ITUPublications

International Telecommunication Union
Telecommunication Standardization Bureau

Recommendation ITU-T G.191

(01/2019)

SERIES G: Transmission Systems and Media, Digital Systems and Networks

International telephone connections and circuits — Software tools for transmission systems

---

Software tools for speech and audio coding standardization

Recommendation ITU-T G.191

Software tools for speech and audio coding standardization

Summary

Recommendation ITU-T G.191 provides source code for speech and audio processing modules for narrowband, wideband and super-wideband telephony applications. The set includes codecs, filters and noise generators.

This edition introduces changes to Annex A, which describes the ITU-T software tool library (STL) containing a high-quality, portable C code library for speech-processing applications. This release of the STL, also known as STL2019, incorporates new basic operators to accommodate state-of-the-art processor architectures that support wide accumulators, single instruction multiple data (SIMD) and very long instruction word (VLIW). Thus, the new operators provide support for 64-bit accumulator, complex numbers, enhanced 32-bit operations and additional control code operators.

The software package was reworked to make it available as a truly open-source project and is therefore hosted on an open-source collaboration platform. The build toolchain now uses CMake to generate platform-dependent and tool-dependent build scripts, as well as to execute regression tests for each module in the STL.

Recommendation ITU-T G.191 includes an electronic attachment containing STL2019 and manual.

History

Edition	Recommendation	Approval	Study Group	Unique ID[a)]
1.0	ITU-T G.191	1993-03-12	XV	11.1002/1000/798
2.0	ITU-T G.191	1996-11-11	15	11.1002/1000/3812
3.0	ITU-T G.191	2000-11-17	16	11.1002/1000/5275
4.0	ITU-T G.191	2005-09-13	16	11.1002/1000/8581
5.0	ITU-T G.191	2010-03-29	16	11.1002/1000/10651
6.0	ITU-T G.191	2019-01-13	12	11.1002/1000/13830

a) To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

Keywords

DSP operators, filters, G.711, G.722, G.726, G.728, MNRU, open source, reverb, STL2019, sv56.

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 not 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

Software tools for speech and audio coding standardization

1. Scope

This Recommendation[1] provides a set of common, coherent and portable signal processing tools to facilitate the development of speech and audio coding algorithms, in particular within the standardization environment, where the following situations often happen:

– experimental results generated with different software tools may not be directly compared;
– software tools used by different organizations may not perfectly conform to related ITU T Recommendations, which may delay ITU-T standardization processes;
– ITU-T Recommendations may leave scope for different implementations;
– new speech and audio coding standards are increasing in complexity, leading to non bitexact specifications; furthermore, appropriate testing procedures to assure interoperability of different implementations are needed.

2. References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T G.192 (03/1996)] Recommendation ITU-T G.192 (03/1996), *A common digital parallel interface for speech standardization activities*, First edition.

[ITU-T G.711 (11/1988)] Recommendation ITU-T G.711 (11/1988), *Pulse code modulation (PCM) of voice frequencies*, Fifth edition.

[ITU-T G.712 (09/1992)] Recommendation ITU-T G.712 (09/1992), *Transmission performance characteristics of pulse code modulation channels*, Sixth edition.

[ITU-T G.718 (06/2008)] Recommendation ITU-T G.718 (06/2008), *Frame error robust narrow-band and wideband embedded variable bit-rate coding of speech and audio from 8-32 kbit/s*, First edition.

[ITU-T G.722 (09/2012)] Recommendation ITU-T G.722 (09/2012), *7 kHz audio-coding within 64 kbit/s*, Third edition.

[ITU-T G.726 (12/1990)] Recommendation ITU-T G.726 (12/1990), *40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM)*, Fourth edition.

[ITU-T G.727 (12/1990)] Recommendation ITU-T G.727 (12/1990), *5-, 4-, 3- and 2-bit/sample embedded adaptive differential pulse code modulation (ADPCM)*, First edition.

[1] This Recommendation includes an electronic attachment containing STL2019 and manual.

[ITU-T G.728 (06/2012)] Recommendation ITU-T G.728 (06/2012), *Coding of speech at 16 kbit/s using low-delay code excited linear prediction*, Second edition.

[ITU-T G.729.1 (05/2006)] Recommendation ITU-T G.729.1 (05/2006), *G.729-based embedded variable bit-rate coder: An 8-32 kbit/s scalable wideband coder bitstream interoperable with G.729*, First edition.

[ITU-T O.41 (03/1993)] Recommendation ITU-T O.41 (03/1993), *Psophometer for use on telephone-type circuits*, Fifth edition.

[ITU-T P.341 (03/2011)] Recommendation ITU-T P.341 (03/2011), *Transmission characteristics for wideband digital loudspeaking and hands-free telephony terminals*, Fourth edition.

[ITU-T P.48 (11/1988)] Recommendation ITU-T P.48 (11/1988), *Specification for an intermediate reference system*, Fourth edition.

[ITU-T P.56 (12/2011)] Recommendation ITU-T P.56 (12/2011), *Objective measurement of active speech level*, Fifth edition.

