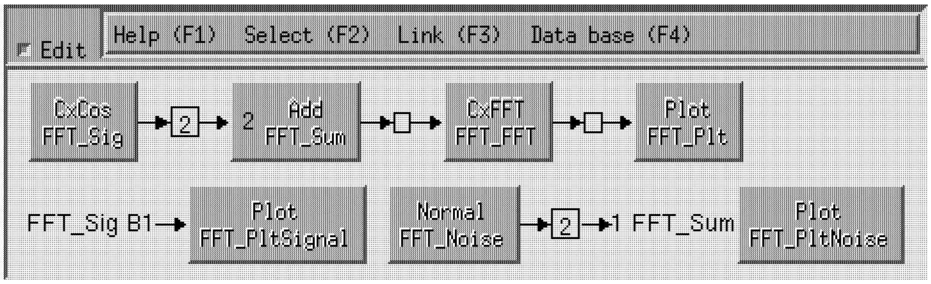
ObjectProDSP Developer's Reference
PRELIMINARY

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.

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 gives a detailed description of ObjectProDSP interactive objects including DSP processing nodes.
- *ObjectProDSP Developer's Reference* This is the document you are reading. 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 Figures

List of Tables

File	Purpose	Section	Page
`xxx.h`	Interactive C++ header	3.4	26
`xxx.C`	Interactive C++ file	3.4	26
`target/xxx.h`	Target C++ header	3.5	26
`target/xxx.C`	Target C++ file	3.5	26
`tex/xxx.tex`	Node library documentation	3.5	26
`texs/xxx.tex`	Node summary documentation	4.2	29
`menus/nodes/xxx.nod`	Menu data base	4.1	28
`xxxI.h`	Data base initialization	4.1	28
`xxx.s`	Text for spelling checker	4.1	28

Table 1: Files generated by ObjectPro`++` from a '`.usr`' file

1 Creating nodes with ObjectPro++

ObjectPro`++` translates a single '`.usr`' file into two pairs of C`++` '`.h`' and '`.C`' files and additional files for documentation and to support integration with the menu data base. The complete list of these files and the sections that describe them are in Table 1 on page 1.

We will describe the syntax of the input file, discuss the semantics and explain how to integrate a node into ObjectProDSP.

1.1 ObjectPro++ class definition

Figure 1 on page 3 shows a sample DSP node definition in ObjectPro`++`. First are 'include' statements. There are several types of 'include' statements that refer to different points in the multiple files that will be generated. These are listed in Table 2 on page 2.

There is then a declaration of the constructor for this class that includes documentation. The class name is followed by a parenthesized list of parameter descriptions. These descriptions have three elements. First is a line that gives the parameter type, default value and bounds. The allowed types are

Name	File	Interactive	Target	Before standard includes
`Cinclude`	'.C'	yes	yes	yes
`Hinclude`	'.h'	yes	yes	yes
`cinclude`	'.C'	yes	yes	no
`hinclude`	'.h'	yes	yes	no
`ICinclude`	'.C'	yes	no	yes
`IHinclude`	'.h'	yes	no	yes
`Icinclude`	'.C'	yes	no	no
`Ihinclude`	'.h'	yes	no	no
`TCinclude`	'.C'	no	yes	yes
`THinclude`	'.h'	no	yes	yes
`Tcinclude`	'.C'	no	yes	no
`Thinclude`	'.h'	no	yes	no

Table 2: ObjectPro++ include options

given in Table 3 on page 5. The key words `Min` and `Max` designate the largest and smallest values for a given data type.

Any ObjectPro++ class can be the type of a parameter. The syntax for this and for type `string` is different from that for numeric values. Only the default and not the bounds are relevant for these data types. The default for a string can be 0 to represent a null pointer.

Either or both of two key words can precede the data type. `Changeable` indicates that the parameter can be modified after the constructor is called. An entry in the menu for the object instance under `set` to change the value is generated for each `Changeable` parameter. `FirstDefault` indicates that this variable and all subsequent variables can be assigned default values when the constructor is typed in. (The menu data base `create default` option assigns all default parameters without regard to `FirstDefault`.)

