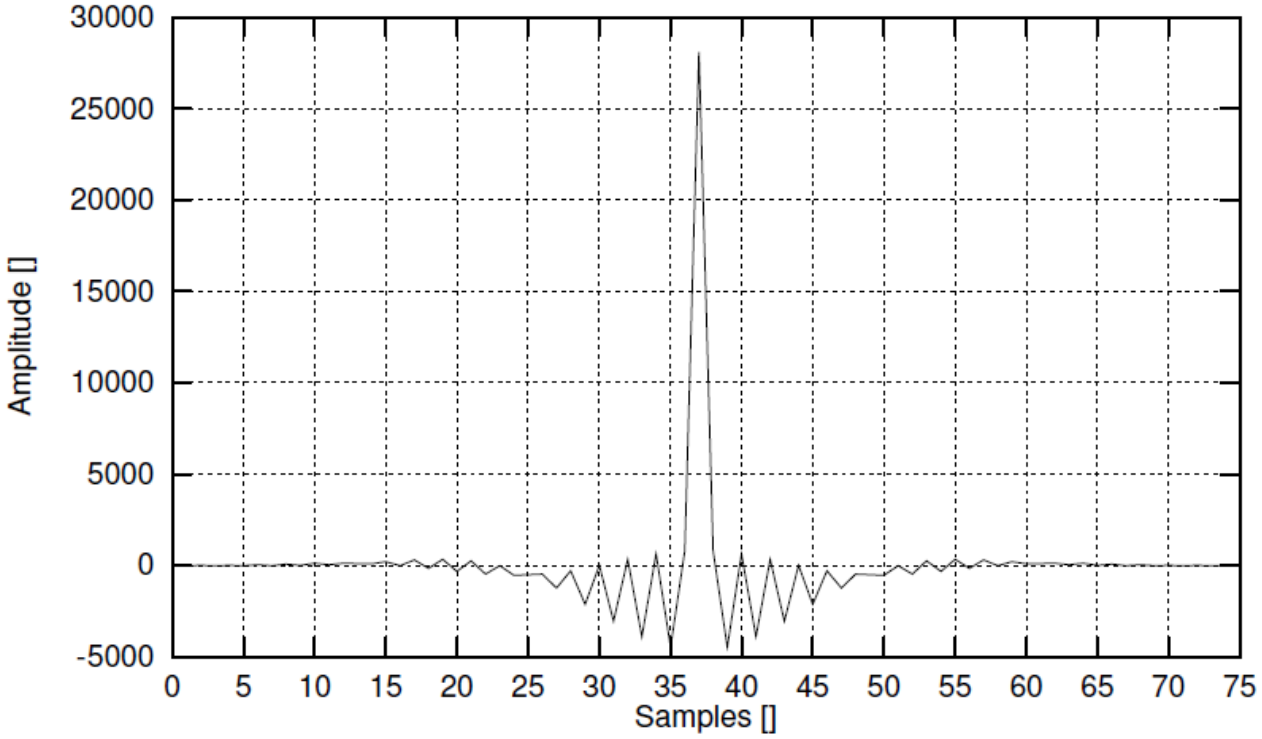



**Figure 36-a — Receive-side modified IRS for input samples at 8 kHz**

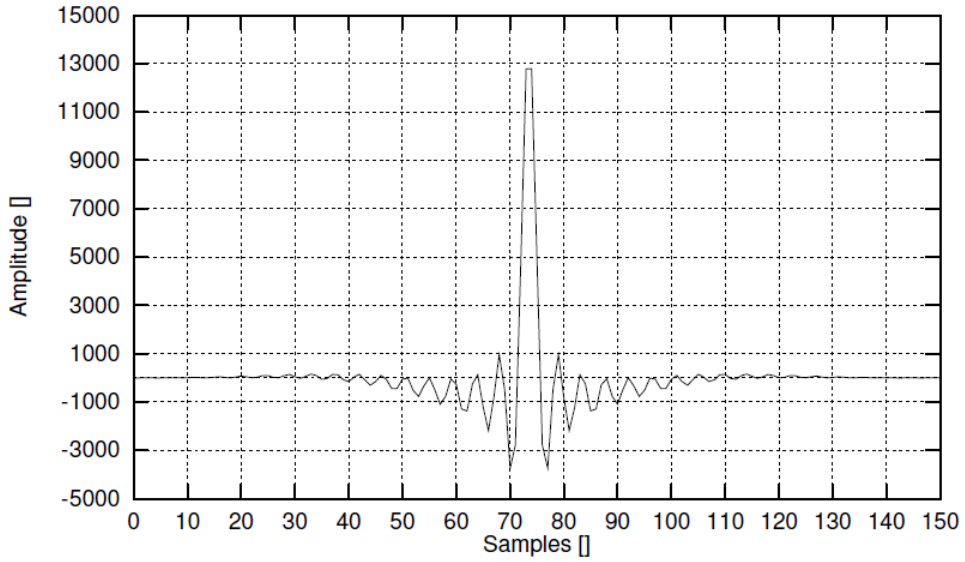



**Figure 36-b — Receive-side modified IRS for input samples at 16 kHz**

**Figure 36 — Impulse response of receive-side modified IRS filters at 8 kHz (top) and 16 kHz (bottom)**

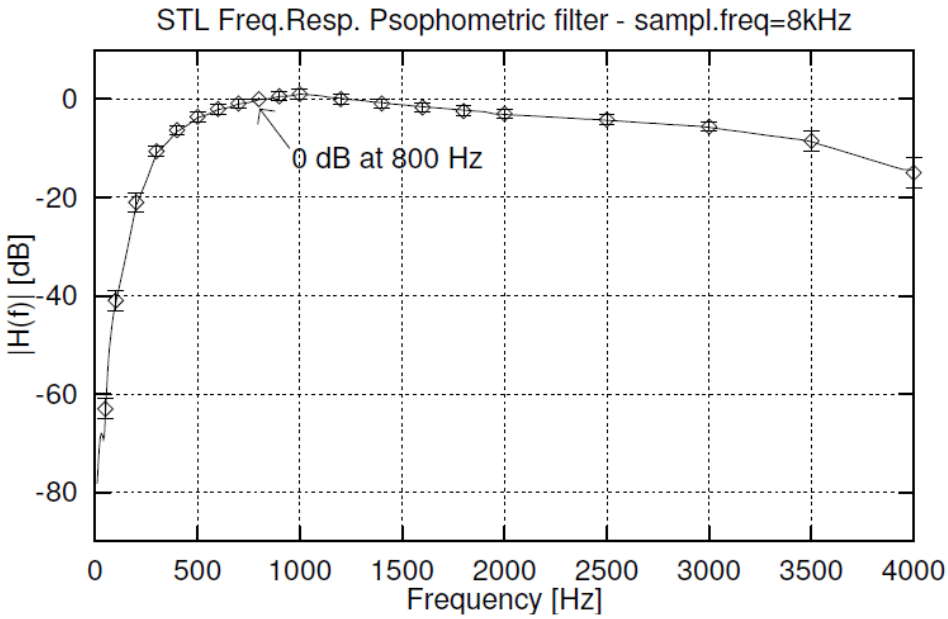



**Figure 37 — Frequency response for the psophometric filter. The points show the average points and the allowed range as per ITU-T Rec. O.41**

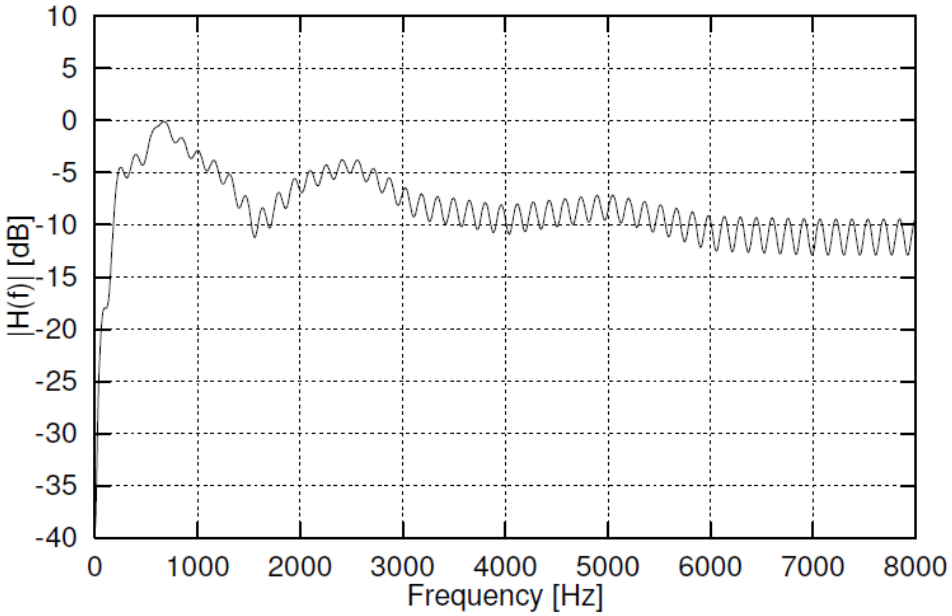



**Figure 38 — Frequency response for the $\Delta_{SM}$ filter.**

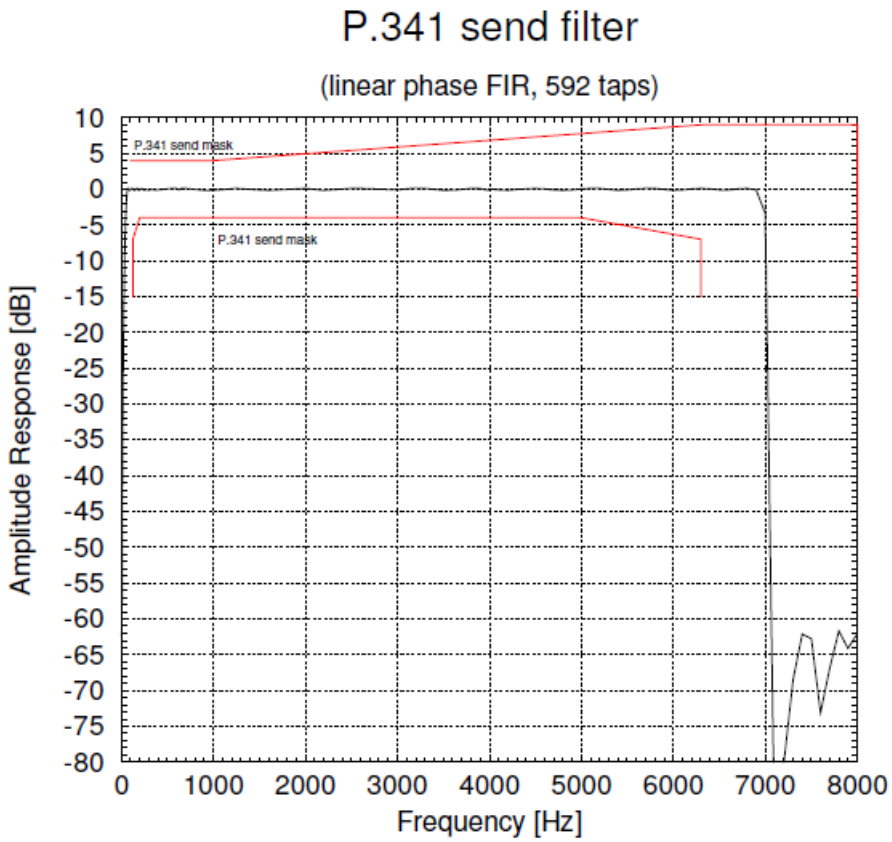



**Figure 39 — STL P.341 send-side filter frequency response for data sampled at 16 kHz (factor 1:1)**

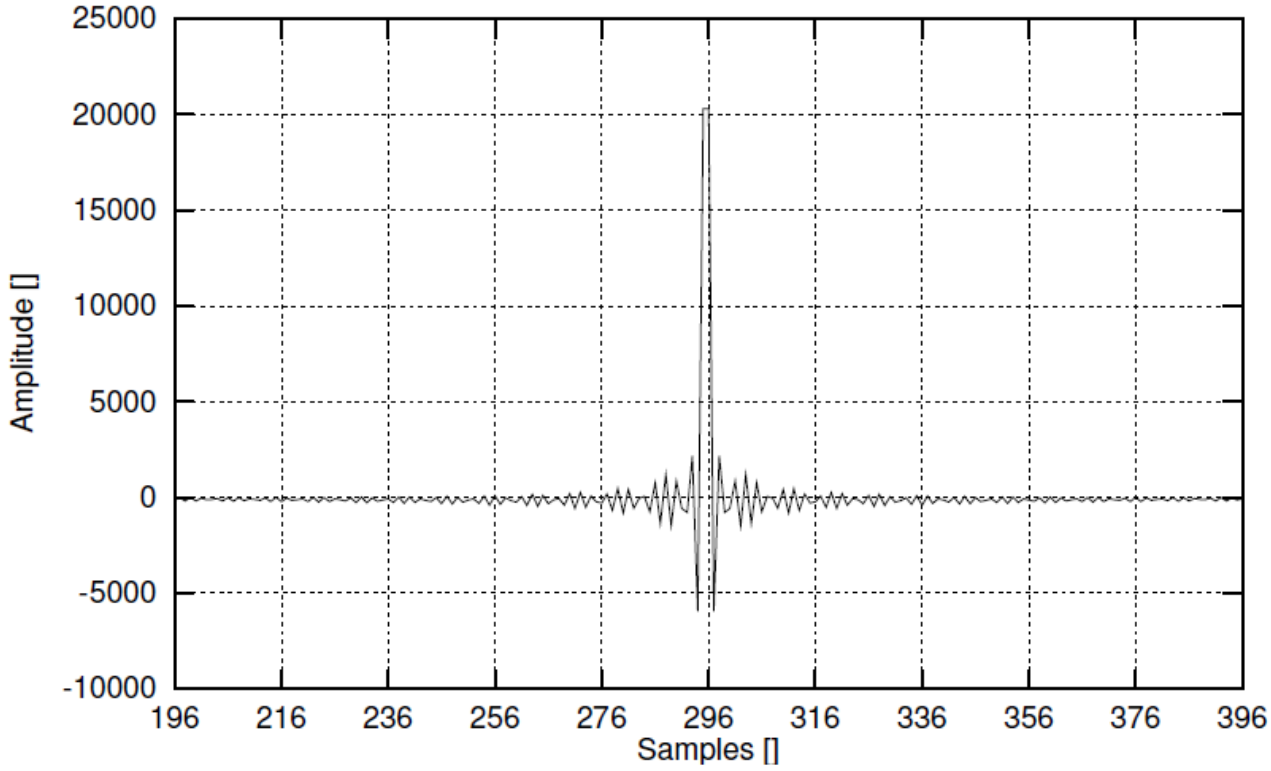



**Figure 40 — STL P.341 send-side filter impulse response**

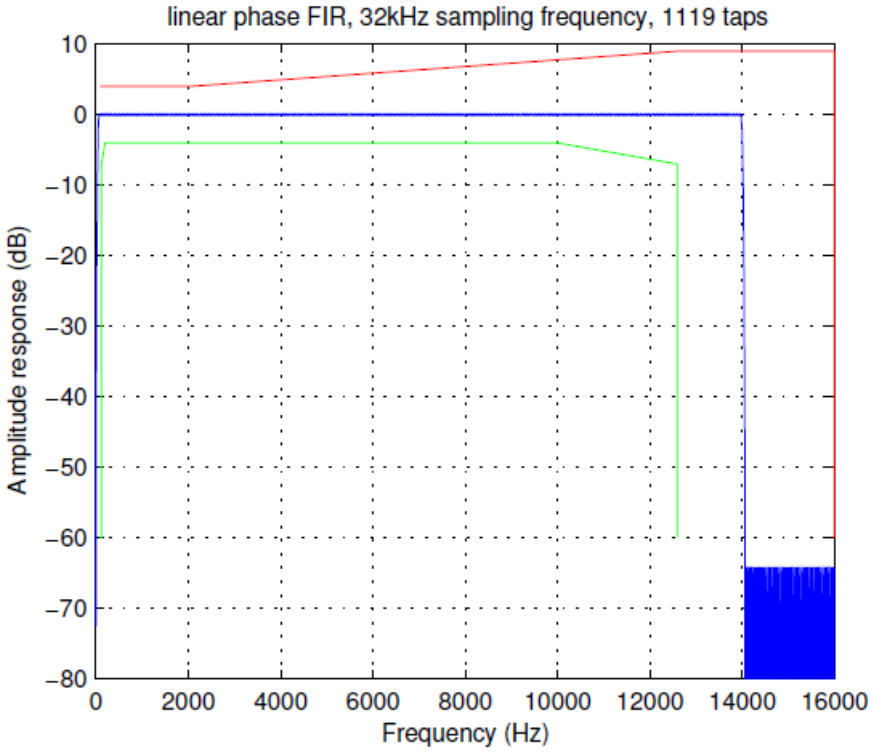



**Figure 41 — STL 50 Hz - 14 kHz band limiting filter frequency response for data sampled at 32 kHz (factor 1:1)**

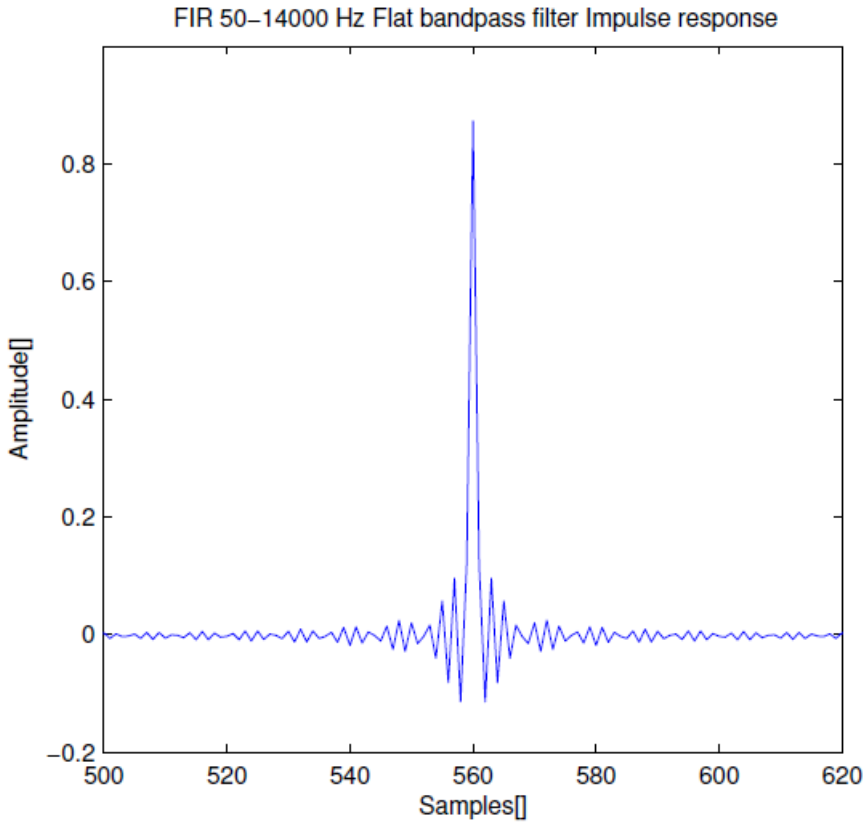



**Figure 42 — STL 50 Hz - 14 kHz band limiting filter impulse response**

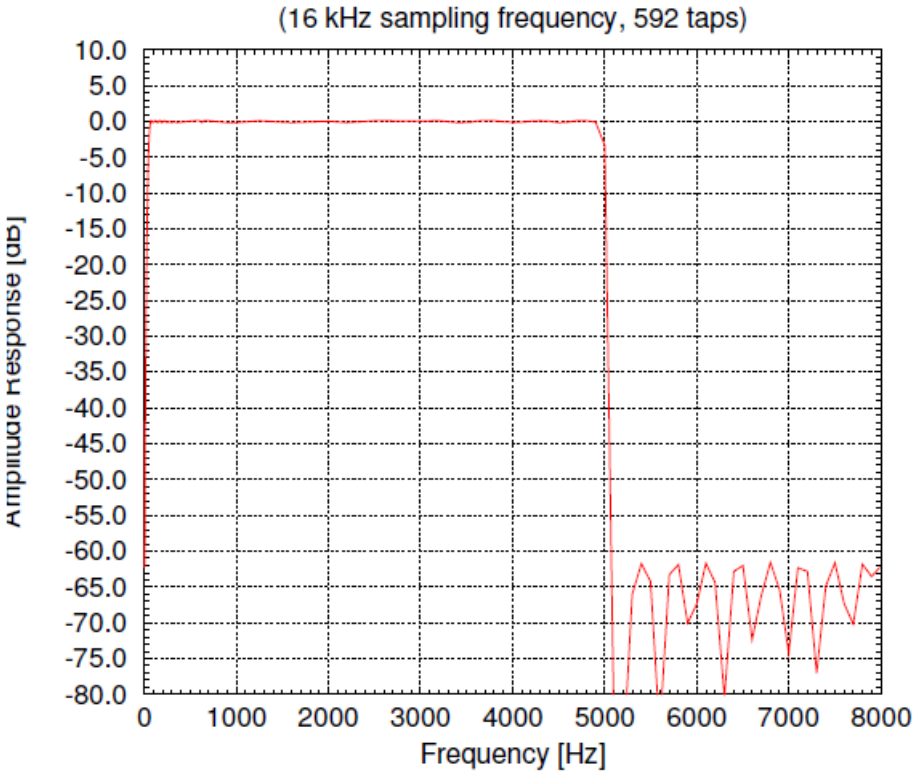



**Figure 43 — STL 50 Hz - 5 kHz band limiting filter frequency response for data sampled at 16 kHz (factor 1:1)**

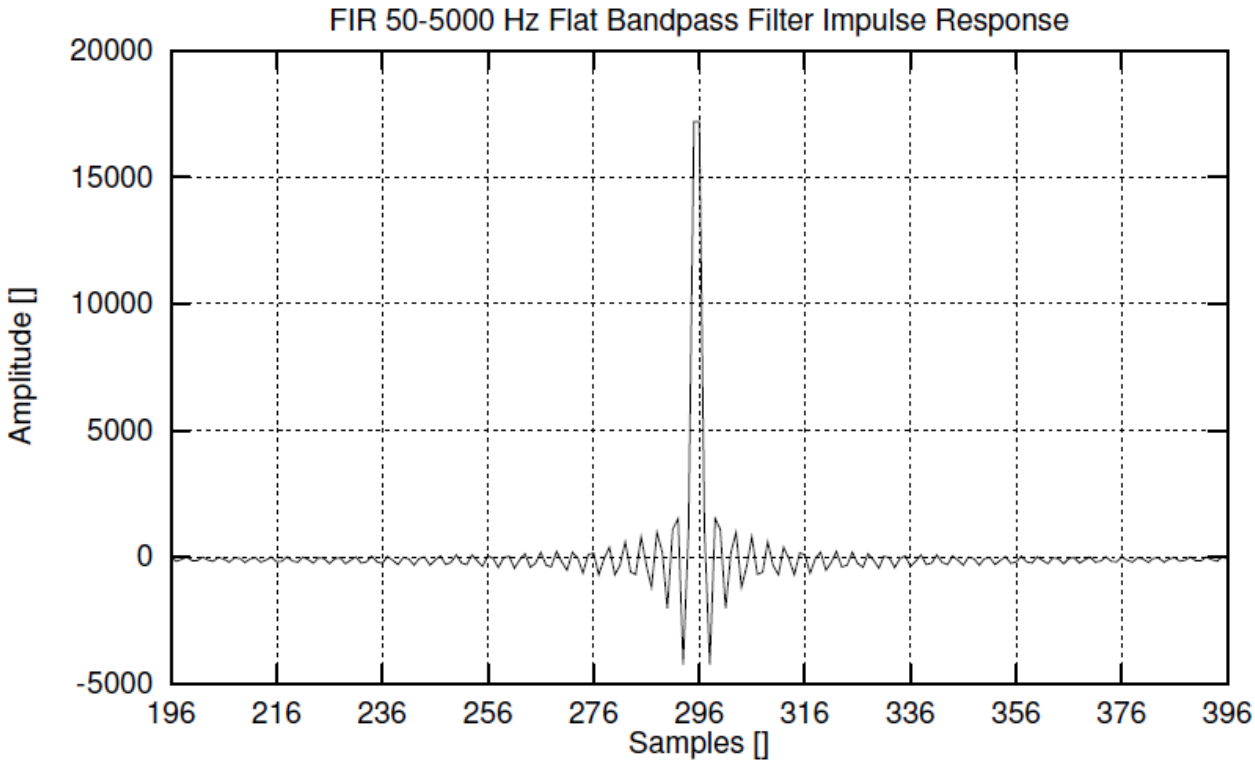



**Figure 44 — STL 50 Hz - 5 kHz band limiting filter impulse response**

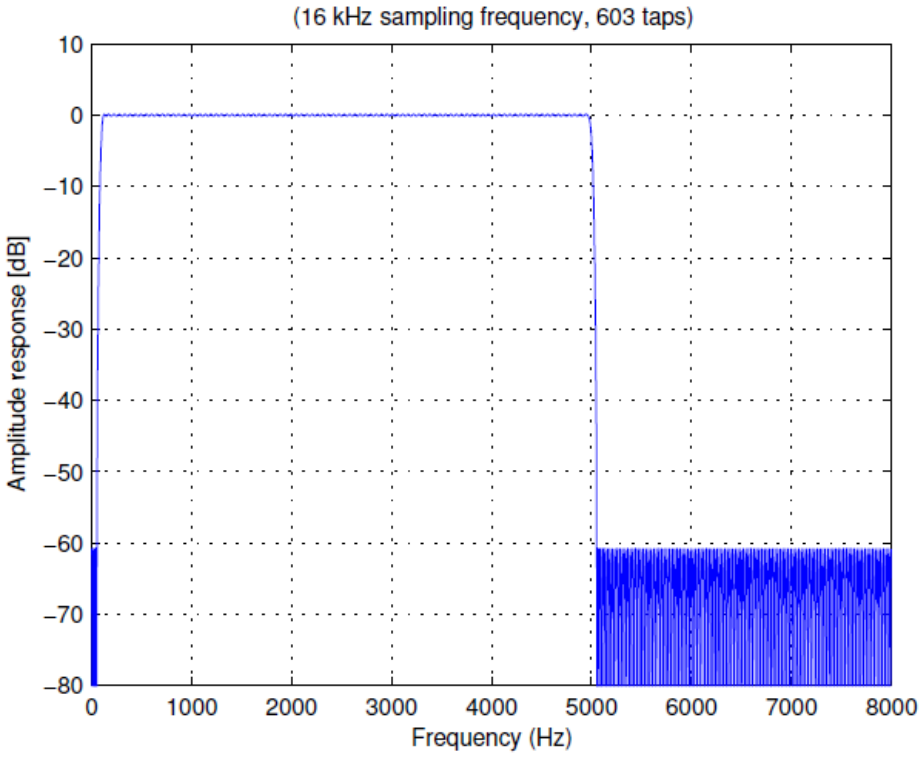



**Figure 45 — STL 100 Hz - 5 kHz band limiting filter frequency response for data sampled at 16 kHz (factor 1:1)**

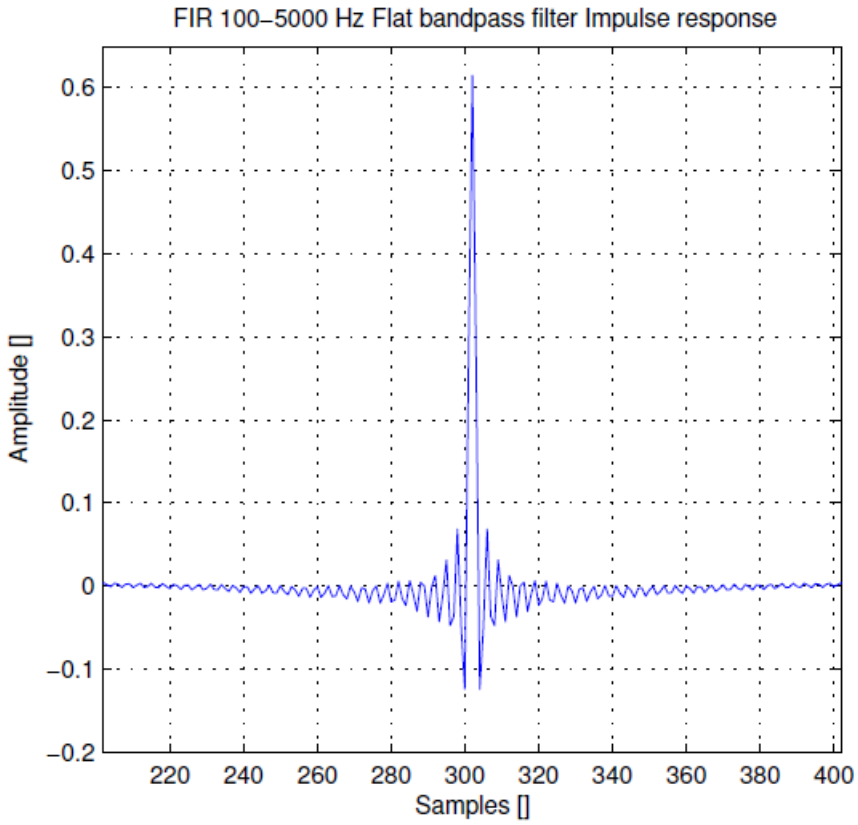



**Figure 46 — STL 100 Hz - 5 kHz band limiting filter impulse response**

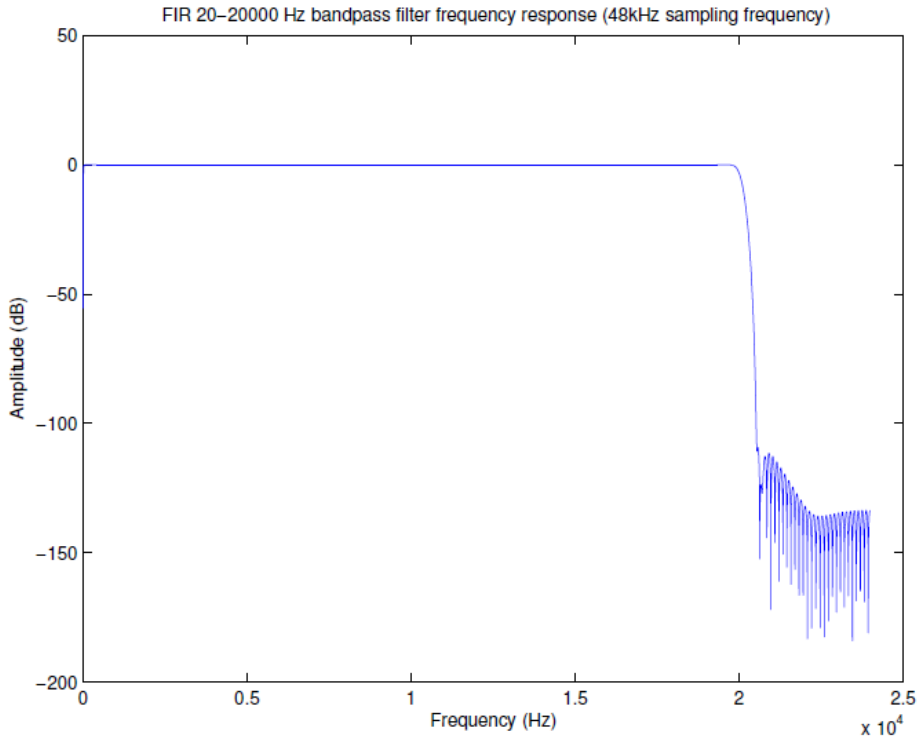



**Figure 47 — STL 20 Hz - 20 kHz band limiting filter frequency response for data sampled at 48 kHz (factor 1:1)**

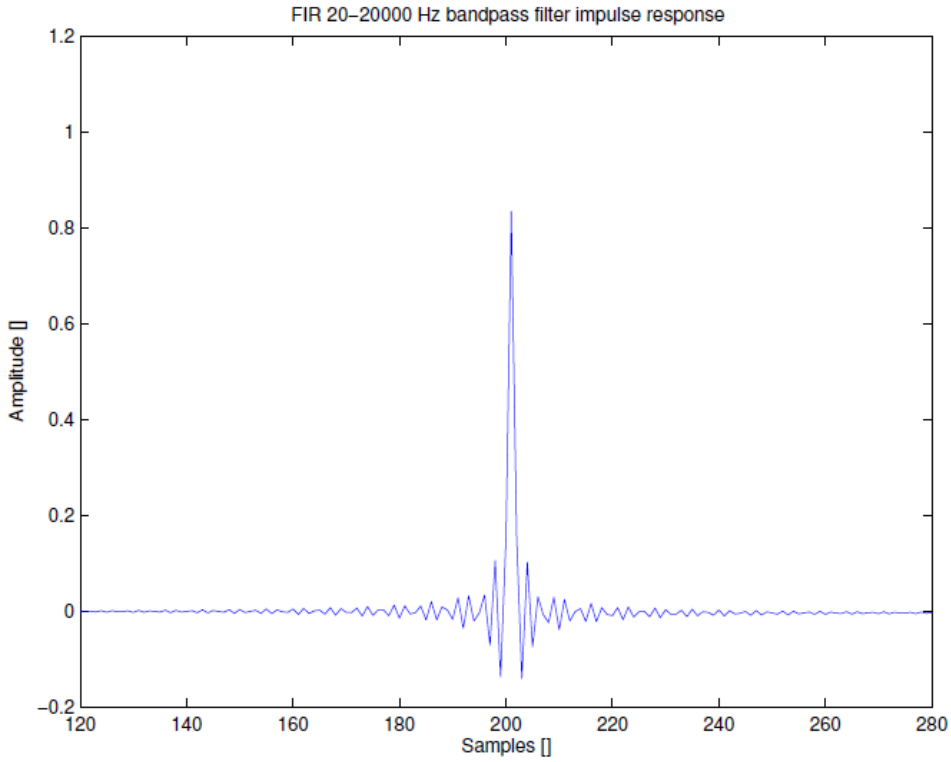



**Figure 48 — STL 20 Hz - 20 kHz band limiting filter impulse response, (the complete impulse response is of length 4001)**

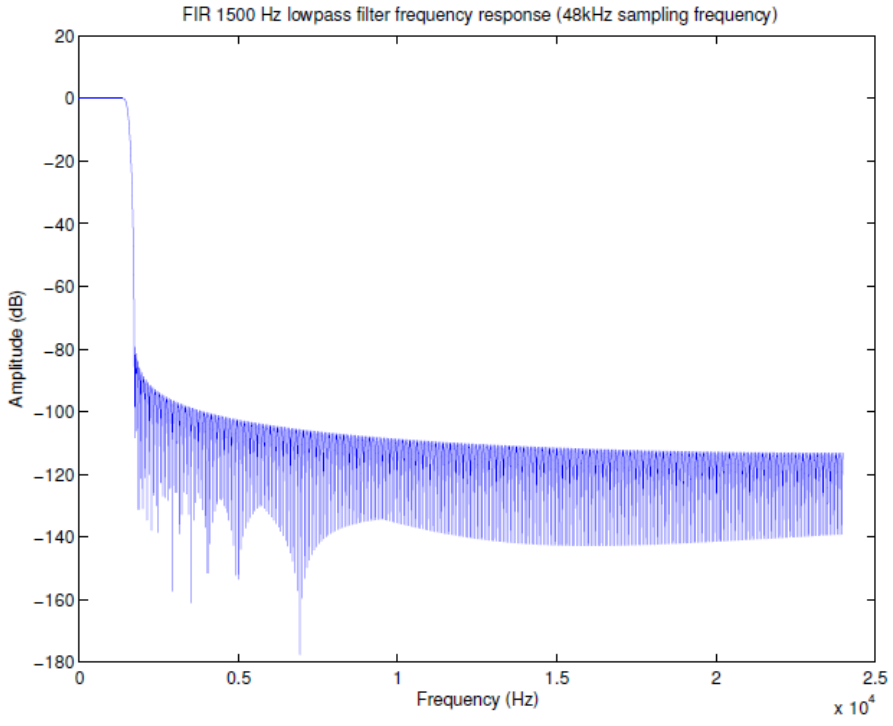



**Figure 49 — Frequency response of the STL MUSHRA anchor LP1.5—Low-pass filter with cut-off frequency 1.5 kHz for a sampling frequency of 48 kHz (factor 1:1)**

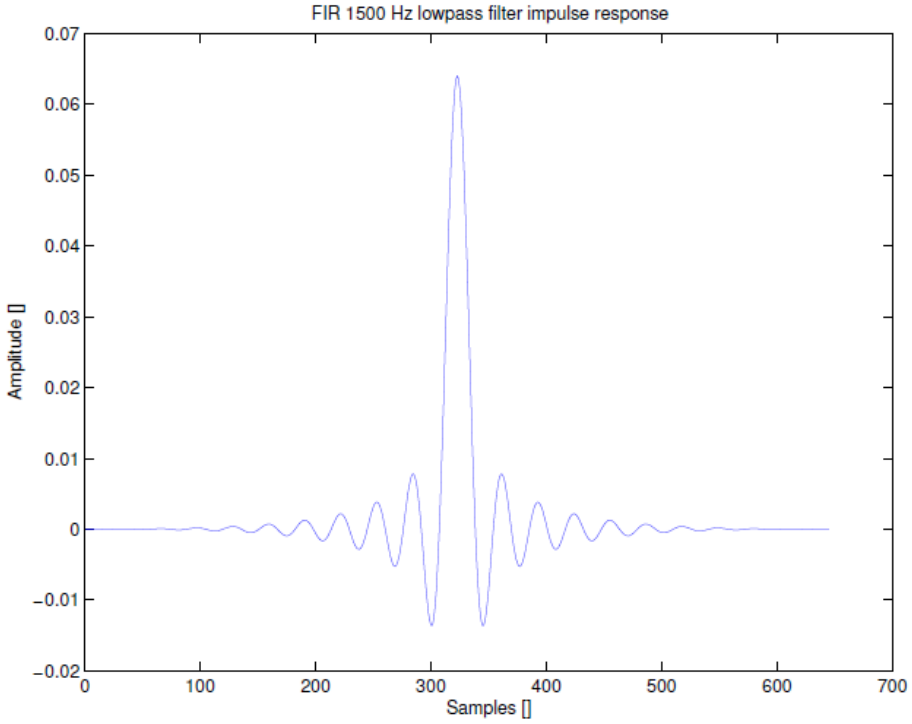



**Figure 50 — Impulse response of the STL MUSHRA anchor LP1.5—Low-pass filter with cut-off frequency 1.5 kHz for a sampling frequency of 48 kHz (factor 1:1)**

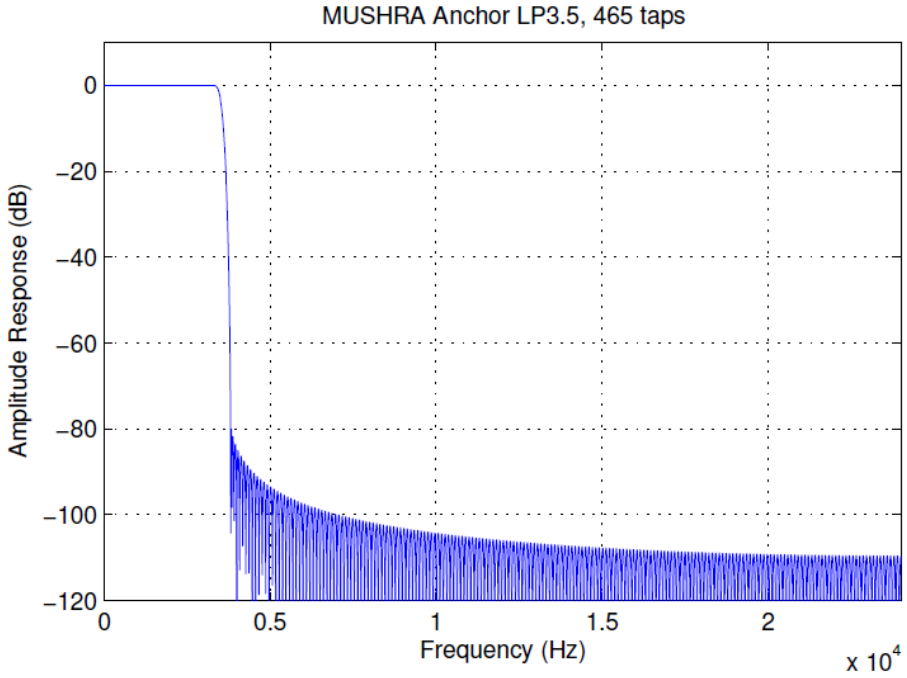



**Figure 51 — Frequency response of the STL MUSHRA anchor LP3.5—Low-pass filter with cut-off frequency 3.5 kHz for a sampling frequency of 48 kHz (factor 1:1)**

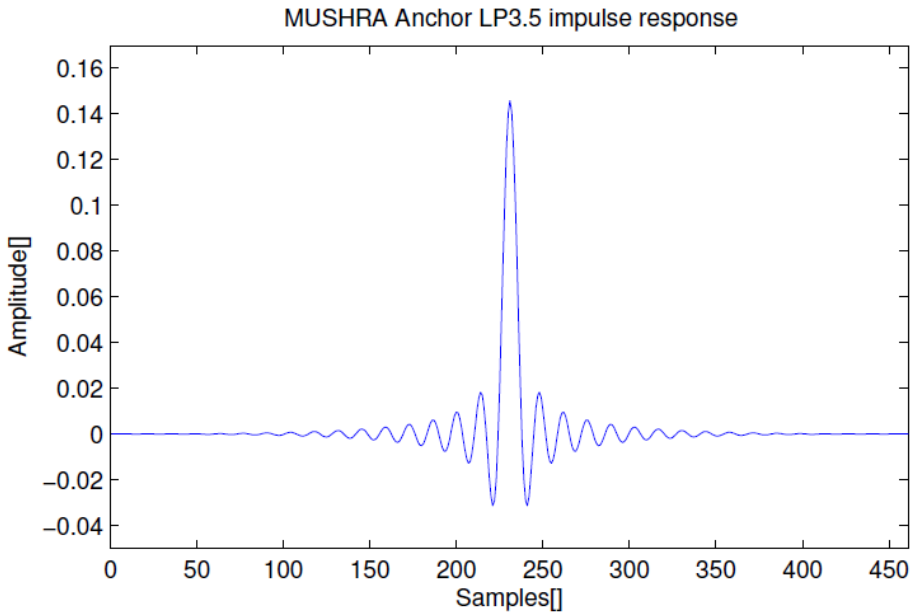



**Figure 52 — Impulse response of the STL MUSHRA anchor LP3.5—Low-pass filter with cut-off frequency 3.5 kHz for a sampling frequency of 48 kHz (factor 1:1)**

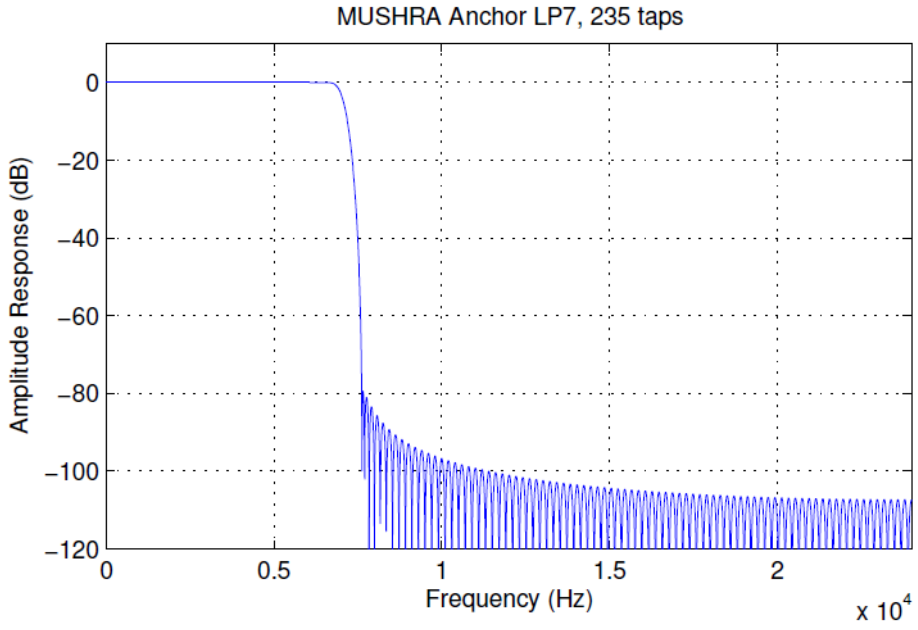



**Figure 53 — Frequency response of the STL MUSHRA anchor LP7—Low-pass filter with cut-off frequency 7 kHz for a sampling frequency of 48 kHz (factor 1:1)**

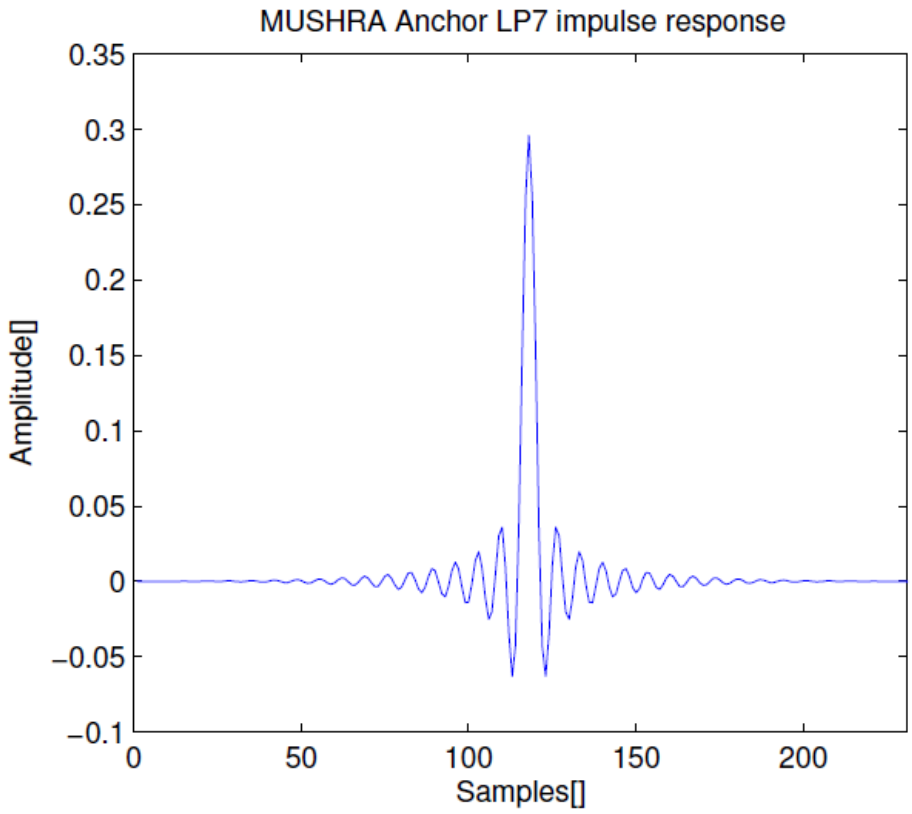



**Figure 54 — Impulse response of the STL MUSHRA anchor LP7 — Low-pass filter with cut-off frequency 7 kHz for a sampling frequency of 48 kHz (factor 1:1)**

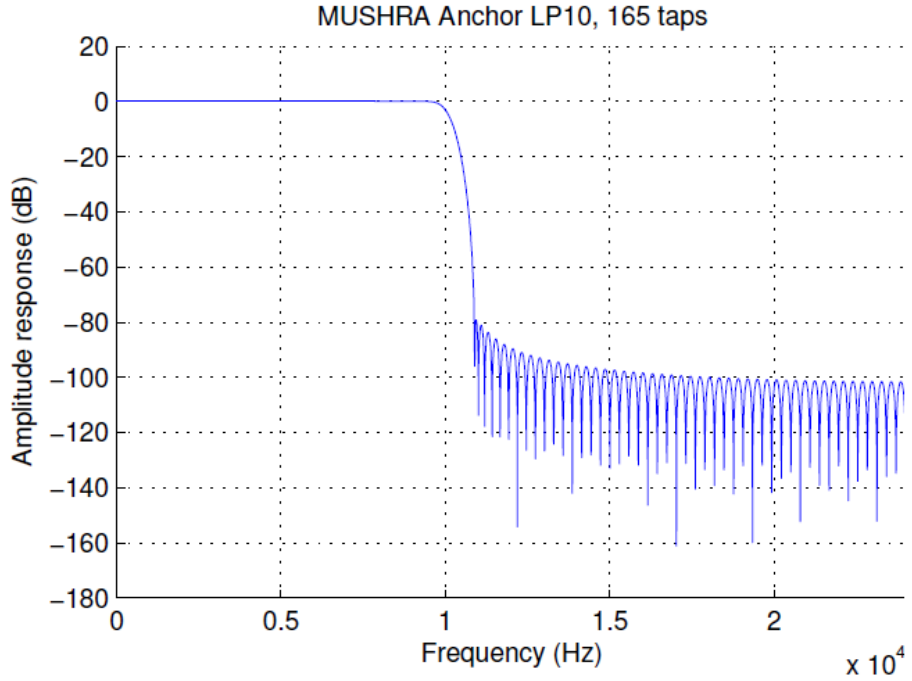



**Figure 55 — Frequency response of the STL MUSHRA anchor LP10 — Low-pass filter with cut-off frequency 10 kHz for a sampling frequency of 48 kHz (factor 1:1)**

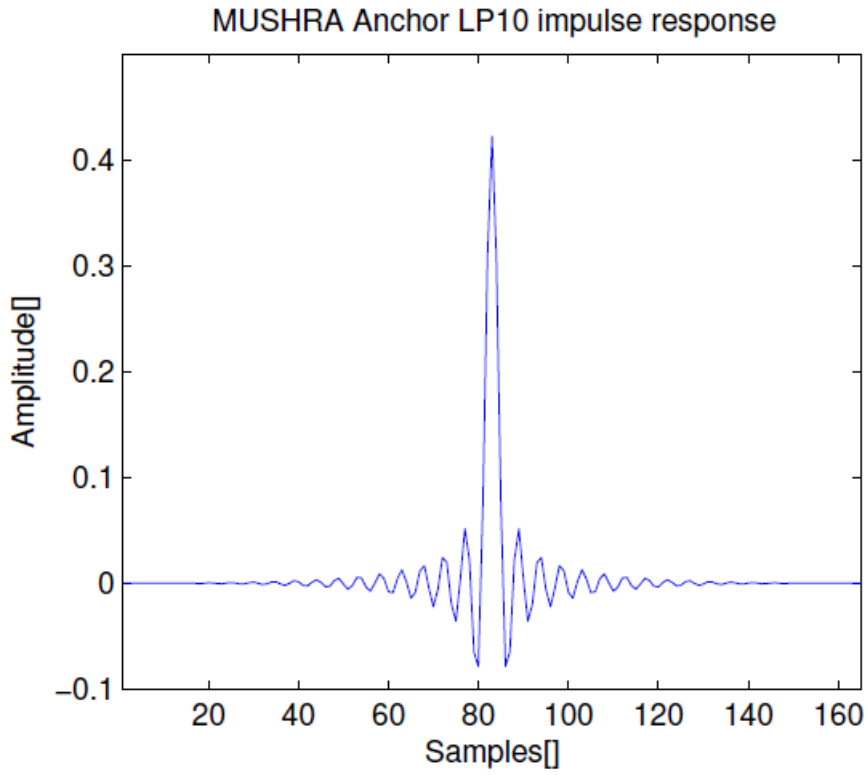



**Figure 56 — Impulse response of the STL MUSHRA anchor LP10 — Low-pass filter with cut-off frequency 10 kHz for a sampling frequency of 48 kHz (factor 1:1)**