[ITU-T P.810 (03/2023)] Recommendation ITU-T P.810 (03/2023), *Modulated noise reference unit (MNRU)*, Fifth edition.

3. Definitions

None.

4. Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FIR-IRS	Finite Impulse Response-Intermediate Reference System
IIR	Infinite Impulse Response
PCM	Pulse Code Modulation
ROM	Read Only Memory
RPE-LTP	Regular Pulse Excitation-Long Term Prediction
STL	Software Tool Library
SIMD	Single Instruction Multiple Data
VLIW	Very Long Instruction Word

5. Conventions

None.

6. Software tools

To clarify the use of the set of software tools arranged as a software tool library (STL), ITU-T makes the following recommendations:

a) The software tools specified in Annex A should be used as building modules of signal processing blocks to be used in the process of generation of ITU-T Recommendations, particularly those concerned with speech and audio coding algorithms.
b) Some of the tools shall be used in procedures for the verification of interoperability of ITU T standards, mainly of speech and audio coding algorithms whose description is in terms of non-bitexact specifications.
c) The use of these modules should be made strictly in accordance with the technical instructions of their attached documentation, and should respect the following terms.

The software tools are maintained on an open-source collaboration platform [b-STLgit]. The build toolchain is implemented using the CMake framework [b-CMake] to generate build scripts crafted for the target platform and to execute regression tests for each module in the STL.

7. License and copyright

The modules in the ITU-T STL are free software; they can be redistributed or modified under the terms of Annex B; this applies to any of the versions of the modules in the STL.

The STL has been carefully tested and it is believed that both the modules and the example programs on their usage conform to their description documents. Nevertheless, the ITU-T STL is provided "as is", in the hope that it will be useful, but without any warranty.

The STL is intended to help the scientific community to achieve new standards in telecommunications more efficiently, and for such must not be sold, entirely or in parts. The original developers, except where otherwise noted, retain ownership of their copyright, and allow their use under the terms and conditions of Annex B.

Annex A

List of software tools available

(This annex forms an integral part of this Recommendation.)

This annex contains a list with a short description of the software tools available in the ITU-T Software Tool Library (STL). The 2019 release is referred to in the associated documentation as STL2019. All the routines in the modules are written in C.

A.1. Example programs available

Associated header file: `ugstdemo.h`

The following programs are examples of the use of the modules.

`g711demo.c`	on the use of the ITU T G.711 module.
`g726demo.c`	on the use of the ITU T G.726 module.
`g727demo.c`	on the use of the ITU T G.727 module
`g722demo.c`	on the use of the ITU T G.722 module.
`g728enc.c`	on the use of the ITU T G.728 floating-point encoder.
`g728dec.c`	on the use of the ITU T G.728 floating-point decoder.
`g728fpenc.c`	on the use of the ITU T G.728 fixed-point encoder.
`g728fpdec.c`	on the use of the ITU T G.728 fixed-point decoder.
`rpedemo.c`	on the use of the full-rate GSM 06.10 speech codec module.
`sv56demo.c`	on the use of the speech voltmeter module, and also the gain/loss routine.
`eiddemo.c`	on the use of the error insertion device for bit error insertion and frame erasure.
`eid-ev.c`	on the use of the error insertion device for bit error insertion for layered bitstreams, which can be used to apply errors to individual layers in layered bitstreams, such as those specified in [ITU-T G.718 (06/2008)] or [ITU-T G.729.1 (05/2006)].
`gen-patt.c`	on the use of generating bit error pattern files for error insertion in serial bitstream encoded files that comply with [ITU-T G.192 (03/1996)].
`gen_rate_profile.c`	on the use of the fast switching rate profile generation tool.
`firdemo.c`	on the use of the finite impulse response (FIR) high-quality low-pass and band-pass filters and of the finite impulse response-intermediate reference system (FIR-IRS) filters, associated with the rate change module.
`pcmdemo.c`	on the use of the ITU T G.712 [standard pulse code modulation (PCM)] infinite impulse response (IIR) filters, associated with the rate change module.
`filter.c`	on the use of both the IIR and the FIR filters available in the rate change module.

`mnrudemo.c`	on the use of the narrow-band and wideband modulated noise reference unity (ITU T P.810) module.
`spdemo.c`	on the use of the serialization and parallelization routines of the utility module.
`g711iplc.c`	on the use of the packet loss concealment module of Appendix I of [ITU-T G.711 (11/1988)].
`reverb.c`	on the use of the reverberation module.
`truncate.c`	on the use of the bitstream truncation module.
`freqresp.c`	on the use of the frequency response computation tool.
`stereoop.c`	on the use of stereo file operations.

NOTE – The module for the basic operators does not have a demo program, but it is supplemented by two tools: one to evaluate program read only memory (ROM) complexity for fixed-point code (`basop_cnt.c`), and another to evaluate complexity (including program ROM) of floating-point implementations (`flc_example.c`). Both reside in the basic operator module.

A.2. Rate change module with finite impulse response routines

Name: `firflt.c`
Associated header file: `firflt.h`
The functions included are as follows.