Following the parameter and bounds check is a `MenuLine` this gives a brief synopsis of the parameter. This synopsis will appear in the menu data base above the buttons when the cursor is positioned over the button for this parameter. This line will be used in the printed documentation to provide

```
#cinclude "ObjProComGui/cgidbg.h"
#cinclude "ObjProGui/yacintfc.h"
Block (int16 ElementSize : 1 <= 2 <= MAX
    MenuLine {output element size}
    Description {
        $ElementSize is the number of words in each output sample
        (1 for real, 2 for complex or larger for other purposes).},
    FirstDefault int16 BlockSize : 1 <= 1 <= MAX
    MenuLine {output block size}
    Description {
        $BlockSize is the number of samples in each output block. If
        set to 1 the output is not blocked.},
    int16 OutputArithmetic : 0 <= 0 <= 2
    MenuLine {output data: 0 - MachWord, 1 - int32, 2 - float}
    Description {
        $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.}
    ): public ProcessNodeStr(STREAM_IN = new StreamStr(
            SizeVariable,SizeVariable,ArithType::ArithCapabilityAny),
            STREAM_OUT= new StreamStr(ElementSize,BlockSize,
            (OutputArithmetic == 2 ? ArithType::ArithFloat :
                (OutputArithmetic == 1 ? ArithType::ArithInt32 :
                    ArithType::ArithTypeUndefined
                )
            )
        )
    );
```

Figure 1: ObjectPro++ example program (part 1 of 3)

```
Class {
    enum convert_type {undefined,no_conversion, float_to_int,
        int_to_float} ;

#ifdef INTERACTIVE
    int InitAfterLinked();
    void input_linked(int in_channel);
    void static_ctor();
#endif
    void ctor();
    ErrCode kernel(int32 k);
};

StaticDeclare {
    int the_convert_type ;
}

StaticInit {
    static_ctor();
}

Constructor {
    ctor();
}
```

Figure 2: ObjectPro++ example program (part 2 of 3)

```
MenuLine {Converts an input stream to a new blocking and sample size}
Description {
    $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. }
HelpFile block
Kernel {
    return kernel(k);
}
```

Figure 3: ObjectPro++ example program (part 3 of 3)

Name	Definition
int16	16 bit integers
int32	32 bit integers
MachWord	native machine word of simulator
double	double precision floating point
string	character string
class	any ObjectPro++ object in *class*

Table 3: ObjectPro++ data types

a brief description of the parameter. Next is an optional `Description` that provides a more complete description of the parameter. This description will appear in the `ObjectProDSP information` window whenever the mouse button is pressed in a button representing this parameter. This description will also be used in the printed documentation.

After the parenthesis that closes the parameter list there is '`: public`' followed by a base class constructor. There must always be a base class to tie this node into the existing ObjectProDSP class structure. Table 5 on page 11 gives the base classes ordinarily used for DSP objects. It is possible to define base classes with '`.usr`' files but all base classes must ultimately be built on classes predefined within ObjectProDSP that do not have a '`.usr`' file. The bottom of this hierarchy is class `UserEntity` which is common to all interactive objects. There are seemingly redundant base classes in Table 5 ending with suffix '`Str`'. These refer to different methods of defining parameters in the base class. Those that end it '`Str`' should be used. The others are for backwards compatibility with nodes that have not been updated. Currently the `BlockPlot` and `Plot` classes are redundant since all plot nodes support blocked and unblocked data. Eventually some specialized plotting routines may not support blocked data.

Following the base class name are the parameters for the base class constructor. These are not assigned in position order as they are in C++ but are assigned by name. There are defaults for all options. Only those for which the defaults are incorrect need to be assigned. The allowed names differ with different base classes, The base classes are described in Section 1.4 on page 8.

Next is a `MenuLine` for the node. This is like the `MenuLine` for each parameter but it describes the node.

1.2 Placing a node in the data base

The key word `MenuLine` may be followed by the name of a menu (defined in `opd.menu` described in Section 5 on page 33) to connect this node to the menu data base. If this is not present the node will be added to menu `DspNodesMenu` (or another standard menu as determined by the optional

`InteractiveEntityList` option described in Table 4.1 on page 28). This also controls where the node will be documented in the printed manuals.

The text of the `MenuLine` is followed by a `Description` of the class which is similar to the `Description` of a parameter,

Next is an optional line that gives a `HelpFile` for the class. This file will appear in a separate window when the *right* mouse button is released with the cursor over the button for this class. See Section 6.1 on page 45 for information on preparing and integrating help files.

Following this are a number of optional sections and one mandatory section. The mandatory section is the `Kernel` which contains the code to implement the kernel processing.