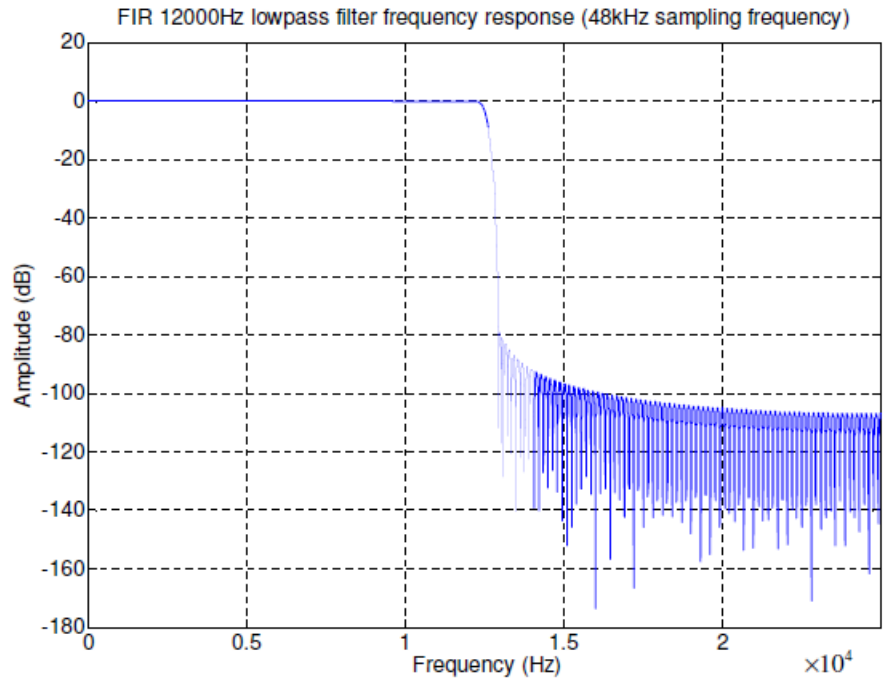



**Figure 57 — Frequency response of the STL MUSHRA anchor LP12 — Low-pass filter with cut-off frequency 12 kHz for a sampling frequency of 48 kHz (factor 1:1)**

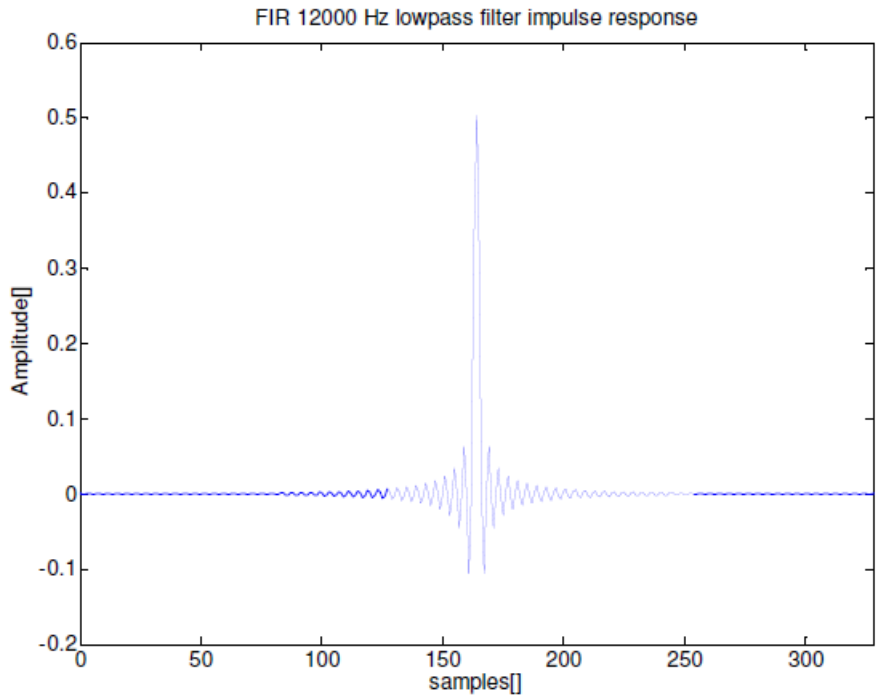



**Figure 58 — Impulse response of the STL MUSHRA anchor LP12 — Low-pass filter with cut-off frequency 12 kHz for a sampling frequency of 48 kHz (factor 1:1)**

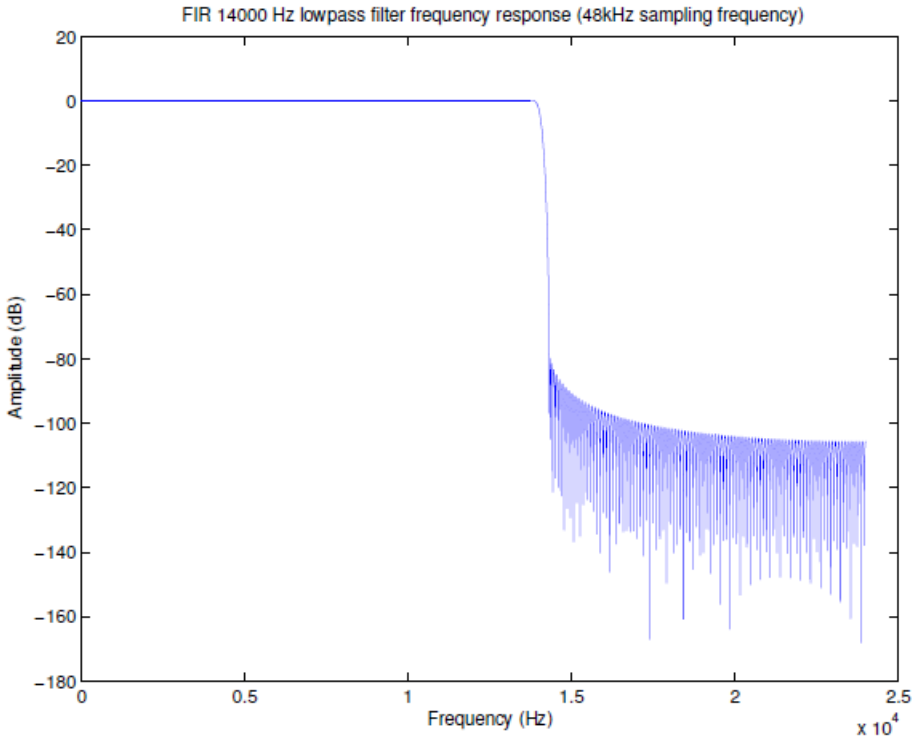



**Figure 59 — Frequency response of the STL MUSHRA anchor LP14 — Low-pass filter with cut-off frequency 14 kHz for a sampling frequency of 48 kHz (factor 1:1)**

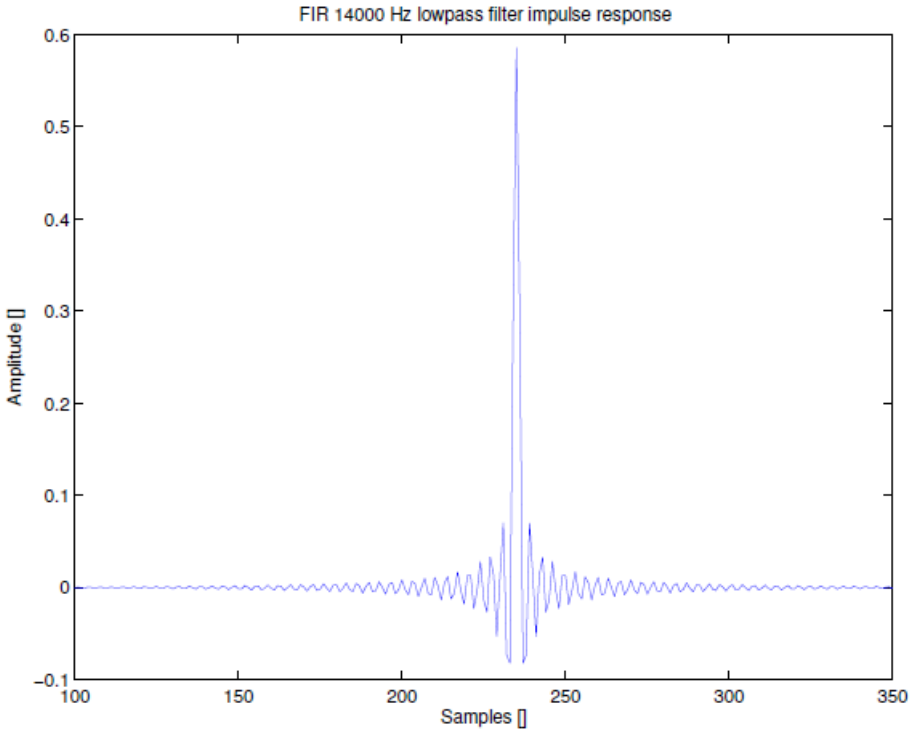



**Figure 60 — Impulse response of the STL MUSHRA anchor LP14—Low-pass filter with cut-off frequency 14 kHz for a sampling frequency of 48 kHz (factor 1:1)**

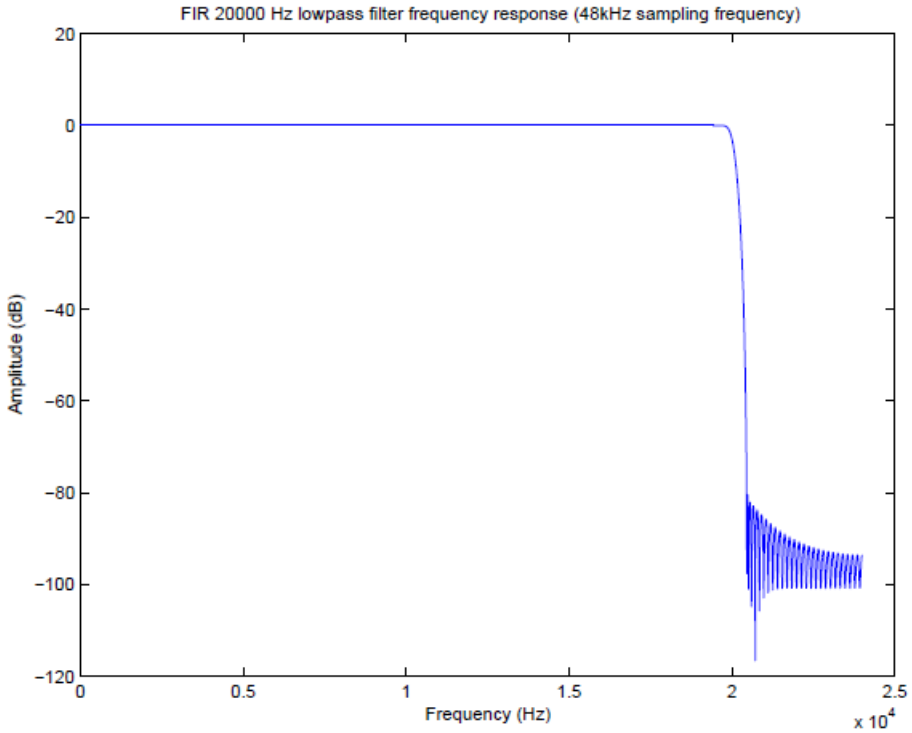



**Figure 61 — Frequency response of the STL MUSHRA anchor LP20—Low-pass filter with cut-off frequency 20 kHz for a sampling frequency of 48 kHz (factor 1:1)**

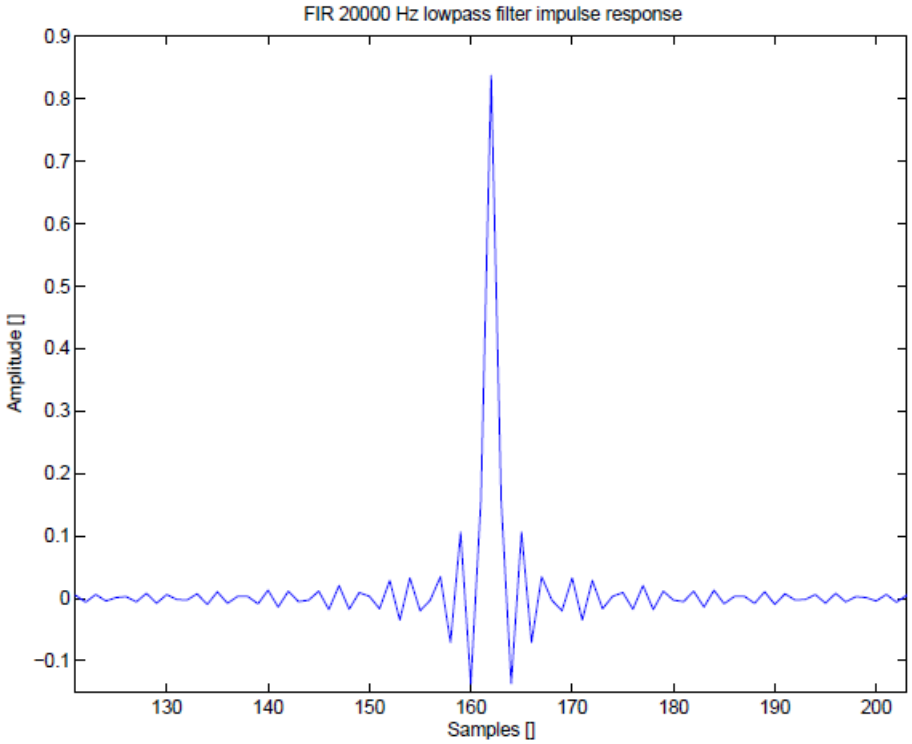



**Figure 62 — Impulse response of the STL MUSHRA anchor LP20—Low-pass filter with cut-off frequency 20 kHz for a sampling frequency of 48 kHz (factor 1:1)**

#### 14.2.1.1. `*_init` for the FIR module

**Syntax:**

```
#include "firflt.h"
SCD_FIR *delta_sm_16khz_init (void);
SCD_FIR *hq_down_2_to_1_init (void);
SCD_FIR *hq_up_1_to_2_init (void);
SCD_FIR *hq_down_3_to_1_init (void);
SCD_FIR *hq_up_1_to_3_init (void);
SCD_FIR *irs_8khz_init (void);
SCD_FIR *irs_16khz_init (void);
SCD_FIR *linear_phase_pb_2_to_1_init (void);
SCD_FIR *linear_phase_pb_1_to_2_init (void);
SCD_FIR *linear_phase_pb_1_to_1_init (void);
SCD_FIR *msin_16khz_init();
SCD_FIR *mod_irs_16khz_init (void);
SCD_FIR *mod_irs_48khz_init (void);
SCD_FIR *rx_mod_irs_8khz_init(void);
SCD_FIR *rx_mod_irs_16khz_init(void);
SCD_FIR *psophometric_8khz_init (void);
SCD_FIR *p341_16k_init (void);
SCD_FIR *bp14k_32khz_init (void);
SCD_FIR *bp5k_16k_init (void);
SCD_FIR *bp100_5k_16khz_init (void);
SCD_FIR *bp20k_48kHz_init (void);
SCD_FIR *LP1p5_48kHz_init (void);
SCD_FIR *LP35_48kHz_init (void);
SCD_FIR *LP7_48kHz_init (void);
SCD_FIR *LP10_48kHz_init (void);
SCD_FIR *LP12_48kHz_init (void);
SCD_FIR *LP14_48kHz_init (void);
SCD_FIR *L20_48kHz_init (void);
```

**Prototypes:** `firflt.h`

**Description:**

`delta_sm_16khz_init` is the initialization routine for the $\Delta_{SM}$ weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. Input and output signals will be at 16 kHz. Code is in file `fir-dsm.c` and its frequency response is given in Figure 38.

`hq_up_1_to_2_init` is the initialization routine for high quality FIR up-sampling filtering by a factor of 2. The -3 dB point for this filter is located at approximately 3660 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures Figure 23-a and Figure 26 (top), respectively.

`hq_down_2_to_1_init` is the initialization routine for high quality FIR down-sampling filtering by a factor of 2. The -3 dB point for this filter is located at approximately 3660 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in Figure 23-b and Figure 27 (top), respectively.

`hq_up_1_to_3_init` is the initialization routine for high quality FIR up-sampling filter by factor of 3. The -3 dB point for this filter is located at approximately 3650 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in Figure 24-a and Figure 26 (bottom), respectively.

`hq_down_3_to_1_init` is the initialization routine for high quality FIR down-sampling filtering by a factor of 3. The -3 dB point for this filter is located at approximately 3650 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in Figure 24-b and Figure 27 (bottom), respectively.

`linear_phase_bp_1_to_2_init` is the initialization routine for bandpass, FIR up-sampling filtering by a factor of 2. The -3 dB points for this filter are located at approximately 98 and 3460 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in Figure 25-b and Figure 28-b, respectively.

`linear_phase_bp_2_to_1_init` is the initialization routine for bandpass, FIR down-sampling filtering by a factor of 2. The -3 dB points for this filter are located at approximately 98 and 3460 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures Figure 25-a and Figure 28-a, respectively.

`linear_phase_bp_1_to_1_init` is the initialization routine for FIR 1:1 bandpass filtering. The -3 dB points for this filter are located at approximately 98 and 3460 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in Figure 25-a and Figure 28-a, respectively.

`msin_16khz_init` is the initialization routine for the high-pass, FIR 1:1 filter that simulates a mobile station input characteristic. The -3 dB point for this filter is located at approximately 195 Hz. Code is in file `fir-flat.c` and its frequency and impulse response are given in figures Figure 29 and Figure 30, respectively.

`irs_8khz_init` is the initialization routine for the transmit-side IRS weighting filter for data sampled at 8 kHz using a linear phase FIR filter structure. Input and output signals will be at 8 kHz. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures Figure 31-a and Figure 32 (bottom), respectively.

`irs_16khz_init` is the initialization routine for the transmit-side IRS weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. Input and output signals will be at 16 kHz. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures Figure 31-b and Figure 32 (top), respectively.

`mod_irs_16khz_init` is the initialization routine for the transmit-side modified IRS weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. Input and output signals will be at

16 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures Figure 33-a and Figure 34-a, respectively.

`mod_irs_48khz_init` is the initialization routine for the transmit-side modified IRS weighting filter for data sampled at 48 kHz using a linear phase FIR filter structure. Input and output signals will be at 48 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures Figure 33-b and Figure 34-b, respectively.

`rx_mod_irs_8khz_init` is the initialization routine for the receive-side modified IRS weighting filter for data sampled at 8 kHz using a linear phase FIR filter structure. The -3 dB points for this filter are located at approximately 285 Hz and 3610 Hz. Input and output signals will be at 8 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures Figure 35-a and Figure 36-a, respectively.

`rx_mod_irs_16khz_init` is the initialization routine for the receive-side modified IRS weighting filter for data sampled at 16 kHz using a linear phase FIR filter structure. The -3 dB points for this filter are located at approximately 285 Hz and 3610 Hz. Input and output signals will be at 16 kHz since no rate change is performed by this function. Code is in file `fir-irs.c` and its frequency and impulse response are given in figures Figure 35-b and Figure 36-b, respectively.

`psophometric_8khz_init` is the initialization routine for the O.41 psophometric weighting filter for data sampled at 8 kHz using a linear phase FIR filter structure. Input and output signals will be at 8 kHz since no rate change is performed by this function. Code is in file `fir-pso.c` and its frequency response is given in Figure 37.

`p341_16khz_init` is the initialization routine for the P.341 send-side weighting filter for data sampled at 16 kHz. Input and output signals will be at 16 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 39 and its impulse response is shown in Figure 40. The -3 dB points for this filter are located at approximately 50 and 7000 Hz. Code is in file `fir-wb.c`.

`bp14k_32khz_init` is the initialization routine for the [50 Hz - 14 kHz] filter for data sampled at 32 kHz. Input and output signals will be at 32 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 41 and its impulse response is shown in Figure 42. The -3 dB points for this filter are located at approximately 50 and 14000 Hz. Code is in file `fir-wb.c`.

`bp5k_16khz_init` is the initialization routine for a 50 Hz - 5 kHz band limiting filter for wideband signals sampled at 16 kHz. Input and output signals will be at 16 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 43 and its impulse response is shown in Figure 44. The -3 dB points for this filter are located at approximately 50 and 4990 Hz. Code is in file `fir-wb.c`.

`bp100_5k_16khz_init` is the initialization routine for a 100 Hz - 5 kHz band limiting filter for wideband signals sampled at 16 kHz. Input and output signals will be at 16 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 45 and its impulse response is shown in Figure 46. The -3 dB points for this filter are located at approximately 100 and 5000 Hz. Code is in file `fir-wb.c`.

`bp20k_48khz_init` is the initialization routine for the [20 Hz - 20 kHz] filter for data sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 47 and its impulse response is shown in Figure 48. The -3 dB points for this filter are located at approximately 20 and 20000 Hz. Code is in file `fir-wb.c`.

`LP1p5_48kHz_init` is the initialization routine for a 1.5 kHz low-pass filter for signals sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 49 and its impulse response is shown in Figure 50. The -3 dB point for this filter is located at approximately 1500 Hz. Code is in file `fir-LP.c`.

`LP35_48kHz_init` is the initialization routine for a 3.5 kHz low-pass filter for signals sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 51 and its impulse response is shown in Figure 52. The -3 dB point for this filter is located at approximately 3500 Hz. Code is in file `fir-LP.c`.

`LP7_48kHz_init` is the initialization routine for a 7 kHz low-pass filter for signals sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 53 and its impulse response is shown in Figure 54. The -3 dB point for this filter is located at approximately 7000 Hz. Code is in file `fir-LP.c`.

`LP10_48kHz_init` is the initialization routine for a 10 kHz low-pass filter for signals sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 55 and its impulse response is shown in Figure 56. The -3 dB point for this filter is located at approximately 10000 Hz. Code is in file `fir-LP.c`.

`LP12_48kHz_init` is the initialization routine for a 12 kHz low-pass filter for signals sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 57 and its impulse response is shown in Figure 58. The -3 dB point for this filter is located at approximately 14000 Hz. Code is in file `fir-LP.c`.

`LP14_48kHz_init` is the initialization routine for a 14 kHz low-pass filter for signals sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 59 and its impulse response is shown in Figure 60. The -3 dB point for this filter is located at approximately 14000 Hz. Code is in file `fir-LP.c`.

`LP20_48kHz_init` is the initialization routine for a 20 kHz low-pass filter for signals sampled at 48 kHz. Input and output signals will be at 48 kHz since no rate change is performed by this function. Its frequency response is shown in Figure 61 and its impulse response is shown in Figure 62. The -3 dB point for this filter is located at approximately 20000 Hz. Code is in file `fir-LP.c`.

**Variables:**

None.

**Return value:**

These functions return a pointer to a state variable structure of type `SCD_FIR`.

#### 14.2.1.2. `hq_kernel`

**Syntax:**

```
#include "firflt.h"
long hq_kernel(long _lseg_, float _*x_ptr_, SCD_FIR _*fir_ptr_, float _*y_ptr_);
```

**Prototype:** `firflt.h`

**Source code:** `fir-lib.c`

**Description:**

This is the main entry routine for generic FIR filtering. It works as a switch to specific up- and down-sampling FIR-kernel functions. The adequate lower-lever filtering routine private to the filtering module (which is not visible by the user) is defined by the initialization routines. Currently, this function does not work properly for sample-by-sample downsampling operation, i.e. when $lseg = 1$. This limitation should be corrected in a future version.

Please note that prior to the first call to `hq_kernel`, one of the initialization routines `hq_*_init` must be called to allocate memory for state variables and the set the desired filter coefficients.

After returning from this function, the state variables are saved to allow segment-wise filtering through successive calls of `hq_kernel`. This is useful when large files have to be processed.

**Variables:**

| | |
|---|---|
| *lseg* | Number of input samples. Should be larger than 1 for proper downsampling operation. |
| *x_ptr* | Array with input samples. |
| *fir_ptr* | Pointer to FIR-struct. |
| *y_ptr* | Pointer to output samples. |

**Return value:**

The number of filtered samples as a `long`.

#### 14.2.1.3. `hq_reset`

**Syntax:**

```
#include "firflt.h"
void hq_reset (SCD_FIR _*fir_ptr_);
```

**Prototype:** `firflt.h`

**Source code:** `fir-lib.c`

**Description:**

Clear state variables in `SCD_FIR` struct; deallocation of filter structure memory is not done. Please note that *fir_ptr* should point to a valid `SCD_FIR` structure, which was allocated by an earlier call to one of the FIR initialization routines `hq_*_init`.

**Variables:**

| | |
|---|---|
| *fir_ptr* | Pointer to a valid structure `SCD_FIR`. |

**Return value:**

None.

#### 14.2.1.4. `hq_free`

**Syntax:**

```
#include "firflt.h"
void hq_free (SCD_FIR _*fir_ptr_);
```

**Prototype:** `firflt.h`

**Source code:** `fir-lib.c`

**Description:**

Deallocate memory, which was allocated by an earlier call to one of the FIR initialization routines `hq_*_init`. Note that the pointer to the structure `SCD_FIR` must not be a null pointer.

**Variables:**

| | |
|---|---|
| *fir_ptr* | Pointer to a structure of type `SCD_FIR`. |

**Return value:**

None.

### 14.2.2. IIR Module

The IIR module contains filters whose main use is for asynchronous filtering. For telephony bandwidth asynchronous filtering, PCM filters are available in both cascade and parallel IIR filter forms. For wideband speech (50—7000 Hz), 3:1 and 1:3 rate-change factor filters are available. A transmit-side IRS filter for speech sampled at 8 kHz is also available in this module as an example of implementation of an IIR cascade-form filter.

The PCM filters have been designed for *sampling rates* of 8 and 16 kHz. It should be noted that the G.712 mask is specified in terms of Hz, rather than normalized frequencies. Therefore this applies only to rate conversions of factor 2, i.e., 8 kHz to 16 kHz and 16 kHz to 8 kHz. The frequency responses of the implemented PCM filters are shown in Figure 67.

Since the digital filters need memory, state variables are needed. In the STL, a type `SCD_IIR` has been defined for parallel-form IIR filters, containing the past memory samples as well as filter coefficients and other control variables. Its fields are as follows:

| | |
|---|---|
| *nblocks* | Number of coefficient sets |
| *idown* | Up-/down-sampling factor |
| *k0* | Start index in next segment |
| *gain* | Gain factor |
| *direct_cof* | Direct path coefficient |
| *b[3]* | Pointer to numerator coefficients |
| *c[2]* | Pointer to denominator coefficients |
| *T[2]* | Pointer to state variables |
| *hswitch* | Switch to IIR-kernel: Up or down-sampling |

For the cascade-form IIR filters, the state variable structure defined is `CASCADE_IIR` which is slightly different from the one for the parallel form structure:

| | |
|---|---|
| *nblocks* | Number of stages in cascade |
| *idown* | Up-/down-sampling factor |
| *k0* | Start index in next segment |
| *gain* | Gain Factor |
| *a[2]* | Pointer to numerator coefficients |
| *b[2]* | Pointer to denominator coefficients |
| *T[4]* | Pointer to state variables |
| *hswitch* | Switch to IIR-kernel: Up or down-sampling |

It should be noted that the values of the fields must not be altered, and for most purposes they are not needed by the user. The relevant routines for each module are described in the next sections.

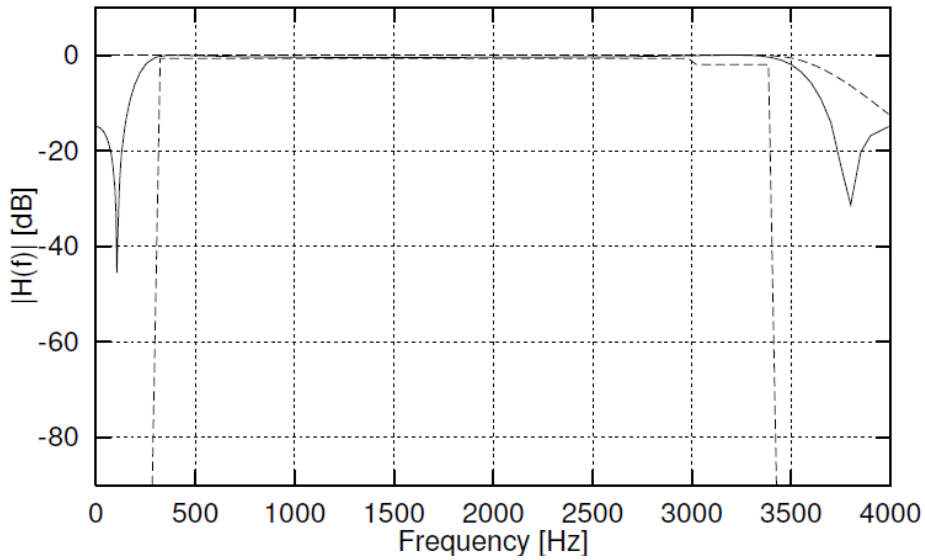



**Figure 63 — Frequency response of the cascade implementation of the G.712 standard PCM filter for data sampled at 8 kHz**

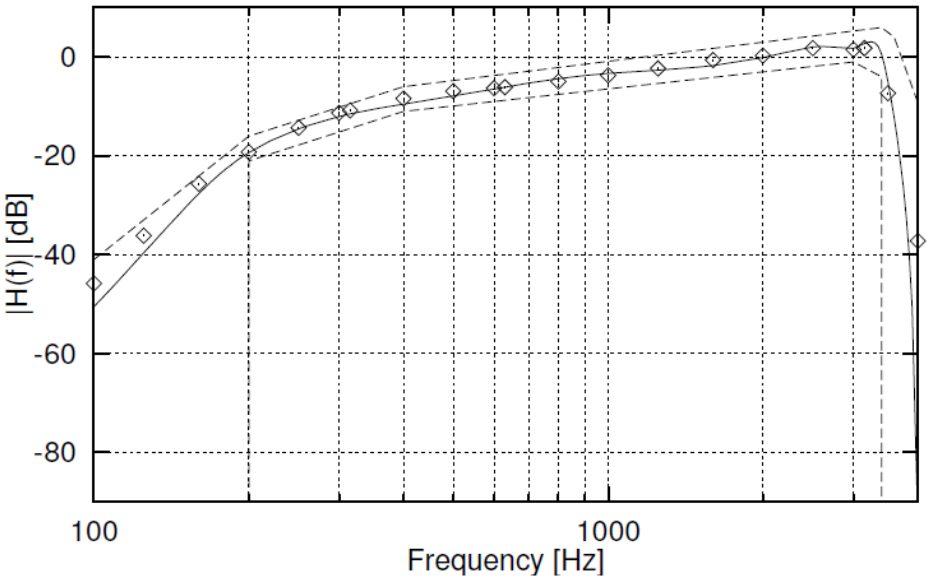



**Figure 64 — Frequency response of an IIR cascade implementation of the P.48 "full" transmitside IRS weighting filter for data sampled at 8 kHz**

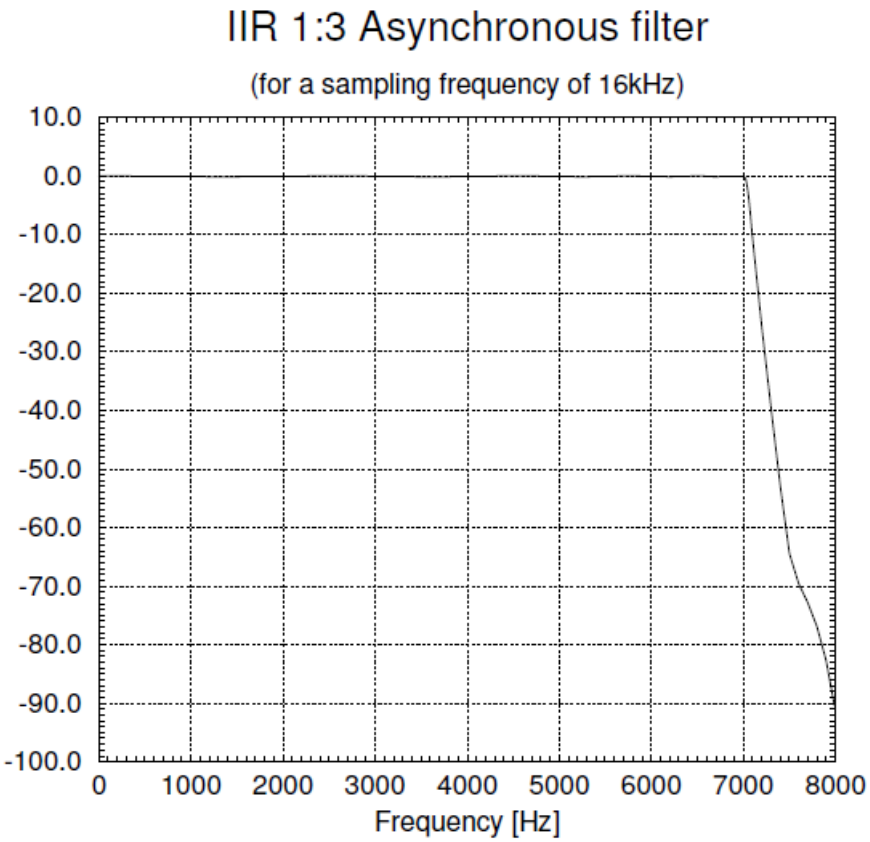



**Figure 65-a — Flat low-pass up-sampling by a factor of 1:3**

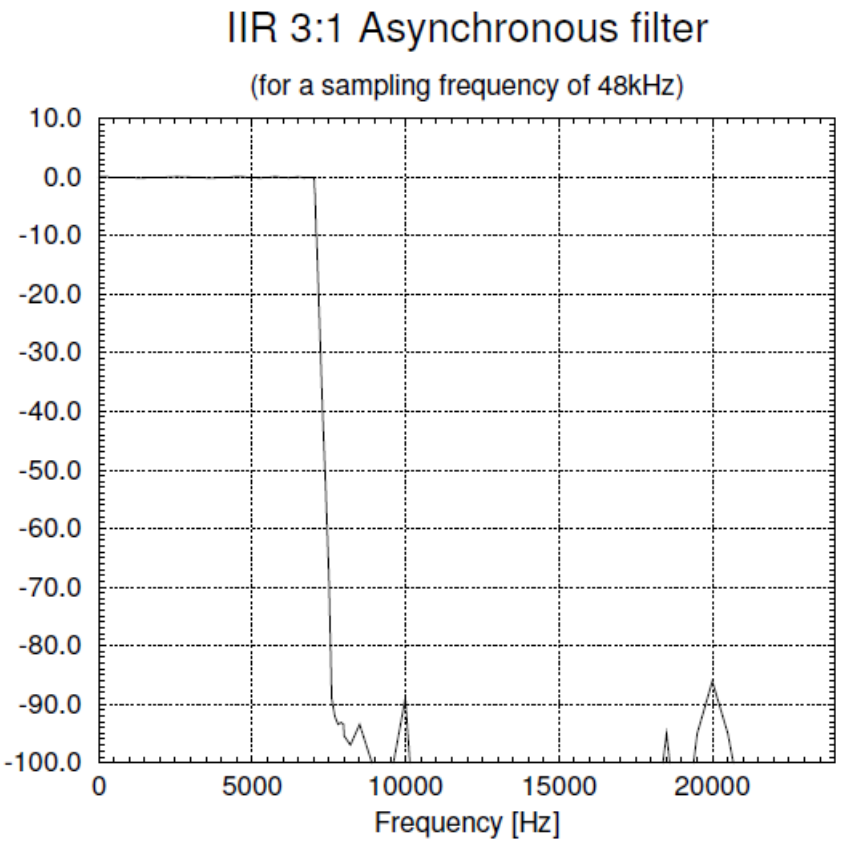



**Figure 65-b — Flat low-pass down-sampling by a factor of 3:1**

**Figure 65 — Flat low-pass IIR filter frequency response with factors 1:3 and 3:1 for sampling rates of 16000 and 48000 Hz**

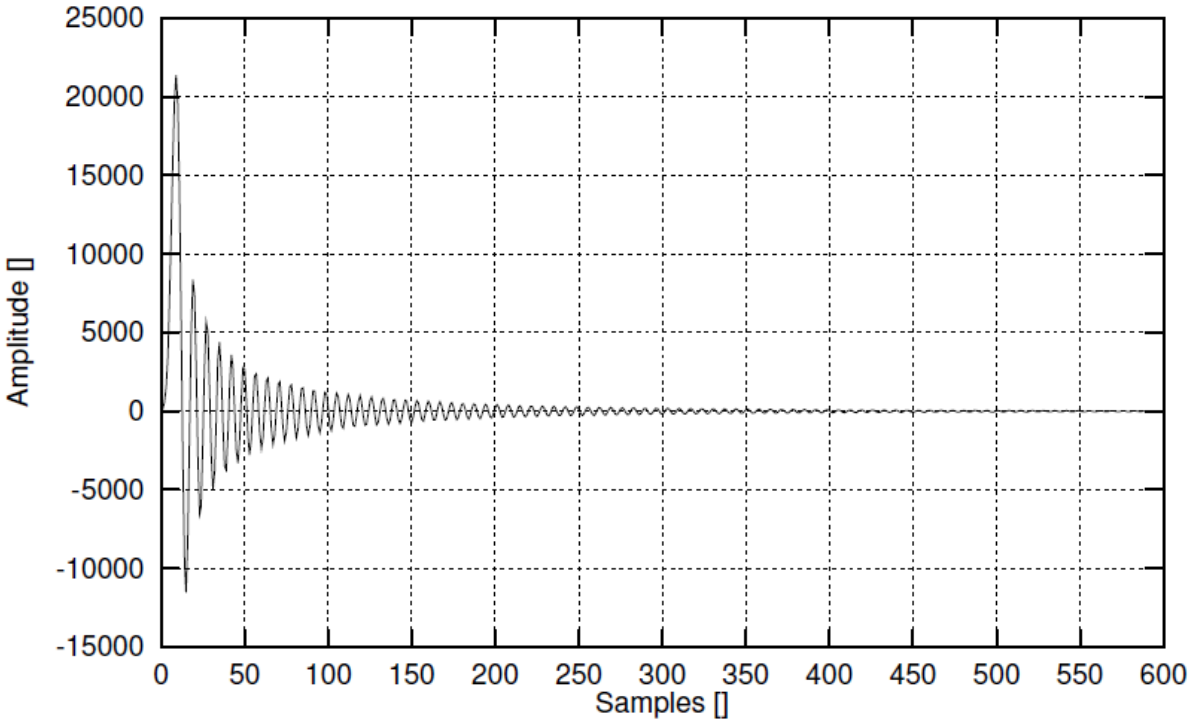



**Figure 66-a — 1:3 up-sampling factor**

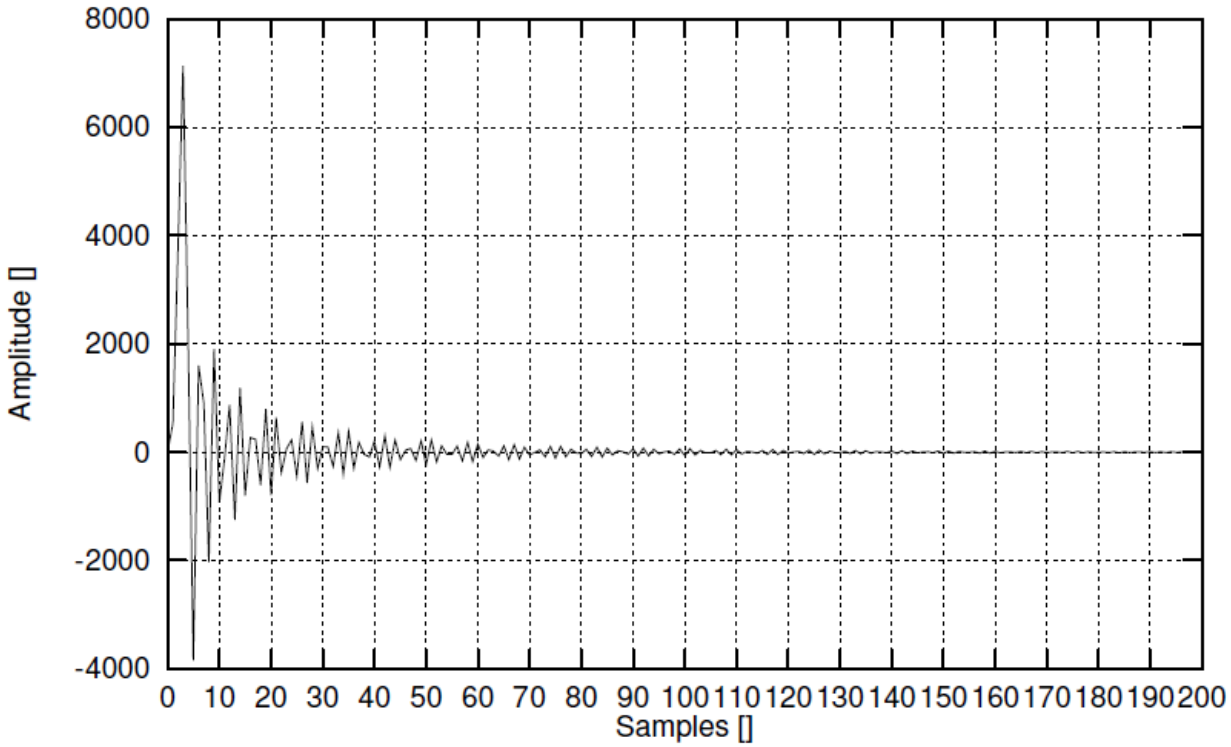



**Figure 66-b — 3:1 down-sampling factor**

**Figure 66 — Impulse response for 1:3 and 3:1 cascade-form low-pass IIR filter**

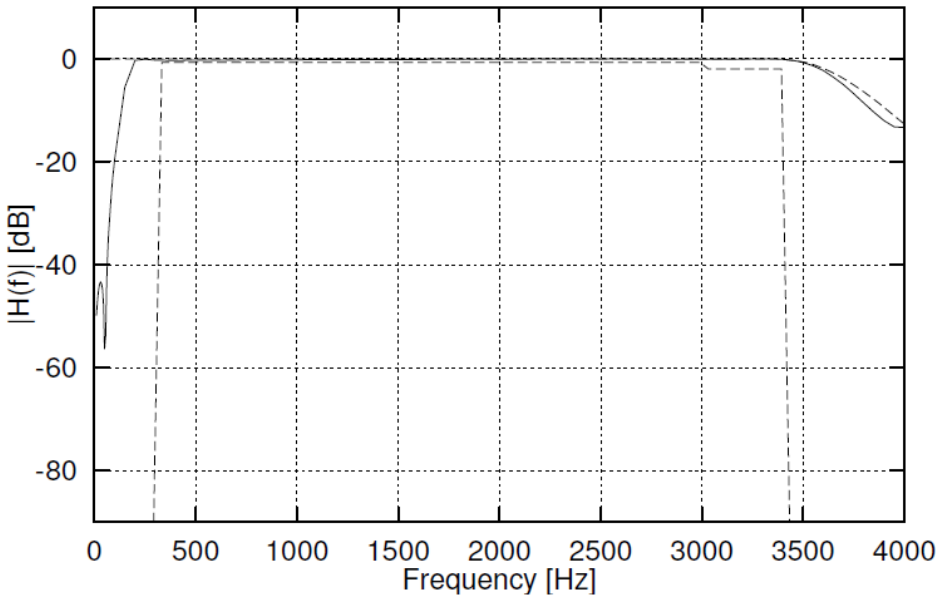



**Figure 67-a — G.712 for input samples at 8 kHz, up-sampling factor 1:2**

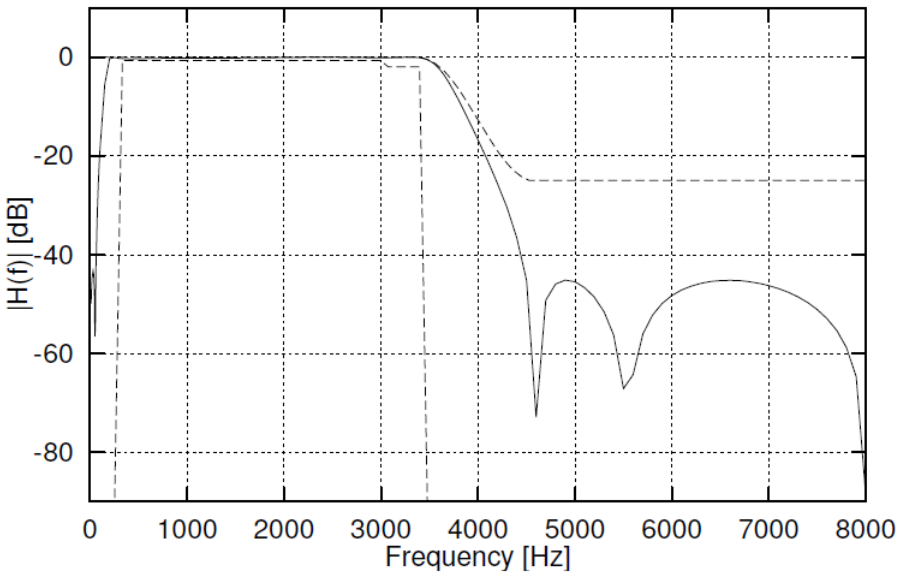



**Figure 67-b — G.712 for input samples at 16 kHz, down-sampling factor 2:1 or 1:1**

**Figure 67 — Standard PCM (G.712) quality filter response**

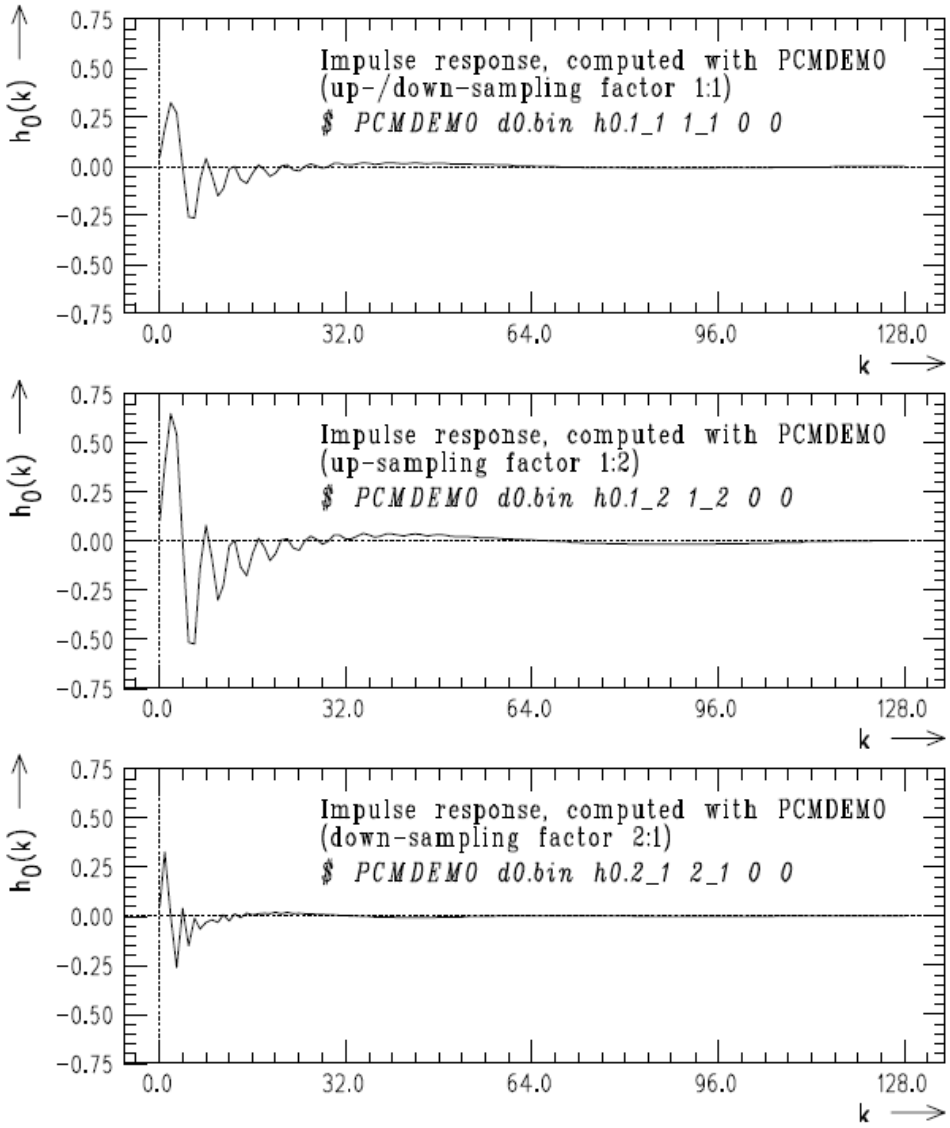



**Figure 68 — Impulse response for G.712 filters (Top: factor 1:1; Middle: factor 1:2; Bottom: factor 2:1)**

#### 14.2.2.1. `iir_*_init`

**Syntax:**

```
#include "iirflt.h"
CASCADE_IIR *iir_G712_8khz_init (void);
CASCADE_IIR *iir_irs_8khz_init (void);
CASCADE_IIR *iir_casc_lp_3_to_1_init(void);
CASCADE_IIR *iir_casc_lp_1_to_3_init(void);
```

**Prototypes:** `iirflt.h`

**Description:**

`iir_G712_8khz_init` initializes an 8 kHz cascade IIR filter structure for a standard PCM (G.712) filtering. Input and output signals will be at 8 kHz since no rate change is performed by this function. The -3 dB points for this filter are located at approximately 230 and 3530 Hz. Its source code is found in file `cascg712.c` and its frequency response is given in Figure 63.

`iir_irs_8khz_init` initializes an 8 kHz cascade IIR filter structure for a transmit-side P.48 IRS non-linear phase filtering. Input and output signals will be at 8 kHz since no rate change is performed

by this function. Its source code is found in file `iir-irs.c` and its frequency response is given in Figure 64.

`iir_casc_lp_3_to_1_init` is the initialization routine for IIR low-pass filtering with a down-sampling factor of 3:1. Although this filter is relatively independent of the sampling rate,[29] it was originally designed for asynchronization filtering of 16 kHz sampled speech. The -3 dB point for this filter is located at approximately 7055 Hz. Its source code is found in file `iir-flat.c` and its frequency and impulse response are given in Figure 65-a and Figure 66-a, respectively.

`iir_casc_lp_1_to_3_init` is the initialization routine for IIR low-pass filtering with a up-sampling factor of 1:3. Although this filter is relatively independent of the sampling rate, it was originally designed for asynchronization filtering of 16 kHz sampled speech. The -3 dB point for this filter is located at approximately 7055 Hz. Its source code is found in file `iir-flat.c` and its frequency and impulse response are given in figures Figure 65-b and Figure 66-b, respectively.

#### 14.2.2.2. `cascade_iir_kernel

**Syntax:**

```
#include "iirflt.h"
long cascade_iir_kernel (long _lseg_, float _*x_ptr_, CASCADE_IIR _*iir_ptr_,
                         float _*y_ptr_);
```

**Prototype:** `iirflt.h`

**Source code:** `iir-lib.c`

**Description:**

General function for implementing filtering using a cascade-form IIR filter previously initialized by one of the `iir_*_init()` routines.

**Variables:**

| | |
|---|---|
| *lseg* | Number of input samples. |
| *x_ptr* | Array with input samples. |
| *iir_ptr* | Pointer to a cascade-form IIR-struct `CASCADE_IIR`. |
| *y_ptr* | Pointer to output samples. |

**Return value:**

The number of output samples is returned as a long.

#### 14.2.2.3. cascade_iir_reset

**Syntax:**

```
#include "iirflt.h"
void cascade_iir_reset (CASCADE_IIR _*iir_ptr_);
```

**Prototype:** `iirflt.h`

**Source code:** `iir-lib.c`

**Description:**

[29] Since this is a low-pass filter, change of sampling rate implies in change of the lower and upper cutoff frequencies.

Clear state variables in CASCADE_IIR structure, which have been initialized by a previous call to one of the initialization functions. Memory previously allocated is not released.

**Variables:**

| | |
|---|---|
| *iir_ptr* | Pointer to struct CASCADE_IIR, previously initialized by a call to one of the initialization routines. |

**Return value:**

None.