`delta_sm_16khz_init`	initialize 16 kHz 1:1 Δ_{SM} weighting filter.
`hq_down_2_to_1_init`	initialize 2:1 low-pass down-sampling filter.
`hq_down_3_to_1_init`	initialize 3:1 low-pass down-sampling filter.
`hq_up_1_to_2_init`	initialize 1:2 low-pass up-sampling filter.
`hq_up_1_to_3_init`	initialize 1:3 low-pass up-sampling filter.
`irs_8khz_init`	initialize 8-kHz ITU-T P.48 IRS weighting filter.
`irs_16khz_init`	initialize 16-kHz ITU-T P.48 IRS weighting filter.
`linear_phase_pb_2_to_1_init`	initialize 2:1 bandpass down-sampling filter.
`linear_phase_pb_1_to_2_init`	initialize 1:2 bandpass up-sampling filter.
`linear_phase_pb_1_to_1_init`	initialize 1:1 bandpass filter.
`mod_irs_16khz_init`	initialize 16-kHz send-side modified IRS weighting filter.
`mod_irs_48khz_init`	initialize 48-kHz send-side modified IRS weighting filter.
`psophometric_8khz_init`	initialize 1:1 ITU T O.41 psophometric weighting filter.
`p341_16khz_init`	initialize 1:1 ITU T P.341 send-part weighting filter for data sampled at 16 kHz.

`rx_mod_irs_16khz_init`	initialize 16-kHz modified IRS receive-side weighting filter.
`rx_mod_irs_8khz_init`	initialize 8-kHz modified IRS receive-side weighting filter.
`tia_irs_8khz_init`	initialize 8-kHz IRS weighting filter using the TIA coefficients.
`ht_irs_16khz_init`	initialize 16-kHz IRS weighting filter with a half-tilt inclination within the ITU T P.48 mask.
`msin_16khz_init`	initialize mobile station weighting filter.
`bp5k_16khz_init`	initialize 50-Hz to 5-kHz-bandpass filter (16 kHz sampling).
`bp100_5k_16khz_init`	initialize a 100-Hz to 5-kHz-bandpass filter (16-kHz sampling).
`bp14k_32khz_init`	initialize a 50-Hz to 14-kHz-bandpass filter (32-kHz sampling).
`bp20k_48khz_init`	initialize a 20-Hz to 20-kHz-bandpass filter (48-kHz sampling).
`LP1p5_48kHz_init`	initialize a low-pass filter with a cut-off frequency of 1.5 kHz (48-kHz sampling).
`LP35_48kHz_init`	initialize a low-pass filter with a cut-off frequency of 3.5 kHz (48-kHz sampling).
`LP7_48kHz_init`	initialize a low-pass filter with a cut-off frequency of 7 kHz (48-kHz sampling).
`LP10_48kHz_init`	initialize a low-pass filter with a cut-off frequency of 10 kHz (48-kHz sampling).
`LP12_48kHz_init`	initialize a low-pass filter with a cut-off frequency of 12 kHz at (48-kHz sampling).
`LP14_48kHz_init`	initialize a low-pass filter with a cut-off frequency of 14 kHz at 48-kHz sampling).
`LP20_48kHz_init`	initialize a low-pass filter with a cut-off frequency of 20 kHz (48-kHz sampling).
`hq_kernel`	FIR filtering function.
`hq_reset`	clear state variables.
`hq_free`	deallocate FIR-filter memory.

A.3. Rate change module with infinite impulse response routines

Name: `iirflt.c`
Associated header file: `iirflt.h`
The functions included are as follows.

`stdpcm_kernel`	parallel-form IIR kernel filtering routine.
`stdpcm_16khz_init`	initialization of a parallel-form IIR standard PCM filter for input and output data at 16 kHz.
`stdpcm_1_to_2_init`	as "stdpcm_16khz_init( )", but needs input with sampling frequency of 8 kHz and returns data at 16 kHz.

`stdpcm_2_to_1_init`	as "stdpcm_16khz_init( )", but needs input with sampling frequency of 16 kHz and returns data at 8 kHz.
`stdpcm_reset`	clear state variables (needed only if another signal should be processed with the same filter) for a parallel-form structure.
`stdpcm_free`	deallocate filter memory for a parallel-form state variable structure.
`cascade_iir_kernel`	cascade-form IIR filtering routine.
`iir_G712_8khz`	initialization of a cascade-form IIR standard PCM filter for data sampled at 8 kHz.
`iir_irs_8khz_init`	initialization of a cascade-form IIR ITU-T P.48 IRS filter for data sampled at 8 kHz.
`iir_casc_1p_3_to_1_init`	initialization of a cascade-form IIR low-pass filter for asynchronization filtering of data and downsampling by a factor of 3:1.
`iir_casc_1p_1_to_3_init`	initialization of a cascade-form IIR low-pass filter for asynchronization filtering of data and upsampling by a factor of 3:1.
`cascade_iir_reset`	clear state variables (needed only if another signal should be processed with the same filter) for a cascade-form structure.
`cascade_iir_free`	deallocate filter memory for a cascade-form state variable structure.
`direct_iir_kernel`	direct-form IIR filtering routine.
`iir_dir_dc_removal_init`	Initialize a direct-form IIR filter structure for a 1:1 DC removal filtering.
`direct_reset`	clear state variables (needed only if another signal should be processed with the same filter) for a direct-form structure.
`direct_iir_free`	deallocate filter memory for a direct-form state variable structure.

