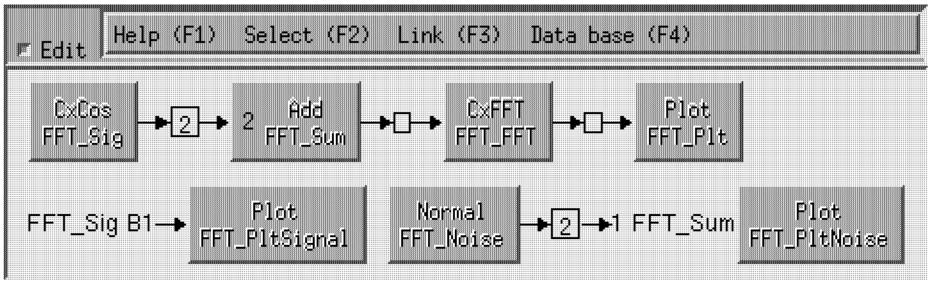
ObjectProDSP Library Reference

Paul P. Budnik Jr. Phd.
Internet: support@MTNMATH.COM

September 1994

'.dvi' file created September 16, 1994



Mountain
Math
Software
P. O. Box 2124, Saratoga, CA 95070
Fax or voice (408) 353-3989

Published by Mountain Math Software, P. O. Box 2124, Saratoga, CA 95070.

Copyright © 1994 by Mountain Math Software. All rights reserved.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "GNU General Public License" and "Licensing" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one, and provided the derived work is clearly identified as a derived work and not solely the creation of either the orignal authors or the authors of the derived work.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the sections entitled "GNU General Public License" and "Licensing", and this permission notice, may be included in translations approved by Mountain Math Software instead of in the original English. Translations of the section entitled "GNU General Public License" must also be approved by the Free Software Foundation which owns the copyright to that text.

Licensing

ObjectProDSP^{TM} is licensed for free use and distribution under version 2 of the GNU General Public License. See Appendix A for the full text of this license. There is absolutely no warranty for ObjectProDSP under this license. ObjectProDSP is a trademark of Mountain Math Software.

You are free to use and distribute ObjectProDSP under the terms of version 2 of the GNU General Public License. Please note that *none* of the ObjectProDSP system is licensed for use under the GNU *Library* General Public License. The Gnu General Public License allows you to distribute executables or librarys linked with or created by ObjectProDSP *only* if you make *all* the *source* code used to create the librarys or executables (other than standard librarys that are part of a compiler or operating system) freely available. Please read the license in Appendix A for the full legal explanation of these conditions.

Mountain Math Software plans to offer, for a fee, a commercial version that will allow you to distribute executables generated with ObjectProDSP under standard commercial terms.

If you wish to extend ObjectProDSP you can distribute your code with ObjectProDSP under the terms of the GNU General Public License. If you include an appropriate copyright notice in your name for your upgrades then no one, including Mountain Math Software, will be able to distribute your code under any terms other than the GNU General Public License without your permission.

If you find ObjectProDSP useful in a commercial environment you are asked to consider purchasing a support contract. This is not shareware and you are under no obligation to do so but you will gain aceess to direct support from Mountain Math Software and you will make a contribution to the continued success of ObjectProDSP and thus to any of your endeavors that benefit from it.

If you are interrested in a custom port of ObjectProDSP to directly support your company's DSP development board or processor please contact us.

Mountain Math Software
P. O. Box 2124
Saratoga, CA 95070
Internet: support@MTNMATH.COM
Fax or voice (408) 353-3989

Documentation

- *ObjectProDSP Overview and Tutorial* This gives a general description of ObjectProDSP's purpose and function. It includes several tutorial examples. There are appendices on the DSP node and class library and Mountain Math Software.
- *ObjectProDSP User's Reference* This describes the user interface and DSP++, a C++ based language for DSP. (You do not need to known DSP++ or C++ to use ObjectProDSP. DSP++ statements are generated for you when you graphically enter a network or execute menu data base commands.) This document includes a reference manual for the menu data base. Appendixes contain a synopsis of menu data base commands and a general index.
- *ObjectProDSP Library Reference* This is the document you are reading. This gives a detailed description of ObjectProDSP interactive objects including DSP processing nodes.
- *ObjectProDSP Developer's Reference* This tells how to write DSP processing nodes and add them to ObjectProDSP. It describes ObjectPro++^{TM}, an extended C++ language for defining interactive objects for DSP or other applications. It explains how to modify the part of the menu data base that does not come from interactive object definitions in ObjectPro++. It describes how to update the ObjectProDSP manuals to include your new nodes and objects. Information about these objects is extracted from your definitions by ObjectPro++ and added to the manuals.

Contents

List of Tables

Name	Signal type
`ConstantData`	constant level
`Cos`	real cosine
`CxCos`	complex cosine
`CxImp`	complex impulse or square wave
`Normal`	Gaussian distributed noise
`Ramp`	ramp function
`UniformNoise`	uniformly distributed noise

Table 1: Signal generation nodes

1 Introduction

This manual describes all user accessible ObjectProDSP objects including DSP processing nodes. Most of the text in this manual is extracted from the extended ObjectPro`++` class definitions.

2 Overview of nodes

This section gives a brief overview of the processing nodes. It contains the information in the help files for the nodes.

2.1 Signal nodes

The nodes under `objects` and `signal` generate standard test signal. They create output data streams as a function of node parameters. They (along with nodes that read their input from disk are the initial source of data for a network.

Signal generation nodes are shown in Table 1 on page 1.

You can use the `Add` node (under `objects` and `dsp processing`) to sum 2 or more of these signal sources.

2.2 Add the output from multiple inputs

The `Add` node sums inputs from any number of nodes to generate a single output stream. All inputs and the output must have the same number of words per sample or `ElementSize`. For example they must all be real or all complex. A check for overflow is made for each addition. At overflow the output is forced to the largest legal magnitude. The number of overflows is periodically reported. The input can be scaled by a uniform factor before the overflow check.

2.3 InputWord, OutputWord, VoiceNode and VocStripOut

`InputWord` and `OutputWorde` read and write binary files as consecutive binary words. The data type is selectable from most standard types. These nodes may be used for importing or exporting binary data to other programs. It is best to use `InputNode` and `OutputNode` for importing and exporting data to ObjecProDSP. These preserve information about data streams such as sample rate and timing as well as the data itself.

`VoiceStripOut` writes files in a format based on the Creative voice format of the SoundBlaster board but with no header.

`VoiceNode` reads `Creative voice file` format files including the header and can display the header contents. These programs were written based on an early version of the SoundBlaster software and may not work with more recent versions.

They nodes are under `objects`, `disk` and `binary`.

2.4 Import ascii data

The `ImportData` node (select `objects`, `disk` and `ascii`) is a flexible node for importing ascii data from other programs. You can specify the format for reading each value (using C conventions such as `%d` and `%x`). You specify the number of fields per line. You can ignore selected columns and selected fields. You can combine multiple fields into a single word.

2.5 Change sample size and blocking of input

`Block` is under `objects` and `dsp processing`.

`Block` converts its input stream to an output stream with a specified number words per sample and a specified number of samples per block. The sample size and block size of the input data stream are ignored. The stream is treated as if it were an unblocked real data stream.

Some nodes that read disk files only generate a real data stream. If such a node is used to read complex data `Block` can do the needed transformation. If a node is used to read FFT output from another process you can use `Block` to structure the data so it will be plotted correctly with one FFT window per plot.

2.6 Filtering operations

Nodes `CxFir`, `RlFir`, `Demod`, `Interpolate` and `Integrate` perform various filtering operations in their input. They are under `objects` and `dsp processing`.

`CxFir` and `RlFir` are standard finite impulse response FIR filters for real and complex data respectively. They both support arbitrary integer resampling of their input data streams.

`DcTrap` is a simple notch filter at 0 hz.

`Demod` does complex demodulation by an arbitrary frequency, i. e. it multiplies its input by $e^{-i\omega N}$ where ω is an angle that translates to a demodulation frequency and N is the sample index. Demodulation is also a built in option in `CxFir`.

`Interpolate` uses linear interpolation to generate an output data stream with a different sample rate then its input data stream.

`Integrate` does a sliding window sum of its input to generate its output. It can produce output at any integral divisor of its input.

2.7 Complex FFT

`CxFft` under `objects` and `dsp processing` converts a complex input data stream to blocked spectral output. The output of the FFT has a sample rate just as any other data stream does but the samples are blocks of complex data. Plot and listing nodes understand this block structure and display the data accordingly. `Plot` uses the sample rate and block size to compute a frequency axis for displaying FFT output.

2.8 Conversion nodes

Nodes `ToInteger`, `ToMach`, `Truncate` and `Power` perform conversion operations. They are under `objects` and `dsp processing`.

`MachWord` is the native type of an ObjectProDSP simulator. For the floating point version this is single precision floating point.

`ToInteger` converts its `MachWord` input stream to 32 bit integers.

`ToMach` converts a 32 bit integer input data stream to type `MachWord`.

`ToInteger` and `ToMach` are only available on the floating point simulator.

`Truncate` limits the accuracy (most significant bit and number of bits) of its input and writes the result to its output. Overflow will optionally saturate or truncate. This node reads and writes data of type `MachWord` but operates internally on 32 bit integers.

`Power` computes the power of a real or complex data stream and writes this to its output. More generally it computes the sum of the squares of each `ElementSize` value in a sample. `ElementSize` is determined from the input data stream.

2.9 Multiplexing, bit manipulation and related operations

Routines `Mux`, `Demux`, `MaskWord`, `PackWord`, `UnpackWord`, `RepackStream` and `GainPad` perform multiplexing bit manipulation and related operations. They are under `objects` and `dsp processing`.

`Mux` and `Demux` do multiplexing and demultiplexing. `Mux` combines an arbitrary number of input channels into a single output channel. `Demux` does the inverse operation. `Demux` can also separate the real and imaginary components of a single data stream into different output channels. `Demux` can invert this operation.

`GainPad` converts an real data stream to complex by padding the imaginary part with 0. It can also apply a gain to the input.

`MaskWord` reads and writes a binary data stream applying a bit mask to each input sample. Use `ToInteger` and `ToMach` to convert input and output streams if needed.

`UnpackWord` unpacks a specified number of words of a specified number of bits from a single physical integer input word and writes each of these as a separate sample to an integer output data stream. `PackWord` does the inverse operation. Both treat the least significant bits in a word as being first in sequence.

`RepackStream` is a more general (and less efficient) operation than `PackWord` and `UnpackWord`. It is its own inverse. It treats its input and output channels as continuous streams of bits with the least significant bits in a physical word occurring first in the continuous sequence. It repacks the input stream with a specified input word size to an output stream with a specified output word size. These word sizes must be less than or equal the physical word size in the stream.

2.10 Miscellaneous DSP processing nodes

There are miscellaneous DSP nodes (under `objects` and `dsp processing`) for creating a delay (necessary for feedback loops) finding the start of a signal in a data stream and applying an arbitrary gain factor.

`SampleDelay` copies its input to its output appending and initial string of constant values. The length of this initial string and the value of the constant are selectable. If you construct a feedback loop in a network you must have a suitably long delay in the loop or the network will block.

`FindStartTail` copies its input to its output after throwing away a fixed initial data segment of selectable length and/or an initial segment that meets selectable signal level constraints.

`Gain` copies its input to its output applying a fixed gain to each sample.

2.11 Read and write disk files

The `disk` menu under `objects` provides access to nodes that read and write binary and ascii files.

Nodes for reading and writing ascii files

`ReadInt` and `ReadFloat` read simply formatted integer or floating point data with one number per line. The outputs stream of `ReadFloat` is a type `MachWord`. The output stream of `ReadInt` is of type integer. You may need to use `ToFloat` if you are working with a floating point simulator. `ImportData` can directly read more complexly structured ascii files.

`AsciiFile` writes ascii files for import to other processes. It optionally includes header information and optionally groups data by samples and blocks. It can output data in hexadecimal format. Under the hexadecimal format input is converted to 32 bit integers and hard limited with a warning if it will overflow 32 bits.

Nodes for reading and writing binary files

Nodes `InputNode` and `OutputNode` are the standard nodes for saving data between ObjectProDSP sessions. In addition to the data itself they preserve all the information that describes the data such as its time base and same rate. This format of these files is documented in the help file for these nodes.

`OutputWord` and `InputWord` are nodes for reading and writing simple binary data files in a form that may be suitable for importing or exporting to other programs.

2.12 ObjectProDSP binary format data files

Output node `OutputNode` writes its input channels to a disk file. This file can be read by input node `InputNode`. The data is written in binary exactly as it is generated.

This file format is convenient for saving multiple channels from a network to a single file. All the data in all the channels directed to this node will be saved as a single file. This data can then be accessed any time in the future in the same sequence in which it was generated.

This node is also used for ObjectProDSP validation and regression testing. See the section on validation for more information on this topic.

OutputNode

The parameters for `OutputNode` include a block size to determine the consecutive words that are written from a single channel to the disk file. Making this larger will improve efficiency at the cost of more memory for buffers.

InputNode

`InputNode` reads most of the characteristics of its input data in the file header. This includes the number of channels. Pamameter `DeltaOut` con-

Field	Size in bits	Contents
Type	8	Code for packet type
Size	8	Length of data field in bytes
Identifier	32	Data identification
Data	Size×8	Packet data

Table 2: Disk file packet format

Packet type	Code	Identifier	Data
`FileEltsHeader`	0	NONE	See Table 4 on page 8
`FileEltsNodeName`	1	NONE	Node name
`FileEltsCaption`	2	NONE	Caption parameter
`FileEltsChannelHeader`	3	NONE	See Table 6 on page 9

Table 3: Disk file header packets

trols how often the node is scheduled and can influence efficiency. You can also ignore the input sample rate and force it be 1 on the base channel.

This node must be able to read its input file at the time its constructor is called or it cannot initialize itself properly from the input file. Any node that does not succeed in doing this read will be unusable later.

Field	Size in bits	Interpretation
`DataType`	32 (enum DataType)	See Table 5 on page 9
`NumberOfChannels`	16	Input channels
`BlockSize`	32	Contiguous data size

Table 4: FileEltsHeader packet structure

Name	Code	Size of one MachWord in bits
`ArithDouble`	1	64
`ArithInt16`	2	16
`ArithInt32`	3	32
`ArithFloat`	4	32

Table 5: DataType codes

Field name	Size in bits	Interpretation
`NumeratorSampling`	32	Samples input (ratio)
`DenominatorSampling`	32	Samples output (ratio)
`FirstSample`	64 (double)	Time of first output
`ElementSize`	32	MachWords in one sample
`NumberWords`	32 (long)	MachWords in file

Table 6: Channel header

File format

The files written by `OutputNode` and read by `InputNode` begin with a header format organized in packets. A sequence of packets that gives general header information is followed by a packet giving information for each channel. The format of these packets is defined in Table 2 on page 8.