#### 14.2.2.4. cascade_iir_free

**Syntax:**

```
#include "iirflt.h"
void cascade_iir_free (SCD_IIR _*iir_ptr_);
```

**Prototype:** iirflt.h

**Source code:** iir-lib.c

**Description:**

Deallocate memory, which was allocated by an earlier call to one of the cascade-form IIR filter initialization routines described before. iir_prt must not be a NULL pointer.

**Variables:**

| | |
|---|---|
| *iir_ptr* | Pointer to struct CASCADE_IIR, previously initialized by a call to one of the initialization routines. |

**Return value:**

None.

#### 14.2.2.5. stdpcm_*_init

**Syntax:**

```
#include "iirflt.h"
SCD_IIR *stdpcm_16khz_init (void);
SCD_IIR *stdpcm_1_to_2_init (void);
SCD_IIR *stdpcm_2_to_1_init (void);
```

**Prototypes:** iirflt.h

**Description:**

stdpcm_16khz_init initializes a 16 kHz IIR filter structure for standard PCM (G712) filtering. Input and output signals will be at 16 kHz since no rate change is performed by this function. The -3 dB points for this filter are located at approximately 174 and 3630 Hz. Source code is found in file iir-g712.c and its frequency and impulse response are given in Figure 67-b and Figure 68 (top), respectively.

stdpcm_1_to_2_init initializes standard PCM filter coefficients for filtering by the generic filtering routine stdpcm_kernel, for input signals at 8 kHz, generating the output at 16 kHz. The -3 dB points for this filter are located at approximately 174 and 3630 Hz. Source code is found in file iir-g712.c and its frequency and impulse response are given in Figure 67-a and Figure 68 (middle), respectively.

stdpcm_2_to_1_init initializes standard PCM filter coefficients for filtering by the generic filtering routine stdpcm_kernel for input signals at 16 kHz, generating the output at 8 kHz. The -3 dB points

for this filter are located at approximately 174 and 3630 Hz. Source code is found in file `iir-g712.c` and its frequency and impulse response are given in Figure 67-b and Figure 68 (bottom), respectively.

**Variables:**

None.

**Return value:**

This function returns a pointer to a state variable structure of type `SCD_IIR`.

#### 14.2.2.6. `stdpcm_kernel`

**Syntax:**

```
#include "iirflt.h"
long stdpcm_kernel (long _lseg_, float _*x_ptr_, SCD_IIR _*iir_ptr_,
                    float _*y_ptr_);
```

**Prototype:** `iirflt.h`

**Source code:** `iir-lib.c`

**Description:**

General function to perform filtering using a parallel-form IIR filter previously initialized by one of the appropriate parallel-form `*_init()` routines available.

**Variables:**

| | |
|---|---|
| *lseg* | Number of input samples. |
| *x_ptr* | Array with input samples. |
| *iir_ptr* | Pointer to a parallel-form IIR-struct `SCD_IIR`. |
| *y_ptr* | Pointer to output samples. |

**Return value:**

This function returns the number of output samples as a `long`.

#### 14.2.2.7. `stdpcm_reset`

**Syntax:**

```
#include "iirflt.h"
void stdpcm_reset (SCD_IIR _*iir_ptr_);
```

**Prototype:** `iirflt.h`

**Source code:** `iir-lib.c`

**Description:**

Clear state variables in `SCD_IIR` structure, which have been initialized by a previous call to one of the init functions. Memory previously allocated is not released.

**Variables:**

| | |
|---|---|
| *iir_ptr* | Pointer to struct `SCD_IIR`, previously initialized by a call to one of the initialization routines. |

**Return value:**

None.

#### 14.2.2.8. `stdpcm_free`

**Syntax:**

```
#include "iirflt.h"
void stdpcm_free (SCD_IIR _*iir_ptr_);
```

**Prototype:** `iirflt.h`

**Source code:** `iir-lib.c`

**Description:**

Release memory which was allocated by an earlier call to one of the parallel-form IIR filter initialization routines described before. The parameter `iir_prt` must not be a null pointer.

**Variables:**

*iir_ptr* Pointer to struct `SCD_IIR`, previously initialized by a call to one of the initialization routines.

**Return value:**

None.

### 14.3. Tests and portability

Compliance with the R&Os was verified by checking the frequency response of the filters and the size of the output files. Frequency response was obtained by feeding the filtering routines with sinewaves and calculating the ratio in dB, for each frequency of interest.

Portability of this module was checked by running the same speech file on a proven platform and on a test platform. Comparison of both processed files should show either no differences or yield equivalent results.[30] Tests were performed in the VAX/VMS environment with VAX-C and gcc, in MSDOS with Borland Turbo C++ Version 1.00 and gcc (DJGPP), in SunOS with cc, acc, and gcc, and in HPUX with gcc.

### 14.4. Examples

#### 14.4.1. Description of the demonstration programs

Three programs are provided as demonstration programs for the RATE module, `firdemo.c`, `iirdemo.c`, and filter.c.

Programs `firdemo.c` and `iirdemo.c` were the first demonstration programs for the rate change module. The former is found in directory `fir` of the STL and contains a cascade processing of the FIR filters available upto the STL96. The latter is found in directory `iir` of the STL and contains a cascade processing of the IIR filters available up to the STL96. However, because of the increasing static memory requirement for cascade processing that came with the introduction of new filters in the STL, these two programs became prohibitive and their maintenance was discontinued. They are still functional, although outdated.

Program `filter.c` is a single demonstration program that incorporates both IIR and FIR filters in the STL and has been kept up-to-date as new filters are added to the STL. Compared to the `firdemo.c` and

[30] Some differences may appear in the output files, but for a few samples and by no more than 1 LSB. As an example, in the tests for checking VAX and SUN-OS, one of the files differed in 3 samples out of 49152 for a cascade of high-quality up- and down-sampling of 1:6 and 6:1. For small rate change factors, differences are unlikely.

`iirdemo.c` programs, `filter.c` can only perform one filtering operation per pass, while `firdemo.c` and `iirdemo.c` could perform a number of 1:1 operations combined with two up-sampling and two downsampling operations. Hence, several calls of the filter program are necessary to implement what was accomplished by a single call of `firdemo.c` and `iirdemo.c`, in addition to the cumulative quantization noise (from the successive float-to-short conversions). In applications where multiple filtering is needed and the user is concerned with the quantization noise accumulation, a custom-made program could be used e.g. based on a specialization of either `firdemo.c`, `iirdemo.c`, or filter.c.

### 14.4.2. Example: Calculating frequency responses

The following C code exemplifies the use of some of the filter functions available in the STL. The C code generates a number of tones which are specified by the user (lower, upper, and step frequencies). The frequency response is obtained by calculating the power change for each single frequency before and after filtered by the selected filter.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* UGST MODULES */
#include "ugstdemo.h"
#include "iirflt.h"
#include "firflt.h"

/* Other stuff */
#define TWO_PI (8*atan(1.0))
#define QUIT(m,code) {fprintf(stderr,m); exit((int)code);}

void main(argc, argv)
  int             argc;
  char           *argv[];
{
  SCD_FIR        *fir_state;
  SCD_IIR        *iir_state;
  float          *BufInp, *BufOut;
  char            F_type[20];
  long            j, N, N2;
  long            inp_size, out_size;
  double          f, f0, fstep, ff, fs, inp_pwr, H_k;
  char            is_fir;

  /* Preamble */
  N = 256; N2 = 20; inp_size = N * N2;

  /* Read parameters for processing */
  GET_PAR_S(1, "_Filter type(irs,hq2,hq3,pcm,pcm1): ... ", F_type);
  GET_PAR_D(2, "_Start frequency [Hz]: ............... ", f0);
  GET_PAR_D(3, "_Stop frequency [Hz]: ................ ", ff);
  GET_PAR_D(4, "_Frequency step [Hz]: ................ ", fstep);
  FIND_PAR_D(5, "_Sampling Frequency [Hz]: ............ ", fs, 8000);

  /* Check consistency of upper and lower frequencies */
  ff = (ff >= fs / 2)? (fs / 2) : ff;
  if (f0 < 2.0 / (double) inp_size * fs && f0 != 0.0)
    f0 = 2.0 / (double) inp_size *fs;

  /* Normalization of frequencies */
  f0 /= fs; ff /= fs; fstep /= fs;
```

```
/* Set flag to filter type: IIR or FIR */
is_fir = (strncmp(F_type,"pcm",3)==0 || strncmp(F_type,"PCM",3)==0)
         ? 0 : 1;

/* ... CHOOSE CORRECT FILTER INITIALIZATION ... */
/*
 * Filter type: irs - IRS weighting 2:1 or 1:2 factor:
 *                     . fs ==  8000 -> up-sample: 1:2
 *                     . fs == 16000 -> down-sample: 2:1
*/
if (strncmp(F_type, "irs", 3) == 0 || strncmp(F_type, "IRS", 3) == 0)
{
  if (fs == 8000)
    fir_state = irs_8khz_init();
  else if (fs == 16000)
    fir_state = irs_16khz_init();
  else
    QUIT("IRS Implemented only for 8 and 16 kHz\n", 15);
}
/*
 * Filter type: hq2 - High-quality 2:1 or 1:2 factor:
 *                    . fs ==  8000 -> up-sample: 1:2
 *                    . fs == 16000 -> down-sample: 2:1
 *              hq3 - High-quality 3:1 or 3:1 factor
 *                    . fs ==  8000 -> up-sample: 1:3
 *                    . fs == 16000 -> down-sample: 3:1
*/
else if (strncmp(F_type,"hq",2)==0 || strncmp(F_type,"HQ",2)==0)
{
  if (fs == 8000)             /* It is up-sampling! */
    fir_state = F_type[2] == '2'
      ? fir_up_1_to_2_init()
      : fir_up_1_to_3_init();
  else                        /* It is down-sampling! */
    fir_state = F_type[2] == '2'
      ? fir_down_2_to_1_init()
      : fir_down_3_to_1_init();
}
/*
 * Filter type: pcm  - Standard PCM quality 2:1 or 1:2 factor:
 *                     . fs ==  8000 -> up-sample: 1:2
 *                     . fs == 16000 -> down-sample: 2:1
 *              pcm1 - Standard PCM quality with 1:1 factor
 *                     . fs ==  8000 -> unimplemented
 *                     . fs == 16000 -> OK, 1:1 at 16 kHz
*/
else if (strncmp(F_type,"pcm",3)==0 || strncmp(F_type,"PCM",3)==0)
{
  if (strncmp(F_type,"pcm1", 4)==0 || strncmp(F_type,"PCM1",4)==0)
  {
    if (fs == 16000)
      iir_state = stdpcm_16khz_init();
    else
      QUIT("Unimplemented: PCM w/ factor 1:1 for given fs\n", 10);
  }
  else
    iir_state = (fs == 8000)
      ? stdpcm_1_to_2_init()  /* It is up-sampling! */
      : stdpcm_2_to_1_init(); /* It is down-sampling! */
}
```

```
  /* Calculate Output buffer size */
  if (is_fir)
    out_size = (fir_state->hswitch=='U')
               ? inp_size * fir_state->dwn_up
               : inp_size / fir_state->dwn_up;
  else
    out_size = (iir_state->hswitch=='U')
               ? inp_size * iir_state->idown
               : inp_size / iir_state->idown;

  /* Allocate memory for input buffer */
  if ((BufInp = (float *) calloc(inp_size, sizeof(float))) == NULL)
    QUIT("Can't allocate memory for data buffer\n", 10);

  /* Allocate memory for output buffer */
  if ((BufOut = (float *) calloc(out_size, sizeof(float))) == NULL)
    QUIT("Can't allocate memory for data buffer\n", 10);

  /* Filtering operation */
  for (f = f0; f <= ff; f += fstep)
  {
    /* Reset memory */
    memset(BufOut, '\0', out_size * sizeof(float));

    /* Adjust top (NORMALIZED!) frequency, if needed */
    if (fabs(f - 0.5) < 1e-8/fs) f -= (0.05*fstep);

    /* Calculate as a temporary the frequency in radians */
    inp_pwr = f * TWO_PI;

    /* Generate sine samples with peak 20000 ... */
    for (j = 0; j < inp_size; j++)
      BufInp[j] = 20000.0 * sin(inp_pwr * j);

    /* Calculate power of input signal */
    for (inp_pwr = 0, j = 0; j < inp_size; j++)
      inp_pwr += BufInp[j] * BufInp[j];

    /* Convert to dB */
    inp_pwr = 10.0 * log10(inp_pwr / (double) inp_size);

    /* Filtering the whole buffer ... */
    j = (is_fir)
        ? fir_kernel(inp_size, BufInp, fir_state, BufOut)
        : stdpcm_kernel(inp_size, BufInp, iir_state, BufOut);

    /* Compute power of output signal; discard initial 2*N samples */
    for (H_k = 0, j = 2 * N; j < out_size - 2 * N; j++)
      H_k += BufOut[j] * BufOut[j];

    /* Convert to dB */
    H_k = 10 * log10(H_k / (double) (out_size - 4 * N)) - inp_pwr;

    /* Printout of gain at the current frequency */
    printf("\nH( %4.0f ) \t = %7.3f dB\n", f * fs, H_k);
  }
}
```

# 15. EID: Error Insertion Device

An error insertion device (EID) is used to study the behaviour of codec over digital transmission systems and equipments under error conditions. This requires a model for the transmission channel, and an error generation algorithm. In the most general case, burst or random bit error generators are needed. In other cases, such as when evaluating mobile and wireless systems, random and bursty frame erasures are of importance.

The EID module implements the four functionalities: random and bursty bit errors, and random and bursty frame erasures. These four models are based on a linear congruential sequence random number generator, and the bit error insertion and random frame erasure are based on a two-state channel model.

The burst frame erasure function requires a more elaborated model. For the specific application of wireless systems, a model based on Markov sequences is used. This is known within ITU as the Bellcore model [78], [79]. This model was first used in the standardization of ITU-T G.729, and has been incorporated in the STL since.

Upon development needs of scalable and embedded-variable bitrate codecs, a more elaborate EID simulation software was added in STL2009 release. This tools allows more complex error insertion simulations: layered application mode where upper layers are dependent on the error of lower layers, and (optional) individual application mode where errors are inserted separately to each layer.

## 15.1. Description of Algorithm

### 15.1.1. Simple Channel Model

The bit error insertion algorithm of the EID is based on a channel model where (binary) data bitstreams are to be transmitted, and is based on the discrete *Gilbert Elliott channel* (GEC) model, described in [80].

This model (see Figure 69) has two states, Good ($G$) and Bad or Burst ($B$). Associated with these two states, there are four parameters (probabilities): two relating to the probability of remaining in state $G$ or $B$, and two relating to the probability of transition from the current state to the other state (i.e., the occurrence of a binary digit transition, or error).

The probabilities associated with the channel states are $P$ and $Q$, $P$ being the probability of transition from state $G$ to $B$, and $Q$ the probability of transition from $B$ to $G$. Hence, the probability of remaining in the same state is $(1-P)$ and $(1-Q)$ for states $G$ and $B$, respectively. For a given state, there are probabilities that a change in a bit occur, and this is $P_G$ for state $G$, and $P_B$ for state $B$.

Therefore, the channel may be either in the good state $G$, where the mean bit error probability $P_G$ is very low ($P_G \approx 0$), or in the bad state $B$, where the mean bit error probability $P_B$ is rather high ($P_B \approx 0.5$).

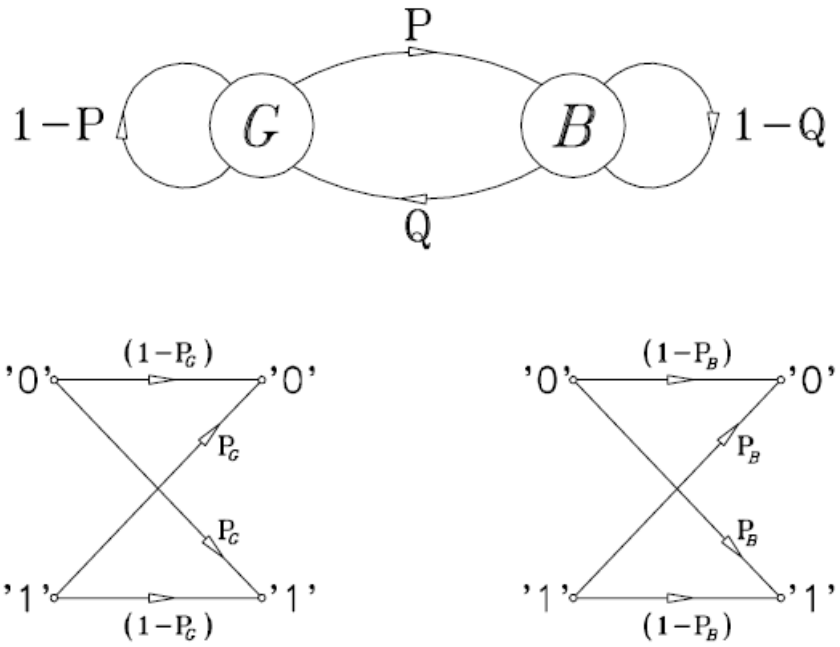



**Figure 69 — Gilbert Elliot Channel Model (GEC)**

The mean bit error probability *BER* generated by this channel model is

$$BER = \frac{P}{1-\gamma} \cdot P_B + \frac{Q}{1-\gamma} \cdot P_G \tag{15-1}$$

where

$$\gamma = 1 - (P + Q) \tag{15-2}$$

is a measure for the correlation of the bit errors, and consequently an indication of the burst or random characteristic of the channel. In this issue, $\gamma \approx 0$ implies a nearly random error channel, while $\gamma \approx 1$ implies a totally bursty channel. Please note that *BER* is reasonable only in the range $0 \leq BER \leq 0.5$.

For many applications, bit error sequences with a distinct mean bit error probability *BER* and a distinct bit error correlation $\gamma$ are of interest. From (Equation (15-1)) and (Equation (15-2)) we get for the remaining parameters of the *GEC* for arbitrarily chosen values of $0 \leq P_G < P_B \leq 0.5$:

$$P = (1-\gamma) \cdot \left(1 - \frac{P_B - BER}{P_B - P_G}\right) \tag{15-3}$$

$$Q = (1-\gamma) \cdot \frac{P_B - BER}{P_B - P_G} \tag{15-4}$$

In the Error Insertion Device (EID) the special values $P_G = 0$ and $P_B = 0.5$ are chosen. This relates to the fact that in the good state no bit changes are expected, hence $P_G = 0$; now, for the bad state, the channel is supposed to be in a totally uncertain state, then $P_B = 0.5$. With this choice, (Equation (15-3)) and (Equation (15-4)) reduce to:

$$P = 2 \cdot (1-\gamma) \cdot BER \tag{15-5}$$

$$Q = (1-\gamma) \cdot (1 - 2 \cdot BER) \tag{15-6}$$

As an example, Figure 70 shows the effect of $\gamma$ on the auto-correlation of a bitstream, generated by the *GEC* (with $BER = 0.02$ in (Equation (15-5)),(Equation (15-6))).

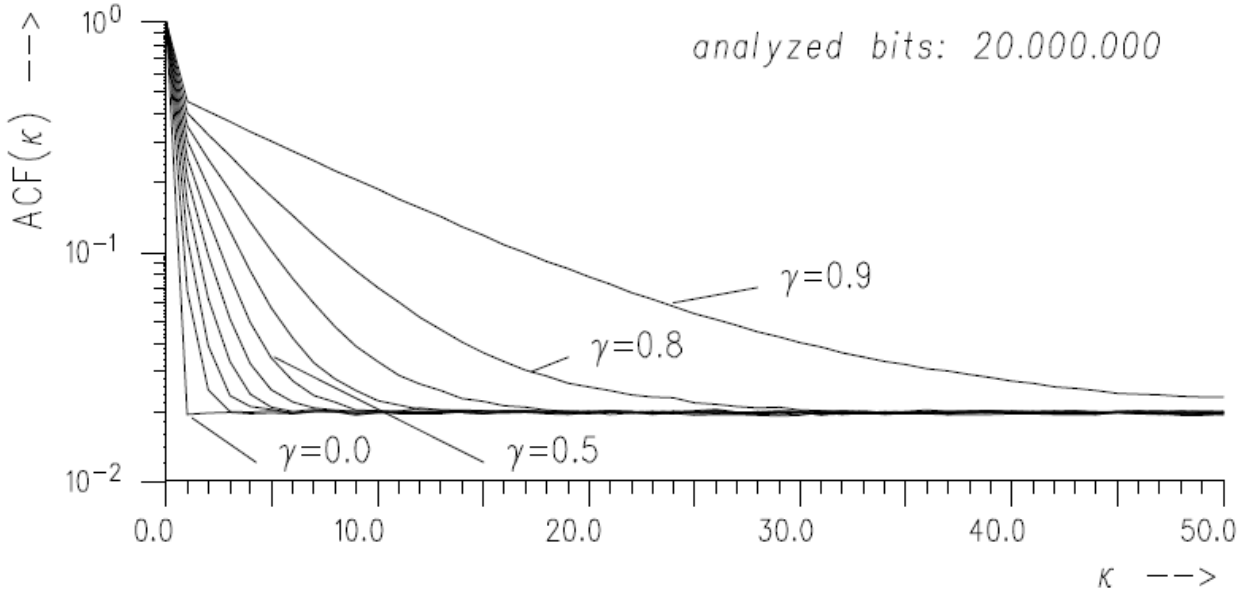



**Figure 70 — Bit Error Correlation for a Bit Error Rate of 2%**

It can be seen that for $\gamma = 0$ the bit errors are statistically independent, because the autocorrelation sequence ACF $(\kappa)$ has a peak (1.0) in $\kappa = 0$, and the remaining coefficients ACF(1),ACF(2),… oscillate around the selected bit error rate of 0.02 $(2 \cdot 10^{-2})$. For $\gamma = 0.5$, slightly bursty errors can be observed, when the initial terms of the correlation sequence build a transition region, and the remaining (higher) terms are around 0.02. Increasing $\gamma$ towards unity, the correlation between the bit errors also increases, leading to totally bursty errors in the limit.

### 15.1.2. Bellcore Model

The following description has been based on [78]. The actual error sequence in a wireless environment will depend upon the carrier frequency, user speed, detection scheme, type of diversity employed, mean SNR, hand-off mechanism, etc. Though a model could be created using the above parameters, it would be impossible to apply because of the wide variance of the model parameters. It was found that a speech coder can be tested using error bursts generated by a much simpler model because the burst error performance of a speech coder can be characterized to a great extend by the way it reacts to short (5—20 ms), medium (30—60 ms) and long (over 80 ms) error bursts. If a coder performs well in the presence of a representative range of short, medium and long error bursts, it can be expected that the coder will behave well in an actual wireless communications environment, even though the actual radio channel generates error bursts with different statistics.

The Bellcore model, rather than modeling the wireless communications channel, models the occurrence of these short, medium, and long error bursts that would enable the characterization of the coder reaction to error bursts and to the error burst patterns it is expected it will encounter in practice.

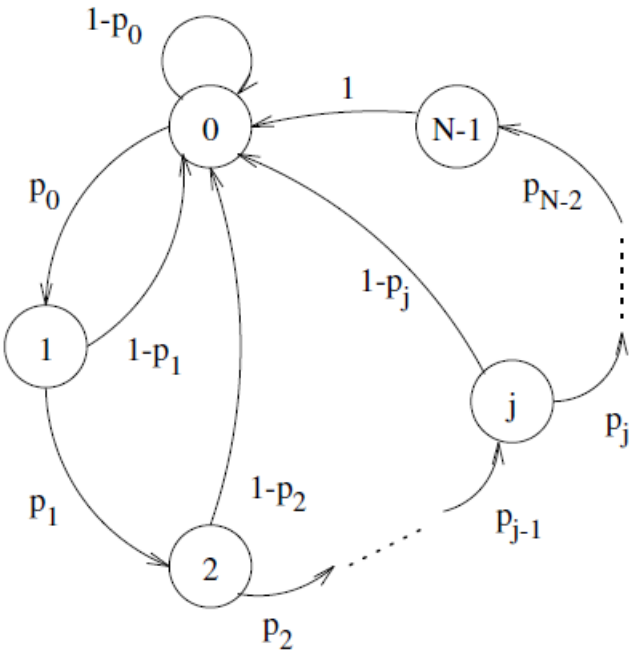



**Figure 71 — N-state Markov Model**

An N-state Markov model, as illustrated in Figure 71, is used in the STL to generate frame erasure bursts. This model has to be adequate to test speech coders using short speech segments (6-8s). In this model, a transition from any state (0..N—1) to state 0 represents a frame received without errors, while a transition from state j—1 to state j indicates that j previous frames have been received in error. A transition from j back to 0 marks the end of an error burst of length j followed by a good frame.

The Markov model with N states for creating a bursty wireless communications channel is capable of erasing up to N—1 frames. The model generates both frames with correlated frame erasures and error-free frames. The value of N will depend on the frame duration of the speech coder under test and the maximum error burst length the coder expects to find in practice. The error statistics can be controlled by selecting the N—1 transition probabilities $p_k$, $k$=0..N—2. The probabilities will also determine the sequence of good and erased frames.

The steady state probabilities can be calculated by solving the state transition matrix or using numerical methods. If $S_j$ denotes the steady state probability that the chain is in state $j$ and $p_j$ is the probability of transitioning from state $j$ to $j+1$, the following relationships can be established:

$$S_{j+1} = p_j S_j, 0 \le j \le N-2$$

$$S_0 = \sum_{j=0}^{N-1} S_j (1-p_j), (p_{N-1} = 0)$$

The equations above can be solved since the probabilities should satisfy:

$$\sum_{j=0}^{N-1} p_j = 1$$

A frame erasure length of $j$ can occur only if the chain first enters state $j$, and then transitions to state 0. The probability $P_{fe}$ of this occur is

$$P_{fe} = S_j (1-p_j)$$

The probability of receiving a frame in error can be calculated as

$$P_e = \sum_{j=1}^{N-1} jP_{fe}(j)$$

and the probability of receiving an error-free frame is $1 - P_e$. It can be seen that the steady state probability of being in state 0, $S_0$, also gives the probability of receiving a frame without error, i.e.,

$$S_0 = 1 - \sum_{j=1}^{N-1} jP_{fe}(j)$$

The frame error distribution can be controlled by selecting the transition probabilities.

### 15.1.3. Error insertion for layered bitstreams

A layered scalable codec provides a multi-layer bitstream that can be modified by application or network entities:

– The core layer of the codec provides a minimum quality.
– Upper layers enable improving quality by increasing bitrate up to a maximum value.

One main feature of a scalable codec lies in the layering flexibility. The layers can be transported over different channels or over the same channel but with different priorities. Further the bitrate can be adjusted between minimal and maximal values by any network element in the communication chain.

To simulate the layering functionality a bitstream error application tool (eid-ev) was added for STL2009 release. This tool applies layer errors at desired levels. For each frame of an input bitstream, this tool performs the following operations:

1) Read and validate the input frame, (each input frame size needs to represent a valid layer boundary)
2) Read the frame error pattern files, (one error pattern file is read for each layer)
3) Apply errors by copying non-distorted layers to the output frame, and setting the bits of distorted layers to the softbit value zero.
4) Truncate output frame if consecutive higher layers are hit by errors.
5) Set the correct synchronization header of the output frame
6) Set the correct length of the output frame.
7) Provide statistics of error application across layers.

## 15.2. Implementation

The EID algorithm is written in C-source code can be found in the module `eid.c`, with prototypes in `eid.h`. This version evolved from previous C implementations developed by PKI[31], and was used in the Host Laboratory Sessions of ETSI's contest for the second generation of the GSM Digital Mobile Radio Systems, and in the Selection Phase of the ITU-T 8 kbit/s speech coder.

The random-number generator is based on a linear congruential technique, as described in [81]. The rule here is:

$$a_n = (69'069 * a_{n-1} + 1) \bmod 2^{32},$$

which is converted (mapped) to a float number between 0 and 1.

Since the random number generator and the channel need their internal state to be saved, two state variable data structure types were defined for the EID module. The structure type called `SCD_EID`

31 Phillips Communications Industry.

is applicable to the burst and random bit erasure functions, as well as to the random frame erasure function. The fields of this structure are:

| | |
|---|---|
| *seed* | Seed for random number generator. |
| *nstates* | Number of states of the channel model (presently 2). |
| *current_state* | Index of current channel state. |
| *BER* | Pointer to array containing thresholds according to the bit error rate in each state. |
| *usrber* | User defined bit error rate. |
| *usrgamma* | User defined correlation factor. |
| *matrix* | Pointer to matrix containing thresholds according to the probabilities for changing from one state to another one. |

For burst frame erasures only, a different state variable structure type called `BURST_EID` has been defined, whose fields are:

| | |
|---|---|
| *seedptr* | Memory for random number generator. |
| *internal* | Array with probabilities for each state of the Markov process. |
| *index* | Channel's current state. |

The values of the fields shall not be altered and are not needed by user.

The random number generator always starts from the same point, if the user does not specify different initial seeds. In order to avoid this, the EID state variables should be saved at the end of the processing of a speech sequence, e.g. to a file by the user. This saving is not implemented by the EID module because this involves I/O to the computer file systems, and this would violate one of the UGST guidelines. Nevertheless, an example of this procedure is described in the demonstration programs that accompany this release of the EID. Therefore, users should keep in mind that, unless they save (e.g. to a file) the EID state at the end of the processing, identical error patterns will be produced, when the processing is re-started.

The EID routines for random bit errors are `BER_generator` and `BER_insertion`; for random frame erasures, `FER_generator_random` and `FER_module`; for burst frame erasures `FER_generator_burst`; and `open_eid`, `open_burst_eid` and `close_eid` for initialization (allocation) and release of EID state structures `SCD_EID` and `BURST_EID`. Their description can be found next. Besides these, there are other routines which are local (private) to the EID module, and therefore are not described.

### 15.2.1. Bitstream format

The EID module operates on *softbits* basis. Softbits are defined as a multi-level representation of the binary ("hard") bits '1' and '0' which are associated to probabilities of being in error. The softbit definition adopted in the ITU-T STL uses 16-bit words as representation of the hardbits `'1'` and `'0'`, where a hardbit `'1'` is represented by the softbit `0x0081` and a hardbit `'0'` is represented by the softbit `0x007F`. This means that 8 significant bits in a softbit is available for other utilities. When soft-decision is not used, the hard bit information can be derived directly from Bit 7 of the 16-bit softbit word. It should also be noted that values `0x0081` and `0x007f` are equally spaced from `0x0000` (in other words, `0x0000` is exactly the middle of the two-complement range for `0x81` and `0x7F`), such that a softbit representation `0x0000` represents *total uncertainty* of the true bit value.

Error patterns produced and used by the EID module use this softbit definition. Input and output data (i.e. signals which are affected by bit errors or frame erasures) also use softbits, but additionally have a so-called synchronization header.

The *synchronization header* is defined as two consecutive 16-bit words, the first one always being a synchronization (sync) word in the range `0x6B21` to `0x6B2F`, followed by the bitstream length word, a two-complement number indicating the number of softbits in the frame. The sync word `0x6B20` is reserved to indicate that a frame erasure happened. For example, for the RPE-LTP algorithm, which uses 260 bits per frame, the soft bitstream would have the format indicated in Figure 72. It can be seen that each RPE-LTP frame will have 262 16-bit words, being one for the sync word (`0x6B21`) in the example), one for the frame length word (whose value here is 260), followed by 260 softbit words (here corresponding to `'1'`, `'0'`, …, `'0'`, `'0'`). This combination of the synchronization header and a softbit "payload" is called the bitstream signal representation and is used in ITU-T Recommendation G.192 [2] to represent encoded signals between speech encoders, error-insertion devices, transmission channel models and speech decoders.

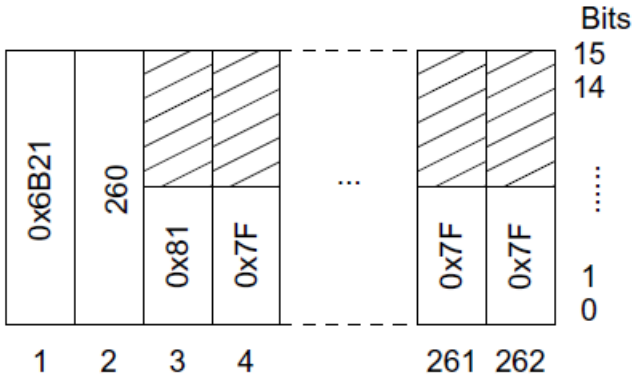



**Figure 72 — Soft bitstream format for the 13 kbit/s RPELTP algorithm, where 260 bits are transmitted per 20 ms transmission frame.**

### 15.2.2. `open_eid`

**Syntax:**

```
#include "eid.h"
SCD_EID *open_eid (double _ber_, double _gamma_);
```

**Prototype:** `eid.h`

**Description:**

Allocate memory for EID struct, set up the transmission matrix according to the selected bit error rate, and initialize the seed for the random number generator. If the symbol `PORT_TEST` is defined at compilation time, then the seed will always be initialized to the same value; otherwise, the seed is initialized with the system time (in seconds). The former is used to test portability of the EID module, since identical patterns will be generated[32].

**Variables:**

*ber* User desired bit error rate;

*gamma* User desired burst factor;

**Return value:**

Returns a pointer to struct `SCD_EID`; if the initialization failed, returns a null pointer.

[32] Another way to force the EID to produce identical bit error patterns is to save the EID state variable (of type `SCD_EID`) e.g. to a file and, in the next call to the routine, initialize the state variable with the saved value.

### 15.2.3. `open_burst_eid`

**Syntax:**

```
#include "eid.h"
BURST_EID *open_burst_eid (long _index_);
```

**Prototype:** `eid.h`

**Description:**

Allocate memory for a state variable structure of type `BURST_EID` and setup the transmission matrix according the burst frame erasure rate (BFER) selected by *index*, and initialize the seed for the random number generator. If the symbol `PORT_TEST` is defined at compilation time, then the seed will always be initialized to the same value; otherwise, the seed is initialized with the system time, in seconds (see note in the description of `open_eid()`).

**Variables:**

*index* Indicates BFER index starting from 0% to 30% with 0.5% steps for the Bellcore model. If *index* is equal to 0, there is no FER, and 1 means that there is 0.5% BFER; incrementally, 60 gives 30% BFER.

**Return value:**

This function returns a pointer to a structure of type `BURST_EID`. If the initialization failed, it returns a null pointer.

### 15.2.4. `reset_burst_eid`

**Syntax:**

```
#include "eid.h"
void reset_burst_eid (BURST_EID _*burst_eid_);
```

**Prototype:** `eid.h`

**Description:**

Reset a `BURST_EID` structure previously initialized by a call to `open_burst_eid()`. By default, only counters are reset; if the symbol `RESET_SEED_AS_WELL` is defined at compilation time, the seed is also reset. However, this is not recommended.

**Variables:**

*burst_eid* `BURST_EID` structure to be reset.

**Return value:** None.

### 15.2.5. `close_eid`

**Syntax:**

```
#include "eid.h"
void close_eid (SCD_EID _*EID_);
```

**Prototype:** `eid.h`

**Description:**

Release the memory previously allocated by `open_eid()` for the specified EID structure.

**Variables:**

*EID* EID state variables' structure to be released.

**Return value:** None.

### 15.2.6. `BER_generator`

**Syntax:**

```
#include "eid.h"
double BER_generator (SCD_EID _*EID_, long _lseg_, short _*EPbuff_);
```

**Prototype:** `eid.h`

**Description:**

Generates a softbit error pattern according to the selected channel model present in *EID*. The introduction of the bit errors in the bitstream is done by the function `BER_insertion`. It should be noted that softbit error pattern buffers do not contain synchronization headers.

**Variables:**

| | |
|---|---|
| *EID* | Structure with channel model. |
| *lseg* | Length of current frame. |
| *EPbuff* | Bit error pattern buffer with softbits. |

**Return value:**

The bit error rate in the current frame is returned as a `double`.

### 15.2.7. `FER_generator_random`

**Syntax:**

```
#include "eid.h"
double FER_generator_random (SCD_EID _*EID_);
```

**Prototype:** `eid.h`

**Description:**

Decides whether a random frame erasure should happen for the current frame according to the state of the GEC model in the channel memory pointed by *EID*.

**Variables:**

| | |
|---|---|
| *EID* | Structure with channel model. |

**Return value:**

Returns a `double` value: 0 if the current frame should not be erased ("good frame") and 1 if the frame should be erased ("bad frame").

### 15.2.8. `FER_generator_burst`

**Syntax:**

```
#include "eid.h"
double FER_generator_burst (BURST_EID _*EID_);
```

**Prototype:** `eid.h`

**Description:**

Decides whether a burst frame erasure should happen for the current frame according to the state of the Bellcore model in the channel memory pointed by *EID*. It should be noted that in the long run, the overall burst frame erasure rate (BFER) may not be consistent with the BFER specified by the user. This is an inherent deficiency of the implemented model and the calling program is responsible for computing the overall BFER and monitoring whether this overall BFER is close enough to the desired BFER.

**Variables:**

*EID* Structure with Bellcore model parameters.

**Return value:**

This function returns a `double` value: 0 if the current frame should not be erased ("good frame") and 1 if the frame should be erased ("bad frame").

### 15.2.9. `BER_insertion`

**Syntax:**

```
#include "eid.h"
void BER_insertion (long _lseg_, short _*xbuff_, short _*ybuff_, short
                    _*error_pattern_);
```

**Prototype:** `eid.h`

**Description:**

Disturbs an input bitstream *xbuff* according to the error pattern provided in *error_pattern*, saving the disturbed bitstream in the output buffer *ybuff*. The input and output bitstream are compliant to the bitstream format described before, i.e. are comprised of a synchronization header (sync word followed by a frame length word) and softbits representing the encoded bitstream. The sync and frame length words are always located in the offsets 0 and 1 of the array, respectively. The error pattern contains only softbits. The following summarizes the bit error insertion rules:

a) input signal (after synchronization header):
   i) hard bit `'0'` represented as `0x007F`;
   ii) hard bit `'1'` represented as `0x0081`.
b) error pattern:
   i) the probability for undisturbed transmission has values in the range `0x0001..-0x007F`, being `0x0001` the lowest probability.
   ii) the probability for disturbed transmission has values in the range `0x00FF..-0x0081`, being `0x00FF` the lowest probability.
c) output signal computation (does not affect the synchronization header, which is copied unmodified from the input buffer to the output buffer):
   For input `'1'` (`0x0081`):
   i) if the error pattern is in the range `0x00FF..0x0081` (255..129), then the output will be `0x0001..0x007F` (1..127), respectively;
   ii) if the error pattern is in the range `0x0001..0x007F` (1..127), then the output will be `0x00FF..0x0081` (255..129), respectively.
   For input `'0'` (`0x007F`):
   i) if the error pattern is in the range ` 0x00FF..0x0081` (255..129), then the output will be `0x00FF..0x0081` (255..129), respectively;
   ii) if the error pattern is in the range `0x0001..0x007F` (1..127), then the output will be `0x0001..0x007F` (1..127), respectively.

**Variables:**

| | |
|---|---|
| *lseg* | Length of current frame (including synchronization header). |
| *xbuff* | Buffer with input bitstream of length *lseg*. |
| *ybuff* | Buffer with output bitstream of length *lseg*. |
| *error_pattern* | Buffer with error pattern (without synchronization header), of length *lseg—2*. |

**Return value:** None.

### 15.2.10. `FER_module`

**Syntax:**

```
#include "eid.h"
double FER_module (SCD_EID _*EID_, long _lseg_, short _*xbuff_, short
                   _*ybuff_);
```

**Prototype:** `eid.h`

**Description:**

Implementation of the frame erasure function based on the GEC model allowing a variable degree of burstiness (as specified by parameter `gamma` in the state variable structure of type `SCD_EID` pointed by *EID*). This function actually erases the current frame (as described below), as opposed to function `FER_generator_random()`, which only indicates whether the current frame should be erased.

– computes the "frame erasure pattern";
– erases all bits in one frame according the current state of the pattern generator.

The input (undisturbed) and output (disturbed) buffers have samples conforming to the bitstream representation description in Annex B of G.192. The input and output bitstream are compliant to the bitstream format described before, i.e. are comprised of a synchronization header (sync word followed by a frame length word) and softbits representing the encoded bitstream. The sync and frame length words are always located in the offsets 0 and 1 of the array, respectively. Should the frame be erased (depending on the frame erasure pattern), all softbits are set to `0x0000`, which corresponds to a total uncertainty about the true bit values.

In addition, the lower 4 bits of the sync word in the synchronization header are set to 0. This makes it easier for the succeeding software to detect an erased frame. The frame length word is copied unmodified to the output buffer.

**Variables:**

| | |
|---|---|
| *EID* | Pointer to a state variable structure structure of type `SCD_EID`. |
| *lseg* | Length of current frame (including synchronisation header). |
| *xbuff* | Pointer to input bitstream. The synchronisation word (*xbuff[0]*) is processed, the frame length word (*xbuff[1]*) is not changed. |
| *ybuff* | Buffer with output bitstream. |

**Return value:**

This function returns a `double` value: 1 if the current frame has been erased, and 0 otherwise.

## 15.3. Tests and portability

Portability may be checked by running the same speech file on a proven platform and on a test platform, for the whole range of input parameters. Results should be identical when the compilation is done with the symbol `PORT_TEST` properly defined and the channel states are set to a same value.

For the eid-ev program, please note that the 16 bit oriented G.192 files require correct byte swapping of inputs and outputs on big/little-endian machines. It checks for inconsistent sync headers and exits the program with a warning if incorrectly swapped synchronism headers are detected.

The original EID routines had portability tested for VAX/VMS with VAX-C, MS-DOS with Turbo C v2.0, Sun-OS with Sun-C, and HPUX with gcc. The newly introduced eid-ev programs were tested for Microsoft C-compiler and gcc on Cygwin.

## 15.4. Examples

### 15.4.1. Description of demonstration programs

Number of programs are provided as demonstration programs for the EID module, `eiddemo.c` (version 3.2), `eid8k.c` (version 3.2), `eid-xor.c` (version 1.0), `gen-patt.c` (version 1.4), `ep-stats.c` (version 2.0), `eid-int.c` (version 1.0), `eid-ev.c` (version 1.0), `gen_rate_profile.c` (version 1.2).

Program `eiddemo.c` uses input and output file in the form of a serial bit stream conforming to the bitstream signal representation, as defined in Annex B of ITU-T G.192. This program will disturb the input bitstream with errors using the Gilbert Elliot Channel model for random or burst bit error insertion and for random frame erasures. The Bellcore model, which is used for burst frame erasures, is supported as a command line option, but not as default. It should be noted that this program uses function `FER_module()`, not function `FER_generator_random()`, for random frame erasures.

Program `eid8k.c` was developed during the standardization process of the ITU-T G.729 8 kbit/s speech codec for the task of producing bit error masks which would be used in the host laboratory hardware-implemented EID. For this program, input files are not generated, but only bit error pattern files. Consistent with the definition in the STL, error patterns do not have synchronization headers (sync word and frame length word), but only softbits representing disturbance of the channel. GEC and the Bellcore model are supported in this program. The output file format is, as was necessary for the G.729 work, different from a serial bitstream as defined in the STL because the softbits are saved as `char` (8-bit words) rather than as `short` (16-bit words). Conversion of this format to the STL 16-bit bitstream format can be accomplished using the unsupported program `ch2sh.c`.

It should be noted that both programs save in files the current state of the EID models under use and also try to read these state files at startup time (if not found, the programs create new ones, which are updated when the programs terminate).

Program `eid-xor.c` is an error-insertion program that simply XORs bits in a bitstream file (in one out of three formats: G.192, byte-oriented G.192, and compact) with error patterns (bit errors or frame erasures in one out of three formats) and saves the disturbed bitstream in a file. The error patterns need not have been produced by any of the EID models implemented in the STL, they only have to be in one of the three input formats. Since error patterns are either bit error EPs or frame erasure EPs, simultaneous bit errors and frame erasures are not allowed by `eid-xor.c`.

The program `gen-patt.c` is used to generate error patterns (EPs) using the EID models implemented in the STL (Gilbert and Bellcore models). The EPs will be either frame erasures or bit errors EPs, since the models in the STL do not support mixed frame erasure/bit error mode.

Program `ep-stats.c` examines an error pattern file (either bit error EPs or frame erasure EPs) and displays the actual BER/FER found in the EP and the distribution of number of consecutive errored bits or erased frames.

Program `eid-int.c` interpolates a frame erasure EP such that each synchronism word found in the EP is repeated a user-specified number of times. This is useful to align the frame erasures for codecs that have frame sizes that are an integer sub-multiple of each other (e.g. 10ms codecs and 20 ms codecs). In the latter example, the master EP will be the 20ms one, and the one generated by eid-int would be used for the 10ms codec.

Program `eid-ev.c` performs an error insertion for layered G.192 files. This program can be used to apply errors to individual layers in layered bitstreams such as G.729.1 or G.718.

Program `gen_rate_profile.c` generates random rate/layer switching file.

As a final note, it should be reinforced that the definition of the symbol `PORT_TEST` at compilation time **will** affect the operation of the programs as explained before. If this symbol is defined, functions `open_eid()` and `open_burst_eid()` will always start from the same seed. Therefore, the output of the programs will be the same, unless EID state files are available. When that symbol is not defined at compilation time, the programs will use the run-time library function `time()` to get the seed used in functions `open_eid()` and `open_burst_eid()`.

### 15.4.2. Using bit-error insertion routine

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "eid.h"

#define OVERHEAD 2
#define LSEG 2048L                      /* Frame length is FIXED! */
#define SYNCword 0x6B21

void main(argc, argv)
  int             argc;
  char           *argv[];
{
  SCD_EID        *ber_st;                /* pointer to EID-structure */
  char            ifile[128], ofile[128];/* input/output file names */
  FILE           *ifilptr, *ofilptr;     /* input/output file pointer */
  static int      EOF_detected = 0;      /* Flag to mark END OF FILE */
  double          ber;                   /* bit error rate factor */
  double          gamma;                 /* burst factor */
  static double   dstbits = 0;           /* distorted bits count */
  static double   prcbits = 0;           /* processed bits count */
  short           err_pat[LSEG];         /* error pattern-buffer */
  short           inp[LSEG+OVERHEAD], out[LSEG+OVERHEAD]; /* bit-buffers */


  GET_PAR_S(1, "_File with input bitstream: ................. ", ifile);
  GET_PAR_S(2, "_File for disturbed bitstream: ............. ", ofile);
  GET_PAR_D(3, "_Bit error rate     (0.0 ... 0.50): ........ ", ber);
  GET_PAR_D(4, "_Burst factor       (0.0 ... 0.99): ........ ", gamma);

  /* Open input and output files */
  ifilptr = fopen(ifile, RB);
  ofilptr = fopen(ofile, WB);

  /* Allocate EID buffer for bit errors */
  ber_st = open_eid(ber, gamma);
  if (ber_st == (SCD_EID *) 0)
    QUIT(" Could not create EID for bit errors!\n", 1);

  /* Now process serial soft bitstream input file */
  while (fread(inp, sizeof(short), LSEG+OVERHEAD, ifilptr) == LSEG+OVERHEAD)
  {
    if (inp[0] == SYNCword && EOF_detected == 0)
    {
      /* Generate Error Pattern */
      dstbits += BER_generator(ber_st, LSEG, err_pat);

      /* Modify input bitstream according the stored error pattern */
```

```
      BER_insertion(LSEG+OVERHEAD, inp, out, err_pat);
      prcbits += (double) LSEG; /* count number of processed bits */

      /* Write disturbed bits to serial soft bitstream output file */
      fwrite(out, sizeof(short), LSEG+OVERHEAD, ofilptr);
    }
    else
      EOF_detected = 1;          /* the next SYNC-word is missed */
  }

  if (EOF_detected == 1)
    printf("   --- end of file detected (no SYNCword match) ---\n");
  printf("\n");

  /* Print message with measured bit error rate */
  if (prcbits > 0)
    printf("Measured BER: %f  (%ld of %ld bits distorted)\n",
           dstbits / prcbits, (long) dstbits, (long) prcbits);
}
```

### 15.4.3. Using frame erasure routine

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "eid.h"

#define QUIT(m,code) {fprintf(stderr,m); exit((int)code);}
#define LSEG 2048L                      /* Frame length is FIXED! */
#define OVERHEAD 2
#define SYNCword 0x6B21

void main(argc, argv)
  int             argc;
  char           *argv[];
{
  SCD_EID        *FEReid;                /* pointer to EID-structure */
  char            ifile[128], ofile[128];/* input/output file names */
  FILE           *ifilptr, *ofilptr;     /* input/output file pointer */
  static int      EOF_detected = 0;      /* Flag to mark END OF FILE */
  double          fer;                   /* frame erasure factor */
  double          gamma;                 /* burst factor */
  static double   ersfrms = 0;           /* total distorted frames */
  static double   prcfrms = 0;           /* number of processed frames */
  short           inp[LSEG+OVERHEAD], out[LSEG+OVERHEAD]; /* bit-buffers */


  GET_PAR_S(1, "_File with input bitstream: ................. ", ifile);
  GET_PAR_S(2, "_File for disturbed bitstream: ............. ", ofile);
  GET_PAR_D(3, "_Frame erasure rate (0.0 ... 0.50): ........ ", fer);
  GET_PAR_D(4, "_Burst factor       (0.0 ... 0.99): ........ ", gamma);

  /* Open input and output files */
  ifilptr = fopen(ifile, RB);
  ofilptr = fopen(ofile, WB);

  /* Allocate EID buffer for bit errors frame erasure */
  FEReid = open_eid(fer, gamma);
  if (FEReid == (SCD_EID *) 0)
```

```
    QUIT(" Could not create EID for frame erasure module\n", 1);

  /* Now process serial soft bitstream input file */
  while (fread(inp, sizeof(short), LSEG+OVERHEAD, ifilptr) == LSEG + OVERHEAD)
  {
    if (inp[0] == SYNCword && EOF_detected == 0)
    {
      /* Generate frame erasure */
      ersfrms += FER_module(FEReid, LSEG+OVERHEAD, inp, out);
      prcfrms++;                /* count number of processed frames */

      /* Write (erased) frames to serial soft bitstream output file */
      fwrite(out, sizeof(short), LSEG+OVERHEAD, ofilptr);
    }
    else
      EOF_detected = 1;          /* the next SYNC-word is missed */
  }

  if (EOF_detected == 1)
    printf("   --- end of file detected (no SYNCword match) ---\n");
  printf("\n");

  /* Print message with measured bit error rate */
  if (prcfrms > 0)
    printf("measured FER: %f  (%ld of %ld frames erased)\n",
           ersfrms / prcfrms, (long) ersfrms, (long) prcfrms);
}
```

### 15.4.4. Using layered bitstream error routine (eid-ev)

The demonstration program, `eid-ev.c` demonstrates the use of this module to apply errors in a scalable layered bit stream to simulate a flexible transport channel.

G.192 bitstreams (with sync header), G.192 byte-oriented bitstreams (with sync header) can be processed with this tool. (The byte oriented G.192 input stream is however limited to a maximum frame size of 255). The supported frame error patterns formats for the layers are the G.192 16-bit softbit format (without synchronism header) and the byte-oriented version of the G.192 format (also without synchronism header).

After processing the `eid-ev` program supplies statistics for the layer erasing operations performed for each layer. The statistics reported are the errors applied in the `eid-ev` error application ('erasing rate') as well as the total erasure rate ('total erasure rate'). The total erasure rate includes both input erasures and erasures performed in the `eid-ev` layer error application. When calculating the `eid-ev` 'erasing rate', applying an layer error to an already erased input layers is not counted as an layer erasure. The statistics can be suppressed by using the -q (quiet option).

To enable the simulation of cascaded network elements, the output of one eid-ev operation can be used as input to a subsequent eid-ev operation.

#### 15.4.4.1. Error Application Modes

By default the tool operates in the layered error application mode, in this case errors that hit one layer will also lead to the erasure of higher layers. Optionally, the tool can operate individual error application mode, for this case it is assumed that layers are transported over individual uncorrelated channels. Thus for the individual error application mode an error in one layer does not result in an error in any other layer.

The layered error application mode is simulated by simply truncating the frame according to the error in the lowest layer for that frame. If the core (lowest layer) is hit by a layer error the frame is marked

as a G192_FER frame, otherwise it is marked as a valid G192_SYNC frame with a shortened frame length.

The individual error application mode is simulated by setting the softbits for the individual layers with errors to the value zero. Subsequently the frame is then truncated if the highest layers have consecutive layer errors. If the frame has remaining layer errors (layers with all zero softbits) after attempting truncation, the frame is tagged as a G192_FER frame. If the frame has no remaining layer errors after truncation it is re-tagged as a valid G192_SYNC frame.

Note that a scalable decoder that wants to utilize the valid bits that may remain in the layers of bitstream after the individual error application operation will need to scan G192_FER tagged frames with a non-zero frame length for the layers with valid bits by discarding the layers with all zero bits.

#### 15.4.4.2. Program options for EID-EV

Usage:

```
eid-ev [options] in_bs e0 [e1, ..., eN] out_bs
```

Where:

| | |
|---|---|
| `in_bs` | input encoded speech bitstream file |
| `eX` | error pattern bitstream file (one for each layer) |
| `out_bs` | disturbed encoded speech bitstream file |

options:

| | |
|---|---|
| `-bs mode` | Mode for bitstream (g192 or byte) |
| `-ep mode` | Mode for error pattern file (g192 or byte) |
| `-ind` | Individual layer error application (individual intermediate layers may be erased) |
| `-layers` | Set layering setup in absolute bits default is "-layers 160,240,320,480,640" |
| `-q` | Quiet operation, skip statistics |
| `-h` | Displays this message |
| `-help` | Displays a complete instructive help message |

#### 15.4.4.3. Examples

For example, one frame of the input bitstream will comprise 640 bits for a 32 kbit/s codec operating a frame size of 20 ms, the core rate is 24 kbit/s (480 bits) and the higher layer is 8 kbit/s (160 bits). To apply correlated layered errors to the two layers of the bitstream the following command line may be issued:

```
eid-ev -layers 480,640 inp f.01.g192 f.30.g192 outp.lay
```

where `f.01.g192` and `f.30.g192` are frame error pattern files with 1 % FER and 30 % FER respectively. And where the `inp` is a valid G192 bitstream file (containing frame sizes 0, 480, 640 only) and `outp.lay` is the resulting G192 bitstream file. If individual layer error application is desired the following command may be issued.