A.4. Error insertion module

Name: `eid.c`
Associated header file: `eid.h`
The functions included are as follows.

`1_eid`	initializes the error pattern generator (for single-bit errors, burst bit-errors or single frame erasures).
`open_burst_eid`	initializes the burst frame erasure pattern generator.
`reset_burst_eid`	reinitializes the burst frame erasure pattern generator.
`BER_generator`	generates a bit error sequence with properties defined by "open_eid".
`FER_generator_random`	generates a random frame erasure sequence with properties, defined by "open_eid".
`FER_generator_burst`	generates a burst frame erasure sequence with properties, defined by "open_burst_eid".
`BER_insertion`	modifies the input data bits according to the error pattern, stored in a buffer.
`FER_module`	frame erasure module.

`close_eid`	frees memory allocated to the EID state variable buffer.

A.5. ITU-T G.711 module

Name: `g711.c`
Associated header file: `g711.h`
The functions included are as follows.

`alaw_compress`	compands one vector of linear PCM samples to A-law; uses 13 most significant bits (MSBs) from input and 8 least significant bits (LSBs) on output.
`alaw_expand`	expands one vector of A-law samples to linear PCM; uses 8 LSBs from input and 13 MSBs on output.
`ulaw_compress`	compands one vector of linear PCM samples to μ-law; uses 14 MSBs from input and 8 LSBs on output.
`ulaw_expand`	expands one vector of μ-law samples to linear PCM; uses 8 LSBs from input and 14 MSBs on output.

A.6. Packet loss concealment module of Appendix I of [ITU-T G.711]

Name: `lowcfe.c`
Associated header file: `lowcfe.h`
The functions included are as follows.

`g711plc_construct`	LowcFE Constructor.
`g711plc_dofe`	generate the synthetic signal.
`g711plc_addtohistory`	a good frame was received and decoded, add the frame to history buffer.

A.7. ITU-T G.726 module

Name: `g726.c`
Associated header file: `g726.h`
The functions included are as follows.

`G726_encode`	ITU T G.726 encoder at 40, 32, 24 and 16 kbit/s.
`G726_decode`	ITU T G.726 decoder at 40, 32, 24 and 16 kbit/s.

A.8. Modulated noise reference unit module

Name: `mnru.c`
Associated header file: `mnru.h`
The functions included are as follows.

`MNRU_process`	module for addition of modulated noise to a vector of samples, according to [ITU-T P.810 (03/2023)], for both the narrow- and wideband models.

A.9. Speech voltmeter module

Name: `sv-p56.c`
Associated header file: `sv-p56.h`
The functions included are as follows.

`init_speech_voltmeter`	initializes a speech voltmeter state variable.
`speech_voltmeter`	measurement of the active speech level of data in a buffer according to [ITU-T P.56 (12/2011)].

A.10. Module with Users' Group on Software Tools utilities

Name: `ugst-utl.c`
Associated header file: `ugst-utl.h`
The functions included are as follows.

`scale`	gain/loss insertion algorithm.
`sh2fl_16bit`	conversion of two's complement, 16-bit integer to floating point.
`sh2fl_15bit`	conversion of two's complement, 15-bit integer to floating point.
`sh2fl_14bit`	conversion of two's complement, 14-bit integer to floating point.
`sh2fl_13bit`	conversion of two's complement, 13-bit integer to floating point.
`sh2fl_12bit`	conversion of two's complement, 12-bit integer to floating point.
`sh2fl`	generic function for conversion from integer to floating point.
`sh2fl_alt`	alternate (faster) implementation of sh2fl, with compulsory range conversion.
`fl2sh_16bit`	conversion of floating point data to two's complement, 16-bit integer.
`fl2sh_15bit`	conversion of floating point data to two's complement, 15-bit integer.
`fl2sh_14bit`	conversion of floating point data to two's complement, 14-bit integer.
`fl2sh_13bit`	conversion of floating point data to two's complement, 13-bit integer.
`fl2sh_12bit`	conversion of floating point data to two's complement, 12-bit integer.
`fl2sh`	generic function for conversion from floating point to integer.
`serialize_left_justified`	serialization for left-justified data.
`serialize_right_justified`	serialization for right-justified data.
`parallelize_left_justified`	parallelization for left-justified data.
`parallelize_right_justified`	parallelization for right-justified data.

A.11. ITU-T G.722 module

Name: `g722.c`
Associated header file: `g722.h`
The functions included are as follows.

`G722_encode`	ITU T G.722 wideband speech encoder at 64 kbit/s.
`G722_decode`	ITU T G.722 wideband speech decoder at 64, 56 and 48 kbit/s.
`g722_reset_encoder`	initialization of the ITU T G.722 encoder state variable.

`g722_reset_decoder` initialization of the ITU T G.722 decoder state variable.

A.12. RPE-LTP module

Name: `rpeltp.c`
Associated header file: `rpeltp.h`
The functions included are as follows.