The general header information and the information for each channel is contained in the sequence of packets defined in Table 3 on page 8.

The data in the FileEltsHeader packet is structured as shown in Table 4 on page 8.

The codes for the DataType field are listed in Table 5 on page 9.

The packet for each channel header contains the data shown in Table 6 on page 9.

This header information is followed with a series of data packets containing

Field	Size in bits	Interpretation
Channel	16	Index of this channel
Word index	32	Index of first word
Data	8×sizeof(MachWord)×BlockSize	Node input data

Table 7: Data packet format

samples from a single channel. The data from each channel is in sequential order. The sequences of the packets from different channels is determined by the order in which the data is written and is not easy to determine. The format of these packets is is shown in Table 7.

3 DSP processing nodes

Most of the text in this section through Section 7 on page 66 was extracted from the ObjectPro`++` input `.usr` files. It duplicates the description of the nodes available in the menu data base.

The nodes in this section have both input and output channels. They provide some form of DSP or utility processing.

3.1 `Add`

Synopsis: `Add` sums two or more input channels.

`Add` sums `Channels` input channels into a single output channel. Each sample is multiplied by `Scale` before being added to the output channel. Overflows are prevented by clipping. The first time clipping occurs a help message is generated. A new help message is generated after every 400 clippings. Each sample consists of `ElementSize` words. `ElementSize` is usually one for real data streams and two for complex streams, but it can be any 16 bit positive integer and used for any purpose.

The parameters for `Add` are:

- `Channels`: number of input channels.

  `Channels` specifies the number of input channels to be added together in a single output channel.

  `Channels` is of type `int16`. The default value is 2. `Channels` must be $\geq$ 2 and $\leq$ 32767.

- `ElementSize`: number of multiplexed elements in each channel.

  `ElementSize` specifies the sample size for each channel. The most common use of `ElementSize` is to set it to two for a complex data stream. All input data streams must have the same sample size. If set to 0 `ElementSize` is set automatically based on its value for the first input node.

  This is the first default parameter. `ElementSize` is of type `int16`. The default value is 0. `ElementSize` must be $\geq$ 0 and $\leq$ 32767.

- `Scale`: scale factor for each channel.

  `Scale` is a scale factor applied to each channel before it is added to the output channel. The channel addition will clip any data that would otherwise create an overflow and generate a help message. Only one help messages is generated, for every 400 input samples regardless of the number of overflows that may occur.

  This is the first default parameter. `Scale` is of type `double`. This parameter can be changed interactively. The default value is 1.0. `Scale` must be $\geq$ -1.e100 and $\leq$ 1.e100.

3.2 Block

Synopsis: Converts an input stream to a new blocking and sample size.

`Block` converts its input stream to an output stream with `ElementSize` words per sample and `BlockSize` samples per block. The sample size and block size of the input data stream are ignored. The stream is treated as if it were an unblocked real data stream. On a 32 bit simulator `Block` can convert an integer or floating point input channel to floating point or integer output. If overflow occurs in converting integer to floating point the result will saturate and no warning message will be given. Some nodes that read disk files only generate a real data stream. If such a node is used to read complex data `Block` can do the needed transformation. If a node is used to read FFT output from another process you can use `BlockSize` to structure the data so it will be plotted correctly with one FFT window per plot. If `BlockSize` is 1 the output is not blocked.

The parameters for `Block` are:

- `ElementSize`: output element size.

  `ElementSize` is the number of words in each output sample (1 for real, 2 for complex or larger for other purposes).

  `ElementSize` is of type `int16`. The default value is 2. `ElementSize` must be ≥ 1 and ≤ 32767.

- `BlockSize`: output block size.

  `BlockSize` is the number of samples in each output block. If set to 1 the output is not blocked.

  This is the first default parameter. `BlockSize` is of type `int16`. The default value is 1. `BlockSize` must be ≥ 1 and ≤ 32767.

- `OutputArithmetic`: output data: 0 - MachWord, 1 - int32, 2 - float.

  `Block` can read data from any input arithmetic type. `OutputArithmetic` selects the output arithmetic type. On a 32 bit simulator `Block` can write output as either 32 bit floating point or 32 bit fixed point. Choose 0 to write output in the default type of the simulator, 1 for 32 bit integers and 2 for 32 bit floating point.

  This is the first default parameter. `OutputArithmetic` is of type `int16`. The default value is 0. `OutputArithmetic` must be ≥ 0 and ≤ 2.

3.3 CxFFT

Synopsis: `CxFFT` computes the complex FFT of a single input channel.

`CxFFT` computes a series of complex FFTs of size N = 2 ∧ `LogSize` on a single complex input channel. A forward FFT is computed if `InverseFlag` is 0 and an inverse FFT if this parameter is 1. Successive FFTs have their first samples separated by by N × `Overlap` samples. If `Overlap` is 1.0 then successive inputs are separated by a single sample.

The parameters for `CxFFT` are:

- `LogSize`: log base 2 of FFT size.

  `LogSize` is the log base 2 of the FFT size.

  `LogSize` is of type `int16`. The default value is 4. `LogSize` must be $\geq$ 1 and $\leq$ 512.

- `Overlap`: fractional overlap of successive FFTs.

  `Overlap` specifies the fractional overlap of successive FFTs. For example `Overlap` = .5 and `LogSize` = 64 would result in each FFT being 32 samples beyond the previous FFT. `Overlap` = 1.0 is interpreted to mean that successive FFTs having their inputs separated by a single sample.

  This is the first default parameter. `Overlap` is of type `double`. The default value is 0.0. `Overlap` must be $\geq$ 0.0 and $\leq$ 1.0.

- `CenterFrequency`: position of center frequency in FFT output bins.

  The true center frequency of the FFT is determined by the data entering it. `CenterFrequency` allows you to rotate the output bins to conform to this. The default value of .5 corresponds to a signal with center frequency of 0 hz in the input sample stream. This value should set to the relative position of the center frequency of the input sample stream.

  This is the first default parameter. `CenterFrequency` is of type `double`. This parameter can be changed interactively. The default value is 0.5. `CenterFrequency` must be $\geq$ -1.0 and $\leq$ 1.0.

- `InverseFlag`: inverse FFT flag.

  `InverseFlag` set to 1 selects an inverse FFT.

  This is the first default parameter. `InverseFlag` is of type `int16`. The default value is 0. `InverseFlag` must be $\geq$ 0 and $\leq$ 1.

3.4 `CxFir`

Synopsis: `CxFir` is a complex symmetric (even or odd) fir filter.

`CxFir` implements two parallel real symmetric FIR filters that operate on a complex data stream. Optional complex demodulation multiplies the input by a complex cosine with phase increment -`DemodFreq` radians between each pair of samples. Input samples can have `ZeroPad` 0's interpolated between them. Output sample can be generated for every `Resample` input (or zero pad) sample. The `Coeff` array contains half of the symmetric coefficients. `Odd` indicates if there are an even(0) or odd(1) total number of coefficients.

The parameters for `CxFir` are:

- `Resample`: filter resampling factor (input rate/output rate).

  `Resample` specifies the filter resampling factor. This is the ratio of the input sampling rate to the output sampling rate.

  `Resample` is of type `int16`. The default value is 1. `Resample` must be ≥ 1 and ≤ 32767.

- `ZeroPad`: zero padding of input.

  `ZeroPad` specifies the number of 0's that are added after each input sample. If Zero padding is done the filter must be designed as an interpolation filter.

  This is the first default parameter. `ZeroPad` is of type `int16`. The default value is 0. `ZeroPad` must be ≥ 0 and ≤ 32767.

- `DemodFreq`: demodulation frequency (0 for no demodulation).

  `DemodFreq` is the demodulation frequency in radians per sample. Input sample N is multiplied by e∧(-i × 2 × Pi × `DemodFreq` × N) before the filtering operation. If `DemodFreq` is 0 then the demodulation step is skipped.

  This is the first default parameter. `DemodFreq` is of type `double`. The default value is 0.0. `DemodFreq` must be $\geq$ -1.e100 and $\leq$ 1.e100.

- `Odd`: if set the filter length is odd.

`Odd` determines if the filter is odd (`Odd`=1) or even (`Odd`=0).

This is the first default parameter. `Odd` is of type `int16`. The default value is 0. `Odd` must be ≥ 0 and ≤ 1.

- `Coeff`: list of unique coefficients.

  `Coeff` is the list of filter coefficients. The filter is symmetric and only half (or half plus one for odd length filters) are specified. The first in the list is the first coefficient of the filter. The middle coefficient is at the end of the list. The default values for the coefficients define a low pass FIR filter with a pass band of .125 times the sample rate and transition band of .375 times the sample rate. The pass band is extremely flat and the stop band is down over 100 db. This is an overdesigned filter for most practical applications but it provides a good test case. The performance will be degraded by 16 bit integer arithmetic.

  This is the first default parameter. `Coeff` is of type `MachWord` ×. `Coeff` is an array of size ≥ 3 and ≤ 1024. The following table contains the default array values.

1.00018552e-04	3.70747893e-04	4.46594395e-04	-5.36969535e-04
-2.67492760e-03	-3.52687595e-03	8.25715193e-04	1.01494638e-02
1.50874056e-02	2.48008436e-03	-2.76876913e-02	-4.93498540e-02
-2.20594765e-02	7.30790837e-02	2.04010323e-01	2.99286359e-01

3.5 `Demod`

Synopsis: `DemodFreq` is a complex modulation/demodulation function.

`Demod` is a complex modulation or demodulation function. Its input can be real or complex as can its output.

The parameters for `Demod` are:

- `DataType`: select real or complex data for input or output.

  `DataType` selects complex input and complex output (0), complex input and real output (1), real input and complex output (2) or real input and real output (3).

`DataType` is of type `int16`. The default value is 0. `DataType` must be $\geq$ 0 and $\leq$ 3.

- `DemodFreq`: demodulation frequency (negative values for modulate).

  `DemodFreq` is the demodulation frequency in radians/sample. Input sample N is multiplied by e∧(-i × N × 2 × Pi × `DemodFreq`)

  `DemodFreq` is of type `double`. This parameter can be changed interactively. The default value is 0.0. `DemodFreq` must be $\geq$ -1.e100 and $\leq$ 1.e100.

3.6 Demux

Synopsis: Demultiplexes 1 input channel to `Channels` output channels.

`Demux` takes one input channel and demultiplexes it onto `Channels` output channels. The input channel must have samples consisting of `InputElementSize` words. For example, complex data streams will generally have two words in each sample. The output data stream will have samples or `OutputElementSize` words. `InputElementSize` and `OutputElementSize` do not affect the kernel loop execution. `InputSampleSize` words are taken from the input and written to the first output channel. The next `InputSampleSize` words of input are written to the next channel. This is done for all output channels and then begins again with the first output channel.

The parameters for `Demux` are:

- `Channels`: number of output channels.

  `Demux` demultiplexes a single input channel into `Channels` output channels.

  `Channels` is of type `int16`. The default value is 2. `Channels` must be $\geq$ 2 and $\leq$ 32767.

- `InputSampleSize`: number of consecutive words demultiplexed in one input sample.

  The input channel has samples of `InputSampleSize` words.

This is the first default parameter. `InputSampleSize` is of type `int32`. The default value is 1. `InputSampleSize` must be ≥ 1 and ≤ 2147483647.

- `InputElementSize`: element size of input channels.

  The parameter does not affect the demultiplexing loop. The input channel must have this value for `InputElementSize`. If you are demultiplexing an element into its component parts (such as demultiplexing complex data to real and imaginary streams) `Channels` and `InputElementSize` must have the same value.

  This is the first default parameter. `InputElementSize` is of type `int32`. The default value is 1. `InputElementSize` must be ≥ 1 and ≤ 2147483647.

- `OutputElementSize`: element size of output channels.

  The parameter does not affect the demultiplexing loop. All output channels have this value for `OutputElementSize`.

  This is the first default parameter. `OutputElementSize` is of type `int32`. The default value is 1. `OutputElementSize` must be ≥ 1 and ≤ 2147483647.

3.7 `FindStartTail`

Synopsis: discard initial input data within bounds.

`FindStartTail` copies its input to its output. Data is copied at the first sample with value > `LowerBound` and < `UpperBound`. If the input data is integer type on a floating point simulator signed input is assumed. If the data is complex then the magnitude of each component of the each sample is checked. The first full sample after the test is passed is output.

The parameters for `FindStartTail` are:

- `LowerBound`: ignore initial input > `LowerBound`.

  A sample containing an initial element > `LowerBound` will be ignored. Once one element of a sample has passed both bound tests all later samples will be passed to the next node. If the first element in a sample passes the tests then that sample will be passed.

`LowerBound` is of type `double`. The default value is 0. `LowerBound` must be $\geq$ -1.e100 and $\leq$ 1.e100.

- `UpperBound`: ignore initial input > `UpperBound`.

  An initial element < `UpperBound` will be ignored. Once one element of a sample has passed both bound tests it and all later samples will be passed to the next node. If the first element in a sample passes the tests then that sample will be passed.

  `UpperBound` is of type `double`. The default value is 0. `UpperBound` must be $\geq$ -1.e100 and $\leq$ 1.e100.

- `Flags`: specieal options.

  If bit 2 is set then the first bounds test is ignored. If bit 3 is set the second test is ignored. If both bits 2 and 3 are set you can skip a fixed number of samples by setting `Skip`.

  This is the first default parameter. `Flags` is of type `int16`. The default value is 0. `Flags` must be $\geq$ -32767 and $\leq$ 32767.

- `Skip`: Samples to skip before processing any data.

  The first `Skip` samples are read but not written.

  This is the first default parameter. `Skip` is of type `int32`. The default value is 0. `Skip` must be $\geq$ 0 and $\leq$ 2147483647.

3.8 Gain

Synopsis: Gain provides a linear gain.

`Gain` copies its input to its output after applying a liner scale factor (`Scale`) to each input sample element.

The parameter for `Gain` is:

- `Scale`: ratio of input amplitude to output amplitude.