```
eid-ev -ind -layers 480,640 inp fer.01.g192 fer.30.g192 outp.ind
```

where `outp.ind` is the resulting G192 bitstream file after individual layer error application.

#### 15.4.4.4. Default layering

Consider a 20-ms frame example given in Figure 73. Here, the default layering is "-layers 160,240,320,480,640", which corresponds to the accumulated layer bitrates of 8, 12, 16, 24 and 32 kbit/s. This default layering requires five error pattern files, e.g.

```
eid-ev inp f.00.g192 f.02.g192 f.06.g192 f.10.g192 f.20.g192 outp
```

Note that a 0 kbit/s input frames are allowed, there are two possible 0-kbit/s frames:
– NoData frame (zero length and a G192_SYNC tag) (see Figure 74).
– A totally erased frame (zero length and a G192_FER tag), (see Figure 77).

#### 15.4.4.5. EID-EV G.192 Input frame examples

Figure 73 and Figure 74 gives input frame examples.

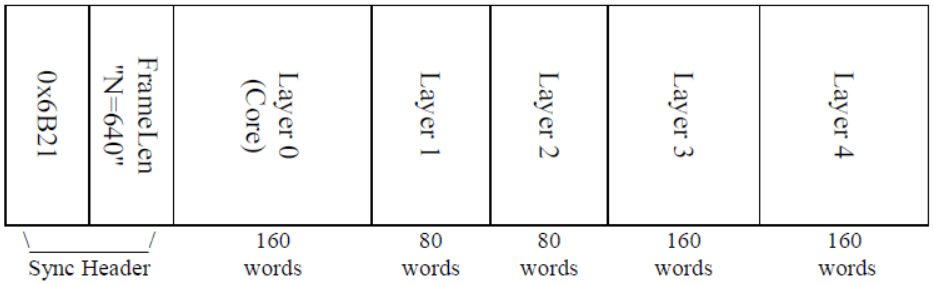



**Figure 73 — Example G.192 input frame with layering "-layers 160,240,320,480,640". This frame has a size of 640 bits and a G192 SYNC(0x6B21) header tag**

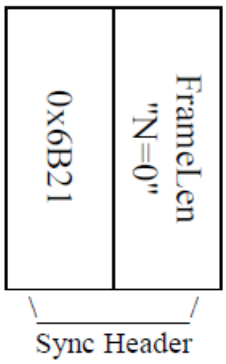



**Figure 74 — Example G.192 NoData frame, sync tag is G192 SYNC and frame length is zero. NoData frames may be used to simulate DTX (Discontinuous Transmission) operation**

#### 15.4.4.6. EID-EV G.192 Output frame examples

Figure 75 gives an output frame example where layer 3 is hit by a layer error in the layered application mode. Since layer 3 is a lower layer to layer 4 and 5, those two layers are also truncated from the frame data. In Figure 76, the layer 1, 3 and 4 are hit by layer errors in layered error application mode. Here, all layers above 0 are truncated because layer 1 is hit. Figure 77 is an example where layer 0, 3 and 4 are hit in layered error application mode. All the layers were deleted because of hit of layer 0, and this is equivalent to a totally erased frame.

In individual error application modes, the hit layers are filled with "0"s. In an example give in Figure 78, where layer 3 is hit, all he bits in layer 3 are changed to "0", and the sync header is changed to G192_FER (`0x6B20`) to indicate the presence of remaining layers with errors. Figure 79 shows an example where layers 1, 3 and 4 are hit in individual error application mode, and layer 3 and 4 are truncated with layer 1 filled with "0"s. Note here that the Sync header is also changed to G192_FER.

Finally, for the case where layers 0, 3 and 4 are hit by layer errors in the individual error application mode, Figure 80 gives the resulting frame. Here, all layers are truncated because layer 0 is hit.

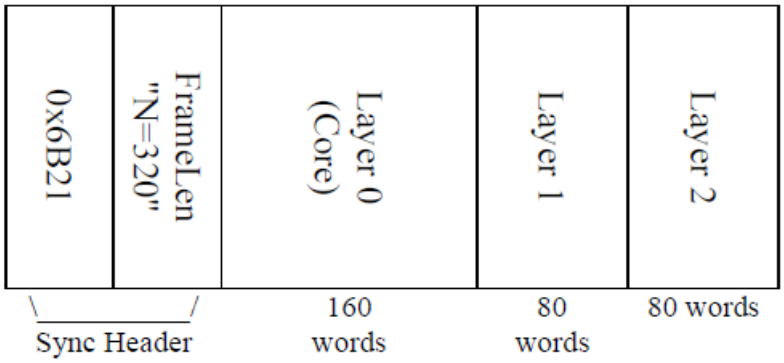



**Figure 75 — Example G.192 output frame when layer 3 is hit by a layer error in the layered error application mode**

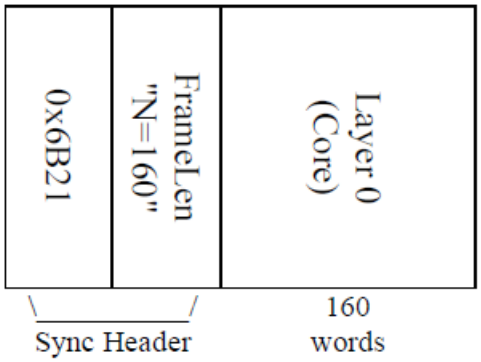



**Figure 76 — Example G.192 output frame when layer 1, layer 3 and layer 4 are hit by layer errors in the layered error application mode**

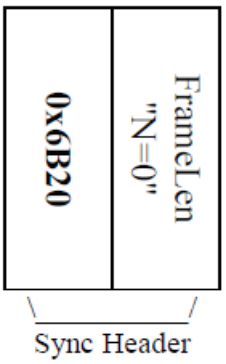



**Figure 77 — Example G.192 output frame when layers 0, 3 and 4 are hit by layer errors in the layered error application mode**

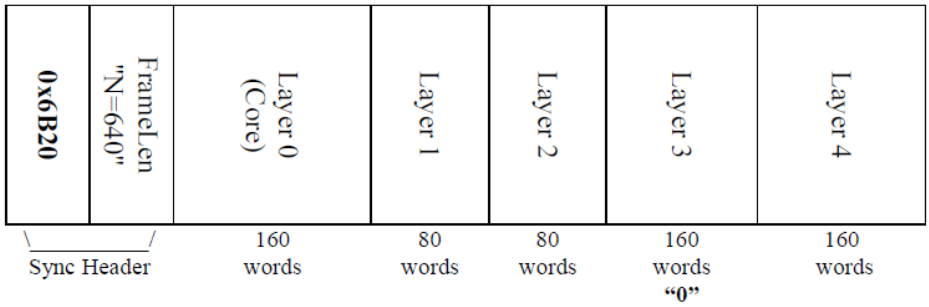



**Figure 78 — Example G.192 output frame when layer 3 is hit by layer errors in the individual error application mode. Note that the Sync header is changed to G192 FER(0x6B20) to indicate the presence of remaining layers with errors**

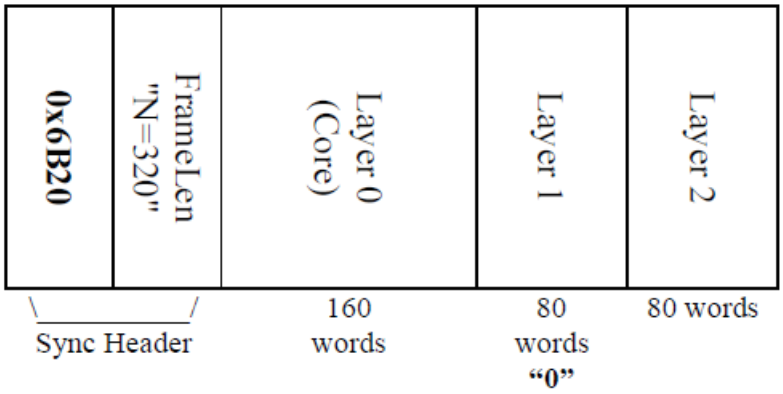



**Figure 79 — Example G.192 output frame when layers 1, 3 and 4 are hit by layer errors in the individual error application mode. Note that the frame is truncated and that the Sync header is changed to G192 FER(0x6B20) to indicate the presence of layers with remaining errors**

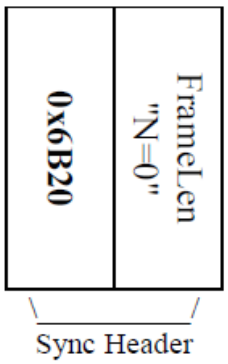



**Figure 80 — Example G.192 output frame when layers 0, 3 and 4 are hit by layer errors in the individual error application mode**

## 16. Duo-MNRU: The Dual-mode Modulated Noise Reference Unit

For evaluation of the quality of a system or equipment, it is important to express the quality measure in a unit suitable for comparison with other reference (or well-known) equipments and systems. A common way of representing these figures is by means of relative units, where the quality is expressed by means of a unique figure, in a unidimensional scale.

But it is insuficient to be unidimensional; the scale must be inequivocal, with a universal meaning. As an example, the ACR scale (*Absolute Category Rating*, [15], Annex B), which is a scale used for listening opinion tests and has five points termed *Excellent*, *Good*, *Fair*, *Poor*, and *Bad*, is inadequate:

besides it shows a continuum of quality points, the meaning of the adjectives are far from universal, varying from language to language, and from person to person. Exchange of information on the performance of these systems and equipments is easier and more consistent with more objective measures. The issue of how the MNRU is to be used as a reference system in subjective tests has been studied in ITU-T Study Group 12, which is described in Recommendation ITU-T P.830 [17] in its Sections 8.2.2 and 11.

The Modulated Noise Reference Unity (MNRU) was introduced as a means to controlled degradations that are representative of the non-linear distortion introduced by waveform coding techniques. Initially aiming at evaluating the quality of log-PCM waveform coding systems, it has been used in the process of generating several ITU-T standards, such as the ITU-T G.726 (32 kbit/s), G.722, G.728, and G.729.

The concept of such reference unit was published in [83]. The first system aimed at was the PCM coding with logarithmic compression (today world-wide available by means of the Recommendation ITU-T G.711), whose main characteristic is to have a considerably uniform signal-to-noise ratio (SNR) over a wide range of amplitudes. Moreover, the quantizing noise is correlated to the signal: if no signal is present, no quantization noise is produced[33], and large signals will produce more quantization noise than small ones. Therefore, the main characteristic of this reference unit should output speech corrupted by a speech-correlated noise.

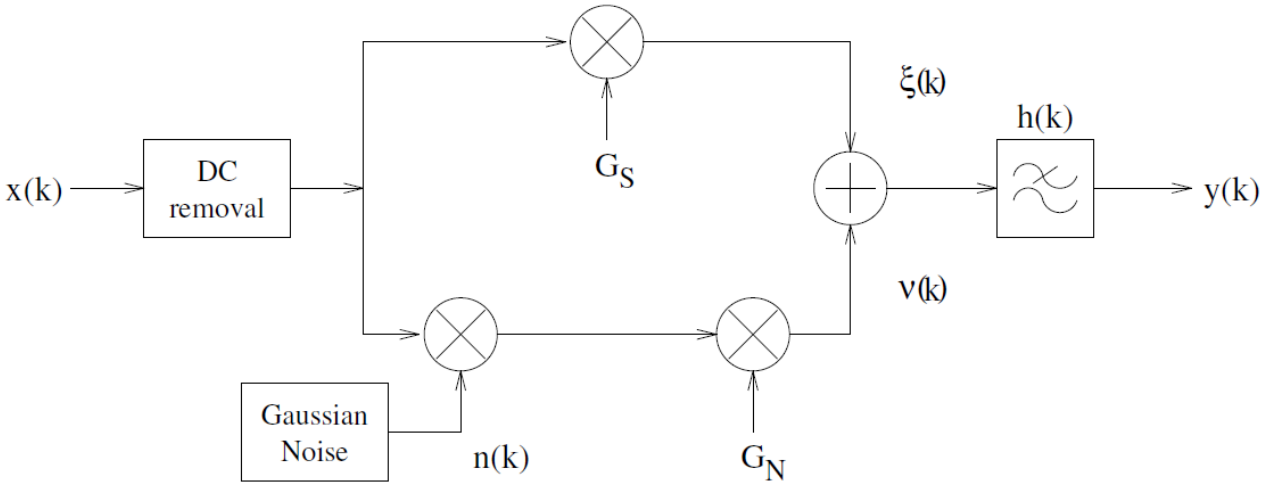



**Figure 81 — Block diagram of the "digital" MNRU. The bandwidth of the output filter h(k) is 0–3400 Hz for the narrowband case, and 0–7000 Hz for the wideband case**

In [83], the speech-correlated noise generation was based on a double-balanced ring modulator, controlled by the input speech signal, which modulates a noise carrier generated by a noise generator having a relatively uniform energy distribution, there in the range of 0—20kHz. This correlated noise is then added to the input signal, with gains applied such that a controlled signal-to-noise ratio is obtained in the output, after the 300—3400Hz band-limiting filter.

With the 1996 revision of the MNRU description published in Recommendation ITU-T P.810[34], specific guidelines were given for "digital implementations"[35], eliminating many of the ambiguities

[33] This is obviously academic, because always there will be idle noise, among others, in the absence of an input signal.

[34] Formerly known as Recommendation ITU-T P.81.

possible in earlier descriptions [84], as explained in the STL92 manual [85], Chapter 8. Figure 81 shows a block diagram of the "digital" MNRU. Also, this implementation allows for transparent operation on narrowband or wideband speech, hence being known as Dual-mode MNRU, or "Duo-MNRU", for short.

```
Special caution must be made when selecting frequency bandwidth of the input
 signal. The current implementation of tool is only useful for narrowband (200 -
- 3400 Hz) and wideband (100 -- 7000 Hz) signal only, and should not be applied
 to superwideband or fullband signal.
```

**Figure 82**

### 16.1. Description of the Algorithm

The de-facto reference implementation of the MNRU[36] is the same of the original description, whose specification can be found in Recommendation ITU-T P.810 [16] (formerly Recommendation ITU-T P.81 [84]). This Recommendation describes two MNRU schemes, one called *Narrow-band MNRU*, and another, *Wideband MNRU*. Wideband MNRU is applicable to systems where wideband speech (70—7000Hz) is expected, whereas Narrow-band MNRU is for telephone bandwidth (300—3400Hz). Both narrowband and wideband MNRUs are implemented in this version of the ITU-T Software Tools Library.

The basic block diagram of the P.810 MNRU is found in Figure 81. In summary, there are two paths, one called *signal path*, another called *noise path*. In the noise path, gaussian noise (uniform in a range at least the cutoff frequency of the low-pass filter in the output of the MNRU) is modulated by the incoming signal. The result is then added with the output from the signal path. The gains are set such that the gain (in dB) applied in the output of the noise path is the signal-to-correlated-noise ratio $Q$, in the output of the band-pass filter, as calculated in the section to follow.

In analytical terms, the signal corrupted by the modulated noise $y(k)$ is

$$y(k) = (G_s x(k) + G_n x(k) n(k)) * h(k)$$

where $G_s$ is the gain of the signal path, $G_n$ is the gain of the noise path, $x(k)$ is the input signal, and $n(k)$ is the gaussian noise signal; the symbol $*$ means convolution, and $h(k)$ is the band-pass filter.

If we suppose that the band-pass filter has $|H(f)| = 1$ in its pass band, and calling $Q$ the signal-to-noise-ratio (SNR) at its output, we may write:

$$10^{Q/10} = \frac{\sigma_\xi^2}{\sigma_\nu^2} = \frac{E\xi^2(k)}{E\nu^2(k)} = \frac{G_s^2 Ex^2(k)}{G_n^2 Ex^2(k) n^2(k)}$$

But $x$ and $n$ are uncorrelated, and the noise is gaussian with mean 0 and variance 1 (N(0,1)):

$$10^{Q/10} = \left(\frac{G_s}{G_n}\right)^2 \frac{\sigma_x^2}{\sigma_x^2 \sigma_n^2} = \left(\frac{G_s}{G_n}\right)^2$$

or

$$Q = \Gamma_s + \Gamma_n$$

$$\Gamma_s = 20\log_{10}(G_s)$$

---

35 The revised P.81 define a "digital implementation" either as a digital hardware implementation or as a software implementation of the MNRU.

36 Developed by the British Telecom and licenced to Malden Electronics.

$$\Gamma_n = -20\log_{10}(G_n)$$

If we set $\Gamma_s = 0$ ($G_s = 1$), $Q$ is exactly $\Gamma_n$ (or, $G_n = 10^{-Q/20}$), i.e., the SNR is the gain (in dB) of the noise path and the previous expression may be written as:

$$y(k) = [x(n) + 10^{-Q/20} x(k) n(k)] * h(k)$$

or approximately

$$y(k) = x(k) + 10^{-Q/20} x(k) n(k)$$

in the passband region of $H(f)|$ .

When both $G_s$ and $G_n$ are non-zero, the MNRU is in an operational mode normally called *Modulated-noise mode*. This is the most common operation mode.

Alternatively, if one consider $G_s = 0$, the output of the algorithm is only the correlated noise, at a level $Q$ dB below the input signal. This is *Noise-only mode*.

If, on the other hand, $G_n = 0$, the output of the algorithm is the input signal filtered by $h(k)$, with a gain $G_s$; this is the *Signal-only mode*.

## 16.2. Implementation

This implementation of the MNRU algorithm can be found in the module `mnru.c`, with prototypes in `mnru.h`. A thorough characterization of this module is presented in [86]. The previous version of the ITU-T STL MNRU was applicable to narrowband signals and evolved from a Fortran implementation which had been used by several laboratories, especially by participants of ETSI's contest for the second generation of Digital Mobile Radio Systems, and was originally written by experts at CSELT/ Italy (sometimes referred as *CSELT MNRU*), an implementation fully compliant with the narrowband MNRU specification available in the then-in-force P.81 [84].

With the revision of MNRU specification, several changes had to be made to the STL92 MNRU:

- The need for an upsampling by a factor of 5 before summation of the modulated noise to the input speech was eliminated because now for digital implementations, the bandwidth of the multiplicative noise shall have the bandwidth of the input signal. In the previous version, the noise bandwidth had to be 20 kHz.
- The output filter for digital implementations shall be a low-pass filter, instead of the bandpass filter of the previous version of the MNRU
- The need of an input speech DC-component removal filter was added to the specification.

These changes, especially the elimination of the 5:1 speech data rate conversion, allowed for the implementation of both the narrowband and the wideband MNRU within the same C function, when the output filter is adequately designed [86], pp.7—12.

The random number generator (RNG) algorithm was also modified to allow for real-time implementations, and the solution adopted was based on Aachen University's approach used by the Host Laboratory for the ITU-T G.729 Selection Tests.

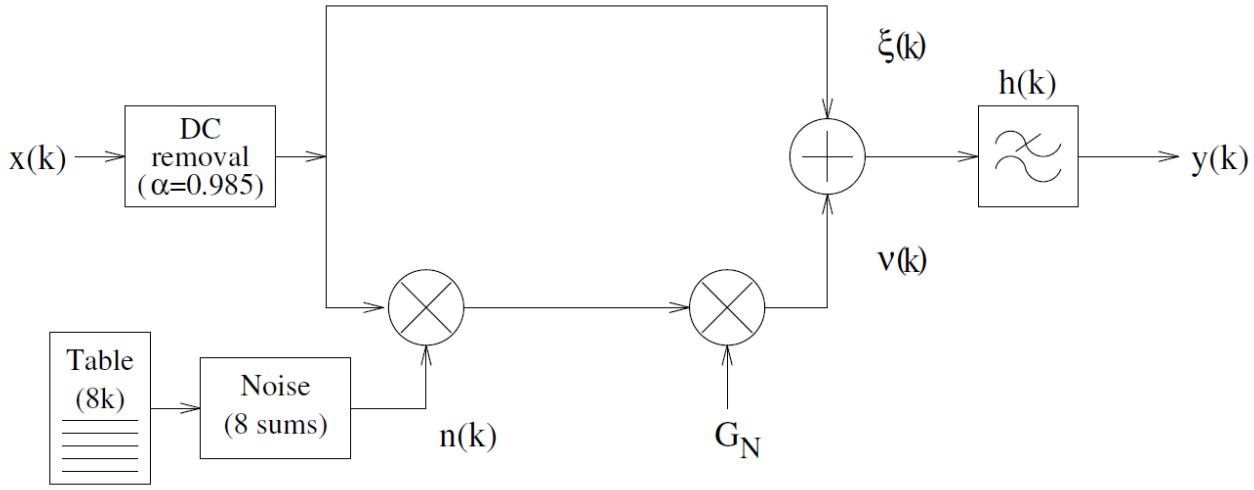



**Figure 83 — STL MNRU implementation**

The block diagram of the MNRU implemented in the STL is in Figure 83.

The MNRU works internally on a sample-by-sample basis but for ease of interface with other speech coding functions, access to it is made on a sample block basis. It should be noted however that the filters have memory, as well as do the random number generator, hence state variables are needed. These state variables have been arranged as fields of a structure whose `type` name is `MNRU_state`. The fields of the structure are:

| | |
|---|---|
| *seed* | RNG's seed |
| *signal_gain* | Gain of the signal path |
| *noise_gain* | Gain of the noise path |
| *vet* | Array for intermediate data |
| *last_xk* | $x(k-1)$ used as memory for the DC-removal filter |
| *last_yk* | $\xi(k-1)$ (see Figure 83), used as memory for the DC-removal filter |
| *DLY[2][2]* | Memory of delayed samples for two second-order stages (first index) for first- and second-order delays (second index) |
| *A[2][2]* | Numerator coefficients for the stage indicated by the first index and delay-order inidcated by the second index |
| *B[2][2]* | Denominator coefficients for the stage indicated by the first index and delay-order inidcated by the second index |
| *rnd_state* | State structure for MNRU's random number generator. Detailed description is found in the section on the random number generator. |
| *rnd_mode* | Operational mode of the random number generator |
| *clip* | Number of samples clipped in the noise-insertion process |

The values of the fields shall not be altered by the user.

**Filters in the MNRU module**

The composite frequency response of the narrowband and wideband MNRU filters is shown in Figure 84. Figure 86 shows the contribution of the output low-pass filter for (a) the narrowband, and (b) the wideband cases. Figure 85 shows the effect of the input DC-removal filter for (a) the narrowband and (b) the wideband operation modes of the MNRU. Details on the design of the output low-pass filters are given in [86]. The frequency responses have been obtained by exciting the MNRU module with digital sinewaves and computing the ratio of input and output signals, in dB.

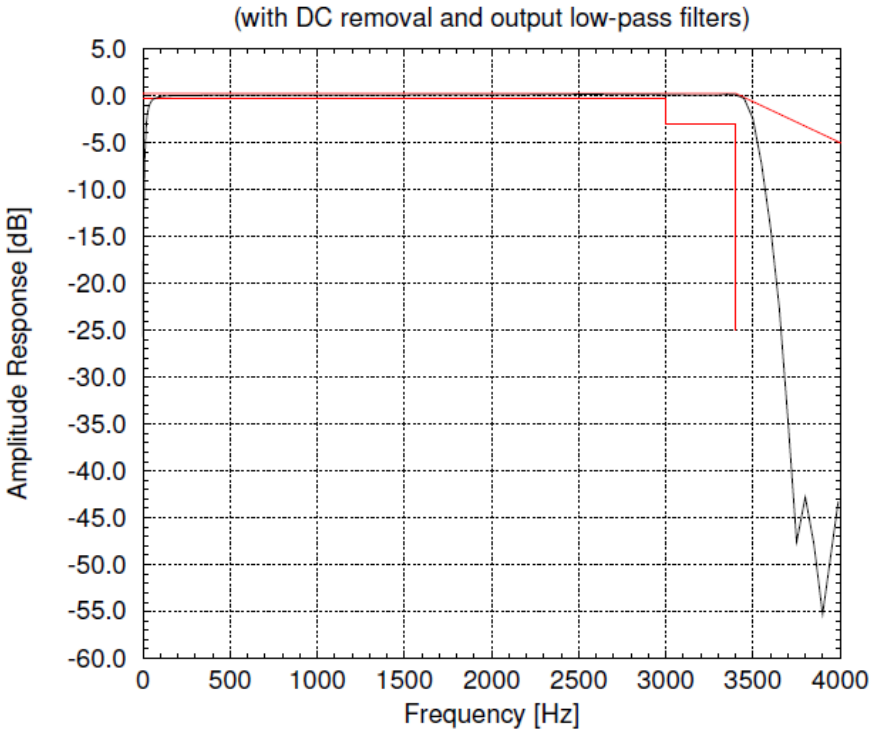



**Figure 84-a — Narrowband Duo-MNRU**

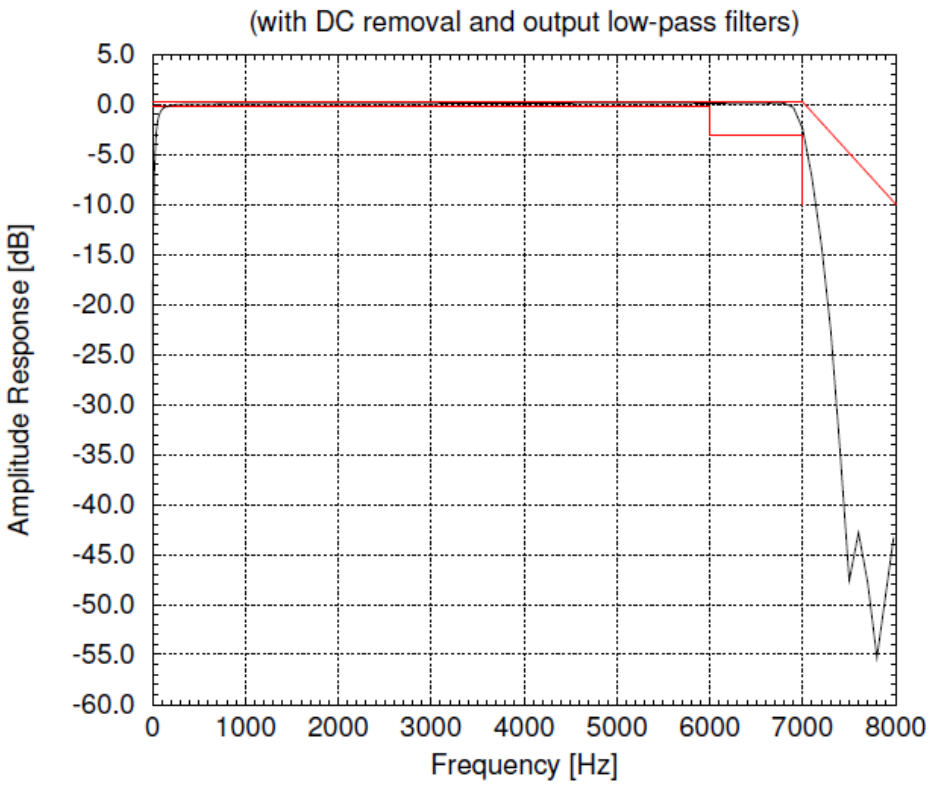



**Figure 84-b — Wideband Duo-MNRU**

**Figure 84 — Total frequency response of the Duo-MNRU filters**

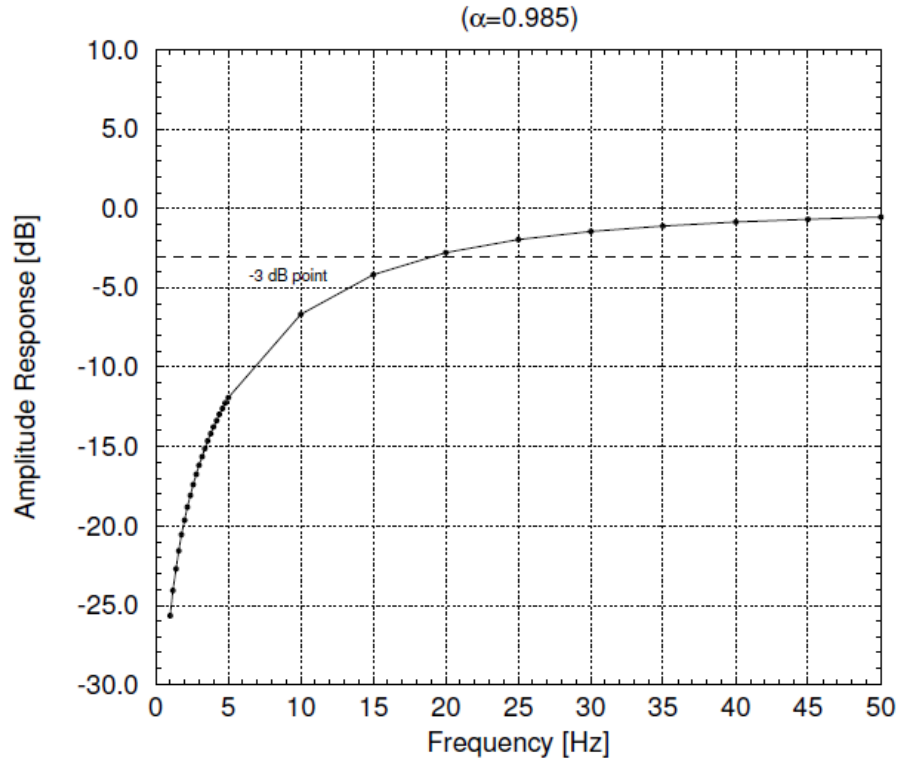



**Figure 85-a — Narrowband Duo-MNRU**

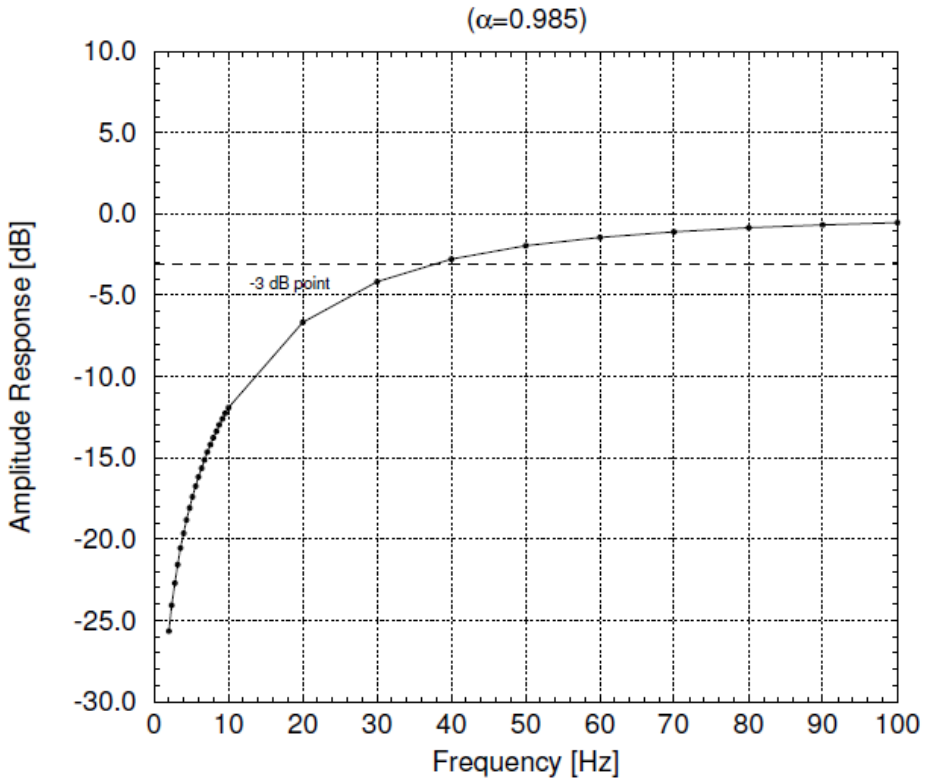



**Figure 85-b — Wideband Duo-MNRU**

**Figure 85 — DC removal filter for the Duo-MNRU**

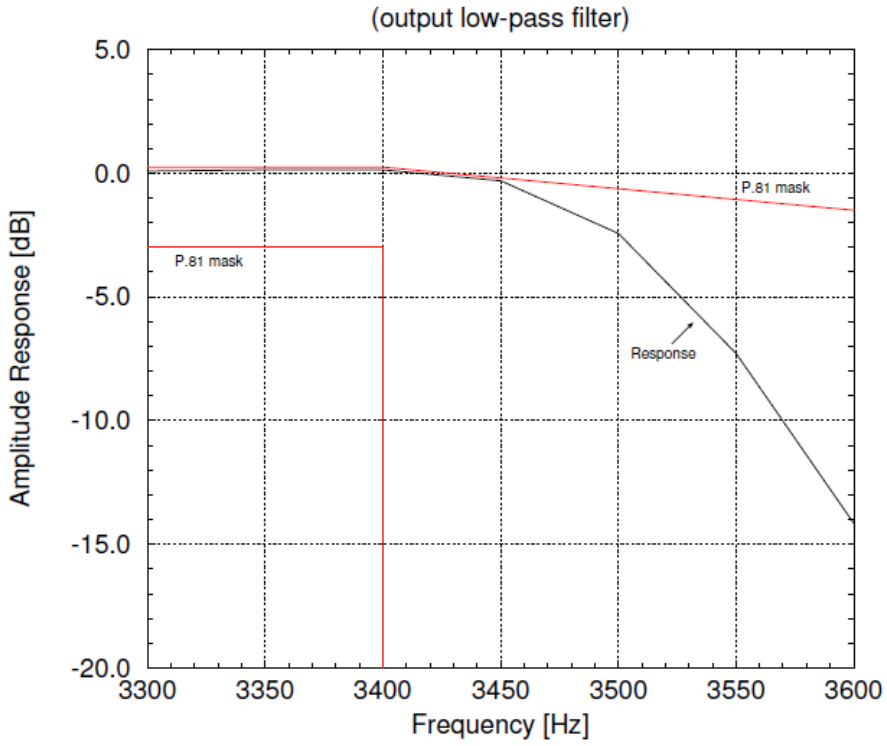



**Figure 86-a — Narrowband Duo-MNRU**

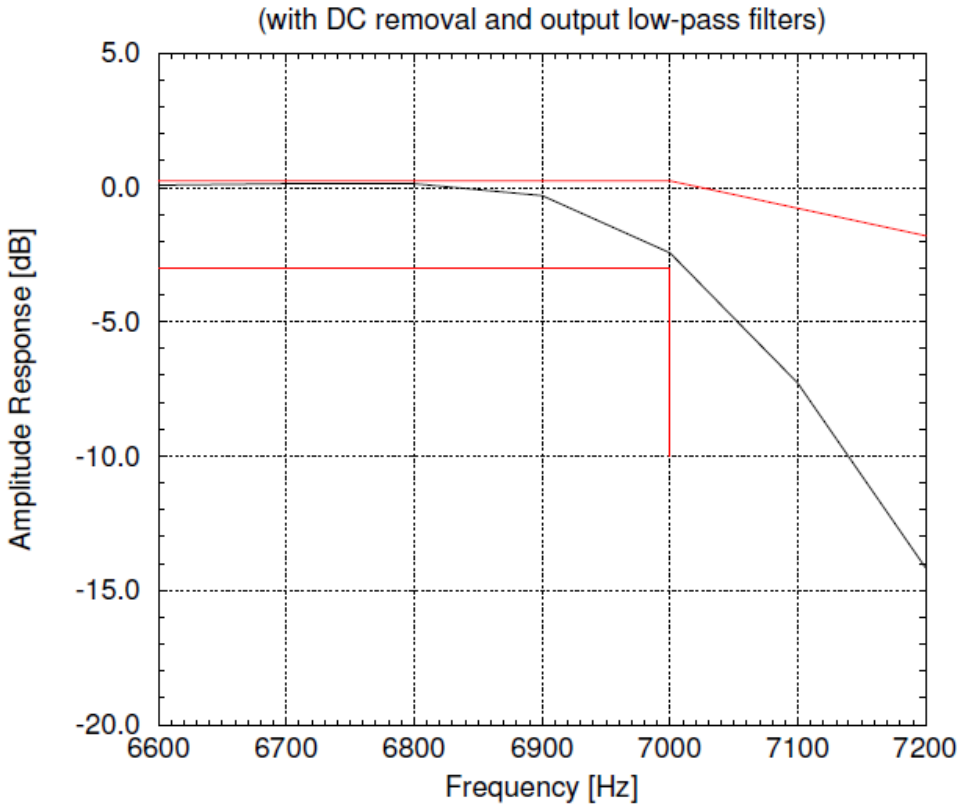



**Figure 86-b — Wideband Duo-MNRU**

**Figure 86 — Output low-pass filter for the Duo-MNRU**

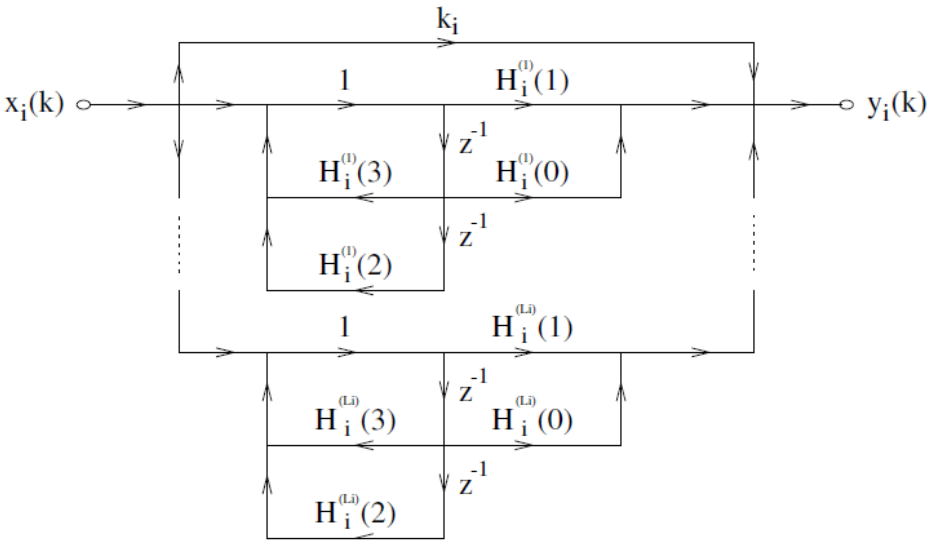



**Figure 87 — MNRU Filters Structure**

The input DC-removal filter was implemented using a first-order IIR pole-zero filter defined by

$$H_i(z) = \frac{1 - z^{-1}}{1 - \alpha z^{-1}}$$

with $\alpha = 0.985$. Its -3dB point is at 16 Hz for the narrowband case and at 38 Hz for the wideband case.

The output low-pass filter was implemented using a second-order cascade-form IIR filter with two-sections as illustrated in Figure 87 and defined by the equation:

$$H_i(z) = A\prod_{k=1}^{2}\frac{a_{0k} + a_{1k}z^{-1} + a_{2k}z^{-2}}{1 + b_{1k}z^{-1} + b_{2k}z^{-2}}$$

IIR filters were chosen because of their low computational complexity when compared to FIR implementations, allowing for a more efficient MNRU implementation.

**Random Number Generator for the MNRU module**

The random number generator (RNG) used in this implementation was chosen using the following criteria:

– the desired value for Q, $Q_d$, and the measured Q, $Q_m$, should be very close for a wide range of Q, e.g., Q from 0 to 50 dB.
– it should show a good approximation of a gaussian distribution. This is needed because it is specified in P.810 and more importantly because uniform distributions do not allow good matching between the desired and measured values of Q.
– the algorithm needed to be portable (i.e., identical results are got in different platforms if the same seed is given).

The RNG chosen to be used in the STL92 version of the MNRU was based on Knuth's Subtractive Method [87], [81], Parts 3.2—3.3, which generates adequate random sequences but is computationally intensive and was too complex to be implemented in a real-time digital hardware MNRU.

The implementation used in the ITU-T G.729 8 kbit/s speech codec selection tests was based on a gaussian-noise table lookup, in a manner similar to Malden Electronic's MNRU implementation.[37] This approach is considerably less computationally intensive than the STL92 approach, and was used to further reduce the complexity of the MNRU implementation.

After several experiments [86], a table with 8192 gaussian samples was chosen to be used, which is randomly and uniformly accessed 8 times (i.e., an eight-time sample accumulation) to be used by the MNRU algorithm. The gaussian table itself is generated in run-time (rather than being stored in the data memory of the source or object code) using the Monte-Carlo substitution algorithm. The Monte-Carlo algorithm uses a linear congruential generation (LCG) algorithm defined by

$$I_j = 69'069 I_{j-1} + 1 \left(\mathrm{mod} 2^{32}\right)$$

which is converted to numbers in the range [0..1] using the upper 24 bits of the 32-bit unsigned long $I_j$. $I_0$ is a fixed seed equal to 314159265. This algorithm is used to generate the necessary initial random samples for the substitution algorithm.

Once the table has been filled, during the normal operation of the MNRU, eight successive samples are drawn (uniformily) from the table using a different LCG algorithm

$$L_j = 253 L_{j-1} + 1 \left(\mathrm{mod} 2^{24}\right)$$

of which the upper 13 bits are used to generate random numbers uniformly distributed between 0 and 8191. $L_0$ is a fixed seed equal to 12345. Both LCGs were implemented as in Aachen University's MNRU implementation.

[37] Malden's MNRU uses a ROM table derived from a Gaussian distribution with 4096 samples uniformly distributed throughout the table. An address in the table is uniformly sampled four times and accumulated to form a gaussian noise sample.

Since different ranges are necessary for table filling and for gaussian sample generation, two different LCG random number generators were used to avoid any additional calculations due to range convertion and to reduce the software load.

Since the Monte-Carlo RNG is used only at startup time, it is not necessary to keep any state variables for it. The sample-drawing RNG however needs to keep stored the previously generated index, which is stored in a structure of type `RANDOM_state`, whose only field is (as defined in `mnru.h`)[38]:

*gauss* Index for next random number;

The field in `RANDOM_state` should not be altered by the user in any situation.

The operational modes are defined in `mnru.h`:

```
#define RANDOM_RUN 0
#define RANDOM_RESET 1
```

The noise modulation routine is `MNRU_process`, which is described next.

### 16.2.1. `MNRU_process`

**Syntax:**

```
#include "mnru.h"
double *MNRU_process (char _operation_, MNRU_state _*s_, float _*input_,
                      float _*output_, long _n_, long _seed_, char _mode_,
                      double _Q_);
```

**Prototype:** `mnru.h`

**Description:**

Module for addition of modulated noise to a vector of *n* samples, according to Recommendation ITU-T P.810, for either the narrowband or the wideband model. Depending on the *mode*, this function:

- adds modulated noise to the *input* buffer at a SNR level of *Q* dB, saving to *output* buffer (*mode*==`MOD_NOISE`);
- puts into *output* only the noise, without addition of the original signal (*mode*==`NOISE_ONLY`);
- produces in the *output* a filtered-only (no noise added) version of the `input' samples` (*mode*==`SIGNAL_ONLY`);

The symbols `MOD_NOISE`, `NOISE_ONLY`, and `SIGNAL_ONLY` are defined in `mnru.h`.

Although the MNRU algorithm operates on a sample-by-sample basis, `MNRU_process` handles the input data in blocks of *n* samples, for better computational efficiency.

The implementation of the MNRU algorithm has three operational states, called `MNRU_START`, `MNRU_CONTINUE` and `MNRU_STOP`. With `MNRU_START`, the state variables are set, as well as memory is allocated for the intermediate data, and this needs to be the first operation with the algorithm. Differently from the speech voltmeter module, after the initialization of the state variables, the normal calculations are carried out for the first block of data. Once reset, the algorithm changes the operation state to `MNRU_CONTINUE`, and the next calls to the MNRU algorithm will skip the reset operation. With the last block, it is adivisable to release the memory allocated to the intermediate data. This is accomplished by calling `MNRU_process` with the operational state set as `MNRU_STOP`. These three operational states are defined in `mnru.h` as follows:

```
#define MNRU_START     1
```

[38] The use of a structure instead of a single variable in the parent structure (`MNRU_state`) allows for unimplemented features to be easily added in a later version of the algorithm.

```
#define MNRU_CONTINUE  0
#define MNRU_STOP     -1
```

**Variables:**

| | |
|---|---|
| *operation* | One of the defined operation status: `MNRU_START`, `MNRU_STOP`, `MNRU_CONTINUE`. |
| *s* | A pointer to a `MNRU_state` structure. |
| *input* | Pointer to input float-data vector; must represent 8 or 16 kHz speech samples. |
| *output* | Pointer to output float-data vector; will represent 8 or 16 kHz speech samples. |
| *n* | Long with the number of samples (`float`) in input. |
| *seed* | Initial value for random number generator. |
| *mode* | Operation mode: `MOD_NOISE`, `SIGNAL_ONLY`, `NOISE_ONLY` (description above). |
| *Q* | Double defining the desired value for the signal-to-modulated-noise *Q* for the output data. |

Please note that new values of *seed*, *mode*, and *Q* are considered only when *operation* is `MNRU_START`, because they are considered as INITIAL state values. Therefore, when the operation is not `MNRU_START`, they are ignored.

**Return value:**

Returns a `(double *)NULL` if not initialized or if initialization failed; returns a `(double *)` to an intermediate data vector if reset was successful or is in the `MNRU_CONTINUE` ("run") operation state.

### 16.3. Portability and compliance

In the development of this module, several steps were taken to assure its compliance to Recommendation ITU-T P.810, which included:

– agreement of expected and measured Q values for tones and speech,
– addition of partial files,
– level of output files,
– frequency response of built-in filters.

Additionally to these objective measurements, a subjective test was performed. The results of this test are found in [86], where it was concluded that the new MNRU implementation conforms to the P.810 and also behaves more closely to the hardware MNRU than the previous STL92 version.

Additionally to the conformance tests, the algorithm was tested for portability using a 1kHz tone file as input to the algorithm with Q values ranging from 0 to 50 dB in 5 dB steps, and also for the algorithm in the `SIGNAL_ONLY` mode. The processed test files were then compared the the reference processed files (generated on a HP workstation). Test and reference files should be identical. The algorithm was found to compile and execute correctly on MS-DOS under Borland Turbo-C++ 1.0 and under the MS-DOS port of the GNU-C compiler (gcc), on a HP UNIX workstation with cc (non-ANSI) and gcc, on a Sun workstation with cc (non-ANSI) and also on VAX VMS and APX computers.

## 16.4. Example code

### 16.4.1. Description of the demonstration programs

One demonstration program is provided for the MNRU module, mnrudemo.c. Irrespective of whether the 16-bit, linear PCM input file is sampled at 8 or 16 kHz, program `mnrudemo.c` will add the multiplicative noise signal to the input signal at the user-defined *Q* level and produce as output a 16-bit, linear PCM file. Optionally, the program can produce a signal-only file (equivalent to a very high Q value) or a noise-only file (the signal path is disconnected).

### 16.4.2. Simple example

The following C code gives an example of a possible use of the Duo-MNRU module. The input file speech is added to a multiplicative noise at a SNR defined by parameter Q. All samples in the file are processed.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "ugstdemo.h"
#include "mnru.c"               /* ... Include MNRU module ... */
#include "ugst-utl.c"           /* ... Include of utilities ... */

#define BLK_LEN 256
main(argc, argv)
  int             argc;
  char           *argv[];
{
  /* File variables */
  char            FileIn[80], FileOut[80];
  FILE           *Fi, *Fo;
  MNRU_state      state;
  short           Buf[BLK_LEN];
  float           inp[BLK_LEN], out[BLK_LEN];
  double          QdB;
  long            l;
  char            MNRU_mode = MOD_NOISE, operation;

  /* Read parameters for processing */
  GET_PAR_S(1, "_Input File: ................. ", FileIn);
  GET_PAR_S(2, "_Output File: ................ ", FileOut);
  GET_PAR_D(3, "_Desired Q: .................. ", QdB);

  /* Check for parameter 4 to change MNRU operation mode */
  if (argc > 4)
  {
    MNRU_mode = toupper(argv[5][0]);
    if (MNRU_mode == 'S')       /* Signal-only mode */
      MNRU_mode = SIGNAL_ONLY;
    else if (MNRU_mode == 'M')  /* Modulated noise, the default mode */
      MNRU_mode = MOD_NOISE;
    else if (MNRU_mode == 'N')  /* Noise-only mode */
      MNRU_mode = NOISE_ONLY;
    else
    {
      fprintf(stderr, "Bad mode chosen; use M,N,or S \n"); exit(2);
    }
  }

  /* Opening input and output files */
  Fi = fopen(FileIn, RB);
```

```
  Fo = fopen(FileOut, WB);

  /* INSERTION OF MODULATED NOISE ACCORDING TO P.810 (FEB.96) */

  /* Set operation as start */
  operation = MNRU_START;

  /* Process for all samples in file */
  while ((l = fread(Buf, sizeof(short), BLK_LEN, Fi)) != NULL)
  {
    /* Convert data from 16-bit short to normalized float */
    sh2fl_16bit((long) l, Buf, inp, 1);

    /* MNRU processing */
    MNRU_process(operation, &state, inp, out, l, 314159265L, MNRU_mode, QdB);

    /* Change operation mode: START --> CONTINUE */
    if (operation == MNRU_START)
      operation = MNRU_CONTINUE;

    /* Convert from normalized float to short (hard clip and rounding) */
    fl2sh_16bit((long) l, out, Buf, 1);

    /* Save data to file */
    fwrite(Buf, sizeof(short), l, Fo);
  }

  /* Stop mode: Deallocation of memory, but process 0 samples */
  operation = MNRU_STOP;
  MNRU_process (operation, &state, inp, out, 0L, 0L, 0, (double) 0.0);

  /* Finalizations */
  fclose(Fi);
  fclose(Fo);
  return(0);
}
```

## 17. SVP56: The Speech Voltmeter

### 17.1. Description of the Algorithm

The specification for the measurement of the active level of speech signals is given in Recommendation ITU-T P.56 [25] [14], and is commonly referred as *speech voltmeter*[39]. Besides the description above, there is complementary information in the ITU-T *Handbook on Telephony* [88], section on `Measurement of Speech'.