`rpeltp_encode` GSM 06.10 full-rate regular pulse excitation-long term prediction (RPE-LTP) speech encoder at 13 kbit/s.

`rpeltp_decode` GSM 06.10 full-rate RPE-LTP speech decoder at 13 kbit/s.

`rpeltp_init` initialize memory for the RPE-LTP state variables.

`rpeltp_delete` release memory previously allocated for the RPE-LTP state variables.

A.13. ITU-T G.727 module

Name: `g727.c`
Associated header file: `g727.h`
The functions included are as follows.

`G727_encode` ITU T G.727 encoder at 40, 32, 24 and 16 kbit/s.

`G727_decode` ITU T G.727 decoder at 40, 32, 24 and 16 kbit/s.

A.14. Basic operators

A.14.1. Basic operators that use 16-bit registers/accumulators

Name: `basop32.c`, `enh1632.c`
Associated header file: `stl.h`, `basop32.h`, `enh1632.h`

Variable definitions:
– v1, v2: 16-bit variables

`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 shifts the 16-bit input v1 left by v2 positions. Zero fills the v2 LSB of the result. If v2 is negative, arithmetically shifts v1 right by -v2 with sign extension. Saturates the result in case of underflows or overflows.

`shr(v1, v2)` Arithmetically shifts the 16-bit input v1 right v2 positions with sign extension. If v2 is negative, arithmetically shifts v1 left by -v2 and zero fills the -v2 LSB of the result:

```
shr(v1, v2) = shl(v1, -v2)
```

Saturates the result in case of underflows or overflows.

`negate(v1)` Negates v1 with saturation, saturate in the case when input is -32768:

```
negate(v1) = sub(0, v1)
```

`s_max(v1, v2)` Compares two 16-bit variables v1 and v2 and returns the maximum value.

`s_min(v1, v2)` 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 normalize the result, the following operation must be done:

```
norm_v1 = shl(v1, norm_s(v1))
```

A.14.2. Basic operators that use 32-bit registers/accumulators

Name: `basop32.c`, `enh1632.c`
Associated header file: `stl.h`, `basop32.h`, `enh1632.h`

Variable definitions:
– v1, v2, v3_l: 16-bit variables
– L_v1, L_v2, L_v3, L_v3_l, L_v3_h: 32-bit variables

`L_add(L_v1, L_v2)` 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 shifts the 32-bit input L_v1 left v2 positions. Zero fills the v2 LSB of the result. If v2 is negative, arithmetically shifts L_v1 right by -v2 with sign extension. Saturates the result in case of underflows or overflows.

`L_shr(L_v1, v2)` Arithmetically shifts the 32-bit input L_v1 right v2 positions with sign extension. If v2 is negative, arithmetically shifts L_v1 left by -v2 and zero fills the -v2 LSB of the result. Saturates the result in case of underflows or overflows.

`L_negate(L_v1)` Negates the 32-bit L_v1 with saturation, saturate in the case where input is -2147483648.

`L_max(L_v1, L_v2)` Compares two 32-bit variables L_v1 and L_v2 and returns the maximum value.

`L_min(L_v1, L_v2)` 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 normalize 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 normalize the result, the following operation must be done:

```
L_norm_v1 = L_shl(L_v1, norm_l(L_v1))
```

`L_mult(v1, v2)` 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 * v2), 1)` Note that L_mult(-32768,-32768) = 2147483647.
`L_mult0(v1, v2)`	L_mult0 implements the 32-bit result of the multiplication of v1 times v2 without left shift, i.e., `L_mult(v1, v2) = (v1 * 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 * v2) + 16384), 15) )`
	and
	`mult_r(-32768, -32768) = 32767.`

`L_mac(L_v3, v1, v2)`	Multiplies v1 by v2 and shifts the result left by 1. Adds the 32-bit result to L_v3 with saturation, returns a 32-bit result: `L_mac(L_v3, v1, v2) = L_add(L_v3, L_mult(v1, v2))`
`L_mac0(L_v3, v1, v2)`	Multiplies v1 by v2 without left shift. Adds the 32-bit result to L_v3 with saturation, returning a 32-bit result: `L_mac(L_v3, v1, v2) = L_add(vL_v3, L_mult0(vv1, v2))`
`L_macNs(L_v3, v1, v2)`	Multiplies v1 by v2 and shifts the result left by 1. Adds the 32-bit result to L_v3 without saturation, returns 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)`	Multiplies v1 by v2 and shifts the result left by 1. Adds the 32-bit result to L_v3 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 result. `mac_r(L_v3, v1, v2) = round(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)`	Multiplies v1 by v2 and shifts the result left by 1. Subtracts the 32-bit result from L_v3 with saturation, returns a 32-bit result: `L_msu(L_v3, v1, v2) = L_sub(L_v3, L_mult(v1, v2))`
`L_msu0(L_v3, v1, v2)`	Multiplies v1 by v2 without left shift. Subtracts the 32-bit result from L_v3 with saturation, returning a 32-bit result: `L_msu(L_v3, v1, v2) = L_sub(L_v3, L_mult0(v1, v2))`
`L_msuNs(L_v3, v1, v2)`	Multiplies v1 by v2 and shifts the result left by 1. Subtracts the 32-bit result from L_v3 without saturation, returns a 32 bit result. Generates carry and overflow values: `L_msuNs(L_v3, v1, v2) = L_sub_c(L_v3, L_mult(v1, v2))`
`msu_r(L_v3, v1, v2)`	Multiplies v1 by v2 and shifts the result left by 1. Subtracts the 32-bit result from L_v3 with saturation. Rounds the 16 least significant bits of the result