`Scale` specifies the ratio of input amplitude to output amplitude. Integer overflows are prevented by clipping. The first time clipping occurs a help message is generated. A new help message is generated after every 400 clippings.

`Scale` is of type `double`. This parameter can be changed interactively. The default value is 1.0. `Scale` must be ≥ -1.e100 and ≤ 1.e100.

3.9 `GainPad`

Synopsis: `GainPad` provides a linear gain.

`GainPad` copies its input to its output after applying a liner scale factor (`Scale`) to each input sample element. It will convert real to complex data by adding a 0 imaginary part to each sample. Set `ElementSize` to 1 for this purpose. Otherwise `ElementSize` is the number of values or elements in each sample.

The parameters for `GainPad` are:

- `Scale`: ratio of input amplitude to output amplitude.

  `Scale` specifies the ratio of input amplitude to output amplitude. Integer overflows are prevented by clipping. The first time clipping occurs a help message is generated. A new help message is generated after every 400 clippings.

  `Scale` is of type `double`. This parameter can be changed interactively. The default value is 1.0. `Scale` must be ≥ -1.e100 and ≤ 1.e100.

- `ElementSize`: number of multiplexed elements.

  `ElementSize` specifies the sample size. The most common use of `ElementSize` is to set it to two for a complex data stream or 1 for real data. Set it to 0 to convert real to complex data by padding the imaginary part with 0.

  This is the first default parameter. `ElementSize` is of type `int16`. The default value is 1. `ElementSize` must be ≥ 0 and ≤ 32767.

- `NullOutputSample`: all samples beyond `NullOutputSample` will be 0.

  If `NullOutputSample` is non zero then all samples after `NullOutputSample` will be zero. If `NullOutputSample` is 0 then the input is written to the output continuously.

  This is the first default parameter. `NullOutputSample` is of type `int32`. The default value is 0. `NullOutputSample` must be ≥ 0 and ≤ 2147483647.

3.10 Integrate

Synopsis: `Integrate` sums consecutive input vector.

`Integrate` sums `IntegrationSize` consecutive samples and outputs this sum for every `OutputStep` input samples. In effect it is a Fir filter with all the coefficients equal to 1. If `IntegrationSize` is 0 then the sum is continuous and the output is normalized by the number of samples. The summation is done in double floating point. The output is scaled by `Scale` before being converted to 'MachWord'.

The parameters for `Integrate` are:

- `IntegrationSize`: Number of samples to sum.

  `IntegrationSize` is the number of samples to sum.

  `IntegrationSize` is of type `int32`. The default value is 1. `IntegrationSize` must be ≥ 0 and ≤ 2147483647.

- `OutputStep`: number of input samples for one output.

  An output is generated for every `OutputStep` inputs. The number of buffers that must be maintained is `IntegrationSize / OutputStep`.

  This is the first default parameter. `OutputStep` is of type `int32`. The default value is 1. `OutputStep` must be ≥ 1 and ≤ 2147483647.

- `Scale`: output is scaled by this factor.

`Scale` multiplies each output sample. The intermediate arithmetic is done in double floating point point. `Scale` is applied to the final output point before it is converted to MachWord.

This is the first default parameter. `Scale` is of type `double`. This parameter can be changed interactively. The default value is 1.0. `Scale` must be $\geq$ -1.e100 and $\leq$ 1.e100.

3.11 `Interpolate`

Synopsis: sample rate conversion with linear interpolation.

`Interpolate` does linear interpolation and sample rate conversion. Each element in a sample is interpolated independently, i. e. for a complex data stream the real and imaginary parts are interpolated independently. For every `DeltaIn` input samples `DeltaOut` output samples are written. Output samples that fall between input samples are computed by linear interpolation between the two nearest neighbors.

The parameters for `Interpolate` are:

- `DeltaIn`: input samples for each `DeltaOut` outputs.

  `DeltaIn` is the number of input samples for `DeltaOut` output samples.

  `DeltaIn` is of type `int16`. The default value is 1. `DeltaIn` must be $\geq$ 1 and $\leq$ 32767.

- `DeltaOut`: output samples for each `DeltaIn` inputs.

  `DeltaOut` is the number of output samples generated from `DeltaIn` input samples.

  `DeltaOut` is of type `int16`. The default value is 1. `DeltaOut` must be $\geq$ 1 and $\leq$ 32767.

3.12 `MaskWord`

Synopsis: applies a mask to a binary data stream.

`MaskWord` copies its input to its output after applying a the mask `Mask` to each sample.

The parameter for `MaskWord` is:

- `Mask`: mask to be be applied to binary data stream.

  `Mask` will be applied to each input sample to create an output sample.

  `Mask` is of type `int32`. The default value is 0. `Mask` must be $\geq$ -2147483647 and $\leq$ 2147483647.

3.13 `Mux`

Synopsis: Multiplexes `Channels` inputs into 1 output channel.

`Mux` takes `Channels` input channels and multiplexes these onto a single output channel. Each input channel must have samples consisting of `InputSampleSize` words. For example, complex data streams will generally have two words in each sample. The output channel will have samples of `OutputSampleSize` words.

The parameters for `Mux` are:

- `Channels`: number of input channels.

  `Mux` combines `Channels` input channels into a single output channel.

  `Channels` is of type `int16`. The default value is 2. `Channels` must be $\geq$ 2 and $\leq$ 32767.

- `InputSampleSize`: number of consecutive words in one sample.

  For each input channel it is assumed that a single sample is made up of `InputSampleSize` words.

  `InputSampleSize` is of type `int32`. The default value is 1. `InputSampleSize` must be $\geq$ 1 and $\leq$ 2147483647.

- `OutputSampleSize`: number of consecutive words in input one sample.

The number of words in an output channel sample is `OutputSampleSize`
.

`OutputSampleSize` is of type `int32`. The default value is 2. `OutputSampleSize` must be ≥ 1 and ≤ 2147483647.

- `MinimumChunk`: minimum number out input samples to process at once.

  `MinimumChunk` can be set to a minimum number of input samples to be processed. This allows for greater efficiency but limits the degree to which data can be flushed.

  This is the first default parameter. `MinimumChunk` is of type `int16`. The default value is 1. `MinimumChunk` must be ≥ 1 and ≤ 32767.

3.14 PackWord

Synopsis: packs multiple input words to a single output word.

`PackWord` packs the least significant `InputWordSize` bits of `InputsPerOutput` consecutive input words to a single output word. The first input word is placed in the least significant output bits. The physical word size must be at least as large as `InputWordSize` × `InputsPerOutput`. If it is not an error is generated and the node will not be usable. The inverse of this operation is `UnpackWord`. `RepackStream` is a a more general but less efficient operation that is its own inverse.

The parameters for `PackWord` are:

- `InputWordSize`: size in bits to pack from each input.

  `InputWordSize` is the number of bits in each input word to pack into the output word.

  `InputWordSize` is of type `int16`. The default value is 8. `InputWordSize` must be ≥ 1 and ≤ 16.

- `InputsPerOutput`: number or input words to pack into one output.

`InputsPerOutput` is the number of input words combined to form a single output word.

`InputsPerOutput` is of type `int16`. The default value is 2. `InputsPerOutput` must be ≥ 2 and ≤ 32.

3.15 Power

Synopsis: Power computes and scales the power in each sample.

`Power` computes the sum of the squares of each input sample element. Before summing the power in each channel is multiplied by a linear scale factor `Scale`. The most common use is to compute the power of a complex signal. The linear scaling and the summation can cause an arithmetic overflow. Overflows cannot occur in the squaring operation (with integer arithmetic) because double precision integer arithmetic is used. Floating point overflows generate an interrupt and end node execution. Integer overflows are prevented by clipping. The first time clipping occurs a help message is generated. A new help message is generated after every 400 clippings.

The parameters for `Power` are:

- `Amplitude`: flag select amplitude(1) or power(0).

  If `Amplitude` is set to 1, the output will be the square root of the power and not the power.

  `Amplitude` is of type `int16`. The default value is 0. `Amplitude` must be ≥ 0 and ≤ 1.

- `Scale`: scale factor applied before summing powers.

  `Scale` is a linear scale factor applied before summing the squared elements in a sample. For integer arithmetic, `Scale` should be set to prevent overflows. Note the squaring operation (in the integer arithmetic model) is done in double precision integer arithmetic. Thus if one is taking the amplitude as the final output (`Amplitude` =1), overflows can only occur from the summation step. If integer overflows do occur the output signal is clipped. The first time clipping occurs, a help message is generated. A new help message is generated after every 400 clippings.

This is the first default parameter. `Scale` is of type `double`. This parameter can be changed interactively. The default value is 1.0. `Scale` must be $\geq$ -1.e100 and $\leq$ 1.e100.

3.16 `RealFir`

Synopsis: `RealFir` is a real symmetric (even or odd) fir filter.

`RealFir` implements a symmetric FIR filter. Input samples can have `ZeroPad` 0's interpolated between them. Output sample can be generated for every `Resample` input (or zero pad) sample. The `Coeff` array contains half of the symmetric coefficients. `Odd` indicates if there are an even(0) or odd(1) total number of coefficients.

The parameters for `RealFir` are:

- `Resample`: filter resampling factor (input rate/output rate).

  `Resample` specifies the filter resampling factor. This is the ratio of the input sampling rate to the output sampling rate.

  `Resample` is of type `int16`. The default value is 1. `Resample` must be $\geq$ 1 and $\leq$ 32767.

- `ZeroPad`: zero padding of input.

  `ZeroPad` specifies the number of 0's that are added after each input sample.

  This is the first default parameter. `ZeroPad` is of type `int16`. The default value is 0. `ZeroPad` must be $\geq$ 0 and $\leq$ 32767.

- `Odd`: if set the filter length is odd.

  `Odd` determines if the filter is if odd (`Odd` =1) or even (`Odd`=0).

  This is the first default parameter. `Odd` is of type `int16`. The default value is 0. `Odd` must be $\geq$ 0 and $\leq$ 1.

- `Coeff`: list of unique coefficients.

  `Coeff` is the list of filter coefficients. The filter is symmetric and only half (or half plus one for odd length filters) are specified. The first in the list is the first coefficient of the filter. The middle coefficient is at the end of the list. The default values for the coefficients define a low pass FIR filter with a pass band of .125 times the total bandwidth and transition band of .375 times the bandwidth. The pass band is extremely flat and the stop band is down over 100 db. This is an overdesigned filter for most practical applications but it provides a good test case. The performance will be degraded by 16 bit integer arithmetic.

  This is the first default parameter. `Coeff` is of type `MachWord` ×. `Coeff` is an array of size ≥ 3 and ≤ 1024. The following table contains the default array values.

1.00018552e-04	3.70747893e-04	4.46594395e-04	-5.36969535e-04
-2.67492760e-03	-3.52687595e-03	8.25715193e-04	1.01494638e-02
1.50874056e-02	2.48008436e-03	-2.76876913e-02	-4.93498540e-02
-2.20594765e-02	7.30790837e-02	2.04010323e-01	2.99286359e-01

3.17 `RepackStream`

Synopsis: repack bit streams to different physical word sizes.

`RepackStream` repacks a bit stream from an input physical word size, `InputWordSize` to an output physical word size, `OutputWordSize`. If `SignedOutput` is set the output will be written as a signed two's compliment value in the full physical word size. Both input and output are treated as continuous bit streams made up of physical words of the specified number of bits. The least significant physical bits occur logically first in the continuous stream. Logical words can cross physical word boundaries on both the input and output streams. One can use this to repack bit streams containing logical words of any length to a different physical word size. One can also use it to unpack a continuous stream of bits of any logical word size (up to 32 bits) to a stream of one physical output word for each each logical input word. Similarly on can pack a stream of physical words into a continuous bit stream, By setting `OutputWordSize` to the logical word size in the continuous input bit stream you unpack that stream placing one logical input word in each physical output word. By setting `InputWordSize` to the input logical word size you pack an input stream that has one logical word in each physical word into a continuous output bit stream. Neither `InputWordSize` and `OutputWordSize` can be greater then the physical word size. If either is the node will not be usable. If `InputWordSize` is an exact multiple of `OutputWordSize` (or vice versa) `UnpackWord` (`PackWord`) is a more efficient process that does the same operation.

The parameters for `RepackStream` are:

- `OutputWordSize`: size in bits of output word.

  `OutputWordSize` is the number of bits in each output word. The output is treated as a continuous stream of bits made up of the least significant `OutputWordSize` bits from each physical output word.

  `OutputWordSize` is of type `int16`. The default value is 8. `OutputWordSize` must be ≥ 1 and ≤ 32.

- `InputWordSize`: size in bits of input word.

  `InputWordSize` is the number of bits in the input word. The input is treated as a continuous stream of bits made up of the least significant `OutputWordSize` bits from each physical output word.

  `InputWordSize` is of type `int16`. The default value is 1. `InputWordSize` must be ≥ 1 and ≤ 32.

- `SignedOutput`: option(1) to treat output as signed two's compliment value.

  If `SignedOutput` is set the output will be written as a signed two's compliment value in the full physical word size.

  `SignedOutput` is of type `int16`. The default value is 0. `SignedOutput` must be ≥ 0 and ≤ 1.

3.18 `SampleDelay`

Synopsis: delays the output by a selected number of samples.

`SampleDelay` initially copies `Delta` samples (with all components set equal to `FillValue`) to its output. After that it simply copies its input to its output. This results in a delay if `Delta` samples.

The parameters for `SampleDelay` are:

- `Delta`: signal delay in samples.

  `Delta` specifies the number of samples the output signal will be delayed relative to the input.

  `Delta` is of type `int32`. The default value is 1. `Delta` must be ≥ 0 and ≤ 2147483647.

- `FillValue`: value to output before input data is copied.

  `FillValue` is the value to be output for each sample component for the first `Delta` samples.

  This is the first default parameter. `FillValue` is of type `MachWord`. The default value is 0. `FillValue` must be $\geq$ DEF_MACH_WORD_MIN and $\leq$ DEF_MACH_WORD_MAX.

3.19 `ToInteger`

Synopsis: converts MachWord data stream to integer.

`ToInteger` converts a MachWord data stream to integer format. The inverse operation can be done with node 'ToMachWord'. This class is not available with the 16 bit arithmetic version of ObjectProDSP. Saturated output is produced on overflow. Overflows are not reported.

This node has no parameters.

3.20 `ToMach`

Synopsis: converts binary data stream to `MachWord`.