In summary, the P.56 algorithm takes samples of a signal in the speech bandwidth and calculates its active speech level. This means that silence and idle noise are not taken into account when calculating the level of the signal. Furthermore, *structural pauses* (pauses in the range of 250 ms which are inherent to the uterance process) are considered in the measurements, but *grammatical pauses* (pauses between phrases or to emphasise words, generally in the range of 300 ms or more) are excluded, because they do not contribute to speech subjective loudness [88].

To decide about the activity or inactivity of a speech segment, the algorithm calculates an envelope waveform, or short-term mean amplitude, such that pauses shorter that 100 ms are not excluded,

[39] After the British Telecom and Malden Ltd.'s SV6 Speech Voltmeter.

but pauses longer than 350 ms are[40]. A signal is considered *active* when its short-term mean level (envelope) exceeds a threshold level (or *margin* of ) 15.9 dB below the prevailing speech voltage[41], and also during short gaps between such bursts of activity.

A word of caution must be given here: the above mentioned margin above of 15.9 dB has been optimized for speech with a low level of background noise. This means that in the case of generation of material for listening subjective tests, once the original files have been processed (already level equalized), especially by processes that add significant amount of noise to the files (e.g. MNRU for low values of *Q*), the P.56 algorithm shall not be utilized for re-equalization. Since the noise introduced by the processing algorithm will be far above the threshold discussed, the P.56 algorithm will generate wrong measurements of the active level and speech activity. A practical way to observe whether the P.56 may be utilized on processed files is to observe the activity factor: if it increases significantly in relation to the original file's activity factor, then the use of the P.56 for re-equalizations should be discarded.

Another operating assumption for the P.56 Recommendation suggests that the input signal be band-limited (300—3400 Hz for telephony band signals and 100—7000 Hz for wideband signals, as given in Table 3 and Figure 2 of P.56. The revision of P.56 made in 2011 [14] now states that the input signal should be bandlimited to 50—14000 Hz for superwideband signals and 20—20000 Hz for fullband signals, as given in Annexes B and C, respectively.

The speech voltmeter algorithm is expressed in terms of discrete operations. Because of this, a minimum sampling frequency must be chosen, and the specification in P.56 gives it as 600 Hz. This is well below Nyquist frequency of the digitized sample's normally used for telephony applications, either 8000 or 16000 Hz, which is explained by the fact that the matter of interest here is not signal's frequency content's information, but only signal statistics. This is one of the unspecified details of the P.56 that may cause implementations to differ.

After considering an input sample $x_i$, the speech voltmeter performs two operations. First, the total energy of the signal is calculated (*sq*), updating also the number of samples *n* and signal's (long-term) mean level *s*. Second, the envelope (or short-term mean level) *q* of the signal is extracted using a second-order exponential filtering:

$$p_i = g \cdot p_{i-1} + (1-g) \cdot |x_i|$$

$$q_i = g \cdot q_{i-1} + (1-g) \cdot p_i$$

with initial states $p_0 = q_0 = 0$ and the quantity $g$ defined as:

$$g = \exp(-1/(f \cdot T))$$

for $f$ as the sampling frequency, in Hz, and $T$, a time constant for smoothing, equal to 0.030s (30 ms).

With the envelope calculated, the algorithm calculates the number of times that the envelope exceeds each of the threshold levels. The thresholds are represented in a vector $c$ of $B-1$ positions, where $B$ is the resolution (number of bits) of the samples. The values in this vector range from half the maximum possible amplitude down to (or less than) one LSB (Least Significant Bit). In terms of practical implementations, the values of $c_j$ are a power of 2:

---

[40] Users will perceive a pause when it lasts more than about 350 ms.

[41] The margin of 15.9 dB has been chosen to be comfortably above the circuit noise, while causing few false detections or failures to detect, having being determined by subjective experiments.

$$c_j = 2^j,\ j = 0 \cdots B-2$$

There are three possible cases[42]:

- **the envelope exceeds the threshold** $c_j$: increment the *activity counter* for the quantization level $j$, $a_j$, and set the timer vector (or *hangover counter*) $h_j$ to zero. This operation means that the segment is active as far as the level $j$ is concerned, and so the hangover counter must be set to zero, as well as the number of active samples ($a_j$) incremented.
- **the envelope does not exceed the threshold level, but the hangover counter** $h_j$ **is less (or shorter) than** $I$ **samples**: this means that, besides the sample being into a pause segment (because the level is below the threshold), it is a structural pause (because the time spent since the last activity burst is less than 200ms). Therefore, the action here is to increment the activity counter, as well as the hangover counter for the level $j$.
- **the envelope does not exceed the threshold level and the hangover time exceeds** $I$ **samples**: this means that the sample is into a pause (because the level is below the threshold); moreover, it is a grammatical pause (because the time spent since the last activity burst is more than 200ms). Therefore, no increments are done.

Then, after all the samples of interest have been considered, three quantities have been accumulated:

1) total number of samples, $n$;
2) signal energy, $sq$;
3) an activity count $a_j$ for each threshold level $c_j$, $j = 0 \cdots B-2$.

The active level can be evaluated from these three parameters, as follows. First, the long-term mean level is calculated:

$$L = 10\log_{10}(sq/n) - 20\log(r)$$

and the activity counter and threshold vectors are converted to dB:

$$A_j = 10\log_{10}(sq/a_j) - 20\log(r)$$

$$C_j = 20\log_{10}(c_j) - 20\log(r)$$

where $r$ is the 0 dB reference point for the measurements[43].

In sequence, the difference between $A_j$ and $C_j$ is calculated for each $j$. When this difference lyes below the margin $M$ (15.9 dB), then the active level[44] $A$ is found by interpolating between this level $\hat{\text{jmath}}$ and level $\hat{\text{jmath}}-1$ (i.e., the nearest level $k$ where $A_k - C_k > M$, what gives $k = \hat{\text{jmath}}-1$), using a bipartition (binary) interpolation algorithm. There are three special cases here:

- When $\hat{\text{jmath}} = 0$, then the active level is zero;
- When $/A_{\hat{\text{jmath}}} - C_{\hat{\text{jmath}}} - M/ \le \delta$ (where $\delta$ is the the given tolerance, or degree of accuracy): the active level is $A_{\hat{\text{jmath}}}$.
- When $/A_{\hat{\text{jmath}}-1} - C_{\hat{\text{jmath}}-1} - M/ \le \delta$: the active level is $A_{\hat{\text{jmath}}-1}$.

The tolerance $\delta$ is not specified in P.56, hence being implementation-dependent.

---

42 It is interesting to remark that the lower the threshold level, the greater the activity count for that level will be.

43 This is another unspecified detail in P.56. This implementation's choice is given in next section.

44 The true active level is defined as the one which exceeds the threshold used for its derivation by a $M = 15.9$ dB.

Once the active level is found, the only remaining point is the calculation of the activity factor,

$$Activity = 10^{L-A}$$

or, in percents,

$$Activity_{\%} = 100 \cdot 10^{L-A}$$

## 17.2. Implementation

This implementation of the speech voltmeter algorithm can be found in the module `sv-p56.c`, with prototypes in `sv-p56.h`. This version evolved from a preliminary Fortran implementation provided by Telebrás, Brazil, which was used by several laboratories, in especial by participants of ETSI's contest for the second generation of Digital Mobile Radio Systems.

In Recommendation P.56, there are several undefined issues needed to be resolved for the implementation of this module. Especially, the rate $f$ used for the averages and the tolerance, or degree of accuracy, $\delta$ to be used for the interpolation of the active level have to be defined. Another undefined parameter is the reference level, or 0 dB reference point $r$. The choices of this implementation are shown in the table below:

| Speech voltmeter parameters | | |
|---|---|---|
| **Parameter** | **Description** | **Value** |
| $f$ | sampling rate | same rate of the input signal. |
| $r$ | dB reference | 0 dBov (see Chapter 2). |
| $\delta$ | tolerance | $\pm 0.5$ dB (the same of $M$). |

The P.56 algorithm operates on a sample-by-sample basis. However, since most software implementations use blocks (or frames) of samples, the speech voltmeter was designed to work with blocks of samples. Measurements are cummulative, therefore state variables are needed in this approach. These state variables have been arranged as fields of a structure whose name is `SVP56_state`. The fields of the structure are[45]:

| | |
|---|---|
| *f* | Sampling frequency, in Hz |
| *a[15]* | Activity count |
| *c[15]* | Threshold level |
| *hang[15]* | Hangover count |
| *n* | Number of samples read since last reset |
| *s* | Sum of all samples since last reset |
| *sq* | Squared sum of samples since last reset |
| *p* | Intermediate quantities |
| *q* | Envelope |
| *max* | Max absolute value found since last reset |
| *refdB* | 0 dB reference point, in [dB] |

[45] All the fields are `double`, except the `float` *f* and the `unsigned long` *a[]*, *hang[]*, and *n*.

| | |
|---|---|
| *rmsdB* | RMS value found since last reset |
| *maxP* | Most positive value since last reset |
| *maxN* | Most negative value since last reset |
| *DClevel* | Average level since last reset |
| *ActivityFactor* | Activity factor since last reset |

The user should note that although some fields are of interest to report signal statistics, such as long-term level, extreme values for file, average (or DC) level, etc., these values shall not be altered. See section clause 17.2.3, which describes macros for safe inpection of the parameters of interest.

The algorithm has two operational parts, one that deals with the initialization of the state variables, and is carried out by the function `init_speech_voltmeter`, and the measuring part (or the algorithm itself), carried out by `speech_voltmeter`. These are presented in the next two sections.

### 17.2.1. `init_speech_voltmeter`

**Syntax:**

```
#include "sv-p56.h"
void init_speech_voltmeter (SVP56_state _*state_, double _f_);
```

**Prototype:** `sv-p56.h`

**Description:**

`init_speech_voltmeter` performs the initialization of the speech voltmeter state variables in the structure pointed by *state* to the appropriate initial values. The only value required from the user is the sampling rate *f* (in Hz) of the signal that the speech voltmeter is supposed to measure. Note that when measuring new speech material, the state variable shall be re-initialized, otherwise accumulation of previous measurements will happen and wrong measurements will be reported.

**Variables:**

| | |
|---|---|
| *state* | Is a pointer to a speech voltmeter state variable. |
| *f* | Is the sampling rate (in Hz) of the signal to be measured in the next calls of `speech_voltmeter`. If zero or negative, the sampling rate is initialized to 16000 Hz. |

**Return value:**

None.

### 17.2.2. `speech_voltmeter`

**Syntax:**

```
#include "sv-p56.h"
double speech_voltmeter (float _*buffer_, long _smpno_, SVP56_state _*state_);
```

**Prototype:** `sv-p56.h`

**Description:**

`speech_voltmeter` performs the measurement of the active level of a speech signal according to ITU-T Recommendation P.56. Other relevant statistics are also available in the state variable (for details, see section clause 17.2.3 ahead):

– average level;
– maximum and minimum amplitude values;
– rms power, in dB;

**Variables:**

| | |
|---|---|
| *buffer* | Is the input sample `float` buffer. |
| *smpno* | Is the number of samples in *buffer*. |
| *state* | Is a pointer to the state variable buffer. This shall have been initialized by a previous call to `init_speech_voltmeter`. |

**Return value:** Returns the active speech level, in dB relative to dBov, as a double.

### 17.2.3. Getting state variable fields

Some macros are provided for the inspection of the speech voltmeter statistics:

**Syntax:**

```
#include "sv-p56.h"
SVP56_get_rms_dB(SVP56_state _state_);
SVP56_get_DC_level(SVP56_state _state_);
SVP56_get_activity(SVP56_state _state_);
SVP56_get_pos_max(SVP56_state _state_);
SVP56_get_neg_max(SVP56_state _state_);
SVP56_get_abs_max(SVP56_state _state_);
SVP56_get_smpno(SVP56_state _state_);
```

**Description:**

`SVP56_get_rms_dB` and `SVP56_get_DC_level` return respectively the long-term level (in dBov) and the DC level (in the normalized range) calculated for the material, both as a `double`.

`SVP56_get_activity` returns the activity factor as a `double`, in percents (0..100%).

`SVP56_get_pos_max`, `SVP56_get_neg_max`, and `SVP56_get_abs_max` returns respectively the maximum positive, negative and absolute amplitudes found for the input data, as normalized `double` values (range --1.0..+1.0).

`SVP56_get_smpno` returns as a `unsigned long` the total number of samples.

**Variables:**

All the macros expect a valid SVP56 state variable structure (not a pointer!).

## 17.3. Portability and compliance

Compliance tests of this module have been done based on the compliance with other existing implementations, especially of the Deutsches Bundespost Telekom Forschungs Institute. Reported results were found to be within the error margins of the P.56 algorithm.

Portability was checked by running the same speech file on a proven platform and on a test platform. Results have to be identical, in especial long-term and active levels, as well as the activity factor. During the development of this tool, the provided demonstration programs (see section clause 17.4) were used to measure and level-equalize a reference file. These test files are provided in the STL distribution.

This module had portability tested for VAX/VMS with VAX-C and GNU C (gcc) and for MS-DOS with a number of Borland C/C compilers (Turbo C v2.0, Turbo-C v1.0, Borland C++ v3.1). Portability was also tested in a number of Unix workstations and compilers: Sun workstation with Sun-OS and Sun-C (`cc`), `acc`, and `gcc`; HP workstation with HP-UX and `gcc`.

## 17.4. Examples

### 17.4.1. Description of the demonstration programs

As a part of the speech voltmeter module, two example programs are provided. They are called `sv56demo.c` and `actlevel.c`.

Both example programs calculate the equalization factor to equalize the active speech level of a file 'NdB' dBs below the 0 dBov reference using the algorithm described in this chapter. However, only program `sv56demo.c` carry out the level-equalization of the input file, which is saved in an aoutput file. Levels are reported in dBov.

In general, input files are in integer representation, 16-bit words, 2's complement (i.e., `short` data). In UGST convention, this data must be left-adjusted, *rather* than right-adjusted. Since the speech voltmeter uses `float` input data, it is necessary to convert from `short` (in the mentioned format) to `float`; this is carried out by the function `sh2fl()`. In addition, the option to `normalize' the input data to the range -1..+1 is selected. After the equalization factor is found, results are reported on the screen, which varies according to the program used and some of the command-line options.

While program `actlevel.c` stops at this point, program `sv56demo.c` proceeds calling the function `scale()` to carry out the (amplitude) equalization using single (rather than double) float precision. After equalization, the samples are converted back to integer (short, right-justified) with the routine `fl2sh()` using truncation, no zero-padding of the least significant bits, left-justification of data, and hard-clipping of data above the overload point. After that, data is saved to the user-specified file .

### 17.4.2. Small example

Following is an simplification of the described demonstration programs. It only measures the statistics for the input file, without carrying out level equalizations and does not implement the several command-line options of `actlevel.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "ugstdemo.h"           /* ... UGST demonstration program defs ... */
#include "sv-p56.h"             /* ... SV-P56 prototypes & defs ... */
#include "ugst-utl.h"           /* ... UGST utilities ... */
#define BLK_LEN 256

void main(argc, argv)
  int             argc;
  char           *argv[];
{
  SVP56_state     state;        /* Speech voltmeter state */
  char            FileIn[180];  /* input file name */
  FILE           *Fi;           /* input file pointers */
  long            N=BLK_LEN, l;
  short           bitno, buffer[BLK_LEN];
  float           Buf[BLK_LEN];
  double          ActiveLeveldB, sf, satur;

  /* Reads parameters for processing */
  GET_PAR_S(1, "_Input File: ..... ", FileIn);

  /* Checks parameters 2, and 3 for specification in command line */
  FIND_PAR_D(2, "_Sampling Frequency: .. ", sf, 16000);
  FIND_PAR_L(3, "_A/D resolution: ..... ", bitno, 16);

  /* Calculate overload point in the non-normalized range */
  satur = pow ((double)2.0, (double)(bitno - 1));
```

```
  /* Reset- variables for speech level measurements */
  init_speech_voltmeter(&state, sf);

  /* Opening input file */
  Fi = fopen(FileIn, RB);

  /* Read samples ... */
  while ((l = fread(buffer, N, sizeof(short), Fi)) > 0)
  {
    /* ... Convert samples to float, normalizing to +1..-1 */
    sh2fl((long) l, buffer, Buf, (long) state.bitno, 1);

    /* ... Get the active level */
    ActiveLeveldB = speech_voltmeter(Buf, (long) l, &state);
  }

  /* If the activity factor is 0, don't report many things */
  if (SVP56_get_activity(state) == 0)
    printf("\n  Activity factor is ZERO -- the file is silence or idle noise");
  else
  {
    printf("\n  DC level: ................ %7.0f [PCM]",
           SVP56_get_DC_level(state) * satur);
    printf("\n  Maximum positive value: .. %7.0f [PCM]",
           SVP56_get_pos_max(state) * satur);
    printf("\n  Maximum negative value: .. %7.0f [PCM]",
           SVP56_get_neg_max(state) * satur);
    printf("\n  Long-term energy (rms): .. %7.3f [dBov]",
           SVP56_get_rms_dB(state);
    printf("\n  Active speech level: ..... %7.3f [dBov]", ActiveLeveldB);
    printf("\n  Activity factor: ......... %7.3f [%%]",
           SVP56_get_activity(state));
  }
  fclose(Fi);
}
```

# 18. ITU-T Reverberation tool

## 18.1. Introduction

In some hands-free applications (video conference for example), the received sound is composed of direct sound from a speaker and its reverberated components. This reverberation effect corresponds to the modification of the speech signal by the acoustic response of the enclosure. The room effect is usually modeled [90] as a finite impulse response that can be measured between a specific source and the position of the receiver. Thus, it is possible to simulate a given room by convolving its measured impulse response with anechoic signals, which is the goal of this tool. Introduced in STL2005 to produce mono reverberant superwideband audio signals, this tool was updated in STL2009 release to also accommodate fullband audio signals and produce stereo signals. Proper saturation of the reverberated signals was also provided.

## 18.2. Description of the algorithm

### 18.2.1. Algorithm

Many approaches are available to add reverberation to a signal. The most realistic of them is to measure a real room impulse response and to convolve anechoic signals with it, which is used in the STL. This is the principle of this tool.

The reverberated signal is computed as

$$s_{rev}(k) = \sum_{l=0}^{N-1} IR(l) \cdot s(k-l),$$

where $s(k)$ is the original signal at time index k, $s_{rev}$ the reverberated signal, $IR$ the impulse response of a room, and $N$ the number of coefficients in $IR$.

The power level of the obtained reverberated signal depends of the experimental conditions of the impulse response measure. As a consequence, the processed signal can be attenuated or amplified. In order to compare reverberated sounds, the user can specify an alignment factor $\alpha$ which will scale the reverberated sound. This factor can be determined with the SV56 speech voltmeter.

The aligned reverberated signal is computed as

$$s_{rev}'(k) = s_{rev}(k) \cdot \alpha,$$

where $s_{rev}'$ and $\alpha$ are the aligned reverberated signal, and the scaling factor, respectively.

### 18.2.2. Mono impulse response

Four mono impulse responses are provided with this tool. The first three impulse responses were measured in typical meeting rooms. They are sampled at 32 kHz. Pictures of the two rooms considered are shown in Figure 88. The last mono impulse response were artificially generated to simulate a reverberant meeting room (90 $m^3$). It is sampled at 48 kHz.

The list of these mono impulse responses is given below:

– File `visio.IR`: sound capture at 100 cm distance in a small video-conference room,
– File `meeting50.IR`: sound capture at 50 cm distance in a large meeting room,
– File `meeting100.IR`: sound capture at 100 cm distance in the same meeting room.
– File `IR48.IR`: artificially generated.

**Figure 88-a — Video-conference room (Small)**

**Figure 88-b — Large meeting room**

**Figure 88 — Pictures of the rooms where mono impulse responses sampled at 32 kHz have been measured**

The geometry characteristics are given in Table 9. These rooms were acoustically treated in order to limit the reverberation (filled carpet, acoustically absorbent wall and ceiling). Note that the reverberation reduces the intelligibility of recorded speech and degrades the performance of acoustic echo canceler in case of hands-free communications.

**Table 9 — Characteristics of the rooms where mono impulse responses sampled at 32 kHz have been measured**

| | Length (m) | Width (m) | Height (m) | Volume ($m^3$) |
|---|---|---|---|---|
| Video-conference room | 4.80 | 4.45 | 2.50 | 53.40 |
| Meeting room | 8.55 | 5.30 | 2.70 | 122.35 |

To give more information concerning the acoustical behaviour, the octave-band reverberation time was computed for frequencies below 8 kHz. The values represented in Table 10 were estimated by the backward integration method applied in each octave-band of the measured impulse response.

**Table 10 — Octave-band reverberation time of the rooms where mono impulse responses sampled at 32 kHz have been measured.**

| | Reverberation time (ms) | | | | | |
|---|---|---|---|---|---|---|
| Octave band | 125 Hz | 250 Hz | 500 Hz | 1 kHz | 2 kHz | 4 kHz |
| Video-conference room | 600 | 450 | 360 | 295 | 280 | 250 |
| Meeting room | 671 | 600 | 518 | 490 | 466 | 440 |

### 18.2.3. Stereo impulse response

In STL 2009, stereo impulse responses were added. These stereo impulse responses were measured in typical meeting rooms with various microphones. The meeting rooms and microphone were selected according to two scenario configurations using two rooms (one per scenario). Pictures of the two rooms are shown in Figure 89 and Figure 91. Three types of microphone configurations were considered: MS microphone (Sony ECM-MS907), Binaural (two omnidirectional DPA 4060 inserted into the ears of a dummy head), AB microphone (two omnidirectional DPA 4060 microphones spaced

1 m apart). Moreover, the two scenarios were simulated in an anechoic room, i.e. the stereo impulse responses were also measured in this anechoic room for all the positions and all the microphones except for the binaural microphone. The geometry characteristics of the large (for scenario 1) and small (for scenario 2) rooms are given in the Table 11. These rooms were acoustically treated in order to limit the reverberation (filled carpet, coustically absorbent wall and ceiling). The acoustic absorption was higher for room of scenario 2 because the room was especially designed for video conferencing.

**Table 11 — Characteristics of the rooms where stereo impulse responses have been measured.**

| | Length (m) | Width (m) | Height (m) | Volume (m3) |
|---|---|---|---|---|
| Meeting room (Large) | 11.15 | 4.35 | 2.90 | 140.65 |
| Video-conference room (Small) | 5.25 | 4.75 | 2.60 | 64.83 |

#### 18.2.3.1. Scenario 1

The first scenario corresponds to a large conference room (see Figure 89) with 12 participants (12 possible positions). Figure 90 shows the microphone configuration - AB microphone (two omnidirectional DPA 4060 microphones spaced 1.5 m apart).

**Figure 89 — Large meeting room where stereo impulse responses sampled have been measured**

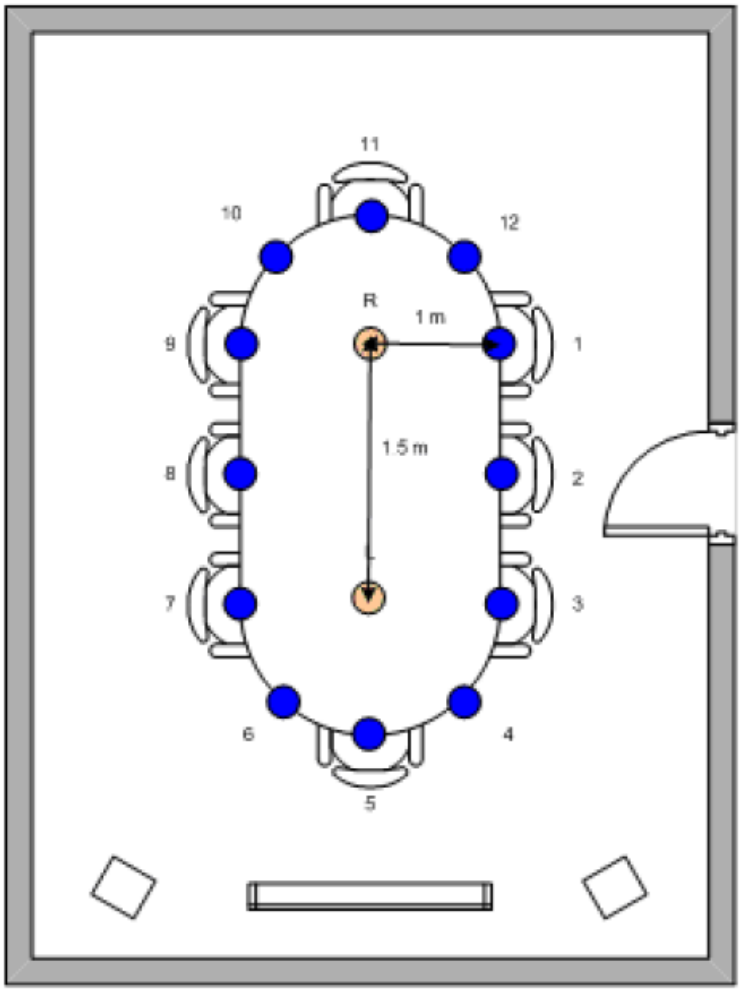



**Figure 90 — Large conference room, AB microphone), talker positions 1 through 12**

### 18.2.3.2. Scenario 2

The second scenario corresponds to a smaller video-conference room (see Figure 91) with 7 seats (these seats can be positioned in 7 possible positions in a +/-45 degree area). Figure 92 shows the microphone configurations: MS microphone (Sony ECM-MS907), Binaural (two omnidirectional DPA 4060 inserted into the ears of a dummy head), AB microphone (two omnidirectional DPA 4060 microphones spaced 1 m apart).

**Figure 91 — Video-conference room (Small) where stereo impulse responses have been measured**

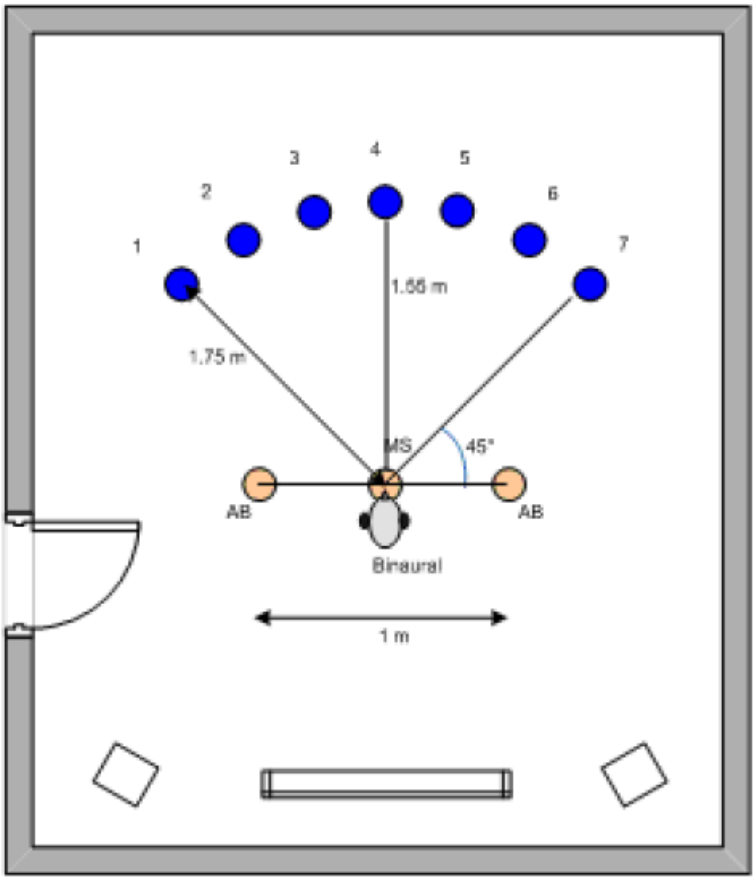



**Figure 92 — Scenario 2 (small video-conference room, AB, MS and Binaural microphones), talker positions 1 through 7, (left though right)**

### 18.2.3.3. List of measured stereo impulse responses

A total of 59 stereo impulse responses have been measured resulting in 118 impulse responses. The naming of mono impulse responses is extended to include the left/right channel, further the connection to the room type and microphone is tabulated.

The naming defined here is: `[Room][Reverb][Mic]P[Position].[channel].IR32`, where:

– `Room` is S=Small or L=Large,
– `Reverb` is E=Echoic or A=Anechoic,

– `Mic` is AB or MS or BI=binaural,
– `Position` is a two digit position number,
– `Channel` is L=Left or R=Right.

The detailed naming of stereo impulse responses is given in Table 12.

#### 18.2.3.4. Office noise recording

Office noises were recorded for the two stereo scenarios with all the microphone configurations [91]. The background noises were the real sounds from air conditioner, video projector, laptop noise (keyboard typing and fan).

### 18.2.4. Impulse response file format

Each sample of the IR is written in the IEEE Standard 754 floating-point representation, with both little-endian and big-endian byte ordering and 32-bit long data type. For example, command lines used to create the impulse responses in Matlab language were:

```
fwrite(file_identifier, impulse_response_vector, 'float',0,'ieee-le')
```

and

```
fwrite(file_identifier, impulse_response_vector, 'float',0,'ieee-be').
```

The mono Impulse Response (IR) are stored into file with ".IR" filename extensions. The stereo IR measures are stored into files with ".IR32" filename extensions and the sampling rate is 32 kHz. When applying these IRs, attention must be paid to have consistency between the sampling frequency of input data and the sampling frequency of an IR.

**Table 12 — Detailed naming of stereo impulse responses**

| Scenario | Main characteristics | Naming of impulse response pairs(with example positions): |
|---|---|---|
| Scenario 1, Large conf. room, 12 positions, AB microphone, no reverb, anechoic. | Large, Anechoic, AB | LAABP12.L.IR32 LAABP12. R.IR32 |
| Scenario 1, Large conf. room, 12 positions, AB microphone, including reverberation. | Large, Echoic, AB | LEABP01.L.IR32 LEABP01.R. IR32 |
| Scenario 2, small conf room, 7 positions, AB microphone, no reverb, anechoic. | Small, Anechoic, AB | SAABP01.L.IR32 SAABP01. R.IR32 |
| Scenario 2, small conf room, 7 positions, MS microphone, no reverb, anechoic. | Small, Anechoic, MS | SAMSP05.L.IR32 SAMSP05. R.IR32 |
| Scenario 2, small conf room, 7 positions, AB microphone, including reverberation. | Small, Echoic, AB | SEABP02.L.IR32 SEABP02.R. IR32 |
| Scenario 2, small conf room, 7 positions, Binaural microphone, including reverberation. | Small, Echoic, Binaural | SEBIP04.L.IR32 SEBIP04.R. IR32 |
| Scenario 2, small conf room, 7 positions, MS microphone, including reverberation. | Small, Echoic, MS | SEMSP07.L.IR32 SEMSP07.R. IR32 |

## 18.3. Implementation

### 18.3.1. `shift`

**Syntax:**

```
#include "reverb-lib.h"
void shift (short* _buff_, long _N_);
```

**Prototype:** `reverb-lib.h`

**Description:**

This routine replaces the first N-1 samples of a buffer by its last N-1 samples. It is useful for the block-based convolution, where N is the length of the blocks.

**Variables:**

| | |
|---|---|
| *buff* | buffer (input/output); |
| *N* | length of each block (input); |

### 18.3.2. `conv`

**Syntax:**

```
#include "reverb-lib.h"
long conv (float* _IR_, short* _buffIn_, short* _buffRvb_, float
           _alignFact_, long _N_, long _L_);
```

**Prototype:** `reverb-lib.h`

**Description:**

This function convolves the input buffer *buffIn* with an impulse response *IR* and stores the processed data into the output buffer *buffRvb*. The alignment factor (multiplicative factor) *alignFact* is used to align the energy of the input file with another file. The return value is used to provide a warning that a 16 bit saturation occurs, a positive value indicates the position of the last overflow occurrence, if no overflow occurs -1 is returned.

**Variables:**

| | |
|---|---|
| *IR* | impulse response buffer ; |
| *buffIn* | input buffer; |
| *buffRvb* | convolved data; |
| *alignFact* | alignment factor; |
| *N* | length of the impulse response buffer; |
| *L* | length of the input buffer to process; |

**Return value:**

Returns the last position of an overflow if any, otherwise -1.

### 18.3.3. Tests and portability

Compiled and tested on a PC (Windows) platform with MS Visual C++ 6.0, in Cygwin with gcc (version 3.4.4), in Fedora 7 with gcc (version 3.4.4).

### 18.4. Example code

The demonstration program uses a room impulse response and a sound file as input to produce a reverberated sound file as output. The input sound is convolved with the room impulse response to produce the reverberated sound. The program can be found in reverb.c.

## 19. ITU-T Bitstream truncation tool

### 19.1. Introduction

A scalable codec is a highly flexible coding technique that is characterized by a multi-layer bitstream:
– The core layer provides the minimum quality. This layer is a minimum requirement for a decoder to operate, or otherwise the frame is considered lost.
– Upper layers improves quality by increasing bitrate.

The main feature of a scalable codec lies in bit rate flexibility. The bitrate can be adjusted between minimal and maximal values by any component in the communication chain. For instance, to cope with network congestion, the bitrate can be adjusted on a frame by frame basis.

The bitrate modification is very simple as it consists of cutting the bitstream at the right rate, i.e., stripping bits. Apart from this straightforward truncation, no signal processing is required.

To simulate this functionality, a bitstream truncation tool (truncate) was introduced into STL2005. G.192 bitstreams (with or without sync header), G.192 byte-oriented bitstreams (with or without sync header) and binary (compact) bitstreams can be processed with this tool.

### 19.2. Description of the algorithm

This tool truncates the bitstream at the desired bitrate (see Figure 93). For each frame of an input bitstream, this tool performs the following operations :
1) Read the synchronization word and copy it to the output bitstream;
2) Write the new frame length word equal to the number N2 of bits to copy to the output bitstream (N2 depends on the desired bit rate);
3) Read the N1 words of the input bitstream and copy the first N2 16-bit words representing the first N2 bits of the input to the output bitstream.

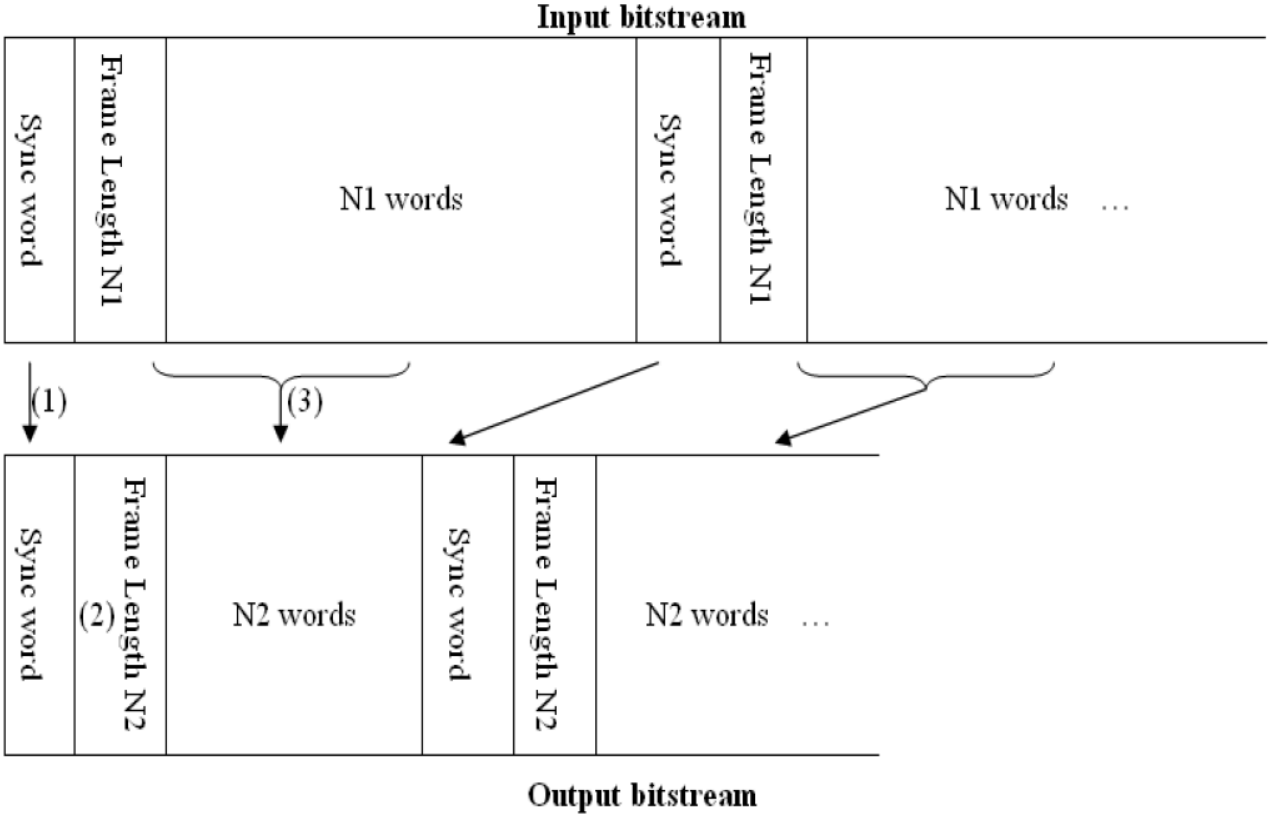



**Figure 93 — Bitstream truncation principle**

For example, one frame of the input bitstream will comprise 640 bits for a 32 kbit/s codec operating a frame size of 20 ms. To truncate the 32 kbit/s input bitstream to 14 kbit/s, the output bitstream frame length will be set to 280 for each frame. Only the first 280 words of the 640 words of the input bitstream will be copied in the output bitstream while the last 360 words will be discarded.

## 19.3. Implementation

### 19.3.1. `trunc`

**Syntax:**

```
#include "trunc-lib.h"
void trunc (short _syncWord_, short _outFrameLgth_, short* _inpFrame_,
            short* _outFrame_);
```

**Prototype:** `trunc-lib.h`

**Description:**

This routine copies the *syncWord* and the first *outFrameLgth* words of the input frame *inpFrame* to the output frame *outFrame*.

**Variables:**

| | |
|---|---|
| *syncWord* | synchronization word to write to the output frame; |
| *outFrameLgth* | length of the output frame; |
| *inpFrame* | input frame to truncate; |
| *outFrame* | output frame; |

### 19.3.2. Tests and portability

Compiled and tested on a PC (Windows) platform with MS Visual C++ 6.0. A bitrate file can be obtains with the tool gen_rate contain in EID module.

## 19.4. Example code

A demonstration program *truncate.c* illustrates the use of this module to truncate a bitstream to the desired bitrate.

# 20. ITU-T frequency response measurement tool

## 20.1. Introduction

In order to measure effective codec bandwidth, a frequency response measurement tool was created for the STL2005 [92]. At Q10/16 January 2008 meeting, a window overlap option was proposed to make this tool more accurate when dealing with signals with quickly varying frequencies while allowing the tool to be used as before [93] and it was wondered whether this window overlap option should be the default mode. At this January 2008 meeting [94], it was also suggested that an option to use other window sizes might be useful. In addition, during G.729.1 Superwideband qualification phase, it was found that the speed of the algorithm can be sometime a bit too slow. Therefore, at Q10/16 September 2008 meeting, an update was provided to allow variable frame size usage and to increase the computation speed [95]. In STL2009, the frequency response measurement tool has been revised to introduce these three options: window overlap, variable frame size, and increased computation speed.

## 20.2. Description of the algorithm

An input signal is encoded and decoded by the Codec under Test. The periodogram method is then used to compute the average amplitude spectrum difference between a reference signal (e.g. the input file to the speech codec) and a test signal (e.g. the speech signal after encoding and decoding by a codec).

The input and output signals can be treated on a frame by frame basis, with size power of 2 which defaults to 2048 if unspecified. A Hanning window is applied to each input and output frame. The resulting windowed signals are transformed to the frequency domain using a various number of point Fast Fourier transform. The input and output amplitude spectra are then computed and averaged. This tool produces the average amplitude spectrum in ASCII and also produces a bitmap file.

### 20.2.1. Discrete Fourier Transform (DFT)

The spectrum is computed using the Discrete Fourier Transform (for real signals). This is performed as follows :

$$X(f) = \sum_{k=0}^{NFFT-1} x(k) \cdot \cos(2\pi f k) - j \cdot \sum_{k=0}^{NFFT-1} x(k) \cdot \sin(2\pi f k)$$

where *NFFT* is the number of DFT coefficients.

### 20.2.2. Hanning window generation (DFT)

The Hanning window of length n is defined as :

$$hanning(k) = 0.5 \cdot 1 - \cos(2\pi \cdot \frac{k+1}{n+1}) \backslash \text{rule}\{15\text{mm}0mm\}(0 \le k \le n-1)$$

### 20.2.3. Window overlap

In STL2005, the frequency measurement tool used non-overlapping windowed frames to perform the signal analysis. With such non-overlapping Hanning windows, some regions of the signal are set to a very weak value at the window edges. Thus some temporal parts of the signal are not taken into account when applying the Fourier transform, and are not visible in the frequency representation. This is a problem when signals with very quick frequency changes, or continuous changes are considered. This case occurred with the sweep tone signal used in G.722.1 fullband qualification phase. To fix this, an option has been added to the tool to introduce overlapping between two consecutive frames. With this option, a temporal region attenuated by the Hanning window in one frame will be present in next frame. The desired percentage of overlapping can be chosen. If the option is not activated, the tool output remains unchanged.

### 20.2.4. Variable frame size

In STL2005, the window size used for discrete Fourier transform was fixed to 2048 samples and could not provide changes in the frequency resolution. In order provide more freedom, a variable window size was implemented using Fast Fourier Transform (FFT) algorithm. The use of FFT implies that the window size is a power of 2, thus, available sizes are: 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 and 8192. The dimensions of the bitmap file created by the tool (with option "-bmp") are adapted to the new variable size: if the size is smaller than 2048, then the bitmap file is in the same dimensions as for a 2048-sample bitmap file, because smaller dimensions would give unreadable figures. For windows sizes larger than 2048, the bitmap dimensions are increased to adapt to better frequency resolution.

The Figure 94, Figure 95, Figure 96 give the bitmap files obtained with `freqresp` tool operating in 256-, 2048- and 8192-sample windowing, respectively. The input file was the artificial female voice standardized by ITU in [13].

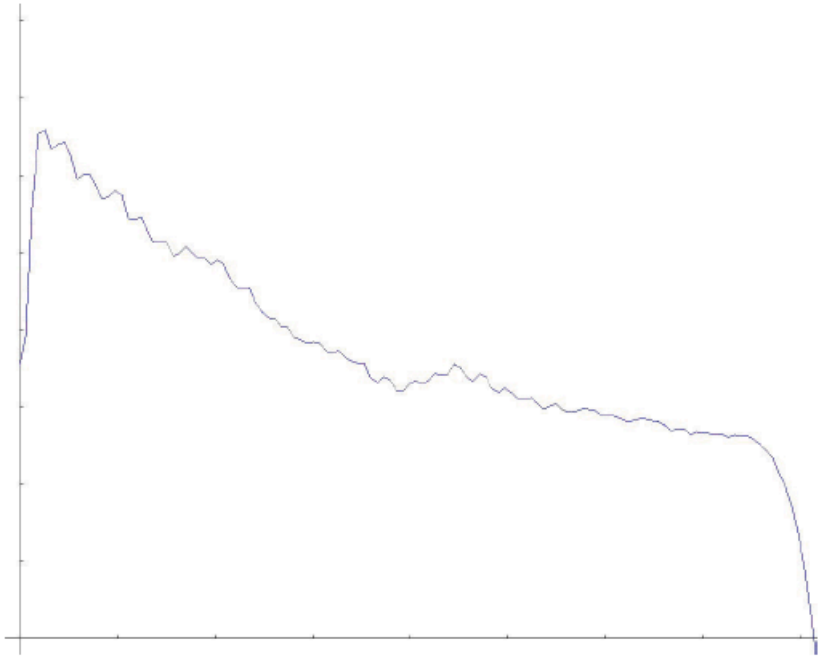


**Figure 94 — P.50 female voice analyzed with a 256-sample window**

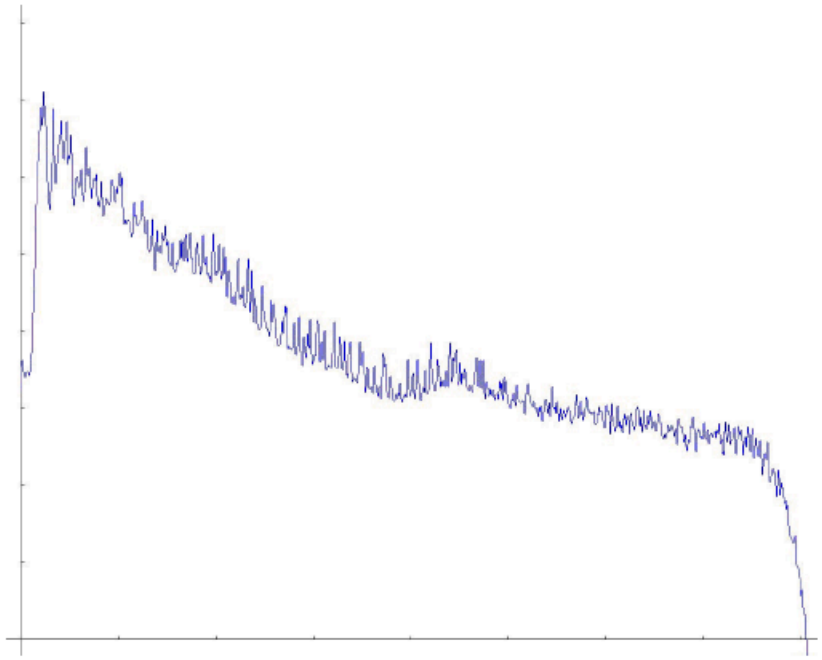


**Figure 95 — P.50 female voice analyzed with a 2048-sample window**

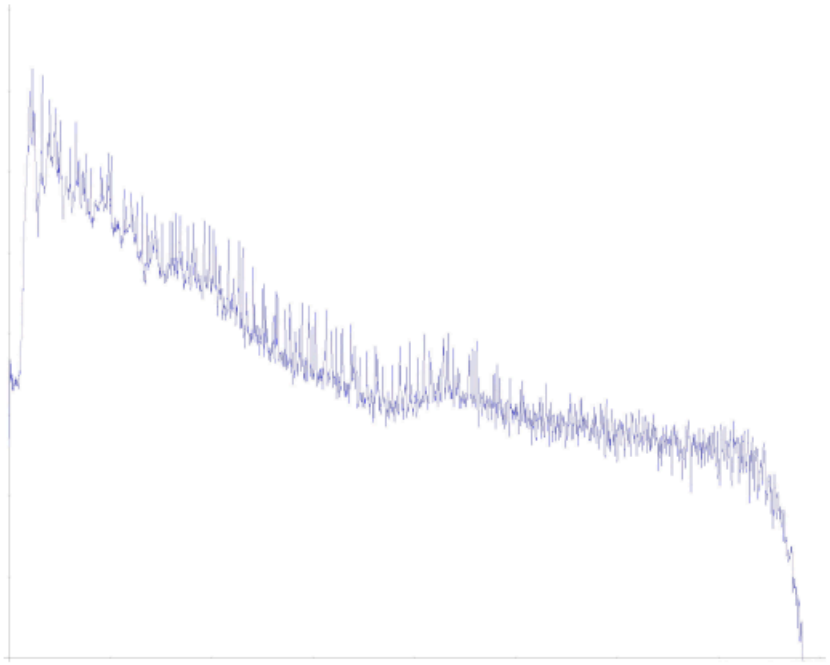


**Figure 96 — P.50 female voice analyzed with a 8192-sample window**

### 20.2.5. Computation optimization

As the time needed for computing the frequency response of a long audio file can be rather long, the Discrete Fourier Transform (DFT) can be replaced by a split radix (4,2) Fast Fourier Transform (FFT) algorithm[46].

When this option is activated the frame size is limited to power of 2 values (16-32-64-128-256-512-1024-2048-4096-8192).

46 Split-radix FFT algorithm, http://en.wikipedia.org/wiki/Split-radix_FFT_algorithm

## 20.3. Test Signals

### 20.3.1. Narrow band and wideband

For narrow band and wideband signals, ITU-T SG 12 has been recommending input signal as Rec. ITU-T **P.50 test signals** (p50_m.16p for male voices, and p50_f.16p for female voices). Those signal are meant to be representative of speech signals.

### 20.3.2. Superwideband and fullband

For audio bandwidths greater than wideband, the Rec. ITU-T **P.50 test signals** does not cover enough frequency range for characterizing superwideband or fullband systems.

Therefore, ITU-T SG12 has been suggesting use of Composite Source (CS) Signal as defined in Rec. ITU-T P.501 as it provides sufficient energy in the frequency range above 8 kHz and below 100 Hz. As audio coding systems must also be capable of transmitting signals other than speech, ITU-T SG12 has provided the following guidelines (more details are given in [96]):

1) In case CS-signal cannot be used, it is recommended to use artificial voice according to ITU-T Rec. P.50 and mix it with a broadband pink noise signal. The mixing should be 50%/50% … If the decoded signal does not sound distorted, it can be used as an input test signal.
2) If the signal sounds distorted replace the artificial voice signal by a voice signal consisting of speech sentences (2 sentences, 2 male, 2 female talkers each), such as ones in Rec. ITU-T P.501.
3) If this does not lead to the desired result replace the pink noise by a broadband background noise signal providing as much energy as possible in the high frequency range, such as ones in Rec. ITU-T P.501 or ETSI EG 202 396-1.
4) Codecs intended to transmit music could be tested with a fullband music signal. Again care should be taken to provide sufficient energy in the high and low frequency range.

## 20.4. Implementation

### 20.4.1. `rdft`

**Syntax:**

```
#include "fft.h"
void rdft (int _NFFT_, float* _x1_, float* _x2_, float* _y2_);
```

**Prototype:** `fft.h`

**Description:**

This routine computes the positive part of the spectrum, using Real Discrete Fourier Transform.

**Variables:**

*NFFT* number of coefficients of the Fourier transform;

*x1* input real signal;

*x2* output real part of the Fourier Transform;

*y2* output imaginary part of the Fourier Transform;

### 20.4.2. `genHanning`

**Syntax:**

```
#include "fft.h"
void genHanning (int _n_, float* _hanning_);
```

**Prototype:** `fft.h`

**Description:**

This routine generates a hanning window.

**Variables:**

| | |
|---|---|
| *n* | number of coefficients of the hanning window; |
| *hanning* | buffer containing the coefficients of the hanning window; |

### 20.4.3. `powSpect`

**Syntax:**

```
#include "fft.h"
void powSpect (float* _real_, float* _imag_, float* _pws_, int _n_);
```

**Prototype:** `fft.h`

**Description:**

This routine computes the power spectrum of a signal with the fast split radix DFT.

**Variables:**

| | |
|---|---|
| *real* | input buffer containing the real part of the DFT; |
| *imag* | input buffer containing the imaginary part of the DFT; |
| *pws* | output buffer containing the power spectrum; |
| *n* | length of the input buffers; |

### 20.4.4. `actrdft`

**Syntax:**

```
#include "fft.h"
void actrdft (int _n_, int _isgn_, float* _a_, int* _ip_, float* _w_);
```

**Prototype:** `fft.h`

**Description:**

This routine computes actual fast discrete Fourier transform for real sequence. This routine replaces `rdft` defined above (newly introduced at STL2009 release).

**Variables:**

| | |
|---|---|
| *n* | data length, must be n $\geq$ 2 and power of 2 |
| *isgn* | Transform type (1 represents forward transform, -1 represents inverse) |
| *a[0...n-1]* | input/output data |
| *ip[0 ... *]* | work area for bit reversal |
| *w[0 ...n/2-1]* | cos/sin table |

### 20.4.5. Tests and portability

Compiled and tested on a PC (Windows) platform with MS Visual C++ 6.0.

## 20.5. Example code

A demonstration program, *freqresp.c* illustrates the use of this module to compute the average power spectrum of two signals (input and output of the codec).

The syntax needed for using the variable size is following:

```
freqresp.exe -nfft 4096 FileInpCodec FileOutCodec ASCIIout
```

If the `-nfft` option is not used, then the default size (2048) is used. The other usual options (e.g. sampling frequency,…) can still be set.

The max length of FFT is defined by the macro "NFFT_MAX" and is set to 8192.

## 21. ITU-T Stereo processing tool

### 21.1. Introduction

For the development and characterization of stereo algorithms, properly conditioned stereo signals are required. To reuse the existing set of STL single-channel processing tools for conditioning the inputs and outputs in a stereo processing chain, a generic stereo operations tool was developed and introduced in STL2009. This stereo processing tool provides the basic functionalities of interleaving single channel files, splitting stereo files into single channel files, and creating specific single channel down-mix signals from a given stereo input.

### 21.2. Description of the algorithm

The tool operates on sample basis on 16-bit resolution input files, and provides the following functionalities:

– Interleaving two single channel files into one single two-channel stereo file.
– Splitting an interleaved two-channel file into individual single channel files
– Down-mixing a two-channel stereo file into a maximum channel energy analysis file (a single channel analysis signal for stereo scene level evaluation purposes).
– Down-mixing a two-channel file into a single channel mono file

Figure 97 illustrates the functionality that interleaves two single channel files into one single two-channel stereo file. Figure 98 illustrates the functionality that splits an interleaved two-channel file into individual single channel files. Figure 99 illustrates the functionality that down-mixes a two-channel file into a maximum channel energy analysis file. Figure 100 illustrates the functionality that down-mixes a two-channel file into a single channel mono file.

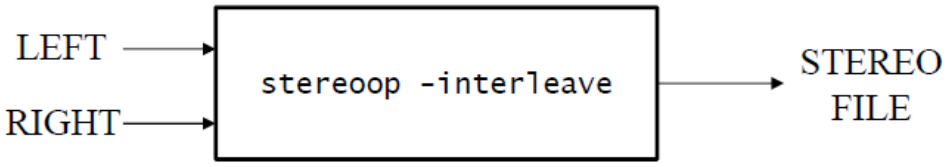



**Figure 97 — Interleaving two single channel files into one single two-channel stereo file**

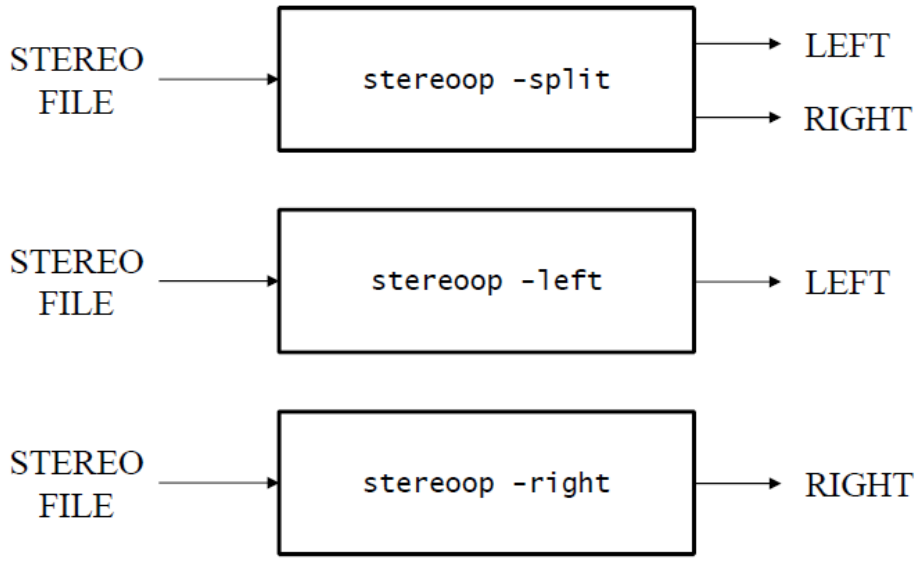



**Figure 98 — Splitting an interleaved two-channel file into two individual single channel files**

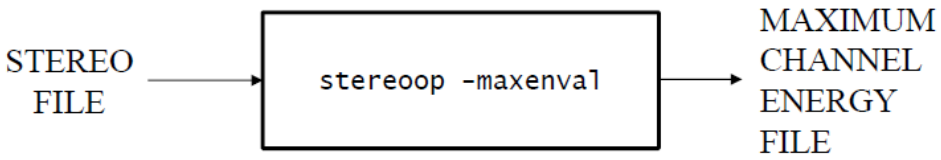



**Figure 99 — Down-mixing a two-channel file into a maximum channel energy analysis file.**

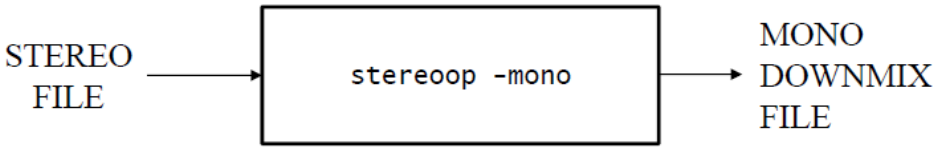



**Figure 100 — Down-mixing a two-channel file into a single channel mono file**

This set of stereo operation functionalities, in combination with the existing STL tools which work on the single channel files, provides a generic framework for stereo file processing.

## 21.3. Implementation

### 21.3.1. Data and file format

The tool works on 16-bit sample data (short), independent of sampling frequency. It is assumed that left and right channels always have the same sampling frequency.

File format: 2-channel data is stored in raw header-less files, with each left channel sample stored before the corresponding right channel sample. Single channel files are stored in raw header-less files as required by most existing STL single channel tools.

The input and output file byte order will be the native system byte order.

### 21.3.2. stereoop module

#### 21.3.2.1. Prototype

```
#include "ugstdemo.h"
```

```
enum Mode NONE = -1,INTER, SPLIT, LEFT, RIGHT, MAXENVAL, MONO, N MODES ;
int main (int argc, char *argv[]);
```

#### 21.3.2.2. Description

The main function opens the input file (or files) and processes them according to a stereo operation option given on the command line and stores the processed data into the output file (or files).

#### 21.3.2.3. Variables

```
FILE   *Fif[MAX_IFILES]; /* Pointer(s) to input files */
FILE   *Fof[MAX_OFILES];      /* Pointer(s) to output files */
char   ifname[MAX_IFILES][MAX_STR]; /* Input file names */
char   ofname[MAX_OFILES][MAX_STR]; /* Output file names */
enum   Mode mode=NONE;  /* Operation mode variable */
short  tmp_short[2],tmp_short_1ch[1]; /* 16bit short read/write buffers */
```

#### 21.3.2.4. Command line syntax for channel manipulation operations

```
stereoop -interleave  infile.L.1ch       infile.R.1ch   outfile.stereo.2ch
stereoop -split       infile.stereo.2ch  outfile.L.1ch  outfile.R.1ch
stereoop -left        infile.stereo.2ch  outfile.L.1ch
stereoop -right       infile.stereo.2ch  outfile.R.1ch
```

#### 21.3.2.5. Command line syntax for channel down-mix operations

```
stereoop -maxenval  infile.stereo.2ch  outfile.maxenval.1ch
stereoop -mono      infile.stereo.2ch  outfile.mono.1ch
```

#### 21.3.2.6. "stereoop -maxenval" detailed operation

The pseudo code below shows the tool operation when creating the maximum channel energy downmix, a downmix that may be used for stereo scene level evaluation purposes. The left or right sample value yielding the maximum absolute value (which equals the value yielding maximum energy) is written to the output file.

Pseudo code

```
[left, right] = ReadSamples(InputStereoChannelFile);

if( |left| > |right|){
  output = left;
} else {
  output = right;
}
WriteSample(OutputSingleChannelFile, output);
}
```

#### 21.3.2.7. "stereoop -mono" detailed operation

The pseudo code below shows the tool operation when creating the mono downmix. The output sample mono value is (left + right) / 2.0, (rounded to a short value).

Pseudo code

```
[left, right] = ReadSamples(InputStereoChannelFile);

output = ((double)left + (double)right)/2.0;

WriteSample(OutputSingleChannelFile, round(output));
```

### 21.4. Tests and Portability

The stereoop tool example code has been evaluated under Win2000/Cygwin(CYGWIN_NT-5.0, v1.5.24/gcc(3.4.4), Linux (2.6.5-7.145)/gcc(2.95.3) and Solaris(SunOS Generic_122300-02)/ gcc(2.95.3).

### 21.5. Example code

A demonstration program, `stereoop.c` implements the described operations of the tool.

## 22. BASOP: ITU-T Basic Operators

### 22.1. Overview of basic operator libraries

Since the standardization of G.729 and G.723.1[47], ANSI-C source codes constitute integral parts of the ITU-T speech and audio coding Recommendations and their specification relies on bit-exact fixed-point C code using library of basic operators that simulates DSP operations. The fixed-point descriptions of G.723.1 and G.729 are based on 16- and 32-bit arithmetic operations defined by ETSI in 1993 for the standardisation of the half-rate GSM speech codec. These operations are also used to define the GSM enhanced full-rate (EFR) and adaptive multi-rate (AMR) speech codecs [19].

In STL2005, the version 2.0 of the ITU-T Basic Operators bears the following additional features compared to the version 1.x:

1) New 16-bit and 32-bit operators;
2) New 40-bit operators;
3) New control flow operators;
4) Revised complexity weight of version 1.x basic operators in order to reflect the evolution of processor capabilities.

In STL2009, that is version 2.3, in addition to some minor fixes in the ITU-T Basic Operators and guidelines of data move counters, the following new additional tools were added:

– Program ROM estimation tool for fixed-point C Code;
– Complexity evaluation tool for floating-point C Code.

In STL2019, new operators were introduced to account for modern DSP architectures:

1) Enhanced 32-bit operators;
2) New unsigned 32-bit operators;
3) New 64-bit operators;
4) New complex operators;
5) New control flow operators;
6) Revised complexity weights in order to reflect the evolution of processor capabilities.

Table 17 provides an overview of the complexity weight history for each basic operator.

### 22.2. Description of the 16-bit and 32-bit basic operators and associated weights

This section describes the different 16-bit and 32-bit basic operators available in the STL, and are organized by complexity ("weights"). The complexity values to be considered (since the publication of the STL2005) are the ones related to the version 2.0 and subsequent versions of the module. When the basic operator did not exist in the previous version of the library (version 1.x), it is highlighted

[47] For older standards (G.711, G.726, G722, G.727, G.728), their C-codes are included in the software tools library.

as follows: → NEW IN v2.0. In STL2009, `round()` operator was renamed as `round_fx()` and `saturate()`, function was made unaccessible from applications, because it was an internal procedure.

### 22.2.1. Variable definitions

The variables used in the operators are signed integers in 2's complements representation, defined by:

`v1` , `v2` 16-bit variables

`L_v1` , `L_v2` , `L_v3` 32-bit variables

### 22.2.2. Operators with complexity weight of 1

#### 22.2.2.1. Arithmetic operators (multiplication excluded)

`add(v1, v2)`

Performs the addition (`v1`+`v2`) with overflow control and saturation; the 16-bit result is set at `+32767` when overflow occurs or at `-32768` when underflow occurs.

`sub(v1, v2)`

Performs the subtraction (`v1`-`v2`) with overflow control and saturation; the 16-bit result is set at `+32767` when overflow occurs or at `-32768` when underflow occurs.

`abs_s(v1)`

Absolute value of v1. If `v1` is `-32768`, returns `32767`.

`shl(v1, v2)`

Arithmetically shift the 16-bit input `v1` left `v2` positions. Zero fill the `v2` LSB of the result. If `v2` is negative, arithmetically shift `v1` right by `-v2` with sign extension. Saturate the result in case of underflows or overflows.

`shr(v1, v2)`

Arithmetically shift the 16-bit input `v1` right `v2` positions with sign extension. If v2 is negative, arithemtically shift `v1` left by `-v2` and zero fill the `-v2` LSB of the result:

```
shr(v1, v2) = shl(v1, -v2)
```

Saturate the result in case of underflows or overflows.

`negate(v1)`

Negate `v1` with saturation, saturate in the case when input is `-32768`:

```
negate(v1) = sub(0, v1)
```

`s_max(v1, v2)`

→ NEW IN v2.0

Compares two 16-bit variables `v1` and `v2` and returns the maximum value.

`s_min(v1, v2)`

→ NEW IN v2.0

Compares two 16-bit variables `v1` and `v2` and returns the minimum value.

`norm_s(v1)`

Produces the number of left shifts needed to normalize the 16-bit variable v1 for positive values on the interval with minimum of 16384 and maximum 32767, and for negative values on the interval with minimum of -32768 and maximum of -16384; in order to normalise the result, the following operation must be done:

```
norm_v1 = shl(v1, norm_s(v1))
```

```
L_add(L_v1, L_v2)
```

This operator implements 32-bit addition of the two 32-bit variables (L_v1+L_v2) with overflow control and saturation; the result is set at +2147483647 when overflow occurs or at -2147483648 when underflow occurs.

```
L_sub(L_v1, L_v2)
```

32-bit subtraction of the two 32-bit variables (L_v1--L_v2) with overflow control and saturation; the result is set at +2147483647 when overflow occurs or at -2147483648 when underflow occurs.

```
L_abs(L_v1)
```

Absolute value of L_v1, with L_abs(-2147483648)=2147483647.

```
L_shl(L_v1, v2)
```

Arithmetically shift the 32-bit input L_v1 left v2 positions. Zero fill the v2 LSB of the result. If v2 is negative, arithmetically shift L_v1 right by -v2 with sign extension. Saturate the result in case of underflows or overflows.

```
L_shr(L_v1, v2)
```

Arithmetically shift the 32-bit input L_v1 right v2 positions with sign extension. If v2 is negative, arithmetically shift L_v1 left by -v2 and zero fill the -v2 LSB of the result. Saturate the result in case of underflows or overflows.

```
L_negate(L_v1)
```

Negate the 32-bit L_v1 parameter with saturation, saturate in the case where input is -2147483648.

```
L_max(L_v1, L_v2)
```

→ NEW IN V2.0

Compares two 32-bit variables L_v1 and L_v2 and returns the maximum value.

```
L_min(L_v1, L_v2)
```

→ NEW IN V2.0

Compares two 32-bit variables L_v1 and L_v2 and returns the minimum value.

```
norm_l(L_v1)
```

Produces the number of left shifts needed to normalise the 32-bit variable L_v1 for positive values on the interval with minimum of 1073741824 and maximum 2147483647, and for negative values on the interval with minimum of -2147483648 and maximum of -1073741824; in order to normalise the result, the following operation must be done:

```
L_norm_v1 = L_shl(L_v1, norm_l(L_v1))
```

#### 22.2.2.2. Multiplication operators

```
i_mult(v1, v2)
```

Multiply two 16-bit words v1 and v2 returning a 16-bit word with overflow control.

```
L_mult(v1, v2)
```

Operator `L_mult` implements the 32-bit result of the multiplication of `v1` times `v2` with one shift left, i.e.

```
L_mult(v1, v2) = L_shl((v1` stem:[xx] `v2), 1)
```

Note that `L_mult(-32768,-32768) = 2147483647`.

### L_mult0(v1, v2)

Operator `L_mult0` implements the 32-bit result of the multiplication of `v1` times `v2` *without* left shift, i.e.

```
L_mult0(v1, v2) = (v1` stem:[xx] `v2)
```

### mult(v1, v2)

Performs the multiplication of `v1` by `v2` and gives a 16-bit result which is scaled, i.e.

```
mult(v1, v2) =  extract_l(L_shr((v1 times v2),15))
```

Note that `mult(-32768,-32768) = 32767`.

### mult_r(v1, v2)

Same as `mult()` but with rounding, i.e.

```
mult_r(v1, v2) = extract_l(L_shr(((v1` stem:[xx] `v2)+16384), 15))
```

and `mult_r(-32768, -32768) = 32767`.

### L_mac(L_v3, v1, v2)

Multiply `v1` by `v2` and shift the result left by 1. Add the 32-bit result to `L_v3` with saturation, return a 32-bit result:

```
L_mac(L_v3, v1, v2) = L_add(L_v3, L_mult(v1, v2))
```

### L_mac0(L_v3, v1, v2)

Multiply `v1` by `v2` *without* left shift. Add the 32-bit result to `L_v3` with saturation, returning a 32-bit result:

```
L_mac0(L_v3, v1, v2) = L_add(L_v3, L_mult0(v1, v2))
```

### L_macNs(L_v3, v1, v2)

Multiply `v1` by `v2` and shift the result left by 1. Add the 32-bit result to `L_v3` without saturation, return a 32-bit result. Generates carry and overflow values:

```
L_macNs(L_v3, v1, v2) = L_add_c(L_v3, L_mult(v1, v2))
```

### mac_r(L_v3, v1, v2)

Multiply `v1` by `v2` and shift the result left by 1. Add the 32-bit result to `L_v3` with saturation. Round the 16 least significant bits of the result into the 16 most significant bits with saturation and shift the result right by 16. Returns a 16-bit result.

```
mac_r(L_v3, v1, v2) =
  round_fx(L_mac(L_v3, v1, v2))=
  extract_h(L_add(L_add(L_v3, L_mult(v1, v2)), 32768))}
```

### L_msu(L_v3, v1, v2)

Multiply `v1` by `v2` and shift the result left by 1. Subtract the 32-bit result from `L_v3` with saturation, return a 32-bit result:

```
L_msu(L_v3, v1, v2) = L_sub(L_v3, L_mult(v1, v2)).
```

### L_msu0(L_v3, v1, v2)

Multiply v1 by v2 *without* left shift. Subtract the 32-bit result from L_v3 with saturation, returning a 32-bit result:

```
L_msu0(L_v3, v1, v2) = L_sub(L_v3, L_mult0(v1, v2)).
                                    L_msuNs(L_v3, v1, v2)
```

The sub-routine L_sub_c evoked from this function is reported to have a carry problem. Use of this operator in current and past release of STL is NOT recommended.

Multiply v1 by v2 and shift the result left by 1. Subtract the 32-bit result from L_v3 without saturation, return a 32-bit result. Generates carry and overflow values:

```
L_msuNs(L_v3, v1, v2) = L_sub_c(L_v3, L_mult(v1, v2))
                                    L_mls(L_v1, v2)
```

Performs a multiplication of a 32-bit variable L_v1 by a 16-bit variable v2, returning a 32-bit value.

```
msu_r(L_v3, v1, v2)
```

Multiply v1 by v2 and shift the result left by 1. Subtract the 32-bit result from L_v3 with saturation. Round the 16 least significant bits of the result into the 16 bits with saturation and shift the result right by 16. Returns a 16-bit result.

```
msu_r(L_v3, v1, v2) =
  round_fx(L_msu(L_v3, v1, v2))=
  extract_h(L_add(L_sub(L_v3, L_mult(v1, v2)), 32768))
```

#### 22.2.2.3. Logical operators

```
s_and(v1, v2)
```

→ NEW IN v2.0

Performs a bit wise AND between the two 16-bit variables v1 and v2.

```
s_or(v1, v2)
```

→ NEW IN v2.0

Performs a bit wise OR between the two 16-bit variables v1 and v2.

```
s_xor(v1, v2)
```

→ NEW IN v2.0

Performs a bit wise XOR between the two 16-bit variables v1 and v2.

```
lshl(v1, v2)
```

→ NEW IN v2.0

Logically shifts left the 16-bit variable v1 by v2 positions:
- if v2 is negative, v1 is shifted to the least significant bits by (-v2) positions with insertion of 0 at the most significant bit.
- if v2 is positive, v1 is shifted to the most significant bits by (v2) positions without saturation control.

```
lshr(v1, v2)
```

→ NEW IN v2.0

Logically shifts right the 16-bit variable v1 by v2 positions:
- if v2 is positive, v1 is shifted to the least significant bits by (v2) positions with insertion of 0 at the most significant bit.
- if v2 is negative, v1 is shifted to the most significant bits by (-v2) positions without saturation control.

```
L_and(L_v1, L_v2)
```

→ NEW IN v2.0

Performs a bit wise AND between the two 32-bit variables `L_v1` and `L_v2`.

```
L_or(L_v1, L_v2)
```

→ NEW IN v2.0

Performs a bit wise OR between the two 32-bit variables `L_v1` and `L_v2`.

```
L_xor(L_v1, L_v2)
```

→ NEW IN v2.0

Performs a bit wise XOR between the two 32-bit variables `L_v1` and `L_v2`.

```
{L_lshl(L_v1, v2)
```

→ NEW IN v2.0

Logically shifts left the 32-bit variable `L_v1` by v2 positions:
– if v2 is negative, `L_v1` is shifted to the least significant bits by (-v2) positions with insertion of 0 at the most significant bit.
– if v2 is positive, `L_v1` is shifted to the most significant bits by (v2) positions without saturation control.

```
L_lshr(L_v1, v2)
```

→ NEW IN v2.0

Logically shifts right the 32-bit variable `L_v1` by v2 positions:
– if v2 is positive, `L_v1` is shifted to the least significant bits by (v2) positions with insertion of 0 at the most significant bit.
– if v2 is negative, `L_v1` is shifted to the most significant bits by (-v2) positions without saturation control.

#### 22.2.2.4. Data type conversion operators

```
extract_h(L_v1)
```

Return the 16 MSB of `L_v1`.

```
extract_l(L_v1)
```

Return the 16 LSB of `L_v1`.

```
round_fx(L_v1)
```

Round the lower 16 bits of the 32-bit input number into the most significant 16 bits with saturation. Shift the resulting bits right by 16 and return the 16-bit number:

```
round_fx(L_v1) = extract_h(L_add(L_v1, 32768))
```

Initially, this operator was named "`round()`", however to avoid the conflict with C standard libraries, this operator was renamed from version 2.3. There are no functionality changes.

```
L_deposit_h(v1)
```

Deposit the 16-bit `v1` into the 16 most significant bits of the 32-bit output. The 16 least significant bits of the output are zeroed.

```
L_deposit_l(v1)
```

Deposit the 16-bit `v1` into the 16 least significant bits of the 32-bit output. The 16 most significant bits of the output are sign-extended.

```
L_sat(L_v1)
```

The 32-bit variable L_v1 is set to `2147483647` if an overflow occurred, or `-2147483648` if an underflow occurred, on the most recent `L_add_c()`, `L_sub_c()`, `L_macNs()` or `L_msuNs()` operations. The carry and overflow values are binary variables which can be tested and assigned values.

### 22.2.3. Operators with complexity weight of 2

`L_add_c(L_v1, L_v2)`

Performs the 32-bit addition with carry. No saturation. Generates carry and overflow values. The carry and overflow values are binary variables which can be tested and assigned values.

`L_sub_c(L_v1, L_v2)`

This `L_sub_c` operator is reported to have a carry problem. This problem has not been mended and volunteers are seeked to correct it. Use of this operator in current and past release of STL is NOT recommended.

Performs the 32-bit subtraction with carry (borrow). Generates carry (borrow) and overflow values. No saturation. The carry and overflow values are binary variables which can be tested and assigned values.

`shr_r(v1, v2)`

Same as `shr()` but with rounding. Saturate the result in case of underflows or overflows.

```
if (v2>0) then
  if (sub(shl(shr(v1,v2),1), shr(v1,sub(v2,1)))==0)
  then shr_r(v1, v2) = shr(v1, v2)
  else shr_r(v1, v2) = add(shr(v1, v2), 1)

  else if (v2 ≤ 0)
  then shr_r(v1, v2) = shr(v1, v2)}
```

`L_shr_r(L_v1, v2)`

Same as `L_shr(v1,v2)` but with rounding. Saturate the result in case of underflows or overflows:

```
if (v2 > 0) then
  if (L_sub(L_shl(L_shr(L_v1,v2),1), L_shr(L_v1, sub(v2,1)))) == 0
  then L_shr_r(L_v1, v2) = L_shr(L_v1, v2)
  else L_shr_r(L_v1, v2) = L_add(L_shr(L_v1, v2), 1)

  if (v2 ≤ 0)
  then L_shr_r(L_v1, v2) = L_shr(L_v1, v2)}
```

`shl_r(v1, v2)`

Same as `shl()` but with rounding. Saturate the result in case of underflows or overflows:

```
shl_r(v1, v2) = shr_r(v1, -v2)
```

NOTE – In v1.x this operator was called **shift_r(v1, v2)**; in the STL2005, both names can be used.

`L_shl_r(L_v1, v2)`

Same as `L_shl(L_v1,v2)` but with rounding. Saturate the result in case of underflows or overflows.

```
L_shl_r(L_v1, v2) = L_shr_r(L_v1, -v2)
```

In v1.x, this operator is called **L_shift_r(L_v1, v2)** ; both names can be used.

### 22.2.4. Operators with complexity weight of 3

#### 22.2.4.1. Logical Operators

`rotl(v1, v2, * v3)`

→ NEW IN V2.0

Rotates the 16-bit variable v1 by 1 bit to the most significant bits. Bit 0 of v2 is copied to the least significant bit of the result before it is returned. The most significant bit of v1 is copied to the bit 0 of v3 variable.

```
rotr(v1, v2, * v3)
```

→ NEW IN V2.0

Rotates the 16-bit variable v1 by 1 bit to the least significant bits. Bit 0 of v2 is copied to the most significant bit of the result before it is returned. The least significant bit of v1 is copied to the bit 0 of v3 variable.

```
L_rotl(L_v1, v2, * v3)
```

→ NEW IN V2.0

Rotates the 32-bit variable L_v1 by 1 bit to the most significant bits. Bit 0 of v2 is copied to the least significant bit of the result before it is returned. The most significant bit of L_v1 is copied to the bit 0 of v3 variable.

```
L_rotr(L_v1, v2, * v3)
```

→ NEW IN V2.0

Rotates the 32-bit variable L_v1 by 1 bit to the least significant bits. Bit 0 of v2 is copied to the most significant bit of the result before it is returned. The least significant bit of L_v1 is copied to the bit 0 of v3 variable.

### 22.2.5. Operators with complexity weight of 18

```
div_s(v1, v2)
```

Produces a result which is the fractional integer division of `v1` by `v2`. Values in `v1` and `v2` must be positive and `v2` must be greater than or equal to `v1`. The result is positive (leading bit equal to 0) and truncated to 16 bits. If `v1`=`v2`, then `div(v1, v2) = 32767`.

### 22.2.6. Operators with complexity weight of 32

```
div_l(L_v1, v2)
```

Produces a result which is the fractional integer division of a positive 32-bit value `L_v1` by a positive 16-bit value `v2`. The result is positive (leading bit equal to 0) and truncated to 16 bits.

### 22.2.7. Basic operator usage across standards

Table 13 contains a survey of the 16-bit and 32-bit basic operators which are used in various standards. Follows some notes associated to Table 13:

1) abs_s(v1) is referred to as abs(v1) in GSM 06.10 (GSM full-rate).
2) shl(v1,v2) is written as v1<<v2 in GSM 06.10.
3) shr(v1,v2) is written as v1>>v2 in GSM 06.10.
4) v2=extract_h(L_v1) is written as v2 = L_v1 in GSM 06.10.
5) negate(v1) is written as --v1 in GSM 06.10.
6) L_negate(L_v1) is written as --L_v1 in GSM 06.10.
7) L_shl(L_v1,v2) is written as L_v1<<v2 in GSM 06.10.
8) L_shr(L_v1,v2) is written as L_v1>>v2 in GSM 06.10.
9) L_v2=deposit_l(v1) is written as L_v2=v1 in GSM 06.10.
10) div_s(v1,v2) is written as div(v1,v2) in GSM 06.10.
11) norm_l(L_v1) is written as norm(L_v1) in GSM 06.10.
12) GSM 06.20 uses shift_r(v1,v2), which can be implemented as shr_r(v1,--v2).
13) GSM 06.20 uses L_shift_r(L_v1,v2), which can be implemented as L_shr_r(L_v1,--v2).
14) div_s(v1,v2) is written as divide_s(v1,v2) in GSM 06.20.
15) Operator is not part of the original ETSI library.

16) Operator is not part of the original ETSI library but was accepted in the TETRA standard.

**Table 13 — Use of 32-bit basic operators in G.723.1, G.729 and ETSI GSM speech coding recommendations**

| Operation | Weight | Fr gsm | Hr gsm | Efr gsm | Amr gsm | G.729 | G.723.1 | Tetra |
|---|---|---|---|---|---|---|---|---|
| add() | 1 | X | X | X | X | X | X | X |
| sub() | 1 | X | X | X | X | X | X | X |
| abs_s() | 1 | X (1) | X | X | X | X | X | X |
| shl() | 1 | X (2) | X | X | X | X | X | X |
| shr() | 1 | X (3) | X | X | X | X | X | X |
| extract_h() | 1 | | X | X | X | X | X | X |
| extract_l() | 1 | X (4) | X | X | X | X | X | X |
| mult() | 1 | X | X | X | X | X | X | X |
| L_mult() | 1 | X | X | X | X | X | X | X |
| negate() | 1 | X (5) | X | X | X | X | X | |
| round_fx() | 1 | | X | X | X | X | X | X |
| L_mac() | 1 | | X | X | X | X | X | X |
| L_msu() | 1 | | X | X | X | X | X | X |
| L_macNs() | 1 | | | X | | X | X | |
| L_msuNs() | 1 | | | | | X | X | |
| L_add() | 1 | X | X | X | X | X | X | X |
| L_sub() | 1 | X | X | X | X | X | X | X |
| L_negate() | 1 | X (6) | X | X | X | X | X | X |
| L_shl() | 1 | X (7) | X | X | X | X | X | X |
| L_shr() | 1 | X (8) | X | X | X | X | X | X |
| mult_r() | 1 | X | X | X | X | X | X | X |
| mac_r() | 1 | | X | | | | X | |
| msu_r() | 1 | | X | | | | X | |
| L_deposit_h() | 1 | | X | X | X | X | X | X |
| L_deposit_l() | 1 | X (9) | X | X | X | X | X | X |
| L_abs() | 1 | | X | X | X | X | X | X |
| norm_s() | 1 | | X | X | X | X | X | |
| norm_l() | 1 | X (11) | X | X | X | X | X | X |
| L_add_c() | 2 | | | | | | X | |
| L_sub_c() | 2 | | | | | | X | |
| shr_r() | 3 | | X (12) | X | X | X | X | |
| L_shr_r() | 3 | | X (13) | X | X | X | X | X |
| L_sat() | 4 | | | | | X | X | |
| div_s() | 18 | X (10) | X (14) | X | X | X | X | X |
| i_mult() | 3 | | | | | | X (15) | |
| L_mls() | 5 | | | | | | X (15) | |
| div_l() | 32 | | | | | | X (15) | |
| L_mult0() | 1 | | | | | | | X (16) |
| L_mac0() | 1 | | | | | | | X (16) |
| L_msu0() | 1 | | | | | | | X (16) |

## 22.3. Description of the basic operators for unsigned data types

This section describes the different basic operators for unsigned data types available in the STL, organized by complexity ("weights").

### 22.3.1. Variable definitions

The variables used in the operators are signed integers in 2's complements representation, defined by:

`U_var1` 16-bit unsigned variables

`,`
`U_varout_l`

`UL_var1` 32-bit unsigned variables

`,`
`UL_var2`

`,`
`var1`

`,`
`UL_varout_h`

`,`
`UL_varout_l`

### 22.3.2. Operators with complexity weight of 1

```
UL_addNs(UL_var1, UL_var2, *var1)
```

Adds the two unsigned 32-bit variables UL_var1 and UL_var2 with overflow control, but without saturation. Returns 32-bit unsigned result. var1 is set to 1 if wrap around occurred, otherwise 0.

```
UL_subNs(UL_var1, UL_var2, *var1)
```

Subtracts the 32-bit usigned variable UL_var2 from the 32-bit unsigned variable UL_var1 with overflow control, but without saturation. Returns 32-bit unsigned result. var1 is set to 1 if wrap around (to "negative") occurred, otherwise 0.

```
norm_ul (UL_var1)
```

Produces the number of left shifts needed to normalize the 32-bit unsigned variable UL_var1 for positive values on the interval with minimum of 0 and maximum of 0xffffffff. If UL_var1 contains 0, return 0.

```
UL_deposit_l(U_var1)
```

Deposit the 16-bit U_var1 into the 16 LS bits of the 32-bit output. The 16 MS bits of the output are not sign extended.

```
UL_Mpy_32_32(UL_var1, UL_var2)
```

Multiplies the two unsigned values UL_var1 and UL_var2 and returns the lower 32 bits, without saturation control. UL_var1 and UL_var2 are supposed to be in Q32 format. The result is produced in Q64 format, the 32 LS bits. Operates like a regular 32x32-bit unsigned int multiplication in ANSI-C.

### 22.3.3. Operators with complexity weight of 2

```
Mpy_32_32_uu(UL_var1, UL_var2, *UL_varout_h, *UL_varout_l)
```

Multiplies the two unsigned 32-bit variables UL_var1 and UL_var2. The operation is performed in fractional mode. UL_var1 and UL_var2 are supposed to be in Q32 format. The result is produced in Q64 format: UL_varout_h points to the 32 MS bits while UL_varout_l points to the 32 LS bits.

```
Mpy_32_16_uu(UL_var1, U_var1, *UL_varout_h, *U_varout_l)
```

Multiplies the unsigned 32-bit variable UL_var1 with the unsigned 16-bit variable U_var1. The operation is performed in fractional mode : UL_var1 is supposed to be in Q32 format. U_var1 is supposed to be in Q16 format. The result is produced in Q48 format: UL_varout_h points to the 32 MS bits while U_varout_l points to the 16 LS bits.

## 22.4. Description of the 40-bit basic operators and associated weights

This section describes the different 40-bit basic operators available in the STL, and are organized by complexity ("weights"). The complexity values to be considered (since the publication of the STL2005) are the ones related to the version 2.0 and subsequent versions of the library. These basic operators did not exist in the previous version of the library (version 1.x).

A set of coding guidelines must be followed in order to avoid algorithm complexity miss-evaluation. This section describes also these guidelines.

### 22.4.1. Variable definitions

The variables used in the operators are signed integers in 2's complements representation, defined by:

| | |
|---|---|
| `v1`<br>`,`<br>`v2` | 16-bit variables |
| `L_v1`<br>`,`<br>`L_v2`<br>`,`<br>`L_v3` | 32-bit variables |
| `L40_v1`<br>`,`<br>`L40_v2`<br>`,`<br>`L40_v3` | 40-bit variables |

### 22.4.2. Operators with complexity weight of 1

#### 22.4.2.1. Arithmetic operators (multiplication excluded)

`L40_set(L40_v1)`

Assigns a 40-bit constant to the returned 40-bit variable.

`L40_add(L40_v1, L40_v2)`

Adds the two 40-bit variables L40_v1 and L40_v2 **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow.

`L40_sub(L40_v1, L40_v2)`

Subtracts the two 40-bit variables L40_v2 from L40_v1 **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow.

`L40_abs(L40_v1)`

Returns the absolute value of the 40-bit variable L40_v1 without 40-bit saturation control. It will exit execution if it detects a 40-bit overflow.

`L40_shl(L40_v1, v2)`

Arithmetically shifts left the 40-bit variable L40_v1 by v2 positions:

– if v2 is negative, L40_v1 is shifted to the least significant bits by (-v2) positions with extension of the sign bit.
– if v2 is positive, L40_v1 is shifted to the most significant bits by (v2) positions **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow.

`L40_shr(L40_v1, v2)`

Arithmetically shifts right the 40-bit variable L40_v1 by v2 positions:

– if v2 is positive, L40_v1 is shifted to the least significant bits by (v2) positions with extension of the sign bit.
– if v2 is negative, L40_v1 is shifted to the most significant bits by (-v2) positions **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow.

```
L40_negate(L40_v1)
```

Negates the 40-bit variable L40_v1 **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow.

```
L40_max(L40_v1, L40_v2)
```

Compares two 40-bit variables L40_v1 and L40_v2 and returns the maximum value.

```
L40_min(L40_v1, L40_v2)
```

Compares two 40-bit variables L40_v1 and L40_v2 and returns the minimum value.

```
norm_L40(L40_v1)
```

Produces the number of left shifts needed to normalize the 40-bit variable L40_v1 for positive values on the interval with minimum of 1073741824 and maximum 2147483647, and for negative values on the interval with minimum of -2147483648 and maximum of -1073741824; in order to normalize the result, the following operation must be done:

```
L40_norm_v1 = L40_shl( L40_v1, norm_L40( L40_v1))
```

#### 22.4.2.2. Multiplication operators

```
L40_mult(v1, v2)
```

Multiplies the 2 signed 16-bit variables v1 and v2 **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow. The operation is performed **in fractional mode**:
– v1 and v2 are supposed to be in 1Q15 format.
– The result is produced in 9Q31 format.

```
L40_mac(L40_v3, v1, v2)
```

Equivalent to: `L40_add( L40_v1, L40_mult( v2, v3))`

```
L40_msu(L40_v3, v1, v2)
```

Equivalent to: `L40_sub( L40_v1, L40_mult( v2, v3))`

#### 22.4.2.3. Logical operators

```
L40_lshl(L40_v1, v2)
```

Logically shifts left the 40-bit variable L40_v1 by v2 positions:
– if v2 is negative, L40_v1 is shifted to the least significant bits by (-v2) positions with insertion of 0 at the most significant bit.
– if v2 is positive, L40_v1 is shifted to the most significant bits by (v2) positions without saturation control.

```
L40_lshr(L40_v1, v2)
```

Logically shifts right the 40-bit variable L40_v1 by v2 positions:
– if v2 is positive, L40_v1 is shifted to the least significant bits by (v2) positions with insertion of 0 at the most significant bit.
– if v2 is negative, L40_v1 is shifted to the most significant bits by (-v2) positions without saturation control.

#### 22.4.2.4. Data type conversion operators

```
Extract40_H(L40_v1)
```

Returns the bits [31..16] of L40_v1.

```
Extract40_L(L40_v1)
```

Returns the bits [15..00] of L40_v1.

```
round40(L40_v1)
```

Equivalent to:

```
extract_h( L_saturate40( L40_round( L40_v1)))
```

```
L_Extract40(L40_v1)
```

Returns the bits [31..00] of L40_v1.

```
L_saturate40(L40_v1)
```

If L40_v1 is greater than 2147483647, the operator returns 2147483647.
If L40_v1 is lower than -2147483648, the operator returns -2147483648.
Otherwise, it is equivalent to `L_Extract40(L40_v1)`.

```
L40_deposit_h(v1)
```

Deposits the 16-bit variable v1 in the bits [31..16] of the return value: the return value bits [15..0] are set to 0 and the bits [39..32] sign extend v1 sign bit.

```
L40_deposit_l(v1)
```

Deposits the 16-bit variable v1 in the bits [15..0] of the return value: the return value bits [39..16] sign extend v1 sign bit.

```
L40_deposit32(L_v1)
```

Deposits the 32-bit variable L_v1 in the bits [31..0] of the return value: the return value bits [39..32] sign extend L_v1 sign bit.

```
L40_round(L40_v1)
```

Performs a rounding to the infinite on the 40-bit variable L40_v1. 32768 is added to L40_v1 **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow. The end-result 16 LSBits are cleared to 0.

### 22.4.3. Operators with complexity weight of 2

```
mac_r40(L40_v1, v2, v3)
```

Equivalent to:

```
round40( L40_mac( L40_v1, v2, v3))
```

```
msu_r40(L40_v1, v2, v3)
```

Equivalent to:

```
round40( L40_msu( L40_v1, v2, v3))
```

```
L40_shr_r(L40_v1, v2)
```

Arithmetically shifts the 40-bit variable L40_v1 by v2 positions to the least significant bits and rounds the result. It is equivalent to `L40_shr( L40_v1, v2)` except that if v2 is positive and the last shifted out bit is 1, then the shifted result is increment by 1 **without** 40-bit saturation control. It will exit execution if it detects a 40-bit overflow.

```
L40_shl_r(L40_v1, v2)
```

Arithmetically shifts the 40-bit variable L40_v1 by v2 positions to the most significant bits and rounds the result. It is equivalent to `L40_shl( L40_var1, v2)` except if v2 is negative. In this case, it does the same as `L40_shr_r( L40_v1, (-v2))`.

```
Mpy_32_16_ss(L_v1, v2, *L_v3_h, *v3_l)
```

**Multiplies the 2 signed values** L_v1 (32-bit) and v2 (16-bit) with saturation control on 48-bit. The operation is performed in **fractional mode**:
When L_v1 is in 1Q31 format, and v2 is in 1Q15 format, the result is produced in 1Q47 format: L_v3_h bears the 32 most significant bits while v3_l bears the 16 least significant bits.

```
Mpy_32_32_ss(L_v1, L_v2, *L_v3_h, *L_v3_l)
```

Multiplies the two signed 32-bit values L_v1 and L_v2 with saturation control on 64-bit. The operation is performed in **fractional mode**: when L_v1 and L_v2 are in 1Q31 format, the result is produced in 1Q63 format; L_v3_h bears the 32 most significant bits while L_v3_l bears the 32 least significant bits.

### 22.4.4. Coding Guidelines

The following recommendations must be followed in the usage of the 40-bit operators:

1) Only 40-bit variables local to functions can be declared. Declaration of arrays and structures containing 40-bit elements must not be done.
2) 40-bit basic operators and 16/32-bit basic operators must not be mixed within the same loop initialized with a FOR(), DO or WHILE() control basic operator.
   When nested loop software structure is implemented, this recommendation applies to the most inner loops. This enables to have, for instance, an outer loop containing 2 inner loops, with the 1st inner loop using 40-bit basic operators and the 2nd inner loop using 16/32-bit basic operators. However, whenever possible, even such 2 level loop structure configuration should only use either 40-bit basic operators or 16/32-bit basic operators.
   Current version (2.0) of the operator implementation does not evaluate the complexity associated to the mixing of 40-bit and 16/32-bit operators. Subsequent versions may do so.

## 22.5. Description of the basic operators which use complex data types

This section describes the complex basic operators available in the STL, organized by complexity ("weights").

### 22.5.1. Variable definitions

The variables used in the operators are signed integer in 2's complements representation, defined by:

| | |
|---|---|
| `var1`<br>`,`<br>`var2`<br>`,`<br>`var3`<br>`,`<br>`re`<br>`,`<br>`im` | 16-bit variables |
| `C_var`<br>`,`<br>`C_var1`<br>`,`<br>`C_var2`<br>`,`<br>`C_coeff` | 16-bit complex variables |
| `L_var2`<br>`,`<br>`L_var3` | 32-bit variables |

,
L_re

,
L_im

CL_var 32-bit complex variables

,
CL_var1

,
CL_var2

### 22.5.2. Operators with complexity weight of 1

CL_shr(CL_var1, var2)

Arithmetically shifts right the real and imaginary parts of the 32-bit complex number CL_var1 by var2 positions.
If var2 is negative, real and imaginary parts of CL_var1 are shifted to the most significant bits by (-var2) positions with 32-bit saturation control.
If var2 is positive, real and imaginary parts of CL_var1 are shifted to the least significant bits by (var2) positions with sign extension.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_shr(CL_var1.re, L_shift_val);
CL_result.im = L_shr(CL_var1.im, L_shift_val);
```

CL_shl(CL_var1, var2)

Arithmetically shifts left the real and imaginary parts of the 32-bit complex number CL_var1 by L_shift_val positions.
If var2 is negative, real and imaginary parts of CL_var1 are shifted to the least significant bits by ( -var2 ) positions with sign extension.
If var2 is positive, real and imaginary parts of CL_var1 are shifted to the most significant bits by (var2) positions with 32-bit saturation control.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_shl(CL_var1.re, L_shift_val);
CL_result.im = L_shl(CL_var1.im, L_shift_val);
```

CL_add(CL_var1, CL_var2)

Adds the two 32-bit complex numbers CL_var1 and CL_var2 with 32-bit saturation control.
Real part of the 32-bit complex number CL_var1 is added to Real part of the 32-bit complex number CL_var2 with 32-bit saturation control.
The result forms the real part of the result variable.
Imaginary part of the 32-bit complex number CL_var1 is added to Imaginary part of the 32-bit complex number CL_var2 with 32-bit saturation control.
The result forms the imaginary part of the result variable.
Following code snippet describe the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_add(CL_var1.re, CL_var2.re);
CL_result.im = L_add(CL_var1.im, CL_var2.im);
```

CL_sub(CL_var1, CL_var2)

Subtracts the two 32-bit complex numbers CL_var1 and CL_var2 with 32-bit saturation control.

Real part of the 32-bit complex number CL_var2 is subtracted from Real part of the 32-bit complex number CL_var1 with 32-bit saturation control.
The result forms the real part of the result variable.
Imaginary part of the 32-bit complex number CL_var2 is subtracted from Imaginary part of the 32-bit complex number CL_var1 with 32-bit saturation control.
The result forms the imaginary part of the result variable.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_sub(CL_var1.re, CL_var2.re);
CL_result.im = L_sub(CL_var1.im, CL_var2.im);
```

```
CL_scale(CL_var, var1)
```

Multiplies the real and imaginary parts of a 32-bit complex number CL_var by a 16-bit var1. The resulting 48-bit product for each part is rounded, saturated and 32-bit MSB of 48-bit result are returned.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = Mpy_32_16_r(CL_var.re, var1);
CL_result.im = Mpy_32_16_r(CL_var.im, var1);
```

```
CL_dscale(CL_var3, var1, var2)
```

Multiplies the real parts of a 32-bit complex number CL_var3 by a 16-bit var1 and imaginary parts of a 32-bit complex number CL_var3 by a 16-bit var2. The resulting 48-bit product for each part is rounded, saturated and 32-bit MSB of 48-bit result are returned.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = Mpy_32_16_r(CL_var.re, var1);
CL_result.im = Mpy_32_16_r(CL_var.im, var2);
```

```
CL_msu_j(CL_var1, CL_var2)
```

Multiplies the 32-bit complex number CL_var2 with j and subtracts the result from the 32-bit complex number CL_var1 with saturation control.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_add( CL_var1.re, CL_var2.im );
CL_result.im = L_sub( CL_var1.im, CL_var2.re );
```

```
CL_mac_j(CL_var1, CL_var2)
```

Multiplies the 32-bit complex number CL_var2 with j and adds the result to the 32-bit complex number CL_var1 with saturation control.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_sub( CL_var1.re, CL_var2.im );
CL_result.im = L_add( CL_var1.im, CL_var2.re );
```

```
CL_move(CL_var1)
```

Copies the 32-bit complex number CL_var1 to destination 32-bit complex number.

```
CL_Extract_real(CL_var1)
```

Returns the real part of a 32-bit complex number CL_var1.

```
CL_scale (CL_var, var1)
```

Multiplies the real and imaginary parts of a 32-bit complex number CL_var by a 16-bit var1. The resulting 48-bit product for each part is rounded, saturated and 32-bit MSB of 48-bit result are returned.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = Mpy_32_16_r(CL_var.re, var1);
CL_result.im = Mpy_32_16_r(CL_var.im, var1);
```

```
CL_dscale(CL_var, var1, var2)
```

Multiplies the real parts of a 32-bit complex number CL_var by a 16-bit var1 and imaginary parts of a 32-bit complex number CL_var by a 16-bit var2. The resulting 48-bit product for each part is rounded, saturated and 32-bit MSB of 48-bit result are returned.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = Mpy_32_16_r(CL_var.re, var1);
CL_result.im = Mpy_32_16_r(CL_var.im, var2);
```

```
CL_msu_j(CL_var1, CL_var2)
```

Multiplies the 32-bit complex number CL_var2 with j and subtracts the result from the 32-bit complex number CL_var1 with saturation control.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_add( CL_var1.re, CL_var2.im );
CL_result.im = L_sub( CL_var1.im, CL_var2.re );
```

```
CL_mac_j(CL_var1, CL_var2)
```

Multiplies the 32-bit complex number CL_var2 with j and adds the result to the 32-bit complex number CL_var1 with saturation control.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_sub( CL_var1.re, CL_var2.im );
CL_result.im = L_add( CL_var1.im, CL_var2.re );
```

```
CL_move(CL_var)
```

Copies the 32-bit complex number CL_var to destination 32-bit complex number.

```
CL_Extract_real(CL_var)
```

Returns the real part of a 32-bit complex number CL_var.

```
CL_Extract_imag(CL_var)
```

Returns the imaginary part of a 32-bit complex number CL_var.

```
CL_form(L_re, L_im)
```

Combines the two 32-bit variables L_re and L_im and returns a 32-bit complex number.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_re;
CL_result.im = L_im;
```

```
CL_negate(CL_var)
```

Negates the 32-bit complex number, saturates and returns.

The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = L_negate(CL_var.re);
CL_result.im = L_negate(CL_var.im);
```

`CL_conjugate(CL_var)`

Negates only the imaginary part of complex number CL_var with saturation. No change in the real part.
The following code snippet describes the operations:

```
CL_result.re = CL_var.re;
CL_result.im = L_negate(CL_var.im);
```

`CL_mul_j(CL_var)`

Multiplication of a 32-bit complex number CL_var with j and return a 32-bit complex number.

`CL_swap_real_imag(CL_var)`

Swaps real and imaginary parts of a 32-bit complex number CL_var and returns a 32-bit complex number.

`C_add(C_var1, C_var2)`

Adds the two 16-bit complex numbers C_var1 and C_var2 with 16-bit saturation control.
The following code snippet describe the operations performed on real & imaginary part of a complex number:

```
C_result.re = add(C_var1.re, C_var2.re);
C_result.im = add(C_var1.im, C_var2.im);
```

`C_sub(C_var1, C_var2)`

Subtracts the two 16-bit complex numbers C_var1 and C_var2 with 16-bit saturation control.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
C_result.re = sub(C_var1.re, C_var2.re);
C_result.im = sub(C_var1.im, C_var2.im);
```

`C_mul_j(C_var)`

Multiplies a 16-bit complex number with j and returns a 16-bit complex number.

`C_form(re, im)`

Combines the two 16-bit variable re and im and returns a 16-bit complex number.

`CL_scale_32(CL_var1, L_var2)`

Multiplies the real and imaginary parts of a 32-bit complex number CL_var1 by a 32-bit L_var2. The resulting 64-bit product for each part is rounded, saturated and 32-bit MSB of 64-bit result are returned.
The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = Mpy_32_32_r(CL_var1.re, L_var2);
CL_result.im = Mpy_32_32_r(CL_var1.im, L_var2);
```

`CL_dscale_32(CL_var1, L_var2, L_var3)`

Multiplies the real parts of a 32-bit complex number CL_var1 by a 32-bit L_var2 and imaginary parts of a 32-bit complex number CL_var1 by a 32-bit L_var3. The resulting 64-bit product for each part is rounded, saturated and 32-bit MSB of 64-bit result are returned.

The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
CL_result.re = Mpy_32_32_r(CL_var1.re, L_var2);
CL_result.im = Mpy_32_32_r(CL_var1.im, L_var3);
```

`CL_round32_16( CL_var1)`

Rounds the lower 16 bits of the 32-bit complex number CL_var1 into the most significant 16 bits with saturation. Shifts the resulting bits right by 16 and returns the 16-bit complex number.
If real and imaginary of CL_var1 is in 1Q31 format, then the result returned will be rounded and saturated to 1Q15 format.

`C_Extract_real(C_var1)`

Returns the real part of a 16-bit complex number C_var1.

`C_Extract_imag(C_var1)`

Returns the imaginary part of a 16-bit complex number C_var1.

`C_scale(C_var1,var2)`

Multiplies the real and imaginary parts of a 16-bit complex number C_var1 by a 16-bit var2. Returns 32-bit complex number.

`C_negate(C_var1)`

Negates the 16-bit complex number, saturates and returns a 16-bit complex number.

`C_conjugate(C_var1)`

Negates only the imaginary part of a 16-bit complex number C_var1 with saturation. No change in the real part.