	into the 16 bits with saturation and shifts the result right by 16. Returns a 16-bit result. `msu_r(L_v3, v1, v2) = round(L_msu(L_v3, v1, v2)) = extract_h(L_add(L_sub(L_v3, L_mult(v1, v2)), 32768))`
`s_and(v1, v2)`	Performs a bit wise AND between the two 16-bit variables v1 and v2.
`s_or(v1, v2)`	Performs a bit wise OR between the two 16-bit variables v1 and v2.
`s_xor(v1, v2)`	Performs a bit wise XOR between the two 16-bit variables v1 and v2.
`lshl(v1, v2)`	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)`	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)`	Performs a bit wise AND between the two 32-bit variables L_v1 and L_v2.
`L_or(L_v1, L_v2)`	Performs a bit wise OR between the two 32-bit variables L_v1 and L_v2.
`L_xor(L_v1, L_v2)`	Performs a bit wise XOR between the two 32-bit variables L_v1 and L_v2.
`L_lshl(L_v1, v2)`	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)`	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.
`extract_h(L_v1)`	Returns the 16 most significant bits of L_v1.
`extract_l(L_v1)`	Returns the 16 least significant bits of L_v1.
`round(L_v1)`	Rounds the lower 16 bits of the 32-bit input number into the most significant 16 bits with saturation. Shifts the resulting bits right by 16 and returns the 16-bit number: `round(L_v1) = extract_h(L_add(L_v1, 32768))`
`L_deposit_h(v1)`	Deposits the 16-bit v1 into the 16-bit most significant bits of the 32 bit output. The 16 least significant bits of the output are zeroed.

`L_deposit_l(v1)` Deposits the 16-bit v1 into the 16-bit least significant bits of the 32 bit output. The 16 most significant bits of the output are sign extended.

`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)` 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(v1, v2) but with rounding. Saturates the result in case of underflows or overflows.

```
if v2 is strictly greater than zero, 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)
```

On the other hand, if v2 is lower than or equal to zero, then

```
shr_r(v1, v2) = shr(v1, v2)
```

`shl_r(v1, v2)` Same as shl(v1, v2) but with rounding. Saturates the result in case of underflows or overflows:

```
shl_r(v1, v2) = shr_r(v1, -v2)
```

In the previous version of the STL-basic operators, this operator is called shift_r(v1, v2); both names can be used.

`L_shr_r(L_v1, v2)` Same as L_shr(v1, v2) but with rounding. Saturates the result in case of underflows or overflows:

```
if v2 is strictly greater than zero, 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)
```

On the other hand,

```
if v2 is less than or equal to zero, then
L_shr_r( L_v1, v2) = L_shr( L_v1, v2)
```

`L_shl_r(L_v1, v2)` Same as L_shl(L_v1, v2) but with rounding. Saturates the result in case of underflows or overflows.

```
L_shift_r(L_v1, v2) = L_shr_r(L_v1, -v2)
```

In the previous version of the STL-basic operators, this operator is called L_shift_r(L_v1, v2); both names can be used.

`i_mult(v1, v2)` Multiplies two 16-bit variables v1 and v2 returning a 16 bit value with overflow control.

`rotl(v1, v2, *v3)` 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)` 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)`	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)`	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.
`L_sat(L_v1)`	Long (32-bit) 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.
`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.
`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 equals v2, then div(v1, v2) = 32767.
`div_l(L_v1, v2)`	Produces a result which is the fractional integer division of a positive 32-bit variable L_v1 by a positive 16-bit variable v2. The result is positive (leading bit equal to 0) and truncated to 16 bits.
`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 bits. 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 2 signed 32-bit values L_v1 and L_v2 with saturation control on 64 bits. 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.

A.14.3. Basic operators for unsigned data types

Name: `enhUL32.c`
Associated header file: `stl.h`, `enhUL32.h`

Variable definitions:
- U_var1, U_varout_l: 16-bit unsigned variables
- UL_var1, UL_var2, var1, UL_varout_h, UL_varout_l: 32-bit unsigned variables

`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 unsigned 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_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.
`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.
`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.