`ToMach` converts a binary data stream to type `MachWord`. If `SignedConversion` is set the input will be treated as two's compliment signed values. Otherwise it will be treated as unsigned. The inverse operation is `ToInteger`. `ToInteger` and `ToMach` are meaningless and not available on the 16 bit integer simulator.

The parameter for `ToMach` is:

- `SignedConversion`: conversion of unsigned(0) or signed(1) integer input to `MachWord`.

  If `SignedConversion` is set the input integer will treated as a two's compliment signed value. Otherwise it will be considered unsigned.

  `SignedConversion` is of type `int16`. The default value is 0. `SignedConversion` must be ≥ 0 and ≤ 1.

3.21 Truncate

Synopsis: Limit the dynamic range and significant bits in a stream.

`Truncate` converts the MachWord input data to a 32 bit integer (after checking for overflow). On the floating point simulator the input can also be 32 bit integers. The dynamic range is then restricted to `Range`. The number of significant bits is restricted to `Accuracy`. Neither `Range` nor `Accuracy` include the sign bit. Although integers are represented is twos compliment format truncation can be thought of as working on signed magnitude data. Overflow will saturate if `OverflowMode` is 0 and cause the high order bits to be ignored if `OverflowMode` is 1. The output is converted back to MachWord format.

The parameters for `Truncate` are:

- `Range`: Bits for dynamic range.

  `Range` is the number of bits in the dynamic range of the output (not counting the sign bit).

`Range` is of type `int16`. This parameter can be changed interactively. The default value is 31. `Range` must be ≥ 1 and ≤ 31.

- `Accuracy`: Significant bits retained in output.

  `Accuracy` is the number of significant bits retained in the output (not counting the sign bit). `Accuracy` will always be <= `Range`. If you specify more accuracy then range accuracy will equal range.

  `Accuracy` is of type `int16`. This parameter can be changed interactively. The default value is 31. `Accuracy` must be ≥ 1 and ≤ 31.

- `OverflowMode`: For overflows: 0 - saturate, 1 - truncate.

  `OverflowMode` selects saturation (0) or truncation (1) mode. In saturation mode an overflow is replaced by the largest positive or negative number representable depending on the sign of the input value. In Truncation mode the high order bits are truncated as if the number was in sign magnitude form.

  `OverflowMode` is of type `int16`. This parameter can be changed interactively. The default value is 0. `OverflowMode` must be ≥ 0 and ≤ 1.

- `Round`: Flag indicates rounding (1) or truncation (0).

  `Round` if set rounds the result. Otherwise it is truncated. Truncation is always towards 0 and not two's compliment truncation.

  `Round` is of type `int16`. This parameter can be changed interactively. The default value is 0. `Round` must be ≥ 0 and ≤ 1.

3.22 UnpackWord

Synopsis: unpack a single input word to multiple output words.

`UnpackWord` unpacks each input word to `OutputsPerInput` output words each containing `OutputWordSize` bits. If `SignedOutput` is set the output will be written as a signed two's compliment value in the full physical word size. The least significant bits of each input word are written first. The physical word size must be at least as large as `OutputWordSize` × `OutputsPerInput`

. If not an error will be generated and the node will be unusable. The inverse of this operation is `PackWord`. `RepackStream` is a more general but less efficient operation that is its own inverse.

The parameters for `UnpackWord` are:

- `OutputWordSize`: size in bits of output word.

  `OutputWordSize` is the number of bits in the output word.

  `OutputWordSize` is of type `int16`. The default value is 8. `OutputWordSize` must be ≥ 1 and ≤ 16.

- `OutputsPerInput`: number or words to unpack from each input.

  `OutputsPerInput` is the number of output words unpacked from each input word.

  This is the first default parameter. `OutputsPerInput` is of type `int16`. The default value is 2. `OutputsPerInput` must be ≥ 2 and ≤ 32.

- `SignedOutput`: option(1) to treat output as signed two's compliment value.

  If `SignedOutput` is set the output will be written as a signed two's compliment value in the full physical word size.

  This is the first default parameter. `SignedOutput` is of type `int16`. The default value is 0. `SignedOutput` must be ≥ 0 and ≤ 1.

4 Input and signal generation nodes

The nodes in this secion have no input channels. They are either signal generators or they read data from a disk file or other source external to ObjectProDSP.

4.1 `ConstantData`

Synopsis: generate a 'MachWord' constant.

`ConstantData` writes parameter `Value` to the output stream repeatedly. It is written as a binary integer constant.

The parameter for `ConstantData` is:

- `Value`: constant output level.

  `Value` is output as an an 'MachWord' constant.

  `Value` is of type `int32`. This parameter can be changed interactively. The default value is 1024. `Value` must be ≥ -2147483647 and ≤ 2147483647.

4.2 Cos

Synopsis: Generates the real function `Amplitude` cos(`Phase` + N `Frequency`).

`Cos` generates the sampled real function: `Amplitude` cos(`Phase` + N `Frequency`). N is the sample index that starts with N = 0.

The parameters for `Cos` are:

- `Frequency`: frequency (in radians per sample).

  `Frequency` specifies the signal frequency in radians per sample. In other words the phase of a given sample is `Frequency` radians plus the phase of the previous sample.

  `Frequency` is of type `double`. This parameter can be changed interactively. The default value is 0.125663706144. `Frequency` must be ≥ -3.1416 and ≤ 3.1416.

- `Phase`: phase (in radians) of first sample.

  `Phase` specifies the initial phase of the first sample of the signal.

  This is the first default parameter. `Phase` is of type `double`. This parameter can be changed interactively. The default value is 0.0. `Phase` must be ≥ -6.2832 and ≤ 6.2832.

- `Amplitude`: absolute maximum amplitude of the continuous cosine function.

  `Amplitude` specifies the maximum amplitude of the continuous cosine function. This may not be the maximum amplitude of the samples generated. If the function is sampled at a phase that is an integer multiple of Pi, then the samples will obtain this maximum.

  This is the first default parameter. `Amplitude` is of type `double`. This parameter can be changed interactively. The default value is 1024. `Amplitude` must be $\geq$ -1.e100 and $\leq$ 1.e100.

4.3 `CxCos`

Synopsis: Generates the function `Amplitude` e∧(2 Pi i(`Phase` + N `Frequency`)).

`CxCos` generates the sampled complex function: `Amplitude` e∧(2 Pi i(`Phase` + N `Frequency`)). N is the sample index. It starts at 0.

The parameters for `CxCos` are:

- `Frequency`: frequency (in radians per sample).

  `Frequency` specifies the signal frequency in radians per sample. In other words the phase of a given sample is `Frequency` radians plus the phase of the previous sample.

  `Frequency` is of type `double`. This parameter can be changed interactively. The default value is 0.125663706144. `Frequency` must be $\geq$ -3.1416 and $\leq$ 3.1416.

- `Phase`: phase (in radians) of first sample.

  `Phase` specifies the initial phase of the first sample of the signal.

  This is the first default parameter. `Phase` is of type `double`. This parameter can be changed interactively. The default value is 0.0. `Phase` must be $\geq$ -6.2832 and $\leq$ 6.2832.

- `Amplitude`: absolute maximum amplitude of the continuous cosine function.

`Amplitude` specifies the maximum amplitude of the continuous cosine function. This may not be the maximum amplitude of the samples generated. If the function is sampled at a phase that is an integer multiple of Pi, then the samples will obtain this maximum.

This is the first default parameter. `Amplitude` is of type `double`. This parameter can be changed interactively. The default value is 1024. `Amplitude` must be $\geq$ -1.e100 and $\leq$ 1.e100.

4.4 CxImp

Synopsis: Generates a periodic impulse or square wave.

`CxImp` generates a periodic impulse or square every `Period` samples. The impulse amplitude is `Amplitude` e$\wedge$(2 Pi i `Phase`). The first transition for 0 to this amplitude occurs at sample `Transition`. The nonzero amplitude is maintained for `Width` $\times$ `Period` samples where `Width` is between 0 and 1. If `Width` $= 1.0$ then the signal is a constant after the first transition.

The parameters for `CxImp` are:

- `Period`: the impulse is repeated after period samples.

  `Period` specifies the number of samples before the impulse is repeated.

  `Period` is of type `int32`. The default value is 32. `Period` must be ≥ 2 and ≤ 2147483647.

- `Phase`: phase (in radians) of first sample.

  `Phase` the relative amplitude of the real and imaginary components of the signal. With `Phase` $= 0$ all the energy is in the real part. With `Phase` $=$ pi/2 all the energy is in the imaginary part.

  This is the first default parameter. `Phase` is of type `double`. The default value is 0.0. `Phase` must be $\geq$ -6.2832 and $\leq$ 6.2832.

- `Amplitude`: magnitude of impulse amplitude.

`Amplitude` specifies the the magnitude of the impulse amplitude. It is the square root of the sum of the squares of the real and imaginary amplitudes.

This is the first default parameter. `Amplitude` is of type `double`. The default value is 1024. `Amplitude` must be $\geq$ -1.e100 and $\leq$ 1.e100.

- `Width`: peak width as a fraction of sample period (0=>impulse).

  `Width` specifies the peak width as a fraction of sample period. `Width` = 0 produces an impulse one sample wide. `Width` = 1 results in a constant amplitude and phase signal. `Width` = .5 results in a standard square wave.

  This is the first default parameter. `Width` is of type `double`. The default value is .5. `Width` must be $\geq$ 0.0 and $\leq$ 1.0.

- `Transition`: sample index where the first transition occurs.

  `Transition` specifies the sample index where the first signal transition from 0 occurs. This may be longer than the sample `Period` .

  This is the first default parameter. `Transition` is of type `int32`. The default value is 0. `Transition` must be $\geq$ 0 and $\leq$ 2147483647.

4.5 ImportData

Synopsis: `ImportData` reads an ascii input file.

`ImportData` reads ascii data in a variety of formats. Selected columns can be be ignored. Each line is broken into fields. A field is a string that has a numeric value. These fields are separated by any characters that do not have an embedded numeric value. Selected fields can be ignored.

The parameters for `ImportData` are:

- `FileName`: name of ascii file to read.

  `FileName` specifies the name of the disk file to be read. If no file name is specified (default 0) you will be prompted for a file name when execution starts.

`FileName` is of type `const char *`. The default value is 0.

- `Format`: ‘C’ format to read data values.

  `Format` specifies the format to use in reading data. Any standard ‘C’ input format can be used. The default, ‘%d’ for integer decimal data, for octal data. If the format string ends in ‘X’ or ‘x’ integer data is written to the output stream on a floating point simulator. If the last character is an ‘s’ and the format is a floating point format the data will be normalized on the 16 bit integer simulator, i. e. .5 will be converted to 16384 or half of full scale. If the last two characters of the format string are an underscore (_) and any other character they will be deleted from the format. You can use this to control the type of output or scaling independent of the format for reading data. The type of format is determined by looking for the first occurrence of ‘%’ ad then the first occurrence of one the letters ‘x’, ‘d’, ‘o’ ,‘f’ or ‘e’ after that. Floating point format characters must be preceded by a ‘l’ and others must not. The letter can be in either upper or lower case. If the format is not recognized and error will be generated. ‘x’ and ‘o’ formats will read data as unsigned 32 bit integers. ‘d’ will it as signed 32 bit integers. ‘e’ and ’f’ will read it as a double floating point value. Overflows will be reported as warnings.

  This is the first default parameter. `Format` is of type `const char *`. The default value is “%d”. `Format` must be legal in this context.

- `Fields`: number of data fields on each line.

  `Fields` specifies the number of data fields on each line. If set to 0 then all data fields that are found on a line will be read (up to a maximum line width of 1024 characters.) A data field is any contiguous string of legal digits separated by white space (blank, tab, the beginning of a line or the end of a line) from other data fields or other information on the line. Decimal and octal fields can start with a ‘+’ or ‘-’. Hexadecimal fields may start with an optional ‘0x’ or ‘0X’ provided the `Format` string is ‘%x’ or ‘%X’. The value `Fields` is an upper limit on the fields on a line. There may be fewer fields on a line and even lines with no valid numeric fields.

  This is the first default parameter. `Fields` is of type `int16`. The default value is 1. `Fields` must be ≥ 0 and ≤ 1024.

- `RepeatFlag`: flag to read the file repeatedly.

  Setting `RepeatFlag` causes the file to be read at the beginning once the end of file is encountered. If the file is short it will only be read once and the data will be retained in memory.

  This is the first default parameter. `RepeatFlag` is of type `int16`. The default value is 0. `RepeatFlag` must be ≥ 0 and ≤ 1.

- `SkipFields`: list of fields to skip (enter single 0 to skip none).

  `SkipFields` is a list of fields (in increasing order) to skip. Fields start at 1. If all values are 0 no fields are skipped.

  This is the first default parameter. `SkipFields` is of type `int32 *`. `SkipFields` is an array of size ≥ 1 and ≤ 1024. The following table contains the default array values.

  0

  `SkipFields` must be legal in this context.

- `SkipColumns`: list of pairs of columns to skip (enter single 0 to skip none.

  `SkipColumns` is a list of pairs of column numbers in increasing order. All data in columns starting at the first column number in the pair and ending at the next column number after the second element in the pair will be ignored. Thus the list {20, 24, 30, 32} would cause the five columns 20 through 24 and the three columns 30 through 32 to be skipped. If there are an odd number of entries all columns at or following the last entry will be skipped. Columns start at 1. If only values of 0 are entered no columns will be skipped.

  This is the first default parameter. `SkipColumns` is of type `int32 *`. `SkipColumns` is an array of size ≥ 1 and ≤ 1024. The following table contains the default array values.

  0

  `SkipColumns` must be legal in this context.

4.6 `InputNode`

Synopsis: `InputNode` reads a disk file written by an `OutputNode`.

`InputNode` reads input from disk file `FileName` . The number of channels and the number of multiplexed elements in a sample are read from the header of the disk file. Member function `DisplayHeader` will give the values of all the information read from the disk file header.

The parameters for `InputNode` are:

- `FileName`: name of disk file to read.

  `FileName` specifies the name of the disk file to be read. If no file name is specified (default 0) then the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

- `Flags`: flags for arithmetic type and other option.

  If `Flags` & 4 is set the sample rate is forced to 1, overwriting the default values.

  This is the first default parameter. `Flags` is of type `int16`. The default value is 0. `Flags` must be $\geq$ -32767 and $\leq$ 32767.

- `DeltaOut`: minimum output chunk size.

  `DeltaOut` is the minimum output size. The node is not scheduled until space for `DeltaOut` words is available in the output buffer. It will only write multiples of `DeltaOut` samples.