`C_shr(C_var1, var2)`

Arithmetically shifts right the real and imaginary parts of the 16-bit complex number C_var1 by var2 positions.
If var2 is negative, real and imaginary parts of C_var1 are shifted to the most significant bits by (-var2) positions with 16-bit saturation control.
If var2 is positive, real and imaginary parts of C_var1 are shifted to the least significant bits by (var2) positions with sign extension.

`C_shl(C_var1,var2)`

Arithmetically shifts left the real and imaginary parts of the 16-bit complex number C_var1 by var2 positions.
If var2 is negative, real and imaginary parts of C_var1 are shifted to the least significant bits by (-var2) positions with sign extension.
If var2 is positive, real and imaginary parts of C_var1 are shifted to the most significant bits by (var2) positions with 16-bit saturation control.

### 22.5.3. Operators with complexity weight of 2

`CL_multr_32x16(CL_var, C_coeff)`

Multiplication of 32-bit complex number CL_var with a 16-bit complex number C_coeff.
The formula for multiplying two complex numbers, `(x+iy)` and `(u+iv)` is:

```
(x+iy)*(u+iv) = (xu – yv) + i(xv + yu);
```

The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
W_tmp1 = W_mult_32_16(CL_var.re, C_coeff.re);
```

```
W_tmp2 = W_mult_32_16(CL_var.im, C_coeff.im);
W_tmp3 = W_mult_32_16(CL_var.re, C_coeff.im);
W_tmp4 = W_mult_32_16(CL_var.im, C_coeff.re);

CL_res.re = W_round48_L( W_sub_nosat (W_tmp1, W_tmp2));
CL_res.im = W_round48_L( W_add_nosat (W_tmp3, W_tmp4));
```

For example, if the real and imaginary part of complex variable CL_var are in 1Q31 format, and C_coeff in 1Q15 format, then the intermediate products would be in 17Q47 format. The round operation will convert the result of addition/subtraction from 17Q47 format to 1Q31 format.

`C_multr(C_var1, C_var2)`

Multiplies the 16-bit complex number C_var1 with the 16-bit complex number C_var2 which results in a 16-bit complex number.
The formula for multiplying two complex numbers, `(x+iy)` and `(u+iv)` is:

```
(x+iy)*(u+iv) = (xu – yv) + i(xv + yu);
```

The following code snippet describes the operations performed on real & imaginary part of a complex number:

```
W_tmp1 = W_mult_16_16(C_var1.re, C_var2.re);
W_tmp2 = W_mult_16_16(C_var1.im, C_var2.im);
W_tmp3 = W_mult_16_16(C_var1.re, C_var2.im);
W_tmp4 = W_mult_16_16(C_var1.im, C_var2.re);

C_result.re = round_fx(W_sat_l (W_sub_nosat (W_tmp1, W_tmp2)));
C_result.im = round_fx(W_sat_l (W_add_nosat (W_tmp3, W_tmp4)));
```

`CL_multr_32x32(CL_var1, CL_var2)`

Complex multiplication of CL_var1 and CL_var2. Multiplication is in fractional mode. Both input and outputs are in 1Q31 format.
The following code snippet describes the performed operations:

```
W_tmp1 = W_mult_32_32(CL_var1.re, CL_var2.re);
W_tmp2 = W_mult_32_32(CL_var1.im, CL_var2.im);
W_tmp3 = W_mult_32_32(CL_var1.re, CL_var2.im);
W_tmp4 = W_mult_32_32(CL_var1.im, CL_var2.re);

CL_res.re = W_round64_L( W_sub (W_tmp1, W_tmp2));
CL_res.im = W_round64_L( W_add (W_tmp3, W_tmp4));
```

`C_mac_r(CL_var1, C_var2, var3)`

Multiplies real and imaginary part of C_var2 by var3 and shifts the result left by 1. Adds the 32-bit result to CL_var1 with saturation. Rounds the 16 least significant bits of the result into the 16 most significant bits with saturation and shifts the result right by 16. Returns a 16-bit complex result.

```
C_result = CL_round32_16( CL_add( Cl_var1, C_scale(C_var2, var3) ) );
```

`C_msu_r(CL_var1, C_var2, var3)`

Multiplies real and imaginary part of C_var2 by var3 and shifts the result left by 1. Subtracts the 32-bit result from CL_var1 with saturation. Rounds the 16 least significant bits of the result into the 16 most significant bits with saturation and shifts the result right by 16. Returns a 16-bit complex result.

```
C_result = CL_round32_16( CL_sub( Cl_var1, C_scale(C_var2, var3) ) );
```

## 22.6. Description of the basic operators which use 64-bit registers/accumulators

This section describes the 64-bit basic operators available in the STL, organized by complexity ("weights").

### 22.6.1. Variable definitions

The variables used in the operators are signed integer in 2's complements representation, defined by:

| | |
|---|---|
| `var1` | 16-bit variables |
| `,` `var2` | |
| `L_var1` | 32-bit variables |
| `,` `L_var2` | |
| `W_var` | 64-bit variables |
| `,` `W_var1` | |
| `,` `W_var2` | |
| `,` `W_acc` | |

### 22.6.2. Operators with complexity weight of 1

`W_add_nosat(W_var1, W_var2)`

Adds the two 64-bit variables W_var1 and W_var2 without saturation control on 64 bits.

`W_sub_nosat(W_var1, W_var2)`

Subtracts the two 64-bit variables W_var1 and W_var2 without saturation control on 64 bits.

`W_shl(W_var1, var2)`

Arithmetically shifts left the 64-bit variable W_var1 by var2 positions:
if var2 is negative, W_var1 is shifted to the least significant bits by (–var2) positions with extension of the sign bit.
if var2 is positive, W_var1 is shifted to the most significant bits by (var2) positions with saturation control on 64 bits.

`W_shl_nosat(W_var1, var2)`

Arithmetically shifts left the 64-bit variable W_var1 by var2 positions:
if var2 is negative, W_var1 is shifted to the least significant bits by (–var2) positions with extension of the sign bit.
if var2 is positive, W_var1 is shifted to the most significant bits by (var2) positions without saturation control on 64 bits.

`W_shr(W_var1, var2)`

Arithmetically shifts right the 64-bit variable W_var1 by var2 positions:
if var2 is negative, W_var1 is shifted to the most significant bits by (–var2) positions with saturation control on 64 bits .
if var2 is positive, W_var1 is shifted to the least significant bits by (var2) positions with extension of the sign bit.

`W_shr_nosat(W_var1, var2)`

Arithmetically shifts right the 64-bit variable W_var1 by var2 positions:
if var2 is negative, W_var1 is shifted to the most significant bits by (–var2) positions without saturation control on 64 bits.
if var2 is positive, W_var1 is shifted to the least significant bits by (var2) positions with extension of the sign bit.

`W_mult_32_16(L_var1, var2)`

**This operator is SIMD and VLIW friendly**

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shifts the product left by 1 and sign extends to 64-bits without saturation control.
The operation is performed in fractional mode.
For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the result is produced in 17Q47 format.

```
W_mac_32_16(W_acc, L_var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shifts the product left by 1 and sign extends to 64-bits without saturation control; adds this 64 bit value to the 64 bit W_acc without saturation control, and returns a 64 bit result.
The operation is performed in fractional mode.
For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then added to W_acc (in 17Q47) format. The final result is in 17Q47 format.

```
W_msu_32_16(W_acc, L_var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Left-shift the product by 1 and sign extends to 64-bit without saturation control;
subtracts this 64 bit value from the 64 bit W_acc without saturation control, and returns a 64 bit result.
The operation is performed in fractional mode.
For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then subtracted from W_acc (in 17Q47) format. The final result is in 17Q47 format.

```
W_mult0_16_16(var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies 16-bit var1 by 16-bit var2, sign extends to 64 bits and returns the 64 bit result.

```
W_mac0_16_16(W_acc, var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies 16-bit var1 by 16-bit var2, sign extends to 64 bits; adds this 64 bit value to the 64 bit W_acc without saturation control, and returns a 64 bit result.

```
W_msu0_16_16(W_acc, var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies 16-bit var1 by 16-bit var2, sign extends to 64 bits; subtracts this 64 bit value from the 64 bit W_acc without saturation control, and returns a 64 bit result.

```
W_mult_16_16(W_acc, var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies a signed 16-bit var1 by signed 16-bit var2, shifts the product left by 1 and sign extends to 64-bits without saturation control and returns a 64 bit result.
The operation is performed in fractional mode.
For example, if var1 is in 1Q15 format and var2 is in 1Q15 format, then the result is produced in 33Q31 format.

```
W_mac_16_16(W_acc, var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies a signed 16-bit var1 by signed 16-bit var2, shifts the result left by 1 and sign extends to 64-bits; add this 64 bit value to the 64 bit W_acc without saturation control, and returns a 64 bit result.
The operation is performed in fractional mode.
For example, if var1 is in 1Q15 format and var2 is in 1Q15 format, then the product is in 33Q31 format which is then added to W_acc (in 33Q31 format) to provide a final result in 33Q31 format.

```
W_msu_16_16(W_acc, var1, var2)
```

**This operator is SIMD and VLIW friendly**

Multiplies a signed 16-bit var1 by signed 16-bit var2, shifts the result left by 1 and sign extends to 64-bit; subtracts this 64 bit value from the 64 bit W_acc without saturation control, and returns a 64 bit result.
The operation is performed in fractional mode.
For example, if var1 is in 1Q15 format and var2 is in 1Q15 format, then the product is in 33Q31 format which is then subtracted from W_acc (in 33Q31 format) to provide a final result in 33Q31 format.

```
W_deposit32_l(L_var1)
```

Deposits the 32 bit L_var1 into the 32 LS bits of the 64-bit output. The 32 MS bits of the output are sign extended.

```
W_deposit32_h(L_var1)
```

Deposits the 32-bit L_var1 into the 32 MS bits of the 64-bit output. The 32 LS bits of the output are zeroed.

```
W_sat_l(W_v1)
```

Saturates the 64-bit variable W_v1 to 32-bit value and returns the lower 32 bits.
For example, a 64-bit wide accumulator is helpful in accumulating 16*16 multiplies without checking for saturation. However, at the end of the multiply-and-accumulate loop, we need to return only the 32-bit value after checking for saturation.
If W_v1 is in 33Q31 format, then the result returned will be saturated to 1Q31 format.

```
W_sat_m(W_v1)
```

Arithmetically shifts right the 64-bit variable W_v1 by 16 bits; saturates the 64-bit value to 32-bit value and returns the lower 32 bits.
For example, a 64-bit wide accumulator is helpful in accumulating 32*16 multiplies without checking for saturation. A 32*16 multiply gives a 48-bit product; at the end of the multiply-and-accumulate loop, the result is in the lower 48 bits of the 64-bit accumulator. Now an arithmetic right shift by 16 bits will drop the LSB 16 bits. Now we should check for saturation and return the lower 32 bits.
If W_var is in 17Q47 format, then the result returned will be saturated to 1Q31 format.

```
W_shl_sat_l(W_1, var1)
```

Arithmetically shifts left the 64-bit W_v1 by v1 positions with lower 32-bit saturation and returns the 32 LSB of 64-bit result.
If v1 is negative, the result is shifted to right by (-var1) positions and sign extended. After shift operation, returns the 32 MSB of 64-bit result.

```
W_extract_l(W_var1)
```

Returns the 32 LSB of a 64-bit variable W_var1.

```
W_extract_h(W_var1)
```

Returns the 32 MSB of a 64-bit variable W_var1.

```
W_round48_L(W_var1)
```

Rounds the lower 16 bits of the 64-bit input number W_var1 into the most significant 32 bits with saturation. Shifts the resulting bits right by 16 and returns the 32-bit number:
If W_var1 is in 17Q47 format, then the result returned will be rounded and saturated to 1Q31 format.

```
W_round32_s(W_var1)
```

Rounds the lower 32 bits of the 64-bit input number W_var1 into the most significant 16 bits with saturation. Shifts the resulting bits right by 32 and returns the 16-bit number:
If W_var1 is in 17Q47 format, then the result returned will be rounded and saturated to 1Q15 format.

```
W_norm(W_var1)
```

Produces the number of left shifts needed to normalize the 64-bit variable W_var1. If W_var1 contains 0, return 0.

```
W_add(W_var1, W_var2)
```

Adds the two 64-bit variables W_var1 and W_var2 with 64-bit saturation control. Sets overflow flag. Returns 64-bit result.

```
W_sub(W_var1, W_var2)
```

Subtracts 64-bit variable W_var2 from W_var1 with 64-bit saturation control. Sets overflow flag. Returns 64-bit result.

```
W_neg(W_var1)
```

Negates a 64-bit variables W_var1 with 64-bit saturation control. Set overflow flag. Returns 64-bit result.

```
W_abs(W_var1)
```

Returns a 64-bit absolute value of a 64-bit variable W_var1 with saturation control.

```
W_mult_32_32(L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shift the product left by 1 with saturation control. Returns the 64-bit result.
The operation is performed in fractional mode.
For example, if L_var1 & L_var2 are in 1Q31 format then the result is produced in 1Q63 format.
Note that `W_mult_32_32(-2147483648, -2147483648) = 9223372036854775807`.

```
W_mult0_32_32(L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Returns the 64-bit result.
For example, if L_var1 & L_var2 are in 1Q31 format then the result is produced in 2Q62 format.

```
W_lshl(W_var1, var2)
```

Logically shift the 64-bit input W_var1 left by var2 positions. If var2 is negative, logically shift right W_var1 by (-var2).

```
W_lshr(W_var1, var2)
```

Logically shift the 64-bit input W_var1 right by var2 positions. If var2 is negative, logically shift left W_var1 by (-var2).

```
W_round64_L(W_var1)
```

Rounds the lower 32 bits of the 64-bit input number W_var1 into the most significant 32 bits with saturation. Shifts the resulting bits right by 32 and returns the 32-bit number.
If W_var1 is in 1Q63 format, then the result returned will be rounded and saturated to 1Q31 format.

## 22.7. Basic operators which use 32-bit precision multiply

Basic operators in this section are useful for FFT and scaling functions where the result of a 32*16 or 32*32 arithmetic operation is rounded, and saturated to 32-bit value. There is no accumulation of products in these functions. In functions that accumulate products, you should use base operators in Section clause 22.6.

All basic operators in this section have a complexity weight of 1.

### 22.7.1. Variable definitions

The variables used in the operators are signed integers in 2's complements representation, defined by:

| | |
|---|---|
| `var2` | 16-bit variables |
| `L_var1`<br>,<br>`L_var2`<br>,<br>`L_var3` | 32-bit variables |

### 22.7.2. Operators

```
Mpy_32_16_1(L_var1, var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shifts the product left by 1 with 48-bit saturation control; Returns the 32 MSB of the 48-bit result after truncation of lower 16 bits. The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, truncated and returned in 1Q31 format.

The following code snippet describes the operations performed:

```
W_var1 = W_mult_32_16 ( L_var1, var2 );
L_var_out = W_sat_m( W_var1 );
```

```
Mpy_32_16_r(L_var1, var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shifts the product left by 1 with 48-bit saturation control; Returns the 32 MSB of the 48-bit result after rounding of the lower 16 bits. The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then rounded, saturated, and returned in 1Q31 format.

The following code snippet describes the operations performed:

```
W_var1 = W_mult_32_16( L_var1, var2 );
L_var_out = W_round48_L (W_var1);
```

```
Mpy_32_32(L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shifts the product left by 1 with 64-bit saturation control; Returns the 32 MSB of the 64-bit result after truncating of the lower 32 bits. The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q31 format, then the product is produced in 1Q63 format which is then truncated, saturated, and returned in 1Q31 format.

The following code snippet describes the operations performed:

```
W_var1 = (( Word64)L_var1 * L_var2);
L_var_out = W_extract_h(W_shl(W_var1, 1) );
```

```
Mpy_32_32_r(L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Adds rounding offset to lower 31 bits of the product. Shifts the result left by 1 with 64-bit saturation control; returns the 32 MSB of the 64-bit result with saturation control.

The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the result is produced in 1Q63 format which is then rounded, saturated, and returned in 1Q31 format.

The following code snippet describes the operations performed:

```
W_var1 = (( Word64)L_var1 * L_var2);
W_var1 = W_var1 + 0x40000000LL;
W_var1 = W_shl ( W_var1, 1 );
L_var_out = W_extract_h( W_var1 );
```

```
Madd_32_16(L_var3, L_var1, var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shift the product left by 1 with 48-bit saturation control; Add the 32-bit MSB of the 48-bit result with 32-bit L_var3 with 32-bit saturation control. The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, truncated to 1Q31 format and added to L_var3 in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_16_1(L_var1, var2);
L_var_out = L_add(L_var3, L_var_out);
```

```
Madd_32_16_r(L_var3, L_var1, var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shifts the product left by 1 with 48-bit saturation control; Gets the 32-bit MSB from 48-bit result after rounding of the lower 16 bits and adds this with 32-bit L_var3 with 32-bit saturation control.

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, rounded to 1Q31 format and added to L_var3 in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_16_r(L_var1, var2);
L_var_out = L_add(L_var3, L_var_out);
```

```
Msub_32_16(L_var3, L_var1, var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shifts the product left by 1 with 48-bit saturation control; Subtracts the 32-bit MSB of the 48-bit result from 32-bit L_var3 with 32-bit saturation control.

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, truncated to 1Q31 format and subtracted from L_var3 in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_16_1(L_var1, var2);
L_var_out = L_sub(L_var3, L_var_out);
```

```
Msub_32_16_r(L_var3, L_var1, var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Shifts the product left by 1 with 48-bit saturation control; Gets the 32-bit MSB from 48-bit result after rounding of the lower 16 bits and subtracts this from 32-bit L_var3 with 32-bit saturation control.

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and var2 is in 1Q15 format, then the product is produced in 17Q47 format which is then saturated, rounded to 1Q31 format and subtracted from L_var3 in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_16_r(L_var1, var2);
L_var_out = L_sub(L_var3, L_var_out);
```

```
Madd_32_32(L_var3, L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shifts the product left by 1 with 64-bit saturation control; Adds the 32 MSB of the 64-bit result to 32-bit signed variable L_var3 with 32-bit saturation control.

The operation is performed in fractional mode. For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and truncated in 1Q31 format which is then added to L_var3 (in 1Q31 format), to provide result in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_32(L_var1, L_var2);
L_var_out = L_add(L_var3, L_var_out);
```

```
Madd_32_32_r(L_var3, L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Adds rounding offset to lower 31 bits of the product. Shifts the result left by 1 with 64-bit saturation control; gets the 32 MSB of the 64-bit result with saturation and adds this with 32-bit signed variable L_var3 with 32-bit saturation control.

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and rounded in 1Q31 format which is then added to L_var3 (in 1Q31 format), to provide result in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_32_r(L_var1, L_var2);
L_var_out = L_add(L_var3, L_var_out);
```

```
Msub_32_32(L_var3, L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Shifts the product left by 1 with 64-bit saturation control; Subtracts the 32 MSB of the 64-bit result from 32-bit signed variable L_var3 with 32-bit saturation control.

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and truncated in 1Q31 format which is then subtracted from L_var3 (in 1Q31 format), to provide result in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_32(L_var1, L_var2);
L_var_out = L_sub(L_var3, L_var_out);
```

```
Msub_32_32_r(L_var3, L_var1, L_var2)
```

Multiplies the signed 32-bit variable L_var1 with signed 32-bit variable L_var2. Adds rounding offset to lower 31 bits of the product. Shifts the result left by 1 with 64-bit saturation control; gets the 32 MSB of the 64-bit result with saturation and subtracts this from 32-bit signed variable L_var3 with 32-bit saturation control.

The operation is performed in fractional mode.

For example, if L_var1 is in 1Q31 format and L_var2 is in 1Q31 format, then the product is saturated and rounded in 1Q31 format which is then subtracted from L_var3 (in 1Q31 format), to provide result in 1Q31 format.

The following code snippet describes the operations performed:

```
L_var_out = Mpy_32_32_r(L_var1, L_var2);
L_var_out = L_sub(L_var3, L_var_out);
```

## 22.8. Description of the basic operators for control operations

The following basic operators should be used in the control processing part of the reference code. They are expected to help compilers generate more efficient code for control sections of the reference C code. In addition, they also help in computing a more accurate representation of control code operations in the total WMOPs (weighted millions of operations) of the reference code.

All operators in this section have a complexity weight of 1.

### 22.8.1. Variable definitions

The variables used in the operators are signed integers in 2's complements representation, defined by:

`var1` , `var2` 16-bit variables

`L_var1` , `L_var2` 32-bit variables

`W_var1` , `W_var2` 64-bit variables

```
LT_16(var1, var2)
```

Returns 1 if 16-bit variable var1 is less than 16-bit variable var2, else returns 0.

```
GT_16(var1, var2)
```

Returns 1 if 16-bit variable var1 is greater than 16-bit variable var2, else returns 0.

```
LE_16(var1, var2)
```

Returns 1 if 16-bit variable var1 is less than or equal to 16-bit variable var2, else return 0.

```
GE_16(var1, var2)
```

Returns 1 if 16-bit variable var1 is greater than or equal to 16-bit variable var2, else returns 0.

```
EQ_16(var1, var2)
```

Returns 1 if 16-bit variable var1 is equal to 16-bit variable var2, else returns 0.

```
NE_16(var1, var2)
```

Returns 1 if 16-bit variable var1 is not equal to 16-bit variable var2, else returns 0.

```
LT_32(L_var1, L_var2)
```

Returns 1 if 32-bit variable L_var1 is less than 32-bit variable L_var2, else returns 0.

```
GT_32(L_var1, L_var2)
```

Returns 1 if 32-bit variable L_var1 is greater than 32-bit variable L_var2, else returns 0.

```
LE_32(L_var1, L_var2)
```

Returns 1 if 32-bit variable L_var1 is less than or equal to 32-bit variable L_var2, else returns 0.

```
GE_32(L_var1, L_var2)
```

Returns 1 if 32-bit variable L_var1 is greater than or equal to 32-bit variable L_var2, else returns 0.

```
EQ_32(L_var1, L_var2)
```

Returns 1 if 32-bit variable L_var1 is equal to 32-bit variable L_var2, else returns 0.

```
NE_32(L_var1, L_var2)
```

Returns 1 if 32-bit variable L_var1 is not equal to 32-bit variable L_var2, else returns 0.

```
LT_64(W_var1, W_var2)
```

Returns 1 if 64-bit variable W_var1 is less than 64-bit variable W_var2, else returns 0.

```
GT_64(W_var1, W_var2)
```

Returns 1 if 64-bit variable W_var1 is greater than 64-bit variable W_var2, else returns 0.

```
LE_64(W_var1, W_var2)
```

Returns 1 if 64-bit variable W_var1 is less than or equal to 64-bit variable W_var2, else returns 0.

```
GE_64(W_var1, W_var2)
```

Returns 1 if 64-bit variable W_var1 is greater than or equal to 64-bit variable W_var2, else returns 0.

```
NE_64(W_var1, W_var2)
```

Returns 1 if 64-bit variable W_var1 is not equal to 64-bit variable W_var2, else returns 0.

```
EQ_64(W_var1, W_var2)
```

Returns 1 if 64-bit variable W_var1 is equal to 64-bit variable W_var2, else returns 0.

## 22.9. Description of the control basic operators and associated weights

This section describes the different control basic operators available in the STL and their associated complexity weights. The complexity values to be considered (since the publication of the STL2005) are the ones related to 2.0 and subsequent versions of the library.

**A set of coding guidelines must be followed** in order to avoid algorithm complexity miss-evaluation. This section describes also these guidelines.

### 22.9.1. Operators and complexity weights

Nine macros are defined to enable the evaluation of the complexity associated to control instructions that are frequently used in C.

– The **IF(expression)** and **ELSE** macros evaluate the cost of the C statement:

```
if (expression) {...}[[else if(expression2) {...}] else {...}]
```

– The **SWITCH(expression)** macro evaluates the cost of the C statement:

```
switch (expression) {...}
```

– The **WHILE (expression)** macro evaluates the cost of the C statement:

```
while (expression) {...}
```

– The **FOR (expr1;expr2; expr3)** macro evaluates the cost of the C statement:

```
for (expr1; expr2; expr3) {...}
```

– The **DO** and **WHILE(expression)** macros evaluates the cost of the C statement:

```
do {...} while (expression)
```

– The **CONTINUE** macro evaluates the cost of the C statement:

```
while (expression) {
...
continue;
...
```

or

```
for (expr1; expr2; expr3) {
...
continue;
...
```

– The **BREAK** macro evaluates the cost of the C statement:

```
while(expression)
{
...
break;
...
```

or

```
for (expr1; expr2; expr3) {
...
break;
...
}
```

or

```
switch(...) {
...
break;
...
}
```

– The **GOTO** macro evaluates the cost of the C statement:

```
goto label;
```

Table 14 summarizes the control basic operators and their associated complexity.

**Table 14 — Control basic operators and associated complexity**

| Complexity weight | Basic operator | Description |
|---|---|---|
| 0 | **DO**{…} while(expression) | The macro DO must be used instead of the 'do' C statement. |
| 3 | **FOR**(expr1; expr2; expr3) {…} | The macro FOR must be used instead of the 'for' C statement. The complexity is **independent** of the number of loop iterations that are performed. |
| 0 | **if**(expression)<br>**one_and_only_one_basic_operator (control operators excluded)** | **The macro IF must not be used** when the 'if' structure does not have any 'else if' nor 'else' statement and it conditions only one basic operator (control operators excluded). |
| 3 | **IF**(expression) {…} | The macro IF must be used instead of the 'if' C statement **in every other case**: when there is an 'else' or 'else if' statement, or when the 'if' conditions several basic operators, or when the 'if' conditions a function call or when the 'if' conditions a control operator. |
| 4 | if(expression) {…} [[<br>**ELSE** if(expression2){…}]<br>**ELSE** {…}] | The macro ELSE must be used instead of the 'else' C statement. |
| 6 | **SWITCH**(expression) {…} | The macro SWITCH must be used instead of the 'switch' C statement. |
| 3 | **WHILE**(expression) {…} | The macro WHILE must be used instead of the 'while' C statement. The complexity is **proportional** to the number of loop iterations that are performed. |
| 2 | while(expression) {<br>… **CONTINUE**; …<br>or<br>for(expr1; expr2; expr3) {<br>…<br>**CONTINUE**;<br>…<br>} | The macro CONTINUE must be used instead of the 'continue' C statement. |
| 2 | while(expression) {<br>…**BREAK**; …<br>or<br>for(expr1; expr2; expr3) {<br>…<br>**BREAK**;<br>…<br>or<br>switch(var) { | The macro BREAK must be used instead of the 'break' C statement. |

| Complexity weight | Basic operator | Description |
|---|---|---|
| | …<br>**BREAK**;<br>…<br>} | |
| 2 | **GOTO** | The macro GOTO must be used instead of the 'goto' C statement. |

### 22.9.2. Coding guidelines

#### 22.9.2.1. When to use IF() instead of if()?

The **IF**() macro must be used instead of the classical C statement **if**(), wherever:
– There is an else or else if statement,
– There is **strictly more than one basic operators** to condition,
– There is at least a function call to condition.
– There is a control basic operator to condition.

An example code:

```
if (x == 0)
z = add(z, sub(y, x)); /* more than one basic ops */
if (z == 0)
Decode(); /* function call */
something();
```

must be written as:

```
IF (x == 0)
z = add(z, sub(y, x));
IF (z == 0)
Decode();
something();
```

While below code must stay untouched since **only one** basic operator is conditioned.

```
if (x == 0)
z = add(z, x); /* one basic op */
something();
```

#### 22.9.2.2. When to use FOR() and WHILE() macros?

The `FOR()` and `WHILE()` macros must be used to differentiate loops which can be handled by a h/w loop controller from complex loops which need to be controlled by additional control software.
– Follows an example of a **simple h/w loop** that must be designed with the **FOR()** macro. It will iterate C-statement E0 to E20 a number of times **known at loop entry** (and at least once). Therefore, for such loops, there is no complexity associated to the computation of the decision to loop back or not:

```
/* var1 > 0 is ensured  */
FOR (n = 0; n < var1; n++) {
E0;
/* never do anything that impacts "var1" nor "n" value */
E20;
}
```

– Follows an example of a **complex s/w loop** that must be designed with the **WHILE()**. It will iterate C-statement E0 to E20 a number of times **undefined** at loop entry (eventually 0 times). Indeed, at the end of one loop iteration, the decision to loop back depends on the processing done within the elapsed iteration.

```
/* do not need to ensure n < var1 at loop entry */
WHILE (n < var1) {
E0;
/* can do anything that impacts "var1" or "n" value */
E20;
}
```

ANSI-C defines **for**() structures with **while**() structures, but by differencing the **FOR**() and **WHILE**() macro usage, a better complexity evaluation of the loop controlling is made.

- A loop defined with FOR() macro:
  - **Only counts the initial set-up** of the h/w loop controller with a complexity weight of 3.
  - **Must iterate at least once**.
  - Has a complexity **independent** of the number of iterations that are performed.
- A Loop defined with WHILE() macro:
  - Counts, at **every single iteration** which is executed, **the complexity associated to the computation of the decision to loop back or not**.
  - **Can be executed 0 times**.
  - Has a complexity **proportional** (by a factor of 4) to the number of iterations that are performed.

#### 22.9.2.3. When to use DO and WHILE() macros?

It is important to **modify** below C code:

```
do {
x = sub(x, y)
} while (x < 0);
```

into following one:

```
DO {
x = sub(x, y)
} WHILE (x < 0);
```

The following code is also possible but, although the associated complexity computation will be identical, it can generate parsing errors by some source code editors which perform on-the-fly syntax checking.

```
do {
x = sub(x, y)
} WHILE (x < 0);
```

#### 22.9.2.4. Testing an expression equality

***if(expression) {…} and while(expression) {…} C statements.***

All arithmetic tests on data must be presented as a comparison to zero. To perform comparison between two variables (or a variable and a non-zero constant), a subtraction (**sub** or **L_sub** or **L40_sub**) must be performed first.

For example, below examples leads to an under evaluation of the complexity:

```
if (a > 3) { }
while (a != 5) { } ...
```

While, below examples leads to a correct evaluation of the complexity:

```
if (sub(a,3) > 0) { }
while (sub(a, 5) != 0) { } ...
```

If multiple condition need to be evaluated and merged, one **test**() operator must be used for each additional test to be done.

Example 1:

The following code …

```
if ( (a > b) && (c > d)) {}
```

…must be modified to:

```
*test*();
if ( (sub( a, b) > 0) && (sub( c, d) > 0)) {}
```

Example 2:

The following code …

```
if ( (a > b)
&& (c > d)
||(e > f)) {}
```

…must be modified to:

```
*test*();
*test*();
if (  (sub( a, b) > 0)
&& ( sub( c, d) > 0)
|| (sub( e, f) > 0)) {
```

***(condition) ? (statement1) : (statement2)***

The ternary operator "**? :**" must not be used since it does not enable the evaluation of the associated complexity.

Therefore, instead of writing:

```
(condition) ? (statement1) : (statement2)
```

One must write:

```
IF(condition)
  statement1;
ELSE
  statement2;
```

Whenever it is possible to avoid the **else** clause, one should write:

```
  statement2;
  IF(condition)
    statement1;
```

And whenever **statement1** is **one and only one basic operator** (control operator excluded), one can write:

```
statement2;
if(condition)
  one_and_only_one_basic_operator;
```

***for(expresion1; expression2; expression3)***

A "**for**" C statement must be limited to initializing, testing and incrementing the loop counter. The following C code statement is an example of incorrect usage:

```
for(i=0, j=0; i<N & w>0 ; i++, j+=3)
```

It must be replaced by:

```
j=0;
for(i=0; i<N ; i++) {
  j = add(j,3);
```

```
  if(w > 0)
    break;
}
```

Actually, in order to respect the other recommendations, it must be replaced by:

```
j=0;
FOR(i=0; i<N ; i++) {
  j = add(j,3);
  if(w > 0){
    BREAK;
  }
}
```

## 22.10. Complexity associated with data moves and other operations

### 22.10.1. Data moves

Each data move between two 16-bit or two 32-bit variables, **move16**() and **move32**() operators respectively, has a complexity weight of 1.

1) A 16-bit variable cannot be directly moved to a 32-bit or 40-bit variable.
2) A 32-bit variable cannot be directly moved to a 16-bit or 40-bit variable.
3) A 40-bit variable cannot be directly moved to a 16-bit or 32-bit variable.

For above 3 types of moves, functions such as the following ones must be used:

| | | |
|---|---|---|
| round_fx() | round40() | L_saturate40() |
| extract_h() | Extract40_H() | L_Extract40() |
| extract_l() | Extract40_L() | L40_deposit32() |
| L_deposit_h() | L40_deposit_h() | |
| L_deposit_l() | L40_deposit_l() | |

There will be no extra weighting for data move when using above functions: the weighting of the data move is already included in the weighting of these functions.

Data moves are only counted in the following cases:

1) A data move from a constant to a variable;
2) A data move from a variable to a variable;
3) A data move of the result of a basic operation to an array;
4) A typecast from a Word8 to a Word16 variable;
5) A typecast from a Word8 to a Word32 variable.

#### 22.10.1.1. Is it necessary to count the complexity of typecast from Word8 to Word16 or Word32?

Following example shows necessity:

```
const Word8 tbl[2] = {0x1, 0x2};

Word16 tmp;
tmp = add((Word16 )tbl[0], (Word16 )tbl[1]); move16(); move16();
a = my_function((Word16 )tbl[0], (Word16 )tbl[1]); move16(); move16();
/* assuming my_function requires 2 Word16 parameters. */
```

#### 22.10.1.2. Is it necessary to count an address calculation when initializing a pointer?

For a case when initializing a pointer with an address, extra weight count, e.g. move16(), is not necessary because address computation is not counted as a part of the complexity. For example:

```
Word16 *p;
p = &(array[1]); /* move16() not necessary */
```

#### 22.10.1.3. Does return value assignment need to be counted?

When a return value of a function is assigned, it does not require move16() count. For example:

```
Word16 a = function(b); /* move16() not necessary */
```

Here is another example that does not require copy count:

```
Word16 function(Word16 b) {
return (add(b, 1)); /* move16() not necessary */.
}
```

Introducing dummy functions to avoid extra basop counts must be avoided.

#### 22.10.1.4. What to do with return value of a macro?

When copying return value of a complex macro to a variable, it does not require extra weight count as in a return value of a function:

```
Word32 w = Mpy_32(); /* move32() not required */
```

### 22.10.2. Other operations

Address computation must be excluded from the complexity evaluation. However, when extremely complex address computations are done, these address computations should be resolved using the basic operations, in order to account for the associated complexity.

For example, incrementation of an address by one does not require an extra count, but followings would be considered complex:

```
j=0; move16();
FOR( i=0; i<5; i++) {
sum = add(sum, array[j]);
j = add(j,i);
}"
```

## 22.11. Program ROM estimation tool for fixed-point C Code

### 22.11.1. Tool Description

This tool is developed to help the estimation of program ROM of applications written using ITU-T Basic Operator libraries. This tool counts the number of calls to basic operators in a C source file, and also the number of calls to user defined functions. The sum of these two numbers gives an estimation of the required PROM for this C source file. Note that RAM and data ROM have to be estimated by other means.

The tool is meant to provide a consistent, platform independent method of obtaining and reporting program ROM estimates.

This tool works as follows:

– In a first step, a pre-processing removes C and C++ style comments to avoid counting functions that are commented out. The result is written in the file with extension `.c_pre`. Note that the pre-processor directives are not taken into account, so functions of the source code deactivated by a pre-processor directive are counted by the tool.
– In a second step, the pre-processed file is analysed and the number of basic operators (first group) and all other functions (second group) are counted. This second group of other functions is divided into two sub-groups as it is possible to define a so called "black list" that contains functions that should not be counted in the program ROM. Typically these functions are memory allocation functions (malloc, free,…), file manipulation functions

(fopen, fclose, fwrite, printf,…) or even functions related to complexity counting (setCounter, WMOPS_output, fwc,…). This list is already initialized and can be further edited and completed if needed. The other subgroup is called user defined functions as it consists of all functions which are neither basic operators nor blacklist functions.

At the end of the execution, the results are printed out on the standard output indicating the names of the C file and of the pre-processed file, the number of calls to basic operators, the number of calls to user defined functions, and the number of calls to blacklist functions.

When a result file name is also given as second optional argument of the command line, a summary of the results is written in this result file in append mode. Thus, to estimate the program ROM of a given application, the tool has to be run for all C files of the application using the same result file. Each call adds a new line in for the input C file. Though the format of the result file is a text file with tab separators, it is advised to name it with .xls extension and open it with Microsoft Excel.

The Excel file has as many rows as the number of C files. For each row, there are four columns: the first column contains the ANSI C input file name, the second the number of calls to basic operators, the third the number of calls to user defined functions and the fourth the number of calls to blacklist functions. In this way the program ROM of an application containing several C source files can be easily estimated as the sum of the numbers in the second and third columns.

By defining the pre-pocessor directives `VERBOSE_BASOP`, `VERBOSE_FUNC` or `VERBOSE_BLACKLIST_FUNC` the detailed list of called function for basic operator functions, user defined functions and blacklist functions respectively, is also printed out. This allows an easy cross-check of the results.

This tool is optimized for the basic operators of STL2005 or later. In case of estimating the program ROM of an application written using earlier STL basic operators, it is advised to define the pre-processor directive `OLD_STL`. In this case some C operators like if, do are not counted directly as they should be accompanied by the basic operator `test()`, and this later is counted by the tool.

### 22.11.2. Tool implementation

The tool consists of one ANSI C file `basop_cnt.c`.

To use this tool the C source file `basop_cnt.c` has to be compiled and linked. The resulted executable file has to be called for each C source file of the examined application, with the following syntax:

```
basop_cnt input.c [result_file_name.txt]
```

where *input.c* is an ANSI C file and *result_file_name.txt* the ouput result file.

If needed to further edit the black list of functions that should not be counted in the program ROM, modify `const char blacklist[][20] = {…}` in `basop_cnt.c` before compilation.

### 22.11.3. Example

This tool was first used to estimate the program ROM of the G.722 PLC candidate algorithms. The example below gives the execution of this tool with part of G.722 appendix IV[48].
- without a result file: `basop_cnt g722.c`
the output is :

```
Input file:                 `g722.c`
Output pre-processed file: g722.c_pre
  87  calls to STL basicops
  28  calls to user-defined functions
```

[48] this example does not give the total complexity of G.722 appendix IV

```
   8  calls to blacklist tokens
```

If the pre-processor directive `VERBOSE_BLACKLIST_FUNC` is defined, the list of the 8 blacklist functions found is also given:

```
line   86: blacklist #   1:                                    return
line  160: blacklist #   2:                                    calloc
line  160: blacklist #   3:                                    sizeof
line  230: blacklist #   4:                                      free
line  271: blacklist #   5:                                    return
line  303: blacklist #   6:                                    calloc
line  303: blacklist #   7:                                    sizeof
line  318: blacklist #  8:                                       free
```

Note that the line numbers correspond to the comment free pre-processed file.

- with a result file g722appIV_summary.xls:
  - `basop_cnt `g722.c` g722appIV_summary.xls`
  - `basop_cnt `funcg722.c` g722appIV_summary.xls`

After the first call, the result file contains:

| | | | |
|---|---|---|---|
| g722.c | 87 | 28 | 8 |

After the second call, the result file contains:

| | | | |
|---|---|---|---|
| g722.c | 87 | 28 | 8 |
| funcg722.c | 389 | 41 | 19 |

## 22.12. Complexity evaluation tool for floating-point C Code

### 22.12.1. Introduction

The Complexity evaluation tool for floating-point C Code enables to estimate the number of WMOPS (Weighted Million Operations per Second) and Program ROM of a floating-point implementation of speech and audio codecs. An estimation of the complexity that would be obtained after conversion of a floating-point source code into the corresponding fixed-point implementation is also computed.

### 22.12.2. Tool Description

This tool consists of complexity counters (macros) collected in an ANSI C library. The library is intended to be included in a codec algorithm, each line of which is instrumented with the complexity counters. The tool measures the computational complexity and program ROM based on a floating-point C source code instrumented with the counters. Note that RAM and table ROM have to be estimated by other means.

The tool is meant to provide a consistent, platform independent method of obtaining and reporting complexity estimates. The weights assigned with arithmetic operations reflect as much as possible those of the ITU-T Fixed-point Basic Operators. It should be noted, however, that the methodology cannot give an exact correspondence with the complexity of the fixed-point implementation and only estimation is given. Among several other reasons, this is because the scaling related operations (including saturation and overflow control) used in the fixed-point implementation have no correspondence in the floating-point implementation.

The essential feature of the tool is that the instructions need to be executed to be counted. Therefore the codec should be executed in conditions that give the broadest possible coverage of the source code (i.e. usually at the highest bitrate in frame-error conditions).

### 22.12.3. Complexity Verification Method

The computational complexity associated with a given speech and audio codec can be specified in terms of the number of instructions required per frame. The type and number of operations returned by the algorithm on a per-frame basis are specified for both the encoder and the decoder. Algorithms are broken down into sub-processing elements, each having a detailed breakdown of the types of operations and the number of operations required to complete the sub-processing element. Certain operations require several instructions in order to be computed. Thus all operations have associated with them a weight, indicative of this expansion, and given in Table 15 (column Complexity Weights).

The total number of instructions required per frame is then given by summing the total number of weighted operations. This number represents the basic computational complexity of the codec in instructions per frame. The complexity in WMOPS is then obtained by dividing the number of operations per frame by the length of the frame in seconds. The complexity estimates are computed assuming average and worst case number of operations both per frame and per second. The complexity is computed separately for the encoder and the decoder.

The operation count performed for the complexity measurement is re-used to get an estimate of the program memory. This operation count specifies operations in loops, loop counters, operations in subroutines, and subroutine counters. Each operation is weighted using the memory weights from the Table 15 (column Memory Weights) to produce a memory usage in words. Operations inside loops are counted only once. Similarly operations inside subroutines are counted only once.

### 22.12.4. Tool implementation

The tool consists of one header file *flc.h* and one ANSI C library file *flc.c*. Both files must be included into the project to use this tool. Further, the code must be instrumented using counters defined in the Table 15 (column Counter). Finally the following functions are needed:

- To initialize internal data structures, the function `FLC_init()` must be called before any counters are defined, usually at the beginning of the codec algorithm.
- The function `FLC_frame_update()` must be called at the end of the frame loop in order the FLC tool can keep track of the per-frame maxima to evaluate the worst-case conditions.
- The function `FLC_end()` computes and prints the complexity of the program and is called usually at the end of the codec algorithm.
- The separate complexity of subroutine/subsection is estimated by calling the function `FLC_sub_start(name_of_subroutine)` that must be matched with calling the function `FLC_sub_end()` at the end of the subsection/subroutine.

Note that once the code is instrumented using the complexity counters and functions, the compilation switch `DONT_COUNT` can be activated to suppress the functionality of the tool with no need of removing the complexity counters and functions from the code.

### 22.12.5. Scaling factor

A scaling factor is used to estimate and print the complexity that would be obtained after conversion of a floating-point source code into the corresponding fixed-point implementation. The scaling factor (`FLC_SCALEFAC`) is defined in file *flc.h* and its value is set to 1.1.

### 22.12.6. List of complexity measurement counters

**Table 15 — Floating-point Complexity Measurement Counters**

| Operation | Counter | Example | Complexity weights (cycles) | Memory weights (words) |
|---|---|---|---|---|
| Addition | ADD() | a=b+c | 1 | 1 |
| Multiplication | MULT() | a=b*c | 1 | 1 |
| Multiplication-Addition | MAC() | a+=b*c | 1 | 1 |
| Move | MOVE() | a=b, a[i]=b[i] ONLY assignment or copy | 1 | 1 |
| Storing Arithmetic Result in Array | STORE() | a[i]=b[i]+c[i] | 1 (for move only) | 0 |
| Logical | LOGIC() | AND, OR, etc. | 1 | 1 |
| Shift | SHIFT() | a=b>>c | 1 | 1 |
| Branch (tested with zero) | BRANCH() | if,<br>if…then…else…Count 1 BRANCH for each "if" possibility | 4 | 2 |
| Division | DIV() | a=b/c, a=b%c | 18 | 2 |
| Square-root | SQRT() | a=sqrt(b), a=isqrt(b), a=1/sqrt(b) | 10 | 2 |
| Transcendental | TRANS() | sine, log, arctan | 25 | 2 |
| Function call | FUNC() | a=func(b, c, d) | 2+i, where i=number of arguments passed & returned | 2 |
| Loop initialization | LOOP() | for (i=0; i<n; i++) | 3 | 1 |
| Indirect addressing | INDIRECT() | a=b.c,<br>a=b[c],<br>a=b[c][d],<br>a=**b,** + **a**=(b+c) | 2 | 2 |
| Pointer initialization | PTR_INIT() | a[i] | 1 (charged outside the loop) | 1 |
| Double Precision Addition | DADD() | a=b+c | 2 | 1 |
| Double Precision Multiplication | DMULT() | a=b*c | 2 | 1 |
| Double Precision Move | DMOVE() | a=b | 2 | 1 |
| Double Precision Division | DDIV() | a=b/c | 36 | 2 |
| Exponential | POWER() | pow, 1.0/x, exp(n) | 25 | 2 |
| Logarithm | LOG() | log2, log10, Ln | 25 | 2 |
| Extra conditional test | TEST() | used in conjunction with BRANCH | 2 | 1 |
| All other operations | MISC() | e.g. ABS | 1 | 1 |

### 22.12.7. Examples of instrumentation of the code

The rules to compute the complexity are general and when they are implemented, some choices must be done. Table 16 contains some examples to show where the counters should be placed.

**Table 16 — Usage of floating-point complexity measurement counters**

| Operation | Counter used | Explanation | Reference |
|---|---|---|---|
| if (a!=b \|\| c==d){…} | ADD(2);<br>BRANCH(1); TEST(1); | BRANCH for if, TEST for additional condition, ADD for two tests against non-zero value | |
| if (a!=b \|\| c==d){} else if( a==c ){} | ADD(3);<br>BRANCH(2);TEST(1); | | |
| b = a / L | MULT(1); | When L is constant; (1/L) is a constant too, So b = a * (1/L) | |
| *a = *b | MOVE(1); | Copy | Table 15, Move |
| for(i=0;i<L;i) + {x[i]= a[i]+b[i];} + for(i=c;i<L;i) {x[i]=a[i]+b[i];} | LOOP(1);(before)<br>ADD(1);<br>STORE(1);(inside)<br>PTR_INIT(3), LOOP(1); (before)<br>ADD(1);<br>STORE(1); (inside) | When a loop begins with an offset, initialization of pointer is counted | |
| &pt = &(a+L)<br>&pt = &(a+b) | PTR_INIT(1);<br>ADD(1); PTR_INIT(1); | if L is constant and b is variable, ADD is counted | |
| a=func(b) or a[i]=func(b) | FUNC(2); | The value returned by function is counted in FUNC(). MOVE or STORE are not counted in that case | |
| a = *b<br>for(i=0;i<L;i) + {a=*b;} a=*b; | INDIRECT(1);<br>MOVE(1) in the loop<br>MOVE(1) outside the loop | Use of MOVE outside the loop because the pointer is already initialized | Table 15, Pointer Initialization<br>Table 15, Move |
| pt_a += M | PTR_INIT(1); | M is constant, so this is equivalent to pt_a=&pt_a+M | |
| pt_a += m | ADD(1); PTR_INIT(1); | m is variable, so this is equivalent to pt_a=&pt_a+m | |
| *a=0.99*b | MULT(1); STORE(1); | A mathematical result is stored with a pointer | |
| for(i=0;i<L;i++) {a=*b+a;} | PTR_INIT(1); (before loop)<br>ADD(1); (inside) | If a pointer is initialized before the loop, no need to count INDIRECT(1) inside | |
| a[b][c]=x[y][z] | MOVE(1); | | Table 15, Move |
| switch(a){<br>case b:break; | ADD(2); BRANCH(2) before switch | Can be replaced by:<br>if (a==b) {…} | |

| Operation | Counter used | Explanation | Reference |
|---|---|---|---|
| case c: break;<br>default:break;<br>} | | else if (a==c) {…}<br>else{…} | |
| st→a[0]=t[2];<br>st→a[1]=t[3]; | INDIRECT(2);<br>MOVE(2) | Use INDIRECT(2) to remove double indirection and call MOVE(2) to copy data | |
| (*rnd_T0)++ | ADD(1); STORE(1); | it can be replaced by *rnd_T0=*rnd_T0+1; | |
| pit_shrp( code, PIT_SHARP, *round_T0, L_SUBFR); | FUNC(4);<br>INDIRECT(1); | *round_T0 is passed by indirection | |
| indice[0] = indirect_dico1[indice[0]]; | INDIRECT(2); | Double indirection | |
| sqr = indice[0] +indice[1] +indice[2] +indice[3] +indice[4]; | PTR_INIT(1); ADD(4); | Can be done in a loop | |

### 22.12.8. Tests and Portability

Compiled and tested on a PC (Windows XP) platform with MS Visual C++ 2005 and in Cygwin with gcc (version 3.4.4).

### 22.12.9. Example code

A demonstration program, *flc_example.c*, serves as an example and guideline for the illustration of the tool usage. To compile a demonstration program, Windows MSVC and Cygwin gcc makefiles are enclosed as *makefile.cl* and *makefile.unx*, respectively.