A.14.4. Basic operators that use 40-bit registers/accumulators

Name: `enh40.c`
Associated header file: `stl.h`, `enh40.h`

Variable definitions:
– v1, v2, v3: 16-bit variables
– L_v1: 32-bit variables
– L40_v1, L40_v2: 40-bit variables

`L40_add(L40_v1, L40_v2)`	Adds the two 40-bit variables L40_v1 and L40_v2 without saturation control on 40 bits. Any detected overflow on 40 bits will exit execution.
`L40_sub(L40_v1, L40_v2)`	Subtracts the two 40-bit variables L40_v2 from L40_v1 without saturation control on 40 bits. Any detected overflow on 40 bits will exit execution.
`L40_abs(L40_v1)`	Returns the absolute value of the 40-bit variable L40_v1 without saturation control on 40 bits. Any detected overflow on 40 bits will exit execution.
`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 saturation control on 40 bits. Any detected overflow on 40 bits will exit execution.
`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 saturation control on 40 bits. Any detected overflow on 40 bits will exit execution.
`L40_negate(L40_v1)`	Negates the 40-bit variable L40_v1 without saturation control on 40 bits. Any detected overflow on 40 bits will exit execution.
`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))`
`L40_mult(v1, v2)`	Multiplies the 2 signed 16-bit variables v1 and v2 without saturation control on 40 bits. Any detected overflow on 40 bits will exit execution. 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_v1, v2, v3)`	Equivalent to: `L40_add(L40_v1, L40_mult(v2, v3))`
`L40_msu(L40_v1, v2, v3)`	Equivalent to: `L40_sub(L40_v1, L40_mult(v2, v3))`
`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.
`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,returns 2147483647. If L40_v1 is lower than -2147483648,returns -2147483648. If not, 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 saturation control on 40 bits. Any detected overflow on 40 bits will exit execution. The end-result 16 LSB are cleared to 0.
`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 incremented by 1 without saturation control on 40 bits. Any detected overflow on 40 bits will exit execution.
`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_v1, v2) except if v2 is negative. In this case, it does the same as L40_shr_r(L40_v1, (-v2)).
`L40_set(L40_v1)`	Assigns a 40-bit constant to the returned 40-bit variable.

A.14.5. Basic operators that use 64-bit registers/accumulators

Name: `enh64.c`
Associated header file: `enh64.h`, `stl.h`

Variable definitions:
- var1, var2: 16-bit variables
- L_var1, L_var2: 32-bit variables
- W_var, W_var1, W_var2, W_acc: 64-bit variables

`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)`	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)`	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)`	Multiplies the signed 32-bit variable L_var1 with signed 16-bit variable var2. Left-shifts 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)`	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)`	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)`	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)`	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)`	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)`	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. Sets 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. Shifts 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 and 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 and L_var2 are in 1Q31 format, then the result is produced in 2Q62 format.

`W_lshl(W_var1, var2)` — Logically shifts 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 shifts the 64-bit input W_var1 right by var2 positions. If var2 is negative, logically shifts 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.

A.14.6. Basic operators which use 32-bit precision multiply

Name: `enh32.c`
Associated header file: `enh32.h`, `stl.h`

Basic operators in this clause are useful for fast Fourier transform (FFT) and scaling functions where the result of a 32*16 or 32*32 arithmetic operation is rounded, and saturated to a 32-bit value. There is no accumulation of products in these functions. In functions that accumulate products, you should use basic operators in Section n.5.

Variable definitions:
– var2: 16-bit variables
– L_var1, L_var2, L_var3: 32-bit variables