  This is the first default parameter. `DeltaOut` is of type `int32`. The default value is 1. `DeltaOut` must be $\geq$ 1 and $\leq$ 65536.

Following are the member functions of this node.

`DisplayHeader`

Synopsis: display the caption and parameters read from the disk.

This function returns type `void`.

`DisplayHeader` displays the parameters read from file `FileName`. These include the original node name that generated the file, the caption for this node, the number of input channels, and the number of scalar elements in a sample. The arithmetic type is also shown. The output channels and sample size for this node are determined by these values. If the data in the file is in a different arithmetic format than that currently in use, the file data will be converted.

This function has no parameters.

`IgnoreHeaderCount`

Synopsis: ignore the word counts in the data file header.

This function returns type `void`.

The data file header contains a count of the number of machine words in each channel. This count is written at the time the node creating the file is deleted. If ObjectProDSPexits abnormally then these counts may never be set and one will not be able to read any of the data in the file. This option causes these counts to be ignored. The result is that data will be read until the physical end of file. This may result in samples of all 0 being read at the end of the file that were never written to it.

This function has no parameters.

4.7 InputWord

Synopsis: `InputWord` reads words in a selected format from a binary file.

`InputWord` reads words from a binary input file. It is intended for importing data to other processes. `OutputWord` writes files that can be read by `InputWord` . However `OutputNode` and `InputNode` are better for preserving data for later use in ObjecProDSP because these retain sample rate, timing, block size and sample size information. You can manually specify the sample and block size of the data in the file with parameters `ElementSize` and `BlockSize`. Each output word is read in a format determined by `FormatIn`. The options are: `MachWord` (0), `int8`(1), `int16`(2), `int32`(3), `float`(4), `double` (5). Integer words are assumed in two's compliment format. You can skip a header or initial segment of a file by setting `InitialSkip` to the number bytes to skip. `IntegerOut` is nonzero the output stream is written as `IntegerMachWord`, otherwise it is written as `MachWord`. Integer output values (`IntegerMachWord`) are treated as signed. If overflow occurs the data is hard limited and a warning is given. The output stream can have any value for sample size, (`ElementSize`) or `BlockSize` but these are not preserved in the output file as they are with `InputNode`.

The parameters for `InputWord` are:

- `FileName`: binary input file to read.

  `FileName` is the binary input file to read. If no default(0) is given the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

- `FormatIn`: format: `MachWord`(0), `int8`(1), `int16`(2), `int32`(3), `float`(4), `double`(5).

  `FormatIn` is the binary input format. The options are: `MachWord`(0), `int8`(1), `int16`(2), `int32`(3), `float` (4), `double`(5). Integer words are written in two's compliment format. Integer input values (`IntegerMachWord`) are treated as signed. If overflow occurs the data is hard limited and a warning is given. The input stream can have any value for sample size, (`ElementSize`)

  This is the first default parameter. `FormatIn` is of type `int16`. The default value is 0. `FormatIn` must be ≥ 0 and ≤ 5.

- `IntegerOut`: output format `MachWord`(0) or `IntegerMachWord`(1).

  If `IntegerOut` is nonzero the output stream is written as `IntegerMachWord`. Otherwise it is written as `MachWord`. If overflow occurs the data is hard limited and a warning is given.

This is the first default parameter. `IntegerOut` is of type `int16`. The default value is 0. `IntegerOut` must be ≥ 0 and ≤ 1.

- `InitialSkip`: bytes to skip at start of file.

  The first `InitialSkip` bytes of the file are ignored.

  This is the first default parameter. `InitialSkip` is of type `int32`. The default value is 0. `InitialSkip` must be ≥ 0 and ≤ 2147483647.

- `ElementSize`: number of words per sample.

  `ElementSize` is the number of words per sample in the input file.

  This is the first default parameter. `ElementSize` is of type `int16`. The default value is 1. `ElementSize` must be ≥ 1 and ≤ 32767.

- `BlockSize`: number of samples per block.

  `BlockSize` is the number of samples per block in the input file.

  This is the first default parameter. `BlockSize` is of type `int32`. The default value is 1. `BlockSize` must be ≥ 1 and ≤ 2147483647.

4.8 `Normal`

Synopsis: Generate normally distributed noise samples.

`Normal` generates noise samples from a normal distribution. Let Z be samples from the standard normal distribution. This process generates samples as `Mean` + `Sigma` × Z. A vector of `ElementSize` samples is generated. `ElementSize` is ordinarily set to 1 for real data and 2 for complex data. `Seed` determines the sequence generated. Each object instance maintains it a separate state for the random number generator.

The parameters for `Normal` are:

- `Sigma`: standard deviation of normally distributed data.

  `Sigma` specifies the standard deviation of the normally distributed samples created by this generator. `Sigma` is a scale factor for values generated in the standard normal distribution.

`Sigma` is of type `double`. This parameter can be changed interactively. The default value is 1024.0. `Sigma` must be $\geq$ 0.0 and $\leq$ 1.e100.

- `Mean`: mean of normally distributed data.

  `Mean` specifies the mean of the normally distributed samples created by this generator. `Mean` is an offset for the values generated in the standard normal distribution.

  This is the first default parameter. `Mean` is of type `double`. This parameter can be changed interactively. The default value is 0.0. `Mean` must be $\geq$ -1.e100 and $\leq$ 1.e100.

- `ElementSize`: number of words in one sample.

  `ElementSize` specifies the number of words in a single sample. It is most commonly 1 for real data or 2 for complex data.

  This is the first default parameter. `ElementSize` is of type `int16`. The default value is 1. `ElementSize` must be $\geq$ 1 and $\leq$ 32767.

- `Seed`: random number generator seed.

  `Seed` seeds the random number generator. Each object instance maintains a separate history state for the random number generator. If the same value of `Seed` is used in different object instances they will generate the same sequence.

  This is the first default parameter. `Seed` is of type `int32`. The default value is 1. `Seed` must be $\geq$ 0 and $\leq$ 2147483647.

4.9 Ramp

Synopsis: generates a linear ramp function.

`Ramp` generates a linear ramp function with the initial value of `Min`, and increment between samples of `Increment` and upper bound on the output of `Max`. At the point where the next output would exceed `Max` it is reset to `Min`.

The parameters for `Ramp` are:

- `Min`: minimum value of ramp sample.

  `Min` is the minimum and initial value of the ramp function.

  `Min` is of type `int32`. This parameter can be changed interactively. The default value is 0. `Min` must be ≥ -2147483647 and ≤ 2147483647.

- `Max`: maximum value of ramp sample.

  `Max` is an upper bound on the ramp function. Before exceeding `Max` a sample will be reset to `Min`.

  `Max` is of type `int32`. This parameter can be changed interactively. The default value is 0. `Max` must be ≥ -2147483647 and ≤ 2147483647.

- `Increment`: increment between samples.

  `Increment` is the amount added to the previous sample to generate the next sample.

  `Increment` is of type `int32`. This parameter can be changed interactively. The default value is 1. `Increment` must be ≥ 1 and ≤ 2147483647.

4.10 ReadFloat

Synopsis: `ReadFloat` reads an ascii float input file.

`ReadFloat` reads a file of floating point ascii formatted values and writes them to a data stream of type ‘MachWord’. You can have as many values as you want on each line but there can only be white space as defined in the C++ input stream functions between numbers.

The parameter for `ReadFloat` is:

- `FileName`: ascii file to read.

  `FileName` specifies the disk file to be read. If no name is specified the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

4.11 `ReadInt`

Synopsis: `ReadInt` reads an ascii integer input file.

`ReadInt` read a file containing ascii formatted integers and writes the values to a `MachWord` data stream. If `Flags`&2 is set on a 32 bit simulator 32 bit integer values will be written. You can have as many values is you want on each line but there can only be white space as defined in the C++ input stream functions between numbers. `ImportData` can process more complex input files. If `Flags`&1 is set a hexadecimal input will be read as unsigned integers and converted as needed.

The parameters for `ReadInt` are:

- `FileName`: ascii file to read.

  `FileName` specifies the the disk file to be read. If no name is specified the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

- `Flags`: flag for hex format input (1) and signed integer output (2).

  `Flags`&1 specifies hex format file with optional 0x or 0X prefix for each value. `Flags`&2 will write 32 bit integer data on a floating point simulator. Decimal format files are read as signed integers and hexadecimal format as unsigned.

  This is the first default parameter. `Flags` is of type `int16`. The default value is 0. `Flags` must be ≥ 0 and ≤ 3.

4.12 `UniformNoise`

Synopsis: Generate uniformly distributed noise samples.

`UniformNoise` generates noise samples of `ElementSize` elements from a uniform distribution. Set `ElementSize` to 1 for real data or 2 for complex. The values generated range between `Minimum` and `Maximum`. `Seed` determines the sequence generated. Each object instance maintains it a separate state for the random number generator.

The parameters for `UniformNoise` are:

- `Maximum`: largest value of uniformly distributed noise.

  `Maximum` is the largest value of uniformly distributed noise.

  `Maximum` is of type `double`. This parameter can be changed interactively. The default value is 1024.0. `Maximum` must be $\geq$ -1.e100 and $\leq$ 1.e100.

- `Minimum`: smallest value of uniformly distributed noise.

  `Minimum` is the smallest value of uniformly distributed noise. Most commonly `Minimum` = - `Maximum`.

  `Minimum` is of type `double`. This parameter can be changed interactively. The default value is -1024.0. `Minimum` must be $\geq$ -1.e100 and $\leq$ 1.e100.

- `ElementSize`: number of words in one sample.

  `ElementSize` specifies the number of words in a single sample. It is most commonly 1 for real data or 2 for complex data.

  This is the first default parameter. `ElementSize` is of type `int16`. The default value is 1. `ElementSize` must be $\geq$ 1 and $\leq$ 32767.

- `Seed`: random number generator seed.

  `Seed` seeds the random number generator. Each object instance maintains a separate history state for the random number generator. If the same value of `Seed` is used in different object instances they will generate the same.

  This is the first default parameter. `Seed` is of type `int32`. The default value is 1. `Seed` must be $\geq$ 0 and $\leq$ 2147483647.

4.13 `VoiceNode`

Synopsis: `VoiceNode` reads ‘Creative Voice’ format files.

`VoiceNode` reads 'Creative Voice' format files. Only files containing uncompressed sampled data are supported. This format has one sample per each byte in offset format, i. e. 128 is subtracted from each unsigned byte to create a signed sample. In the 16 bit integer simulator you will probably need to rescale the data (most likely by dividing by 128) before writing it in this format. Member function `DisplayHeader` displays the file header. If `NoHeader` is set then no header is read. Node `VoiceStripOut` writes files in this format.

The parameters for `VoiceNode` are:

- `FileName`: input file in 'Creative Voice' file format.

  `FileName` must be the name of an existing file in the Creative Voice File Format. Only uncompressed files of block type 1 (New Voice Block) are supported.

  `FileName` is of type `const char *`. The default value is 0.

- `NoHeader`: file does(0) or does not(1) contain a header.

  `NoHeader` if set to 1 indicates the file does not contain a header. `VoiceStripOut` writes files in Creative Voice format files with no header.

  `NoHeader` is of type `int16`. The default value is 0. `NoHeader` must be ≥ 0 and ≤ 1.

Following is the member function of this node.

`DisplayHeader`

Synopsis: display file header.

This function returns type `void`.

`DisplayHeader` displays the information in the file header.

This function has no parameters.

5 Output and data display nodes

The nodes in this section have no output channels. They either display output, write it to a disk file or write to some other external device.

5.1 `AsciiFile`

Synopsis: `AsciiFile` writes an ascii file of data sent to it.

`AsciiFile` writes an ascii file of the data sent to it. By default the data is grouped as one sample per line with the words in each sample separated by a space. If blocks are larger than one sample they are grouped in side curly brackets '{}'. This formatting information can be removed with the `NoGroup` option. The output format is accurate enough to fully represent the internal data value. `Hex` uses a hexadecimal format. (Values that do not fit in a 32 bit integer are hard limited and generate a warning with `Hex` format.) describes the data format and the data source. It can be omitted using the `NoHeader` option.

The parameters for `AsciiFile` are:

- `FileName`: output file name (defaults to node name).

  `FileName` specifies the output ascii file name head of the listing. If no caption is specified (default 0) then the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

- `Hex`: option to output data in hex format.

  `Hex`, when set, writes output in hexadecimal format. If the value will not fit in a 32 bit integer it is hard limited and a warning is generated.

  This is the first default parameter. `Hex` is of type `int16`. This parameter can be changed interactively. The default value is 0. `Hex` must be ≥ 0 and ≤ 1.

- `NoGroup`: option to not group complex elements on one line.

`NoGroup`, if set, writes data one word per line. The default is to write the words in a sample on a single line (up to 20 words per sample) and to put brackets '{}' around blocks if the block size is larger then the sample size. If `NoGroup` is set then the data is written one per line with no grouping information.

This is the first default parameter. `NoGroup` is of type `int16`. The default value is 0. `NoGroup` must be ≥ 0 and ≤ 1.

- `NoHeader`: option to omit data header.

  The default is to write a data header that describes the data format, the time it was created and gives the names of the creating node and network. If this option is set this header is omitted.

  This is the first default parameter. `NoHeader` is of type `int16`. The default value is 0. `NoHeader` must be ≥ 0 and ≤ 1.

5.2 CompareDisk

Synopsis: `CompareDisk` compares input to a file written by an `OutputNode`.

`CompareDisk` compares the contents of disk file `FileName` with data from its input channels. All discrepancies are reported in a text window. `CompareDisk`'s primary purpose is for regression testing. The number of channels and the number of multiplexed elements in a sample are read from the header of the disk file. Member function `DisplayHeader` will give the values of all the information read from the disk file header.

The parameters for `CompareDisk` are:

- `FileName`: name of disk file to read.

  `FileName` specifies the name of the disk file to be compared with the data read from the input channels.

  `FileName` is of type `const char *`. The default value is 0.

- `MaxReport`: maximum number of errors to report.

  Only the first `MaxReport` errors will be reported.

This is the first default parameter. `MaxReport` is of type `int32`. The default value is 1000. `MaxReport` must be ≥ 1 and ≤ 2147483647.

- `Tolerance`: absolute value of minimum difference for an error.

  `Tolerance` is the absolute value of the smallest difference that constitutes an error. Ordinarily this value is 0.0. It might be set to a value larger than 0 to compare slightly different algorithms or results on two different computers with different arithmetic.