Below you can find the output screen when executing the demonstration program.

```
===== Call Graph and total ops per function =====
Function                      Calls              Ops         Ops/Call
-----------
ROOT                              1                0                0
-Autocorr                       100           975900             9759
--Set_Zero                      200             4100             20.5
-Lev_dur                        100            99600              996

===== Program Memory Usage by Function =====

Function            ADD  MULT   MAC  MOVE STORE LOGIC SHIFT BRNCH   DIV
-----------
Set_Zero              0     0     0     1     0     0     0     0     0
Autocorr              1     4    17     0    21     0     0     2     0
Lev_dur               0     2     5     5     3     0     1     1     2
ROOT                  0     0     0     0     0     0     0     0     0
-----------
totals                1     6    22     6    24     0     1     3     2

--------------------------------


Function           SQRT TRANC  FUNC  LOOP   IND   PTR  MISC
-----------
Set_Zero              0     0     0     1     0     1     0
Autocorr              0     0     2     2     1     6     0
```

```
Lev_dur                0      0      0      3      4      5      0
ROOT                   0      0      2      0      0      0      0
-----------
totals                 0      0      4      6      5     12      0


===== SUMMARY =====
Total Ops: 1.0796e+06
Total Program ROM usage: 83  (word)


===== Per Frame Summary =====
Number of Frames: 100
Average Ops/frame: 10796.00   Max Ops/frame: 10796

===== ESTIMATED COMPLEXITY (Frame length is 20.00 ms) =====
Maximum complexity: 0.539800 WMOPS
Average complexity: 0.539800 WMOPS

Estimated fixed point complexity with 1.1 scaling factor:
Maximum complexity: 0.593780 WMOPS
Average complexity: 0.593780 WMOPS
```

**Table 17 — Complexity weight history for each basic operator**

| Operator | Stl2000 | Stl2005 | Stl2009 | Stl2019 |
|---|---|---|---|---|
| add | 1 | 1 | 1 | 1 |
| sub | 1 | 1 | 1 | 1 |
| abs_s | 1 | 1 | 1 | 1 |
| shl | 1 | 1 | 1 | 1 |
| shr | 1 | 1 | 1 | 1 |
| extract_h | 1 | 1 | 1 | 1 |
| extract_l | 1 | 1 | 1 | 1 |
| mult | 1 | 1 | 1 | 1 |
| L_mult | 1 | 1 | 1 | 1 |
| negate | 1 | 1 | 1 | 1 |
| round / round_fx | 1 | 1 | 1 | 1 |
| L_mac | 1 | 1 | 1 | 1 |
| L_msu | 1 | 1 | 1 | 1 |
| L_macNs | 1 | 1 | 1 | 1 |
| L_msuNs | 1 | 1 | 1 | 1 |
| L_add | 2 | 1 | 1 | 1 |
| L_sub | 2 | 1 | 1 | 1 |
| L_add_c | 2 | 2 | 2 | 2 |
| L_sub_c | 2 | 2 | 2 | 2 |
| L_negate | 2 | 1 | 1 | 1 |
| L_shl | 2 | 1 | 1 | 1 |
| L_shr | 2 | 1 | 1 | 1 |
| mult_r | 2 | 1 | 1 | 1 |
| shr_r | 3 | 3 | 3 | 2 |
| mac_r | 2 | 1 | 1 | 1 |
| msu_r | 2 | 1 | 1 | 1 |
| L_deposit_h | 2 | 1 | 1 | 1 |
| L_deposit_l | 2 | 1 | 1 | 1 |
| L_shr_r | 3 | 3 | 3 | 2 |
| L_abs | 2 | 1 | 1 | 1 |

| Operator | Stl2000 | Stl2005 | Stl2009 | Stl2019 |
|---|---|---|---|---|
| L_sat | 4 | 4 | 4 | 1 |
| norm_s | 15 | 1 | 1 | 1 |
| div_s | 18 | 18 | 18 | 18 |
| norm_l | 30 | 1 | 1 | 1 |
| move16 | 1 | 1 | 1 | 1 |
| move32 | 2 | 2 | 2 | 1 |
| Logic16 | 1 | 1 | 1 | 1 |
| Logic32 | 2 | 2 | 2 | 1 |
| Test | 2 | 2 | 2 | 1 |
| s_max | - | 1 | 1 | 1 |
| s_min | - | 1 | 1 | 1 |
| L_max | - | 1 | 1 | 1 |
| L_min | - | 1 | 1 | 1 |
| L40_max | - | 1 | 1 | 1 |
| L40_min | - | 1 | 1 | 1 |
| shl_r / shift_r | 2 | 3 | 3 | 2 |
| L_shl_r / L_shift_r | 3 | 3 | 3 | 2 |
| L40_shr_r | - | 3 | 3 | 2 |
| L40_shl_r | - | 3 | 3 | 2 |
| norm_L40 | - | 1 | 1 | 1 |
| L40_shl | - | 1 | 1 | 1 |
| L40_shr | - | 1 | 1 | 1 |
| L40_negate | - | 1 | 1 | 1 |
| L40_add | - | 1 | 1 | 1 |
| L40_sub | - | 1 | 1 | 1 |
| L40_abs | - | 1 | 1 | 1 |
| L40_mult | - | 1 | 1 | 1 |
| L40_mac | - | 1 | 1 | 1 |
| mac_r40 | - | 2 | 2 | 2 |
| L40_msu | - | 1 | 1 | 1 |
| msu_r40 | - | 2 | 2 | 2 |
| Mpy_32_16_ss | - | 2 | 2 | 2 |
| Mpy_32_32_ss | - | 4 | 4 | 2 |
| L_mult0 | 1 | 1 | 1 | 1 |
| L_mac0 | 1 | 1 | 1 | 1 |
| L_msu0 | 1 | 1 | 1 | 1 |
| lshl | - | 1 | 1 | 1 |
| lshr | - | 1 | 1 | 1 |
| L_lshl | - | 1 | 1 | 1 |
| L_lshr | - | 1 | 1 | 1 |
| L40_lshl | - | 1 | 1 | 1 |
| L40_lshr | - | 1 | 1 | 1 |
| s_and | - | 1 | 1 | 1 |
| s_or | - | 1 | 1 | 1 |
| s_xor | - | 1 | 1 | 1 |
| L_and | - | 1 | 1 | 1 |
| L_or | - | 1 | 1 | 1 |
| L_xor | - | 1 | 1 | 1 |
| rotl | - | 3 | 3 | 3 |
| rotr | - | 3 | 3 | 3 |
| L_rotl | - | 3 | 3 | 3 |
| L_rotr | - | 3 | 3 | 3 |
| L40_set | - | 3 | 3 | 1 |
| L40_deposit_h | - | 1 | 1 | 1 |
| L40_deposit_l | - | 1 | 1 | 1 |

| Operator | Stl2000 | Stl2005 | Stl2009 | Stl2019 |
|---|---|---|---|---|
| L40_deposit32 | - | 1 | 1 | 1 |
| Extract40_H | - | 1 | 1 | 1 |
| Extract40_L | - | 1 | 1 | 1 |
| L_Extract40 | - | 1 | 1 | 1 |
| L40_round | - | 1 | 1 | 1 |
| L_saturate40 | - | 1 | 1 | 1 |
| round40 | - | 1 | 1 | 1 |
| IF | - | 4 | 4 | 3 |
| GOTO | - | 4 | 4 | 2 |
| BREAK | - | 4 | 4 | 2 |
| SWITCH | - | 8 | 8 | 6 |
| FOR | - | 3 | 3 | 3 |
| WHILE | - | 4 | 4 | 3 |
| CONTINUE | - | 4 | 4 | 2 |
| L_mls | 6 | 5 | 5 | 1 |
| div_l | 32 | 32 | 32 | 32 |
| i_mult | 1 | 3 | 3 | 1 |
| CL_shr | - | - | - | 1 |
| CL_shl | - | - | - | 1 |
| CL_add | - | - | - | 1 |
| CL_sub | - | - | - | 1 |
| CL_scale | - | - | - | 1 |
| CL_dscale | - | - | - | 1 |
| CL_msu_j | - | - | - | 1 |
| CL_mac_j | - | - | - | 1 |
| CL_move | - | - | - | 1 |
| CL_Extract_real | - | - | - | 1 |
| CL_Extract_imag | - | - | - | 1 |
| CL_form | - | - | - | 1 |
| CL_multr_32x16 | - | - | - | 2 |
| CL_negate | - | - | - | 1 |
| CL_conjugate | - | - | - | 1 |
| CL_mul_j | - | - | - | 1 |
| CL_swap_real_imag | - | - | - | 1 |
| C_add | - | - | - | 1 |
| C_sub | - | - | - | 1 |
| C_mul_j | - | - | - | 1 |
| C_multr | - | - | - | 2 |
| C_form | - | - | - | 1 |
| CL_scale_32 | - | - | - | 1 |
| CL_dscale_32 | - | - | - | 1 |
| CL_multr_32x32 | - | - | - | 2 |
| C_mac_r | - | - | - | 2 |
| C_msu_r | - | - | - | 2 |
| CL_round32_16 | - | - | - | 1 |
| C_Extract_real | - | - | - | 1 |
| C_Extract_imag | - | - | - | 1 |
| C_scale | - | - | - | 1 |
| C_negate | - | - | - | 1 |
| C_conjugate | - | - | - | 1 |
| C_shr | - | - | - | 1 |
| C_shl | - | - | - | 1 |
| move64 | - | - | - | 1 |
| W_add_nosat | - | - | - | 1 |
| W_sub_nosat | - | - | - | 1 |

| Operator | Stl2000 | Stl2005 | Stl2009 | Stl2019 |
|---|---|---|---|---|
| W_shl | - | - | - | 1 |
| W_shr | - | - | - | 1 |
| W_shl_nosat | - | - | - | 1 |
| W_shr_nosat | - | - | - | 1 |
| W_mac_32_16 | - | - | - | 1 |
| W_msu_32_16 | - | - | - | 1 |
| W_mult_32_16 | - | - | - | 1 |
| W_mult0_16_16 | - | - | - | 1 |
| W_mac0_16_16 | - | - | - | 1 |
| W_msu0_16_16 | - | - | - | 1 |
| W_mult_16_16 | - | - | - | 1 |
| W_mac_16_16 | - | - | - | 1 |
| W_msu_16_16 | - | - | - | 1 |
| W_shl_sat_l | - | - | - | 1 |
| W_sat_l | - | - | - | 1 |
| W_sat_m | - | - | - | 1 |
| W_deposit32_l | - | - | - | 1 |
| W_deposit32_h | - | - | - | 1 |
| W_extract_l | - | - | - | 1 |
| W_extract_h | - | - | - | 1 |
| W_round48_L | - | - | - | 1 |
| W_round32_s | - | - | - | 1 |
| W_norm | - | - | - | 1 |
| W_add | - | - | - | 1 |
| W_sub | - | - | - | 1 |
| W_neg | - | - | - | 1 |
| W_abs | - | - | - | 1 |
| W_mult_32_32 | - | - | - | 1 |
| W_mult0_32_32 | - | - | - | 1 |
| W_lshl | - | - | - | 1 |
| W_lshr | - | - | - | 1 |
| W_round64_L | - | - | - | 1 |
| Mpy_32_16_l | - | - | - | 1 |
| Mpy_32_16_r | - | - | - | 1 |
| Mpy_32_32 | - | - | - | 1 |
| Mpy_32_32_r | - | - | - | 1 |
| Madd_32_16 | - | - | - | 1 |
| Madd_32_16_r | - | - | - | 1 |
| Msub_32_16 | - | - | - | 1 |
| Msub_32_16_r | - | - | - | 1 |
| Madd_32_32 | - | - | - | 1 |
| Madd_32_32_r | - | - | - | 1 |
| Msub_32_32 | - | - | - | 1 |
| Msub_32_32_r | - | - | - | 1 |
| UL_addNs | - | - | - | 1 |
| UL_subNs | - | - | - | 1 |
| UL_Mpy_32_32 | - | - | - | 1 |
| Mpy_32_32_uu | - | - | - | 2 |
| Mpy_32_16_uu | - | - | - | 2 |
| norm_ul | - | - | - | 1 |
| UL_deposit_l | - | - | - | 1 |
| LT_16 | - | - | - | 1 |
| GT_16 | - | - | - | 1 |
| LE_16 | - | - | - | 1 |
| GE_16 | - | - | - | 1 |

| Operator | Stl2000 | Stl2005 | Stl2009 | Stl2019 |
|---|---|---|---|---|
| EQ_16 | - | - | - | 1 |
| NE_16 | - | - | - | 1 |
| LT_32 | - | - | - | 1 |
| GT_32 | - | - | - | 1 |
| LE_32 | - | - | - | 1 |
| GE_32 | - | - | - | 1 |
| EQ_32 | - | - | - | 1 |
| NE_32 | - | - | - | 1 |
| LT_64 | - | - | - | 1 |
| GT_64 | - | - | - | 1 |
| LE_64 | - | - | - | 1 |
| GE_64 | - | - | - | 1 |
| EQ_64 | - | - | - | 1 |
| NE_64 | - | - | - | 1 |

## 23. UTILITIES: UGST utilities

This module does not relate to any ITU-T Recommendation, but implements several general-purpose routines, that are needed when using other STL modules.

In the process of implementing the STL modules, it was found that the interfacing between data representations (`float` and `short`; `serial` and `parallel`) could present problems. Hence, algorithms implementation these functions have been made available in the ITU-T STL. Additionally, a scaling routine for application of gain and loss to speech samples is included.

### 23.1. Some definitions

Some functions in this module convert between a serial format and a parallel format. The *parallel format* is defined to be a representation in which all the bits in a computer word have an information content, as in a multi-level representation of data. Speech samples in a computer file are a typical example of a parallel representation. A *serial format* is defined as the representation of the data where each computer word refer to a single bit of information. An example would be the sequence of bits sent in a communication channel refering to an encoded digital signal. A *serial bitstream*, in the context of the ITU-T STL, refers to a multi-level representation of information bits in which each of the "hard" bits `'0'` or `'1'` are mapped respectively to the so-called softbits `0x007F` and `0x0081`, to which an error probability is associated. These softbits are stored in 16-bit right-justified words. In addition, if the bitstream is compliant to the bitstream signal representation in Annex B of Recommendation ITU-T G.192, the serial bitstream "payload" described above will be preceed by a synchronization header. A *synchronization header* is composed by a synchronization word followed by a frame length word. *Synchronization words* are words in the bitstream in the range `0x6B21` to `0x6B2F`. A synchronization word equal to `0x6B20` indicates a frame loss. The *frame length word* is a two-complement word representing the number of softbits in the payload. Therefore, the frame length word does not account for the synchronization header length (which equals two, by definition). Typically (as in the EID module), encoded signals are represented using the bitstreams with a synchronization header, while error patterns are represented without a synchronization header.

### 23.2. Implementation

The functions implemented in this module are:

`scale` for level change of a float data stream;

`sh2fl*` for conversion from `short` to `float`;

`fl2sh` for conversion from `float` to `short`;

`serialize_*` for conversion from parallel to serial data representation;

`parallelize_*` for conversion from serial to parallel data representation;

Following you find a summary of calls to these functions.

### 23.2.1. `scale`

**Syntax:**

```
#include "ugst-utl.h"
long scale (float _*buffer_,long _smpno_,double _factor_);
```

**Prototype:** `ugst-utl.h`

**Description:**

Gain/loss insertion algorithm that scales the input buffer data by a given factor. If the factor is greater than 1.0, it means a gain; if less than 1.0, a loss. The basic algorithm is:

$$y(k) = x(k) \cdot \text{factor}$$

Please note that:
- the scaled data is put into the same location of the original data, in order to save memory space, thus overwriting original samples;
- input data buffer is an array of `floats`;
- scaling precision is single (rather than double precision).

**Variables:**

| | |
|---|---|
| *buffer* | Float data vector to be scaled. |
| *smpno* | Number of samples in *buffer*. |
| *factor* | The `float` scaling factor. |

**Return value:**

Return the number of scaled samples.

### 23.2.2. `sh2fl`

**Syntax:**

```
#include "ugst-utl.h"
void sh2fl (long _n_, short _*ix_, float _*y_, long _resolution_, char _norm_);
```

**Prototype:** `ugst-utl.h`

**Description:**

Common conversion routine. The conversion routine expects the fixed point data to be in the range between --32768..32767. Conversion to float is done by taking into account only the most significant bits (i.e., input samples shall be left-justified), normalizing afterwards to the range --1..+1, if *norm* is 1.

In order to maintain a match with its complementary routine `fl2sh`, a set of macros have been defined for resolutions in the range of 16 to 12 bits (see below for the complementary definitions):
- *sh2fl_16bit*: conversion from 16 bit to float
- *sh2fl_15bit*: conversion from 15 bit to float
- *sh2fl_14bit*: conversion from 14 bit to float
- *sh2fl_13bit*: conversion from 13 bit to float
- *sh2fl_12bit*: conversion from 12 bit to float

**Variables:**

| | |
|---|---|
| *n* | Is the number of samples in ix[ ]; |
| *ix* | Is input short array pointer; |
| *y* | Is output float array pointer; |
| *resolution* | Is the resolution (number of bits) desired for the input data in the floating-point representation. |
| *norm* | Flag for normalization:<br>*1*: normalize float data to the range --1..+1;<br>*0*: convert from short to float, leaving data in the range: -32768>>(16 —resolution) .. 32767>>(16—resolution), where >> is the right-shift operation. |

**Return value:**

None.

### 23.2.3. `sh2fl_alt`

**Syntax:**

```
#include "ugst-utl.h"
void sh2fl_alt (long _n_, short _*ix_, float _*y_, short _mask_);
```

**Prototype:** `ugst-utl.h`

**Description:**

Common conversion routine alternative to routine `sh2fl`. This conversion routine expects the fixed-point data to be in the range -32768..32767. Conversion to float is done by taking into account only the most significant bits, indicated by *mask*. Conversion to float results necessarily in normalised values in the range $-1.0 \leq y < +1.0$.

**Variables:**

| | |
|---|---|
| *n* | Number of samples in ix[ ]. |
| *ix* | Pointer to input short array. |
| *y* | Pointer to output float array. |
| *mask* | Mask determining how many bits of the input samples are to be considered for convertion to float. Bits `'1'` in *mask* indicate that this bit in particular will be used in the conversion. For example, *mask* equal to `0xFFFF` indicates that all 16 bits of the word are used in the convertion, while *mask* equal `0xFFFE`, `0xFFFC`, `0xFFF8`, or `0xFFF0` will force respectively only the upper 15, 14, 13, or 12 most significant bits to be used. |

**Return value:**

None.

### 23.2.4. `fl2sh`

**Syntax:**

```
#include "ugst-utl.h"
long fl2sh (long _n_, float _*x_, short _*iy_, double _half_lsb_, unsigned
            _mask_);
```

**Prototype:** `ugst-utl.h`

**Description:**

Common quantisation routine. The conversion routine expects the floating-point data to be in the range between --1..+1, values outside this range are limited. Quantization is done by taking into account only the most significant bits. Therefore, the quantized (or converted) data are located left justified within the 16-bit word, and the results are in the range:

– -32768, …, -1, 0, +1, …, +32767, if quantized to 16 bit
– -32768, …, -2, +2, …, +32766, if quantized to 15 bit
– -32768, …, -4, +4, …, +32763, if quantized to 14 bit
– -32768, …, -8, +8 …, +32760, if quantized to 13 bit
– -32768, …, -16,+16, …, +32752, if quantized to 12 bit

The operation may be summarized as:

$$y_k = (x_k \pm h) \& m$$

where $x_k$ is the float number, $y_k$ is the quantized number, $h$ is the value of half-LSb for the resolution desired (which is added to $x_k$ if the latter is positive or zero, or subtracted otherwise), and $m$ is the bit mask (to assure that the bits below the LSb are `0`). The operation $=$ is a truncation, and *&* is a bit-wise AND operation. The appropriate values for $h$ are determined by:

$$h = 0.5 \cdot 2^{16-B} = 2^{15-B}$$

where $B$ is the desired resolution in bits. As an example, if data is to be stored with 15 bits of resolution (equivalent to -16384..+16383, in right-justified notation), the rounding number $h$ is 1.0, because the smallest number in the output buffer can be +1 or -1. The mask $m$, by its turn, is

$$m = 0xFFFF << (16 - B)$$

where $<<$ is the left-shift bit operation with zero-padding from the right. For the same example, $m$ is `0xFFFE`, i.e., only bit 0 of the samples is zeroed.

To facilitate to the use of the `fl2sh`, a set of macros has been defined for quantizations in the range of 16 to 12 bits (see `ugst-utl.h`):

– *fl2sh_16bit*: conversion from float to 16 bit
– *fl2sh_15bit*: conversion from float to 15 bit
– *fl2sh_14bit*: conversion from float to 14 bit
– *fl2sh_13bit*: conversion from float to 13 bit
– *fl2sh_12bit*: conversion from float to 12 bit

In some cases truncated data is needed, what can be accomplished by setting $h = 0$. For example, at the input for A-law encoding, truncation is necessary, not rounding. On the other hand within recursive filters rounding is essential. Hence, this routine serves both cases.

Concerning the location of the fixed-point data within one 16 bit word, it is more practical to have the decimal point immediately after the sign bit (between bit 15 and 14, if the bits are ordered from 0..15). Since this is well defined, software that processes the quantized data needs no knowledge about the resolution of the data. It is not important whether tha data comes from A or $\mu$ law decoding routines or from 12-bit (13, 14, 16-bit) A/D converters.

It should be noted that this routine only processes data in a normalized form ($-1.0 \leq x < +1.0$); it shall not be used if data is in the `short` range (-32768.0 .. 32767.0).

**Variables:**

| | |
|---|---|
| *n* | Number of samples in x[ ]. |
| *x* | Pointer to input `float` array. |

*iy* is output `short` array pointer.

*half_lsb* A `double` representation of half LSb for the desired resolution (quantization).

*mask* The `unsigned` masking of the lower (right) bits.

**Return value:**

Returns the number of overflows that happened in the quantization process.

### 23.2.5. `serialize_*_justified`

**Syntax:**

```
#include "ugst-utl.h"
long serialize_right_justified (short _*par_buf_, short _*bit_stm_, long _n_,
                                long _resol_, char _sync_);
long serialize_left_justified (short _*par_buf_, short _*bit_stm_, long _n_,
                               long _resol_, char _sync_);
```

**Prototype:** `ugst-utl.h`

**Description:**

Routines `serialize_right_justified` and `serialize_left_justified` convert a frame of *n* right- or left-justified samples with a resolution *resol* into a right-justified, serial soft bitstream of length *n.resol*. If the parameter *sync* is set, a serial bitstream compliant to the Annex B of Recommendation ITU-T G.192 will be generated. In this case, the the length of the bitstream is increased to *(n+2).resol*.[49] It should be noted that the least significant bits of the input words are serialized first, such that the bitstream is a stream with less significant bits coming first.

The only difference between these functions is that function `serialize_right_justified` serializes right-justified parallel data and function `serialize_left_justified` serialize left-adjusted data.

It is supposed that all parallel samples have a constant number of bits, or resolution, for the whole frame. If this does not happen, the bitstream cannot be serialized by these functions. As an example, this is the case of the RPE-LTP bitstream: the 260 bits of the encoded bitstream are not divided equally among the 76 parameters of the bitstream. In cases like this, users must write their own serialization function.

**Variables:**

*par_buf* Input buffer with right- or left-adjusted, parallel samples to be serialized.

*bit_stm* Output buffer with serial bitstream. It should be noted that *bit_stm* must point to an appropriately allocated memory block, which should be a block of *n. resol* `short`s if *sync* is `0`, or a block of *(n+2).resol* `short`s otherwise.

*n* Number of words in the input buffer, i.e., the number of parallel samples/ frame.

*resol* Resolution (number of bits) of the samples in *par_buf*.

*sync* If 1, a synchronization header is to be used (appended) at the boundaries of each frame of the bitstream. If 0, a synchronization header is not used.

**Return value:**

[49] The option of adding only the synchronization word, as implemented in the STL92, is no longer available with this function since the STL96.

This function returns the total number of softbits in the output bitstream, including the synchronization word and frame length. If the value returned is 0, the number of converted samples is zero.

### 23.2.6. `parallelize_*_justified`

**Syntax:**

```
#include "ugst-utl.h"
long parallelize_right_justified (short _*bit_stm_, short _*par_buf_, long _bs_
len_,
                                  long _resol_, char _sync_);
long parallelize_left_justified (short _*bit_stm_, short _*par_buf_, long _bs_
len_,
                                 long _resol_, char _sync_);
```

**Prototype:** `ugst-utl.h`

**Description:**

Functions `parallelize_right_justified` and `parallelize_left_justified` convert the samples in input buffer *bit_stm* from the ITU-T softbit representation to its parallel representation, given a number of bits per sample, or *resolution*. The input serial bitstream of length *bs_len* is converted into a frame with *bs_len/resol* samples (if *sync==0*) or *(bs_len—2)/resol* samples (if *sync!=0*), with a resolution *resol*. It should be noted that softbits in lower positions in the input buffer are supposed to represent less significant bits of the parallel word (considering bits that would compose the same parallel word). In other words, the softbits that come first are less significant than the next ones, when referring to the same parallel word (as defined by the parameter *resol*). Therefore, when generating a word from the bitstream, bits from the bitstream that comes first are converted to lower significant bits. Frames with the synchronization flag but without the frame length cause the function to exit with an error code equal to *--bs_len*.

The difference between both functions is that `parallelize_right_justified` converts the serial bitstream to a parallel data in a right-justified format, i.e., data is aligned to the right, while the routine `parallelize_left_justified` parallelizes samples with left-justification.

If the G.192 Annex B bitstream format is used (parameter *sync==1*), a synchronization header is present at frame boundaries in the input buffer. In this case, the synchronization and frame length words are not copied from the bitstream to the output buffer.

Note that all parallel samples are supposed to have a constant number of bits, or resolution, for the whole frame. This means that, by construction, the number of softbits divided by the resolution must be an integer number, or *(bs_len—2)%resol==0*. If this does not happen, probably the serial bitstream was not generated by one of the `serialize_…()` routines, and cannot be parallelized by these functions. An example is the case of the RPE-LTP bitstream: the 260 bits of the encoded bitstream are not divided equally among the 76 parameters of the bitstream. In cases like this, users must write their own parallelization function.

If an erased frame is found, the function returns without performing any action.

**Variables:**

| | |
|---|---|
| *bit_stm* | Input buffer with bitstream to be parallelized. |
| *par_buf* | Output buffer with right- or left-adjusted samples. |
| *bs_len* | Number of bits per frame (i.e., size of input buffer, which includes the synchronization header length if *sync==1*). |
| *resol* | Resolution (number of bits per parallel sample) of the right- or left-adjusted samples in *par_buf*. |

*sync* If 1, a synchronization header is expected in the boundaries of each frame of input the bitstream. If 0, synchronization headers are not expected.

**Return value:**

On success, this function returns the number of samples of the output parallel sample buffer.

## 23.3. Portability and compliance

Since these tools do not refer to ITU-T Recommendations, no special compliance tests are needed. As for portability, it may be checked by running the same speech file on a proven platform and on a test platform. Files processed this way should match exactly. A preferred data file would be the ramp described in the compliance test description.

The routines in this module had portability tested for VAX/VMS with VAX-C and GNU C (gcc) and for MS-DOS with a number of Borland C/C compilers (Turbo C v2.0, Turbo-C v1.0, Borland C++ v3.1). Portability was also tested in a number of Unix workstations and compilers: Sun workstation with Sun-OS and Sun-C (cc), acc, and gcc; HP workstation with HP-UX and gcc.

## 23.4. Example code

### 23.4.1. Description of the demonstration programs

Two programs are provided as demonstration programs for the UTL module, `scaldemo.c` (version 1.3) and `spdemo.c` (version 3.2).

Program `scaldemo.c` scales a 16-bit, linear PCM input file by a user-specified linear or dB gain value. Default resolution is 16 bits per sample, and rounding is used by default when converting from float to short. When resolutions different from 16 bits are used with rounding, versions 1.2 and earlier of the program might not produce the "expected" results. The program used to limit the resolution of the samples (by masking the *16-resolution* least significant bits) when converting from short to float. Additional rounding is applied after scaling when converting from float to short. If the desired operation is, actually, scale and then reduce the resolution with rounding, masking before the scaling operation should be disabled. In version 1.3 and later, the default behavior is **not** to apply such mask, (same as the option *-nopremask*) for backward compatible behavior, the option *-premask* should be explicitly used.

Program `spdemo.c` converts files between serial and parallel formats using a user-specified resolution and frame (or block) size. A known issue with spdemonstration version 3.2 is that the command-line option -frame does not work properly for parallel-to-serial conversion. In this case, the desired frame size has to be specified as parameter *N* in the command line.

### 23.4.2. The master header file for the STL demonstration programs

The module also contains the common demonstration program definition file `ugstdemo.h` (version 2.2), which is used by all STL demonstration programs. This header file contains the definition of a number of pseudo-functions and symbols that facilitate the use of a more homogeneous user interface for the different demonstration programs in the STL.

The available pseudo-functions include:

`GET_PAR_*` Pseudo-functions for printing a user prompt and reading a positional parameter from the command line. The parameters can be char (C), integers (I), long integers (L), unsigned long integers (LU), floats (F), doubles (D), and strings (S).

`FIND_PAR_*` Pseudo-functions for printing a user prompt and reading a positional parameter from the command line if it was specified by the user, or to

assume a default value defined by the programer. The parameters can be char (\C), integers (I), long integers (L), floats (F), doubles (D), and strings (S).

`ARGS()` The pseudo-function `ARGS()` allows that the list of parameters that show up as its arguments be passed on to ANSI-C compliant compilers, or be discarded for old-vintage, K&R-style compilers that do not accept parameter list in function prototypes. This pseudo-function allows for safer function prototypes in compilers that support parameter declaration in function prototypes and avoids the need to edit function declarations (or long `#if/#else/#end` for prototype sections) for non-ANSI C compilers.

Some of the symbols defined in `ugstdemo.h` include:

– Symbols `WB`, `RB`, `WT`, `RT`, and `RWT` for file open (`fopen()`) operation. These symbols are portable across a large number of platforms and permit write-binary, read-binary, write-text, read-text, and read-write-text file mode operations.
– Symbol `MSDOS`, which is necessary for proper compilation of some of the programs under the MS-DOS environment. The symbol is defined in case MS-DOS is detected, and undefined in case MS-DOS is not detected.
– Symbol `COMPILER`, which contains a text string describing the compiler used to generate an executable.

### 23.4.3. Short and float conversion and scaling routines

The following C code exemplifies the use of the short and float number format interchange routines, as well as of the gain scaling routine. This program is a simplified version of the example program `scaldemo.c` provided in the STL distribution. This program reads 16-bit, 2-complement, left-justified input samples, converts them to a float representation in the range of -1..+1, applies a gain (or loss) factor to these samples, converts the scaled samples back to an integer representation (16 bit, 2's complement, left-justified) using rounding and hard-clip of the floating-point numbers. The number of most significant bits to be used is also specified by the user.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "ugst-utl.h"

#define LENGTH 5

main(argc, argv)
  int             argc;
  char           *argv[];
{
  long            i, NrSat;
  long            B, round;
  double          h;
  unsigned        m;
  short           ix[LENGTH];
  float           y[LENGTH];
  float           factor;


  GET_PAR_F(1, "_Factor: ..... ", factor);
  GET_PAR_L(2, "_Resolution: ..... ", B);
```

```
  GET_PAR_L(3, "_Round(1=yes,0=no): ... ", round);

  /* Initialize short's buffer, BUT left-ajusted!  */
  for (i = 0; i < LENGTH; i++)
    ix[i] = i << (16 - B);

  /* Choose rounding number */
  h = 0.5 * (round << (16 - B));

  /* Find mask */
  m = 0xFFFF << (16 - B);

  /* Print original data */
  printf("ix before normalization\n");
  printf("=======================\n");
  for (i = 0; i < LENGTH; i++)
    printf("ix[%3d]=%5d\n", i, ix[i]);

  /* Convert samples to float, normalizing */
  sh2fl(LENGTH, ix, y, B, 1);

  /* Normalizes vector */
  scale(y, LENGTH, (double) factor);

  /* Convert from float to short */
  NrSat = fl2sh(LENGTH, y, ix, h, m);

  /* Inform about overflows */
  if (NrSat != 0)
    printf("\n  Number of clippings: .......... %ld [] ", NrSat);


  /* Print new data */
  printf("after normalization ... \n");
  printf("=======================\n");
  for (i = 0; i < LENGTH; i++)
    printf("y[%3d]= %e -> ix[%3d]=%5d\n", i, y[i], i, ix[i]);

  return (0);
}
```

### 23.4.4. Serialization and parallelization routines

The following C code implements an example of use of the serialization and parallelization routines available in the STL. Input data is generated within the program. The program takes the number of bits per sample, the justification, and whether synchronization headers should be generated. The input data is printed on the screen in its parallel representation, which is then converted to the serial format and back to the parallel format. Then, the serialized version of the data is printed on the screen, and the program ends.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "ugstdemo.h"
#include "ugst-utl.h"

#define LENGTH 5

void main(argc, argv)
```

```
  int             argc;
  char           *argv[];
{
  long            i, j, k, smpno, bitno, init;
  long            B, just;
  double          h;
  unsigned        m;
  char            c;
  short           par[LENGTH];
  short           ser[16 * LENGTH + 2];
  char            sync;
  long            (*ser_f) ();  /* pointer to serialization function */
  long            (*par_f) ();  /* pointer to parallelization function */


  GET_PAR_L(1, "_Resolution: ............................... ", B);
  GET_PAR_L(2, "_Data is Right (1) or Left (0) justified? ... ", just);
  GET_PAR_L(3, "_Use sync header? .......................... ", sync);

  /* Initialize flag "OFF" */
  init = 0;
  c = sync ? 1 : 0;
  smpno = LENGTH;
  bitno = LENGTH * B + sync ? 2 : 0;

  /* Initialize data and choose pointers to appropriate functions */
  if (just)
  {                             /* Right-justified data */
    ser_f = serialize_right_justified;
    par_f = parallelize_right_justified;
    for (i = 0; i < LENGTH; i++)
      par[i] = i;
  }
  else
  {                             /* Left-justified data */
    ser_f = serialize_left_justified;
    par_f = parallelize_left_justified;
    for (i = 0; i < LENGTH; i++)
      par[i] = i << (16 - B);
  }

  /* Print original data */
  printf("\npar[] before serialization\n");
  printf("==========================\n");
  for (i = 0; i < LENGTH; i++)
    printf("par[%3d]=%5d\n", i, par[i]);

  bitno = ser_f
    (par,                       /* input buffer pointer */
     ser,                       /* output buffer pointer */
     smpno,                     /* no. of samples (not bits) per frame */
     B,                         /* number of bits per sample */
     sync);                     /* whether sync header is present or not */

  smpno = par_f
    (ser,                       /* input buffer pointer */
     par,                       /* output buffer pointer */
     bitno,                     /* number of softbits per frame */
     B,                         /* number of bits per sample */
     sync);                     /* whether sync header is present or not */
```

```
  /* Print new data */
  printf("=======================\n");
  printf("| 0x81 represents a `1'| \n| 0x7F represents a `0'|\n");
  printf("=======================\n");
  printf("after serialization ... \n");
  printf("=======================\n");
  if (sync)
  {
    printf("Sync word is ser[%d]= %04X", 0, ser[0]);
    printf("Frame length is ser[%d]= %04X", 1, ser[1]);
  }

  for (k = 2, i = 0; i < LENGTH; i++)
  {
    printf("\npar[%3d]=%5d\n", i, par[i]);
    for (j = 0; j < B; j++, k++)
      printf("ser[%3d]= %04X\t", sync ? k : k - 2, ser[k]);
  }
  printf("\n");
}
```

# Appendix I

## Unsupported tools

(This appendix does not form an integral part of this Recommendation.)

This Appendix to the ITU-T Software Tool Library (STL) Manual describes the unsupported tools provided in the ITU-T STL. The tools are named as "unsupported" because they do not follow the initial modularity concept of STL. These tools are provided "as is" and without any warranties or implied suitability to use. However, any feedback on problems with these tools will be welcome and accomodated as possible, as will any improvements made which can be shared and incorporated in the STL.

### I.1. Source code

| | |
|---|---|
| *asc2bin.c* | converts decimal or hex ASCII data into short/long/float or double binary numbers. Input data must be one number per line. |
| *astrip.c* | strips off a segment of a file. Can operate on block or sample-based parameters and can apply windowing to theborders of the extracted segment. Tested in Unix/ MSDOS. |
| *bin2asc.c* | converts short/long/float or double binary numbers into octal, decimal or hex ASCII numbers, printing one perline. For Unix/MSDOS. |
| *compfile.c* | compare word-wise binary files. For VMS/Unix/MSDOS. |
| *dumpfile.c* | dump a binary file. For VMS/Unix/MSDOS. |
| *chr2sh.c* | convert `char`-oriented files to `short`-oriented (16-bit words) files by padding the upper byte of each word of the output file with zeros. For Unix/MSDOS. |
| *endian.c* | verify whether the current platform is big or little endian (i.e. high-byte first or low-byte first). For Unix/MSDOS. |
| *fdelay.c* | flexibly introduce delay into a file. Delay can be specified in value and length, or can be taken from a user-specified file. For Unix/MSDOS. |
| *g728-vt* | a directory with software tools for use with the G. 728 floating-point verification package. Not all tools are functional; preserved here for future reference. |
| *getcrc32.c* | 32-bit CRC calculation function and program (depending on how it is compiled). Uses the same polynomial as ZIP. Checked for portability across a number of platforms. Makefile compiles it into an executable called `crc`. For Unix/MSDOS. |
| *measure.c* | measure statistics/CRC for a bunch of files. For VMS/ Unix/MSDOS. |

| | |
|---|---|
| *oper.c* | implement arithmetic operation on two files: add, subtract, multiply or divide two files applying scaling factors (linear or dB), and adding a DC level. For Unix/MSDOS. |
| *sb.c* | swap bytes for word-oriented files. For VMS/Unix/MSDOS. |
| *sh2chr.c* | convert `short`-oriented (16-bit words) files to `char`-oriented files by ignoring the upper byte of each word of the input file. For Unix/MSDOS. |
| *sine.c* | generate a sinewave file for a given speco of AC / DC / phase / frequency / sampling frequency values. For VMS/Unix/MSDOS. |
| *signal-diff.c* | subtract/add files (depending on the compilation, see makefiles). For VMS/Unix/MSDOS. |
| *g729e_convert_synch.c* | convert G.192 word oriented analysis frames without SYNCH_WORD into a G.192 word oriented file with SYNCH_WORD and zeroed payload bits. |

## I.2. Test files

*unsup/test_data* — Test files for testing some of the unsupported tools:

```
cf:
  3200    cftest1.dat
  3200    cftest2.dat
  3200    cftest3.dat
   186    delay-15.ref
   186    delay-a.ref
   214    delay-u.ref
   186    delaydft.ref
   200    delayfil.ref

sb:
   100    bigend.src
   100    litend.src
```

# Bibliography


[1] ITU-T G.111, *Loudness ratings (LRs) in an international connection*

[2] ITU-T G.192, *A common digital parallel interface for speech standardization activities*

[3] ITU-T G.712, *Transmission performance characteristics of pulse code modulation channels*

[4] ITU-T G.722, *7 kHz audio-coding within 64 kbit/s*

[5] ITU-T G.725, *System aspects for the use of the 7 kHz audio codec within 64 kbit/s*

[6] ITU-T G.726 Annex A, *Extensions of Recommendation G.726 for use with uniform-quantized input and output*

[7] ITU-T G.727 Annex A, *Extensions of Recommendation G.727 for use with uniform-quantized input and output*

[8] ITU-T G.728 Annex G, *16 kbit/s fixed point specification*

[9] ITU-T G.728 Annex I, *Frame or packet loss concealment for the LD-CELP decoder*

[10] ITU-T P.11, *Effect of transmission impairments*

[11] ITU-T P.341, *Transmission characteristics for wideband digital loudspeaking and hands-free telephony terminals*

[12] ITU-T P.48, *Specification for an intermediate reference system*

[13] ITU-T P.50, *Artificial voices*

[14] ITU-T P.56, *Objective measurement of active speech level*

[15] ITU-T P.800, *Methods for subjective determination of transmission quality*

[16] ITU-T P.810, *Modulated noise reference unit (MNRU)*

[17] ITU-T P.830, *Subjective performance assessment of telephone-band and wideband digital codecs*

[18] IETF RFC 3551, *RTP Profile for Audio and Video Conferences with Minimal Control*

[19] ITU G.191, ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.191, Software Tools for Speech and Audio Coding Standards*. ITU, Geneva, Switzerland, March 1993.

[20] ITU G.711, ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.711, Pulse code modulation (PCM) of voice frequencies, volume Fascicle III.4 of Blue Book,* pages 175–184. ITU, Geneva, Switzerland, 1989.

[21] ITU G.726, ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.726, 40, 32, 24, 16 kbit/s adaptive differential pulse code modulation (ADPCM)*. ITU, Geneva, Switzerland, December 1990.

[22] ITU G.727, ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.727, 5-, 4-, 3- and 2-bits/sample embedded adaptive differential pulse code modulation (ADPCM)*. ITU, Geneva, Switzerland, December 1990.

[23] ITU G.728, CCITT. Recommendation ITU-T G.728, *Coding of Speech at 16 kbit/s using Low Delay Code Excited Linear Prediction (LD-CELP)*. ITU, Geneva, Switzerland, June 2012.

[24] ITU-R BS.1534-1, ITU-R. Recommendation ITU-R BS.1534-1: *Method for the subjective assessment of intermediate quality level of coding systems.* ITU, January 2003.

[25] ITU-T P.56, ITU Telecommunication Standardization Sector. *Recommendation ITU-T P.56, Objective measurement of active speech level,* volume V of *Blue Book*, pages 110–120. ITU, Geneva, Switzerland, 1989.

[26] ITU Telecommunication Standardization Sector. Comparison of ADPCM Algorithms. In *Recommendation G.726*, chapter Appendix III. ITU, Geneva, Switzerland, November 1994.

[27] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A Fortran–to–C Converter. Technical Report Computing Science 149, AT&T Bell Laboratories, August 1990.

[28] Study Group XV. Q.21/XV- Requirements and Objectives. report COM XV-R 73-E, CCITT, July 1990.

[29] V. Iyengar and P. Kabal. A Low Delay 16 kbit/sec Speech Coder. In *Proc. ICASSP,* 1988.

[30] AT&T. Description of 16 kbit/s Low-Delay Code-Excited Linear Predictive Coding (LDCELP) algorithm. contribution AH.89-D02, CCITT, March 1989.

[31] BNR-Canada. Letter to the Chairman of CCITT Ad-hoc Group for Q.21/XV. contribution AH.89-D13, CCITT, March 1989.

[32] Consortium for Speech Coding. Letter to the Chairman of CCITT Ad-hoc Group for Q.21/XV. contribution AH.89-D13, CCITT, March 1989.

[33] Consortium for Speech Coding. A High Level Description of the Consortium's Low Delay Vector Excitation Coder (LD-VCX). contribution AH.89-D08, CCITT, March 1989.

[34] Consortium for Speech Coding. Description of the Consortium's Low Delay Vector Excitation Coder (LD-VCX), Version 2. contribution AH.89-D21, CCITT, July 1989.

[35] V. Cuperman, A. Gersho, R. Pettigrew, J. J. Shynk, and J.-H. Yao. Backward Adaptive Configurations for Low-delay Vector Excitation Coding. In *Advances in Speech Coding.* Kluwer Academic Publishers, 1989.

[36] C. R. South and P. Usai. Subjective Performance of CCITT's 16 kbit/s LD-CELP algorithm with voice signals. In *Proc. IEEE Globecom,* 1992.

[37] M. R. Schroeder and B. S. Atal. Code-Excited Linear Prediction (CELP): high speech quality at very low rates. In *Proc. ICASSP,* 1985.

[38] R. V. Cox and F. T. Johansen. Test Verification of LD-CELP: an objective measurement approach to a non-bit exact standard. In *Proc. IEEE Globecom,* 1992.

[39] L. R. Rabiner and R. W. Schafer. *Digital Processing of Speech Signals.* Prentice-Hall, Englewood Cliffs, 1978.

[40] Study Group XV. Report of Working Party XV/2. report COM XV-R 73-E, CCITT, November 1991.

[41] J.-H. Chen and Y.-C. Lin and R. V. Cox. A fixed-point 16 kb/s LD-CELP algorithm. In *Proc. ICASSP,* 1991.

[42] T. P. Barnwell III. Recursive windowing for generating autocorrelation coefficients for LPC analysis. *Transactions on Acoust., Speech, Signal Processing,* ASSP-29(5), October 1981.

[43] J.-H. Chen, R. V. Cox, Y.-C. Lin, N. Jayant, and M. J. Melchner. A Low-Delay CELP Coder for the CCITT 16 kb/s Speech Coding Standard. *IEEE Journal.on Selected Areas in Communications,* 10(5), June 1992.

[44] AT&T. Improvements to AT&T's 16 kbit/s Low-Delay Code-Excited Linear Predictive Coding (LD-CELP) algorithm. contribution, CCITT, July 1989. AH.89-D26.

[45] J.-H. Chen. A robust Low-Delay CELP Speech Coder at 16 kb/s. In *Advances in Speech Coding*. Kluwer Academic Publishers, 1989.

[46] J.-H. Yao and A. Gersho. Real-time vector APC speech coding at 4800 bps with adaptive post-filtering. In *Proc. ICASSP,* 1987.

[47] D. Grant and M. Young and A. Gersho. Real Time Vector Excitation Coding of Speech at 4800 bps. In *Proc. ICASSP,* 1987.

[48] S. F. Campos Neto and H. Irii and J. Rosenberger and J. Sotcsheck and P. Usai. Effect of Tandeming and Input Level. technical report 2–2, CSELT, 1993.

[49] J.-H. Chen and R. V. Cox. LD-CELP: A high quality 16 kb/s speech coder with low delay. In *Proc. IEEE Globecom,* 1990.

[50] J.-H. Chen. A robust Low-Delay CELP Speech Coder at 16 kb/s. In *Proc. IEEE Globecom,* 1989.

[51] J. Fennick. *Quality Measures and the design of telecommunications systems* . Artech House, 1988.

[52] V. Ramamoorthy and N. S. Jayant and R. V. Cox and M. M. Sondhi. Enhancement of ADPCM Speech Coding using Backward-Adaptive Algorithms for Post-Filtering and Noise Feedback. *Journal on Selected Areas in Communications,* 6(2), February 1988.

[53] AT&T. Detailed description of AT&T's LD-CELP algorithm. contribution AH.89-D02, CCITT, November 1989.

[54] P. Mermelstein. G.722, a new CCITT coding standard for digital transmission of wideband audio signals. *IEEE Communications Magazine,* 26(1):8–15, January 1988.

[55] M. Taka, et. al. Overview of the 64 kbit/s (7 kHz) audio coding standard. In *Proc. IEEE Globecom*, pages 593–598, Houston, Texas, December 1986.

[56] G. Modena, A. Coleman, P. Usai, and P. Coverdale. Subjective Performance Evaluation of the 7 KHz Audio Coder. In *Proc. IEEE Globecom*, pages 599–604, Houston, Texas, December 1986.

[57] G. Le Tourneur, et. al. Implementation of the 7 kHz Audio Codec and its Transmission Characteristics. In *Proc. IEEE Globecom*, pages 605–609, 1986.

[58] M. Dietrich, et. al. Initialization and Mode Switching of 7 kHz Audio Terminals. In *Proc. IEEE Globecom*, pages 610–614, 1986.

[59] K. R. Harrison, et. al. Possible Applications for networking considerations relating to the 64 kbit (7 kHz) audio coding system. In *Proc. IEEE Globecom*, pages 615–622, 1986.

[60] X. Maitre. 7 kHz audio coding within 64 kbit/s. *IEEE Journal on Selected Areas in Communications,* 6(2):283–298, February 1988.

[61] ETSI. *ETSI TS 102 527-1 V1.2.1 (2008-06) Digital Enhanced Cordless Telecommunications (DECT); New Generation DECT; Part 1: Wideband speech.* ETSI, France, June 2008.

[62] ETSI. *ETSI TS 102 527-1 V1.1.1 (2008-06) Digital Enhanced Cordless Telecommunications (DECT); New Generation DECT; Part 3: Extended wideband speech services.* ETSI, France, June 2008.

[63] ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.722/Appendix III, A high quality packet loss concealment algorithm for G.722.* ITU, Geneva, Switzerland, 2006.

[64] ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.722/Appendix IV, A low-complexity algorithm for packet loss concealment with G.722.* ITU, Geneva, Switzerland, 2006.

[65] P. Kroon, R. J. Sluyter, and E. F. Deprettere. Regular-pulse excitation: a novel approach to effective and efficient multipulse coding of speech. *IEEE Trans. Acoust., Speech, Signal Processing,* ASSP-34(5):1054–1063, October 1986.

[66] P. Vary et al. Speech codec for the European Mobile Radio System. In *Proc. ICASSP,* pages 227–230, 1988.

[67] Ulrich Reute (Guest Editor). Special Issue on Medium Rate Speech Coding for Digital Mobile Technology. *Speech Communication,* 7(2), July 1988.

[68] GSM-06.10. *Full Rate Speech Transcoding.* ETSI, France, October 1992. Released July 1, 1993.

[69] N. S. Jayant and P. Noll. *Digital Coding of Waveforms.* Prentice-Hall, 1984.

[70] C. South and P. Usai. Subjective Performance of CCITT's 16 kbit/s LD-CELP Algorithm with Voice Signals. In *Proc. IEEE Globecom,* 1992.

[71] H. J. Braun, S. Feldes, and G. Schröder. Preselection for the Half-Rate GSM Standard. In *IEEE Workshop on Speech Coding for Telecommunications,* pages 90–92, September 11-13 1991.

[72] *ITU-T Q.10/16 Rapporteur. ITU-T AC-05-16 Processing Test Plan of the 14kHz Low-Complexity Audio Coding Algorithm at 24, 32 and 48 kbps Extension to ITU-T G.722.1.* ITU, Strasbourg, France, April 2005.

[73] Bell Northern Research (Canada). Frequency response characteristics for low bit-rate codec testing. Technical report, UIT-T SG 12, Geneva, Switzerland, December 1994. Delayed Contribution D,38 (SG12).

[74] S. Dimolitsas, F. Corcoran, and C. Ravishankar. Correlation between headphone and telephone-handset listener opinion scores for single-stimulus voice coder performance assessments. *IEEE Signal Processing Letters,* 2(3):41–43, March 1995.

[75] S. Dimolitsas, F. Corcoran, and C. Ravishankar. Dependence of opinion scores on listening sets used in degradation category rating assessments. *IEEE Transactions on Speech and Audio Processing,* 3(5):421–424, September 1995.

[76] D. Byrne, et. al. An international comparison of long-term average speech spectra. *Journal of the Acoutical Society of America,* 96(4):2108–2120, October 1994.

[77] Aachen University. An Implementation of the Signal Conditioning Device. Technical Report TD91/23, ETSI/TM/TM5/TCH-HS, April 1991.

[78] V. K. Varma. Testing speech coders for usage in wireless communications systems. In *Second IEEE Workshop on Speech Coding for Telecommunications, "Speech Coding for the Network of the Future",* Quebec, Canada, October 13–15 1993.

[79] Bellcore. Proposed model for simulating radio channel burst errors. Technical report, CCITT SG XII, Geneva, Switzerland, October 1992. Doc.SQ-15.92(Rev.).

[80] E. N. Gilbert. Capacity of a burst-noise channel. *Bell Syst. Tech. J.,* pages 1253–1265, 1960.

[81] D. Knuth. Seminumerical Algorithms. *In The Art of Computer Programming.* Addison-Wesley, Massachusetts, 1981.

[82] ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.711/Appendix I, A high quality low-complexity algorithm for packet loss concealment with G.711*. ITU, Geneva, Switzerland, September 1999.

[83] H. B. Law and R. A. Seymour. A reference distortion system using modulated noise. *Proc. Institution of Electrical Engineers (IEE),* 109B:484–485, Nov 1962.

[84] ITU Telecommunication Standardization Sector. *Recommendation ITU-T P.81, Modulated Noise Reference Unit (MNRU)* , volume V of Blue Book, pages 198–203. ITU, Geneva, Switzerland, 1989.

[85] User's Group on Software Tools. CCITT Software Tool Library Manual. report COM XV-R 87-E, CCITT SG XV, May 1992.

[86] S. F. Campos Neto. Characterization of the revised implementation of the Modulated Noise Reference Unit (MNRU) for the ITU-T Software Tool Library. White Contribution COM 15-182-E, ITU Telecommunication Standardization Sector, 1993-1996.

[87] W. H. Press, B. P. Flannery, S. A. Teukolky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing* . Cambridge University Press, Cambridge, 1990.

[88] ITU Telecommunication Standardization Sector. *Handbook on Telephonometry.* ITU, Geneva, 1992. 2nd. Edition.

[89] M. Bonnet, O. Macchi, and M. Jaidane-Saidane. Theoretical analysis of the ADPCM CCITT algorithm. *IEEE Trans.on Communications,* 38(6):847–858, June 1990.

[90] H. Kuttruff. *Room acoustics.* Elsevier, 1991.

[91] France Telecom. Proposed update of ITU-T reverberation software tool. Input document ITU-T Q.10/16 Rapporteur meeting AC-0809-Q10-26, Geneva, Switzerland, September 2008.

[92] ITU Telecommunication Standardization Sector. *Recommendation ITU-T G.191, - Software Tool Library 2005 User's Manual. ITU,* Geneva, Switzerland, July 2005.

[93] ITU-T Q.10/16 Rapporteur. Proposed update of ITU-T frequency response measurement tool. input document Q.10/16 Rapporteur meeting AC-0801-Q10-31, ITU-T, Geneva, Switzerland, January 2008.

[94] ITU-T Q.10/16 Rapporteur. Report of Q10/16 Rapporteur's meeting report (Geneva, 28 January-1 February 2008). report TD297R1 (WP3/16), Geneva, Switzerland, February 2008.

[95] Huawei Technologies. Update of the frequency response measurement Software Tool. contribution ITU-T Q.10/16 Rapporteur meeting, AC-0809-Q10-35, Geneva, Switzerland, 2008.

[96] Study Group 12. Gen-431-16: Reply ls on wideband/fullband test signals and babble noise (com 16-ls 242). input document ITU-T Q.10/16 Rapporteur meeting, AC-0801-Q10-04, Geneva, Switzerland, 2008.

[97] W. R. Daumer, X. Maitre, P. Mermelstein, and I. Tokizawa. Overview of the ADPCM coding algorithm. Proc. *IEEE Globecom,* pages 774–777, 1984.