`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. Shifts the product left by 1 with 48-bit saturation control; Adds 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 a 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 a 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 a 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 a 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);
```

A.14.7. Basic operators that use complex data types

Name: `complex_basop.c`
Associated header file: `complex_basop.h`, `stl.h`

Variable definitions:
- var1, var2, var3, re, im: 16-bit variables
- C_var, C_var1, C_var2, C_coeff: 16-bit complex variables
- L_var2, L_var3, L_re, L_im: 32-bit variables
- CL_var, CL_var1, CL_var2: 32-bit complex variables

`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 the real and imaginary parts 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 and imaginary parts 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 the real and imaginary parts 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 and 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 the real and imaginary parts 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 the real and imaginary parts 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 the real and imaginary parts 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 the real and imaginary parts 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 the real and imaginary parts 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 the real and imaginary parts 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 the real and imaginary parts 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 the real and imaginary parts 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 the real and imaginary parts of a complex number:
	`CL_result.re = L_re;`
	`CL_result.im = L_im;`

`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 the real and imaginary parts 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 parts of a complex variable CL_var are in 1Q31 format, and C_coeff is in 1Q15 format, then the intermediate products would be in the 17Q47 format. The round operation will convert the result of addition/subtraction from 17Q47 format to 1Q31 format.

`CL_negate(CL_var)`	Negates the 32-bit complex number, saturates and returns.
	The following code snippet describes the operations performed on the real and imaginary parts 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 describes the operations performed on the real and imaginary parts 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 the real and imaginary parts 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_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 the real and imaginary parts 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)));`

`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 the real and imaginary parts 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 the real and imaginary parts 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_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 parts 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 parts 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)));`
`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, the 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, the 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, the real and imaginary parts of C_var1 are shifted to the least significant bits by (-var2) positions with sign extension.
If var2 is positive, the real and imaginary parts of C_var1 are shifted to the most significant bits by (var2) positions with 16-bit saturation control.

A.14.8. Basic operators for control operation

Name: `control.c`
Associated header file: `control.h`, `stl.h`

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.

Variable definitions:
– 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.

The basic operators module is supplemented by two tools: one to evaluate program ROM complexity for fixed-point code, and another to evaluate complexity (including program ROM) of floating-point implementations.

A.14.9. Program ROM estimation tool for fixed-point C code

Name: `basop_cnt.c`
Associated header file: None.
Usage: `basop cnt input.c [result_file_name.txt]`

The basop_cnt tool estimates the program ROM of applications written using the ITU-T basic operator libraries. It counts the number of calls to basic operators in the input 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.

A.14.10. Complexity evaluation tool for floating-point C code

Name: `flc.c`
Associated header file: `flc.h`
The functions included are as follows.

`FLC_init`	Initialize the floating-point counters.
`FLC_sub_start`	Marks the start of a subroutine/subsection.
`FLC_sub_end`	Marks the end of a subroutine/subsection.
`FLC_end`	Computes and prints complexity, i.e., floating-point counter results.
`FLC_frame_update`	Marks the end of a frame processing to keep track of the per-frame maxima.

A.15. Reverberation module

Name: `reverb-lib.c`
Associated header file: `reverb-lib.h`
The functions included are as follows.

`conv`	Convolution routine.
`shift`	Shift elements of a vector for the block-based convolution.

A.16. Bit stream truncation module

Name: `trunc-lib.c`
Associated header file: `trunc-lib.h`
The functions included are as follows.

`trunc`	Frame truncation routine.

A.17. Frequency response calculation module

Name: `fft.c`
Associated header file: `fft.h`
The functions included are as follows.

`rdft`	Discrete Fourier transform for real signals.
`genHanning`	Hanning window generation routine.
`powSpect`	Power spectrum computation routine.

Annex B

ITU-T software tools General Public Licence

(This annex forms an integral part of this Recommendation.)

Terms and conditions

B.1.

This Licence Agreement applies to any module or other work related to the ITU-T Software Tool Library, and developed by the User's Group on Software Tools. The term "Module" refers to any such module or work, and a "work based on the Module" means either the Module or any work containing the Module or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".

B.2.

You may copy and distribute verbatim copies of the Module's source code as you receive it, in any medium, provided that you:

– conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty;
– keep intact all the notices that refer to this General Public Licence and to the absence of any warranty; and
– give any other recipients of the Module a copy of this General Public Licence along with the Module.

You may charge a fee for the physical act of transferring a copy.

B.3.

You may modify your copy or copies of the Module or any portion of it, and copy and distribute such modifications under the terms of clause B.1, provided that you also do the following:

– cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
– cause the whole of any work that you distribute or publish, that in whole or in part contains the Module or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public Licence (except that you may choose to grant warranty protection to some or all third parties, at your option);
– if the modified module normally reads commands interactively when run, you must cause it, on start-up for such interactive use, in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the module under these conditions, and telling the user how to view a copy of this General Public Licence.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Mere aggregation of another independent work with the Module (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

B.4.

You may copy and distribute the Module (or a portion or derivative of it, under clause B.2) in object code or executable form under the terms of clauses B.1 and B.2, provided that you also do one of the following:

– accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of clauses B.1 and B.2; or
– accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of clauses B.1 and B.2; or
– accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for non-commercial distribution and only if you received the module in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules that are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definition files that accompany that operating system.

B.5.

You may not copy, modify, sublicense, distribute or transfer the Module except as expressly provided under this General Public Licence. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Module is void, and will automatically terminate your rights to use the Module under this Licence. However, parties who have received copies, or rights to use copies, from you under this General Public Licence will not have their licences terminated so long as such parties remain in full compliance.

B.6.

By copying, distributing or modifying the Module (or any work based on the Module) you indicate your acceptance of this licence to do so, and all its terms and conditions.

B.7.

Each time you redistribute the Module (or any work based on the Module), the recipient automatically receives a licence from the original licensor to copy, distribute or modify the Module subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

B.8.

The ITU-T may publish revised and/or new versions of this General Public Licence from time to time. Such new versions will be similar in spirit to this version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Module specifies a version number of the licence that applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by ITU-T. If the Module does not specify a version number of the licence, you may choose any version ever published by ITU T.

B.9.

If you wish to incorporate parts of the Module into other free modules whose distribution conditions are different, write to the author to ask for permission. For software that is copyrighted by the ITU-

T, write to the ITU-T Secretariat; exceptions may be made for this. This decision will be guided by the two goals of preserving the free status of all derivatives of this free software and of promoting the sharing and reuse of software generally.

B.10.

Because the Module is licensed free of charge, there is no warranty for the Module, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the Module "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Module is with you. Should the Module prove defective, you assume the cost of all necessary servicing, repair or correction.

B.11.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the Module as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the Module (including, but not limited to, loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the Module to operate with any other modules), even if such holder or other party has been advised of the possibility of such damages.

Bibliography

[b-CMake] b-CMake, Kitware (2018), *CMake*. https://cmake.org/.

[b-GSM 06.10] b-GSM 06.10, ETSI Recommendation GSM 06.10 (1992), *GSM full-rate speech transcoding*.

[b-STLgit] b-STLgit, ITU (2019), *ITU-T software tool library (G.191)*, GitHub repository. https://github.com/openitu/STL.