  This is the first default parameter. `Tolerance` is of type `double`. The default value is 0.0. `Tolerance` must be ≥ 0.0 and $\leq$ 1.e100.

- `ErrorFile`: if set errors will be written to this file.

  `ErrorFile` is a file in which errors will be reported instead of displaying them in a window.

  This is the first default parameter. `ErrorFile` is of type `const char *`. The default value is 0.

Following are the member functions of this node.

`DisplayHeader`

Synopsis: display the caption and parameters read from the disk.

This function returns type `void`.

`DisplayHeader` displays the parameters read from file `FileName`. These include the original node name that generated the file, the caption for this node, the number of input channels, and the number of scalar elements in a sample. The arithmetic type is also shown. The output channels and sample size for this node are determined by these values. If the data in the file is in a different arithmetic format than that currently in use, the file data will be converted.

This function has no parameters.

`IgnoreHeaderCount`

Synopsis: ignore the word counts in the data file header.

This function returns type `void`.

The data file header contains a count of the number of machine words in each channel. This count is written at the time the node creating the file is deleted. If ObjectProDSPexits abnormally then these counts may never be set and one will not be able to read any of the data in the file. This option causes these counts to be ignored. The result is that data will be read until the physical end of file. This may result in samples of all 0 being read at the end of the file that were never written to it.

This function has no parameters.

5.3 `EyePlot`

Synopsis: `EyePlot` plots complex signal in eye plot (X versus Y) form.

`EyePlot` displays a complex signal on a single eye plot or X versus Y format. The real value of a sample determines the X coordinate and the imaginary value determines the Y coordinate.

The parameters for `EyePlot` are:

- `SamplesPerPlot`: the number of samples in one diaplay.

  `EyePlot` generates a series of plots with `SamplesPerPlot` in each display. If the input block size is not 1 then that value is overrides this parameter.

  This is the first default parameter. `SamplesPerPlot` is of type `double`. The default value is 400. `SamplesPerPlot` must be $\geq$ 3. and $\leq$ 1.e100.

- `Caption`: plot caption.

  The plot caption is a string that will be displayed at the base of the plot. The default value of 0 causes the plot node name to be used for the caption.

This is the first default parameter. `Caption` is of type `const char *`. The default value is 0.

5.4 HexList

Synopsis: `HexList` lists a specified number of channels to a display window.

`HexList` displays `Channels` signals on a single list. If you only want to display a single channel the 'HexList' option in 'Listing' may be more convenient. In a floating point simulator the input to `HexList` must be an integer format. You can use 'ToInteger' to do the conversion. 'BlockSize' and 'ElementSize' are set when the first input channel is linked. All subsequent input channels must have the same values.

The parameters for `HexList` are:

- `Channels`: number of input channels.

  `HexList` lists `Channels` signals of integer data in a hexadecimal format.

  `Channels` is of type `int16`. The default value is 1. `Channels` must be ≥ 1 and ≤ 32.

- `Caption`: list caption.

  `Caption` specifies a caption that will appear at the head of the listing. If no caption is specified (default 0) then the node name will be used. The caption cannot contain blanks. Use underscore instead.

  `Caption` is of type `const char *`. The default value is 0.

5.5 Listing

Synopsis: `Listing` lists a specified number of channels to a display window.

`Listing` displays its input data streams. The `ElementSize` words in each sample are enclosed in parenthesis and separated by commas. Each sample is given an index of one to three levels. These are the element within a block, the block index and the channel index. If the time flag is set then the time of each sample is listed. If `Hex` is set the data is displayed in hexadecimal format and hard limited if it cannot fit in a 32 bit integer. `Caption` is a label the listing window.

The parameters for `Listing` are:

- `Hex`: option(1) to output data in hex format.

  `Hex`, when set, displays data in hexadecimal format. If a value will not fit in a 32 bit integer it is hard limited.

  `Hex` is of type `int16`. This parameter can be changed interactively. The default value is 0. `Hex` must be ≥ 0 and ≤ 1.

- `Caption`: list caption.

  `Caption` specifies a caption that will appear at the head of the listing. If no caption is specified (default 0) then the node name will be used. The caption cannot contain blanks. Use underscore instead.

  `Caption` is of type `const char *`. The default value is 0.

5.6 `OutputNode`

Synopsis: `OutputNode` writes a specified number of channels to a disk file.

`OutputNode` writes `Channels` of input to disk file `FileName`. Each channel contains ‘ElementSize’ multiplexed elements. The data is blocked in as `FileBlockSize` consecutive samples from each channel. An `Caption` can be used to provide annotation of the file contents. If a reset is done after the output file is opened the last character of the file name is changed by adding 1 to it in the character sequence 0-9,A-Z,a-z. Once the last character ‘z’ is reached the next character is ‘0’. If the option is set not to overwrite a file then the open will fail if a file with the new name already exists. This cycling though file names allows testing of reset in a stand alone program that may need to process some data and then be reset to process the next input stream. cycle through the sequence 0-9,A-Z,-a-z or numeric that is the new file name. If not

The parameters for `OutputNode` are:

- `FileName`: name of disk file to create.

  `FileName` specifies the name of the disk file to be created. If no caption is specified (default 0) then the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

- `Flags`: flags to overwrite exiting file and other options.

  if bit 1 of `Flags` is set then an existing file with the same name is overwritten without comment. If the overwrite option is not set and a file with the same name exists this node will not be initialized properly in non interactive mode and network execution will fail. In interactive mode, you will be asked to supply a new name. if bit 8 is set (Flags&8) is true then the data will be converted from whatever format the input channel is to 32 bit integer data and written in that format. The data is hard limited.

  This is the first default parameter. `Flags` is of type `int16`. The default value is 1. `Flags` must be $\geq$ -32767 and $\leq$ 32767.

- `Channels`: number of input channels.

  `OutputNode` writes `Channels` of data to a disk file.

  This is the first default parameter. `Channels` is of type `int16`. The default value is 1. `Channels` must be ≥ 1 and ≤ 32.

- `FileBlockSize`: number of consecutive samples from one channel.

  `FileBlockSize` determines the number of consecutive samples written to each channel. Making this larger reduces the number of disk accesses at the cost of larger memory buffers.

  This is the first default parameter. `FileBlockSize` is of type `int32`. The default value is 128. `FileBlockSize` must be ≥ 1 and ≤ 16384.

- `Caption`: descriptive caption.

  `Caption` is a one line description of the contents of the file. It is displayed by executing a member function of an input node that reads this file.

This is the first default parameter. `Caption` is of type `const char *`. The default value is 0.

- `DeltaIn`: minimum input chunk size.

  The node will read chunks of multiples of `DeltaIn` samples. Note for complex data there are two words in each sample. The node will not be executed unless `DeltaIn` input samples are available.

  This is the first default parameter. `DeltaIn` is of type `int32`. The default value is 1. `DeltaIn` must be ≥ 1 and ≤ 16384.

5.7 OutputWord

Synopsis: `OutputWord` writes words in a selected format to a binary file.

`OutputWord` writes words to a binary file. It is intended for exporting data to other processes. `OutputNode` is better for preserving data for later use in ObjectProDSP because `Outputnode` retains sample rate, timing, block size and sample size information. Each input word is written in a format determined by `FormatOut`. The options are: `MachWord`(0), `int8`(1), `int16`(2), `int32`(3), `float` (4), `double`(5). Integer words are written in two's compliment format. Integer input values (`IntegerMachWord`) are treated as signed. If overflow occurs the data is hard limited and a warning is given. The input stream can have any value for sample size, (`ElementSize`) or `BlockSize` but these are not preserved in the output file as they are with `OutputNode`.

The parameters for `OutputWord` are:

- `FileName`: file to create.

  `FileName` is the file to be created. If no default(0) is given the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

- `FormatOut`: format: `MachWord`(0), `int8`(1), `int16`(2), `int32`(3), `float`(4), `double`(5).

`FormatOut` is the binary output format. The options are: `MachWord`(0), `int8`(1), `int16`(2), `int32`(3), `float` (4), `double`(5). Integer words are written in two's compliment format. Integer input values (`IntegerMachWord`) are treated as signed. If overflow occurs the data is hard limited and a warning is given. The input stream can have any value for sample size, (`ElementSize`)

`FormatOut` is of type `int16`. The default value is 0. `FormatOut` must be ≥ 0 and ≤ 5.

5.8 Plot

Synopsis: `Plot` creates graphs of real, complex and two dimensional data streams.

`Plot` displays signals in graphical form. The plot can be captioned by setting `Caption`. If `Caption` is the default (0) a caption is constructed from the plot node name and the name of its driver node. The data is auto-scaled. You can display it with a fixed scale and alternate between the default full scale display and a fixed scale. See the help file on 'change plot detail' accessible under 'Help' in the menu bar in any plot window.

The parameter for `Plot` is:

- `Caption`: plot caption.

  The `Caption` is displayed at the base of the plot. The default value of 0 causes the plot node name to be used for the caption. The caption cannot contain blanks. Use underscore instead.

  `Caption` is of type `const char *`. The default value is 0.

5.9 VoiceStripOut

Synopsis: `VoiceStripOut` writes a Creative Voice format file with no header.

`VoiceStripOut` writes its input to disk file `FileName` in Creative Voice file format but with no header. This format has one sample per each byte in offset format, i. e. 128 is added to each input sample. (After this sum any value < 0 is set to 0 and any value > 255 is set to 255 producing hard limit on overflows. In the 16 bit integer simulator you will probably need to rescale the data (most likely by dividing by 128) before processing it. Input can be `MachWord` or `IntegerMachWord`. (`IntegerMachWord` values are treated as signed integers.) Input have any value for `BlockSize` and sample size (`ElementSize`) but these values will not be recorded in the file as they are with `OutputNode`.

The parameter for `VoiceStripOut` is:

- `FileName`: file to create.

  `FileName` is the file to be created. If no default(0) is given the node name will be used.

  `FileName` is of type `const char *`. The default value is 0.

6 Base class nodes

The nodes in this section are base classes for ObjectProDSP processing nodes. They provide member functions that are available to all nodes in derived classes. For example, the function to specify a non default output channel for linking the next node could be defined in a base class that includes all nodes that have output channels. There are three base classes for the three groups of nodes defined in the previous three sections.

6.1 `DisplayNodeStr`

`DisplayNodeStr` is a base class for all nodes that have no output channels. Such nodes either display data or write it to an external device. All member functions of this base class are shared by such nodes.

Following are the member functions of this node.

`Raise`

Synopsis: raise any window referencing this node.

This function returns type `void`.

`Raise` will cause a window displaying this network to be raised to the top level over any overlapping windows. Examples of windows that will be affected are a network display containing this node or a plot window for this node.

This function has no parameters.

`DisplayInputTiming`

Synopsis: display the timing of the selected input channel.

This function returns type `void`.

Member function `DisplayInputTiming` displays the timing of the selects input channel for this node.

The parameters of this function are:

- `Channel`: input channel to display timing for.

  Parameter `Channel` selects input channel to display timing for.

  This is the first default parameter. `Channel` is of type `int16`. The default value is 0. `Channel` must be $\geq$ -32767 and $\leq$ 32767.

`Edit`

Synopsis: make this node available for editing in a graphical network.

This function returns type `void`.

If this node is not linked in an existing network it will be added to the display of the network currently being edited. If there is no such network one will be created.

This function has no parameters.

`Unlink`

Synopsis: unlink this node.

This function returns type `void`.

Member function `Unlink` disconnects this node from the DSP network it is linked in.

This function has no parameters.

`LinkIn`

Synopsis: select the next input channel to link to.

This function returns type `Node&`.

The `LinkIn` member function selects the next input channel to link to. It's single parameter (`Channel`) specifies the channel index. Ordinarily the first unused channel is linked to. This function overrides that default.

The parameters of this function are:

- `Channel`: input channel to link to.

  The `Channel` determines the next input channel to link to.

  `Channel` is of type `int16`. The default value is 0. `Channel` must be ≥ 0 and ≤ 32767.

`NextFreeInput`

Synopsis: display next free input link.

This function returns type `void`.

This function displays the next available input link for this node.

This function has no parameters.

6.2 `ProcessNodeStr`

This is a base class for all ObjectProDSPprocessing nodes that have both input and output channels. Thus the member functions in this base class are common to all such ObjectProDSPnodes.

Following are the member functions of this node.

`Raise`

Synopsis: raise any window referencing this node.

This function returns type `void`.

`Raise` will cause a displayed window referencing this node to raised to the top level over any overlapping windows. Examples of windows that will be affected are a network display containing this node or a plot window for this node.

This function has no parameters.

`SetSampleRate`

Synopsis: set the sample rate for the network.

This function returns type `void`.

The `SetSampleRate` member function sets the sample rate for the specified output channel of this node. In turn the rates for all input and output channels connected to this node are adjusted with one exception. The adjustment will $\times$ b + c not be made through a node output channel that specifies a timing relationship of `TimingTypeRandom`.

The parameters of this function are:

- `Rate`: sample rate for node output.

  Parameter `Rate` is the sample rate for node output channel `Channel`.

  `Rate` is of type `double`. The default value is 1.0. `Rate` must be $\geq$ 1.e-60 and $\leq$ 1.e60.

- `Channel`: output channel to set sample rate for.

  Parameter `Channel` determines the the output channel to set the sample rate for.

  This is the first default parameter. `Channel` is of type `int16`. The default value is 0. `Channel` must be ≥ 0 and ≤ 32767.

`DisplayInputTiming`

Synopsis: display the timing of the selected input channel.

This function returns type `void`.

Member function `DisplayInputTiming` displays the timing of the selected input channel for this node.

The parameters of this function are:

- `Channel`: input channel to display timing for.

  Parameter `Channel` selects input channel to display timing for.

This is the first default parameter. `Channel` is of type `int16`. The default value is 0. `Channel` must be $\geq$ -32767 and $\leq$ 32767.

`DisplayOutputTiming`

Synopsis: display the timing of the selected output channel.

This function returns type `void`.

Member function `DisplayOutputTiming` displays the timing of the selected output channel for this node.

The parameters of this function are:

- `Channel`: output channel to display timing for.

  Parameter `Channel` selects output channel to display timing for.

  This is the first default parameter. `Channel` is of type `int16`. The default value is 0. `Channel` must be $\geq$ 0 and $\leq$ 32767.

`Edit`

Synopsis: make this node available for editing in a graphical network.

This function returns type `void`.

If this node is not linked in an existing network it will be added to the display of the network currently being edited. If there is no such network one will be created.

This function has no parameters.

`Unlink`

Synopsis: unlink this node and connected nodes.

This function returns type `void`.

Member function `Unlink` disconnects this node from the DSP network it is linked in. All nodes that are connected as outputs from this node will be unlinked. This process continues recursively up to the terminal output nodes of all affected threads. Unlinking the first node in a single thread network will unlink every node in the network. If a node has two or more inputs and only one of these is unlinked the node will remain connected to the network on the unaffected input channel or channels.

This function has no parameters.

`LinkIn`

Synopsis: select the next input channel to link to.

This function returns type `Node&`.

The `LinkIn` member function selects the next input channel to link to. It's single parameter (`Channel`) specifies the channel index. Ordinarily the first unused channel is linked to. This function overrides that default.

The parameters of this function are:

- `Channel`: input channel to link to.

  Parameter `Channel` determines the input channel to link to.

  This is the first default parameter. `Channel` is of type `int16`. The default value is 0. `Channel` must be ≥ 0 and ≤ 32767.

`NextFreeInput`

Synopsis: display next free input link.

This function returns type `void`.

This function displays the next available input link for this node.

This function has no parameters.

`NextFreeOutput`

Synopsis: display next free output link.

This function returns type `void`.

This function displays the next available output link for this node.

This function has no parameters.

6.3 `SignalStr`

`SignalStr` is a base class for all signal generation and data input nodes. The member functions in this class are available in all these nodes.

Following are the member functions of this node.

`Raise`

Synopsis: raise any window referencing this node.

This function returns type `void`.

`Raise` will cause a window displaying this network to be raised to the top level over any overlapping windows. Examples of windows that will be affected are a network display containing this node or a plot window for this node.

This function has no parameters.

`SetSampleRate`

Synopsis: set the sample rate for the network.

This function returns type `void`.

The `SetSampleRate` member function sets the sample rate for the specified output channel of this node. In turn the rates for all input and output channels connected to this node are adjusted with one exception. The adjustment will not be made through a node output channel that specifies a timing relationship of `TimingTypeRandom`.

The parameters of this function are:

- `Rate`: sample rate for node output.

  Parameter `Rate` is the sample rate for node output channel `Channel`.

  `Rate` is of type `double`. The default value is 1.0. `Rate` must be $\geq$ 1.e-60 and $\leq$ 1.e60.

- `Channel`: output channel to set sample rate for.

  Parameter `Channel` determines the the output channel to set the sample rate for.

  This is the first default parameter. `Channel` is of type `int16`. The default value is 0. `Channel` must be ≥ 0 and ≤ 32767.

`DisplayOutputTiming`

Synopsis: display the timing of the selected output channel.

This function returns type `void`.

Member function `DisplayOutputTiming` displays the timing of the selected output channel for this node.

The parameters of this function are:

- `Channel`: output channel to display timing for.

  Parameter `Channel` selects output channel to display timing for.

  This is the first default parameter. `Channel` is of type `int16`. The default value is 0. `Channel` must be $\geq$ -32767 and $\leq$ 32767.

`Edit`

Synopsis: make this node available for editing in a graphical network.

This function returns type `void`.

If this node is not linked in an existing network it will be added to the display of the network currently being edited. If there is no such network one will be created.

This function has no parameters.

`Unlink`

Synopsis: unlink this node and nodes connected to its output .

This function returns type `void`.

Member function `Unlink` disconnects this node from the DSP network it is linked in. All nodes that are connected as outputs from this node will be unlinked. This process continues recursively up to the terminal output nodes of all affected threads. Unlinking the first node in a single thread network will unlink every node in the network. If a node has two or more inputs and only one of these is unlinked the node will remain connected to the network on the unaffected input channel or channels.

This function has no parameters.

`NextFreeOutput`

Synopsis: display next free output link.

This function returns type `void`.

This function displays the next available output link for this node.

This function has no parameters.

7 Additional ObjectProDSP objects

The objects in this section do not perform DSP processing or data I/O functions. They provide auxillar function such as specifying buffering options or control options.

7.1 `CircBufDes`

Synopsis: Circular buffer descriptor.

After a network has been completely described, it is necessary to provide physical buffers for all the connection paths between nodes. This is done by first defining a `BufferDescriptor`. And then using this descriptor in the network member function `AssignBuffers`. (If you do not do this manually it will be done automatically before executing a network using the default buffer descriptor.) `CircBufDes` describes circular buffers. There is a single parameter for this descriptor. It specifies the size (`Size`) of the buffer. Other parameters describe the constraints on buffer size for generating target processor code.

The parameters for `CircBufDes` are:

- `Size`: buffer size in words.

`Size` determines the buffer size for interactive execution. The larger the buffer is the more efficiently the nodes can execute and the greater the delay that is possible in the network. If the buffers are too small the network may lock up without processing all input data.

`Size` is of type `int32`. The default value is 2048. `Size` must be ≥ 1 and ≤ 2147483647.

- `TargetSize`: desired buffer size in target code.

  `TargetSize` is the desired size of the buffer used in code prepared for execution on the target processor. An analysis will determine if this size is adequate and if it is larger than will be of benefit. The size will be optimized based on this target size as an approximate goal.

  This is the first default parameter. `TargetSize` is of type `int32`. This parameter can be changed interactively. The default value is 512. `TargetSize` must be ≥ 0 and ≤ 2147483647.

- `TargetSizeGoal`: target buffer size goal (0 minimize size, 1 maximize size).

  `TargetSizeGoal` determines whether the buffer will be made larger or smaller then the selected size when the selected size is not optimal. If `TargetSizeGoal` is 0 that space will be minimized. If `TargetSizeGoal` is 1 then execution overhead will be minimized.

  This is the first default parameter. `TargetSizeGoal` is of type `int16`. This parameter can be changed interactively. The default value is 0. `TargetSizeGoal` must be ≥ 0 and ≤ 1.

- `TargetControlGoal`: control goal (0 fixed buffer size, 1 fixed sequence).

  `TargetControlGoal` determines how execution is controlled. If it is 0 then the buffer size is fixed and the amount of data available in the buffer is computed before each execution step. If `TargetControlGoal` is 1 then each node is executed according to a fixed predetermined schedule and the buffer size is optimized to this schedule. Nodes with feedback or that require excessively large buffers are defaulted to execute without fixed sequences. Fixed sequence execution provides more efficient execution at the expense of larger buffer requirements.

This is the first default parameter. `TargetControlGoal` is of type `int16`. This parameter can be changed interactively. The default value is 1. `TargetControlGoal` must be ≥ 0 and ≤ 1.

- `MaxTargetSize`: maximum size for any single buffer.

  `MaxTargetSize` specifies the maximum size allowed for any single buffer. It can force the use of a non fixed sequence scheduler or even result in an error message if the network cannot run without deadlock with this size buffer.

  This is the first default parameter. `MaxTargetSize` is of type `int32`. This parameter can be changed interactively. The default value is 4096. `MaxTargetSize` must be ≥ 1 and ≤ 2147483647.

- `MinTargetSize`: minimum size for any single buffer.

  `MinTargetSize` specifies the minimum size allowed for any single buffer. Setting this to a larger value can improve execution efficiency at the cost of more memory.

  This is the first default parameter. `MinTargetSize` is of type `int32`. This parameter can be changed interactively. The default value is 32. `MinTargetSize` must be ≥ 1 and ≤ 2147483647.

Following is the member function of this node.

`AssignToEdit`

Synopsis: assign this descriptor to the network currently being edited.

This function returns type `void`.

`AssignToEdit` makes this the descriptor for the network currently being edited. If a previous descriptor was assigned it will be overwritten.

This function has no parameters.

7.2 `DataFlow`

Synopsis: Data flow based network control and scheduling.

`DataFlow` is a ‘Scheduler’ based on a data flow model. Its Only argument is the `TheNet` to control.

The parameter for `DataFlow` is:

- `TheNet`: Network to control.

  `TheNet` is the ‘Network’ to be controlled.

  `TheNet` is of type `ProcessNet&`. The default value is `NetworkDef`. `TheNet` must be legal in this context.

Following are the member functions of this node.

`GraphDisplay`

Synopsis: display network topology and timing.

This function returns type `void`.

Member function `GraphDisplay` displays the network topology and timing.

The parameters of this function are:

- `Option`: display option (0 - short, 1 - full, 2- timing).

  The `Option` parameter selects a network display that includes network topology (`Option` $= 0$) plus network data flow parameters (`Option` $=$ 1) or network timing analysis (`Option` =2).

  `Option` is of type `int16`. The default value is 0. `Option` must be ≥ 0 and ≤ 2.

`Execute`

Synopsis: execute network generating the specified number of samples.

This function returns type `void`.

Member function `Execute` executes the network being controlled. It causes the signal generator (or first node in the network) to produce a specified number of input samples. Each node is executed for as many iterations as possible given the available input data and output buffer space. Execution halts when no node generates any new samples after a complete pass throughout the network.

The parameters of this function are:

- `InputSamples`: input samples to process.

  `InputSamples` speicfies the samples to be processes in the first node during this network execution. If there are multiple threads in a network than `InputSamples` is relative to the first thread.

  `InputSamples` is of type `int32`. The default value is 128. `InputSamples` must be ≥ 1 and ≤ 2147483647.

`AssignBuffers`

Synopsis: assign buffers to a completely defined network.

This function returns type `void`.

Member function `AssignBuffers` buffers to a completely defined data flow network. The network is first checked for completeness. Buffers will not be assigned if the network fails this test. The single parameter of this function specifies the buffer characteristics.

The parameters of this function are:

- `Descriptor`: specify buffer characteristics.

  The `Description` parameter specifies the buffer characteristics.

  `Descriptor` is of type `BufferDescript&`. The default value is `CircBufDesDef`.

`ClearBuffers`

Synopsis: remove all buffers from network.

This function returns type `void`.

You can not change the topology of a network while buffers are assigned. Member function `ClearBuffers` removes all buffers so that the network can be edited or different buffers assigned.

This function has no parameters.

`ClearNetwork`

Synopsis: delete all network links.

This function returns type `void`.

Member function `ClearNetwork` removes all links in the network being controlled. The nodes freed in this way can then be used in a different network.

This function has no parameters.

7.3 Network

Synopsis: Data flow network objects.

`Network` is the class that allows you to combine nodes to form a DSP process. Member functions allow you to add threads ('operator+') and add nodes ('operator>>'). It is easiest to use these functions by editing a network graphically. There are other member functions for editing and manipulating networks. Most of these can also be called by graphically editing a network. Member function `Execute` executes a network for a specified number of input blocks. It can only be accesses from the menu data base.

This node has no parameters.

Following are the member functions of this node.

`GraphDisplay`

Synopsis: display network topology.

This function returns type `void`.

Member function `GraphDisplay` displays network topology.

This function has no parameters.

`Execute`

Synopsis: execute network generating the specified number of samples.

This function returns type `void`.

Member function `Execute` executes this network. It causes the first node in the network to produce a specified number of input blocks. Each node is executed for as many iterations as possible given the available input data and output buffer space. Execution halts when no node generates any new samples after a complete pass throughout the network.

The parameters of this function are:

- `InputSamples`: input samples to process.

  `InputSamples` is the samples to be generated by the first node of the first thread. `InputSamples` determines the input parameter `k` that the kernel for this driver is called with. `InputSamples` is not the same as `k`. It must be divided by (`DeltaOut` × `BlockSize`). If there are multiple threads in an independent subnetwork, the others will have their driver input count adjusted according to their sample rates relative to this first node in the first thread of the subnetwork. This adjustment is made by analyzing the input and output sample ratios for all nodes in a connected subnet of the network.

`InputSamples` is of type `int32`. The default value is 128. `InputSamples` must be ≥ 1 and ≤ 2147483647.

`Raise`

Synopsis: raise a window containing this network display.

This function returns type `void`.

`Raise` will cause a window displaying this network to be raised to the top level over any overlapping windows.

This function has no parameters.

`ReplaceNode`

Synopsis: replace a single node in a network.

This function returns type `void`.

`ReplaceNode` will substitute node `Replacement` for node `ToReplace` in the network. The nodes must have the same number of input and output channels and be compatible in all other respects. If an error occurs the original node will remain in the network.

The parameters of this function are:

- `ToReplace`: node to replace.

  `ToReplace` is the node to be replaced.

  `ToReplace` is of type `Node&`. The default value is `DefaultNotLegal`.

- `Replacement`: node to replace.

  `Replacement` will be substituted in the network for `ToReplace`.

  `Replacement` is of type `Node&`. The default value is `DefaultNotLegal`.

`MakeTarget`

Synopsis: create target processor code for this network.

This function returns type `void`.

Member function `MakeTarget` will create source and executable code for this network for a supported target. See the description of parameter `Target` for a list of the available targets. See parameter `Directory` for a description of the files created.

The parameters of this function are:

- `Target`: target processor.

  `Target` is the target processor name. The default generates stand alone generic C++ code. Currently no other processors are supported.

  This is the first default parameter. `Target` is of type `const char *`. The default value is "generic_cpp".

- `Create`: create executable flag.

  if `Create` is 0 only the network source files and a 'domakemake' and 'Makefile' will be created. These can be used to create an executable. If `Create` is 1 a 'make' command will also be issued to create an executable.

  This is the first default parameter. `Create` is of type `int16`. The default value is 1. `Create` must be ≥ 0 and ≤ 1.

- `Directory`: directory of created files.

  If the default is chosen the network name is used for `Directory`. If directory `Directory` does not exist it will be created. The target network topology, sequencing tables, and a short main program are written to `Directory`. A shell script 'OPDmkmkmk' is also written and executed. This creates a 'domakmake' file which it then executes invoking 'makemake" to create a 'Makefile'. If `Create` is selected 'OPDmkmkmk' will also invoke 'make' to build an executable. If not go to `Directory`/sub (where 'sub' is a subdirectory determined by the type of simulator arithmetic) and type 'make' to build an executable.

  This is the first default parameter. `Directory` is of type `const char *`. The default value is 0.

`MakeValidate`

Synopsis: create a validation test case from this network.

This function returns type `void`.

Member function `MakeValidate` first replaces each display output node in this network (such as plotting or listing nodes) with an `OutputNode`. This network is then saved to directory `DirName`. Next the same nodes are replaced with a `CompareDisk` node. This new network is written to the same directory under a different name. The first network saved state will include a statement to execute for `ExecuteCount` + `ExtraCountCreator` blocks of input. The second network will execute for `ExecuteCount` blocks. These networks are used to generate baseline regression test data and to run tests against this data.

The parameters of this function are:

- `DirName`: directory to place validation files.

  Validation files are written to `DirName`. If the default is taken a directory name will be constructed from the network name. If the directory does not exist it will be created.

This is the first default parameter. `DirName` is of type `const char *`. The default value is 0.

- `ExecuteCount`: number of inputs to execute for.

  The test network state generated will include a statement to execute for `ExecuteCount` input blocks.

  This is the first default parameter. `ExecuteCount` is of type `int32`. The default value is 8192. `ExecuteCount` must be ≥ 1 and ≤ 2147483647.

- `ExtraCountCreator`: number of additional inputs to create data.

  If the creator and validation program execute for the same number of times then and end of file may be encountered on some channels. This provides a margin to prevent end of file error messages. The creator network will execute for `ExecuteCount` + `ExtraExecuteCount` times.

  This is the first default parameter. `ExtraCountCreator` is of type `int32`. The default value is 2048. `ExtraCountCreator` must be ≥ 1 and ≤ 2147483647.

- `MaxReport`: maximum number of errors to report.

  Only the first `MaxReport` errors will be reported.

  This is the first default parameter. `MaxReport` is of type `int32`. The default value is 1000. `MaxReport` must be ≥ 1 and ≤ 2147483647.

- `Tolerance`: absolute value of minimum difference for an error.

  `Tolerance` is the absolute value of the smallest difference that constitutes an error. Ordinarily this value is 0.0. It might be set to a value larger than 0 to compare slightly different algorithms or results on two different computers with different arithmetic.

  This is the first default parameter. `Tolerance` is of type `double`. The default value is 0.0. `Tolerance` must be ≥ 0.0 and $\leq 1.e100$.

- `errorFile`: if set errors will be written to this file.

  `ErrorFile` is a file in which errors will be reported instead of displaying them in a window. If the default is taken a name will be constructed from the network name and node name.

  This is the first default parameter. `errorFile` is of type `const char *`. The default value is 0.

`TargetValidate`

Synopsis: create a target validation test case from this network.

This function returns type `void`.

Member function `TargetValidate` first replaces each display output node in this network (such as plotting or listing nodes) with an `OutputNode`. This network is then used to generate a target system in directory `DirName`/create'. Next the same nodes are replaced with a `CompareDisk` node. The code for this network is written to the directory `DirName`/test. Shell scripts will be written to directory `DirName` to execute the networks for `ExecuteCount` blocks (test network) and (`ExecuteCount` + `ExtraCountCreator`) blocks (test data creation network). `DirName` will contain all test data and error files and network state descriptions.

The parameters of this function are:

- `Target`: target processor.

  `Target` is the target processor name. The default generates stand alone generic C++ code. Currently no other processors are supported.

  This is the first default parameter. `Target` is of type `const char *`. The default value is "generic_cpp".

- `Create`: create executable flag.

  if `Create` is 0 a generic description of the network and a make file will be created. These can be used to create an executable for the specified target processor. If `Create` is 1 a 'make' command will also be issued to create an executable file for the target processor.

  This is the first default parameter. `Create` is of type `int16`. The default value is 0. `Create` must be ≥ 0 and ≤ 1.

- `DirName`: directories to place validation files.

  The test cases will be written under directory `DirName`. Test data, scripts to create and run tests and the network states will be in this directory. Code for generateion and testing will be in subdirectories 'create' and 'test'. If the default is taken a directory name will be constructed from the network name. Directories are created if they do not exist.

This is the first default parameter. `DirName` is of type `const char *`. The default value is 0.

- `ExecuteCount`: number of inputs to execute for.

  A shell script will be created to execute the test network state for `ExecuteCount` input blocks.

  This is the first default parameter. `ExecuteCount` is of type `int32`. The default value is 8192. `ExecuteCount` must be ≥ 1 and ≤ 2147483647.

- `ExtraCountCreator`: number of additional inputs to create data.

  If the creator and validation program execute for the same number of times then and end of file may be encountered on some channels. This provides a margin to prevent end of file error messages. A shell script will be created to execute the test network state for `ExecuteCount` + `ExtraCountCreator` input blocks.

  This is the first default parameter. `ExtraCountCreator` is of type `int32`. The default value is 2048. `ExtraCountCreator` must be ≥ 1 and ≤ 2147483647.

- `MaxReport`: maximum number of errors to report.

  Only the first `MaxReport` errors will be reported.

  This is the first default parameter. `MaxReport` is of type `int32`. The default value is 1000. `MaxReport` must be ≥ 1 and ≤ 2147483647.

- `Tolerance`: absolute value of minimum difference for an error.

  `Tolerance` is the absolute value of the smallest difference that constitutes an error. Ordinarily this value is 0.0. It might be set to a value larger than 0 to compare slightly different algorithms or results on two different computers with different arithmetic.

  This is the first default parameter. `Tolerance` is of type `double`. The default value is 0.0. `Tolerance` must be ≥ 0.0 and $\leq 1.e100$.

- `ErrorFile`: if set errors will be written to this file.

`ErrorFile` is a file in which errors will be reported instead of displaying them in a window. If the default is taken a name will be constructed from the network name and node name.

This is the first default parameter. `ErrorFile` is of type `const char *`. The default value is 0.

`SetTimingExact`

Synopsis: set exact or loose timing constraints.

This function returns type `void`.

If the timing analysis cannot resolve a network it may still execute correctly. If `Exact` is one no attempt will be made to execute the network. Instead an error will be generated. This is usually set for for validation tests to make sure that timing analysis errors are not overlooked.

The parameters of this function are:

- `Exact`: timing constraints loose(0) or exact(1).

  If the timing analysis cannot resolve a network it may still execute correctly. If `Exact` is one no attempt will be made to execute the network.

  `Exact` is of type `int16`. The default value is 0. `Exact` must be ≥ 0 and ≤ 1.

`ReplaceWithOutput`

Synopsis: replaces plot and listing nodes with an `OutputNode`.

This function returns type `void`.

Member function `ReplaceWithOutput` replaces all plot and listing nodes with a new output node with a name derived from the node it is replacing. The file name is the same as the node name. This is used to create regression tests. First `ReplaceWithOutput` creates a network to generate test data. Then `ReplaceWithCompare` creates a network for running a regression test against the data. All three networks should be saved in separate state files.

This function has no parameters.

`ReplaceWithCompare`

Synopsis: replace each `OutputNode` with a 'Compare' node.

This function returns type `void`.

Member function `ReplaceWithCompare` replaces each `OutputNode` in a network with a `CompareDisk` node. If an `OutputNode` has more than one input channel the operation will fail. This is useful in converting a network used to generate a regression test case to a network for running the regression test. The new node name is created from the node replaced. The file name is the output file written by the node being replaced. The other parameters are set to be the same as the corresponding parameters of this function.

The parameters of this function are:

- `MaxReport`: maximum number of errors to report.

  Only the first `MaxReport` errors will be reported.

  This is the first default parameter. `MaxReport` is of type `int32`. The default value is 1000. `MaxReport` must be ≥ 1 and ≤ 2147483647.

- `Tolerance`: absolute value of minimum difference for an error.

  `Tolerance` is the absolute value of the smallest difference that constitutes an error. Ordinarily this value is 0.0. It might be set to a value larger than 0 to compare slightly different algorithms or results on two different computers with different arithmetic.

  This is the first default parameter. `Tolerance` is of type `double`. The default value is 0.0. `Tolerance` must be ≥ 0.0 and $\leq 1.e100$.

- `ErrorFile`: if set errors will be written to this file.

  `ErrorFile` is a file in which errors will be reported instead of displaying them in a window.

  This is the first default parameter. `ErrorFile` is of type `const char *`. The default value is 0.

`operator+`

Synopsis: the '+' operator adds a thread to a `Network`.

This function returns type `Network&`.

The '+' operator appends its right operand (a signal generation node) `TheNode` to its left operand (a data flow `Network`).

The parameters of this function are:

- `TheNode`: node to append to.

  `TheNode` is a signal generator that starts a thread in the network to the right of the '+' operator.

  `TheNode` is of type `SignalStr&`. The default value is `CosDef`.

`operator>>`

Synopsis: operator>> appends a node to a network.

This function returns type `Network&`.

The '>>' operator appends its right operand (a processing or signal generation node) `TheNode` to its left operand (a `DataFlow` Network).

The parameters of this function are:

- `TheNode`: node to append to.

  `TheNode` is the node to append to the network on the right of operator '>>'.

  `TheNode` is of type `Node&`. The default value is `GainDef`.

`GraphDisplayWindow`

Synopsis: display network topology in a specified window size.

This function returns type `void`.

Member function `GraphDisplay` displays the network topology. In a window of up to `Width` x `Height` pixels. The window may start out smaller and grow larger as nodes are added to it but it will not exceed these dimensions. (If needed a vertical scrollbar will be added to the window.)

The parameters of this function are:

- `Width`: maximum width in pixels of display.

  `Width` is the maximum width in pixels the window can grow to. It may start out smaller but will not exceed this width as nodes are added to it. (If needed a vertical scrollbar will be added to the window.)

  `Width` is of type `int16`. The default value is 550. `Width` must be ≥ 150 and ≤ 2048.

- `Height`: maximum height in pixels of display.

  `Height` is the maximum height in pixels the window can grow to. It may start out smaller but will not exceed this height as nodes are added to it. (If needed a vertical scrollbar will be added to the window.)

  `Height` is of type `int16`. The default value is 700. `Height` must be $\geq$ 150 and ≤ 2048.

`DisplayNames`

Synopsis: display the name of the controller and buffer descriptor.

This function returns type `void`.

Member function `DisplayNames` displays the names of the controller and buffer descriptor for this node in the help window.

This function has no parameters.

`SetBufferDescriptor`

Synopsis: assign a buffer descriptor to this network.

This function returns type `void`.

`SetBufferDescriptor` assigns descriptor `Descriptor` to this network. The network need not be complete. No buffers are allocated.

The parameters of this function are:

- `Descriptor`: buffer characteristics.

  The `Description` parameter specifies the buffer characteristics.

  `Descriptor` is of type `BufferDescript&`. The default value is `CircBufDesDef`.

`AssociateNode`

Synopsis: put this node in this networks display window.

This function returns type `void`.

`AssociateNode` associates node `TheNode` with this network. It does not link the node into the network. Its only effect is to have the node displayed in the window in which the network appears.

The parameters of this function are:

- `TheNode`: node to associate with this network.

  `TheNode` will be displayed in the window for this network. This member function does not link the node into the network or affect anything other than the display.

  `TheNode` is of type `Node&`. The default value is `DefaultNotLegal`.

`Link`

Synopsis: make next link in network from specified node and channel.

This function returns type `Network&`.

In building a data flow topology network the connection operator '>>' always links the output of the last node accessed (from the network to the left of '>>') to the node on the right side of '>>'. For simple linear networks this is adequate. For nodes with more than one output one must specify when to use channels other than 0 using `Link`. `Link` causes the next link from the network to begin at the specified `TheNode` and `OutChannel`.

The parameters of this function are:

- `TheNode`: processing or signal generator to link to.

  The `TheNode` parameter specifies a node within this network to establish the next link to. This is used in building the network topology when the default option of linking to the last node accessed will not work. `Link` is needed whenever a node has multiple output channels.

  `TheNode` is of type `Node&`. The default value is `GetFreeNodeOut`.

- `OutChannel`: output channel of selected node to link to.

  The `OutChannel` parameter specifes the index of the output channel from the selected node. The next link in the network will originate from this channel and node.

This is the first default parameter. `OutChannel` is of type `int16`. The default value is 0. `OutChannel` must be ≥ 0 and ≤ 32767.

`SelfLink`

Synopsis: establish a feedback link in a data flow network.

This function returns type `Network&`.

`SelfLink` establishes a feedback link in a data flow network. Input parameters include the source node and its output channel and the destination node and its input channel.

The parameters of this function are:

- `NodeOut`: node to link from.

  The `NodeOut` parameter specifies a node within this network to establish the feedback link from.

  `NodeOut` is of type `Node&`. The default value is `GetFreeNodeOut`.

- `NodeIn`: node to link to.

  The `NodeIn` parameter specifies a node within this network to establish the feedback link to.

  `NodeIn` is of type `Node&`. The default value is `GetFreeNodeIn`.

- `ChannelOut`: output channel of output node to link from.

  The `ChannelOut` parameter specifes the index of the output channel of `NodeOut` to link from.

  This is the first default parameter. `ChannelOut` is of type `int16`. The default value is 0. `ChannelOut` must be ≥ 0 and ≤ 32767.

- `ChannelIn`: input channel to link to.

  The `ChannelIn` parameter specifes the index of the output channel of `NodeIn` to link to.

  This is the first default parameter. `ChannelIn` is of type `int16`. The default value is 1. `ChannelIn` must be ≥ 0 and ≤ 32767.

`AssignBuffers`

Synopsis: assign buffers to a completely defined network.

This function returns type `void`.

Member function `AssignBuffers` buffers to a complete network. Buffers will not be assigned if the network is not complete. `Descriptor` determines the buffer characteristics.

The parameters of this function are:

- `Descriptor`: specify buffer characteristics.

  The `Description` parameter specifies the buffer characteristics.

  `Descriptor` is of type `BufferDescript&`. The default value is `CircBufDesDef`.

`GetBufferDescriptor`

Synopsis: get the buffer descriptor associated with this network.

This function returns type `BufferDescript&`.

Member function `GetBufferDescriptor` returns the buffer descriptor associated with this network.

This function has no parameters.

`ClearBuffers`

Synopsis: remove all buffers from network.

This function returns type `void`.

You can not change the topology of a network while buffers are assigned. Member function `ClearBuffers` removes all buffers so that the network can be edited or different buffers assigned.

This function has no parameters.

`GetNetController`

Synopsis: get the network controller associated with this network.

This function returns type `NetControl&`.

Member function `GetNetController` returns the network controller associated with this network.

This function has no parameters.

`ClearNetwork`

Synopsis: delete all network links.

This function returns type `void`.

Member function `ClearNetwork` removes all links in the network being controlled. The nodes freed in this way can then be used in a different network.

This function has no parameters.

A GNU GENERAL PUBLIC LICENSE

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program'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 License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

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.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

> a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
>
> b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
>
> c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary 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 program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

> a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
>
> b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
>
> c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

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

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions

of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which 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 the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "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 PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. 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 PROGRAM 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 PROGRAM (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 PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.> Copyright (C) 19yy <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index