Unless the code for the node is short you may want to use include or additional '`.C`' files for most of the kernel and initialization code. The mandatory `Kernel` section can call another subroutine that does the processing. The best choice is usually to use '`.C`' files. The directory in which the '`.usr`' file occurs is only used for interactive code and not for target code. Any '`.C`' file that is common to both should be paced in the subdirectory `common` under the directory containing the '`.usr`' file. If you use header files they should be put in directory
`$OPD_ROOT/src/include/ObjProDSPcom`
and referenced as `"ObjProDSPcom/`*file_name*.

1.3 Simplified node syntax

Figure 4 on page 9 gives a simplified `yacc` (or `bison`) syntax for a node. The rules are in alphabetical order. The top level symbol is DfNode. Literal terminal symbols are in `typewriter` font and enclosed in single quotes. Standard terminal symbols like *integer* are in *italicized font*. If a rule ends with '|' it can be reduced to the null symbol.

Table 4 on page 10 gives a brief explanation for each nonterminal symbol and references for some. The full `yacc` (or `bison`) syntax is in file
`$OPD_ROOT/src/util/mknode/mknod_b.y`.

Note some constructs in the full syntax are for future expansion (such as `TargetDesignator`) and others are obsolete but retained for backwards compatibility.

1.4 Base classes

Base classes in ObjectPro++ function like and are implemented with base classes in C++. The concept is extended to provide interactive member functions which are shared by all nodes with a common base class. Multiple inheritance is not supported for interactive functions but can be used for C++ only constructs. The base classes and their purpose are in Table 5 on page 11. This table also lists the parent class and the header file that defines the class. These header files are in directory
`$OPD_ROOT/src/include/ObjProDSPint/ObjProUsr`
(for base classes defined with a '`.usr`' file) and
`$OPD_ROOT/src/include/ObjProGui/ObjProNet`
(for base classes defined directly in C++). The `.usr`' for the base class (if it exists) will be in some directory under `$OPD_ROOT/src/dsp` or
`$OPD_ROOT/src/dsp/dsp_gui`.

1.4.1 Base constructor

The base class constructor parameters determine the number of input and output channels (`IN` and `OUT`) the type of data on these channels (`STREAM_IN` and `STREAM_OUT`) and the amount of output data generated for a given amount of input data (`NODE_DELAY`, `DELTA_IN`, `DELTA_OUT` and `OVERLAP`) and other properties such as the node timing. The use of these values to do process scheduling are described in *ObjectProDSP User's Reference*[2] in the 'Data flow model' subsection of the chapter on 'The DSP++ language".

Additional constructor parameters are for special classes of nodes such as those for plotting.

The parameters used in each base class are in Table 5 on page 11. The parameters are described in alphabetical order in Table 6 on page 12 along with

BaseDescription : '`BaseDescription`' '`{`' *text of description* '`}`'
Body : StateEmit Timing Kernel Instances
Comment : '`//`' *text to end of line*
Cpp : '`{`' *C++ code* '`}`'
Ctor : '`Constructor`' Cpp
Declaration : ReferencedNameList DeclarationCode | DeclarationCode
DeclarationCode : '`Class`' Cpp ';'
DefaultList : NumParmValue | '{' NumberList '}'
Description : MenuLine FullDescription HelpFile
  | MenuLine FullDescription
DfNode : Header NodeDescription Body | Header
DfNodeCtor : BaseDfNodeCtor ',' OtherBaseList | BaseDfNodeCtor |
Dtor : '`Destructor`' Cpp
FullDescription : '`Description`' '`{`' *text of description* '`}`'
Header : Includes ClassName NodeName Parameters HeaderEnd
HeaderEnd : Declaration StaticDeclare StaticInit Ctor Dtor Safe
Help : '`HelpDefaultFile`' *string*
Instance : Name '(' ConstantList ')' InstanceDescription
InstanceDescription : Description |
Instances : Instances '`Instance`' Instance ';' | '`NoDefaultInstance`' ';' |
InteractiveClass : '`InteractiveEntityList`' *name* ';'
Kernel : '`Kernel`' Cpp
Member : *type* MemberName '(' ParameterList ')' '`Wait`' Description ';'
MemberHelp : Member | Help
MenuLine : '`MenuLine`' '{' *text of description* '}'
NodeDescription : Description | BaseDescription
NodeDescription : InteractiveClass DfNode | DfNode
Parameter :
  type Name ':' ParamValue '`<=`' ParmValue '`<=`' ParmValue Description |
  type Name ':' ParmValue Description |
  type Name Size ':' ParmValue '`<=`' DefaultList '`<=`' ParmValue
ParameterAndCheck : '`FirstDefault`' Parameter |
  '`FirstDefault`' Parameter `CheckParameter`' Cpp
ParameterList : ParameterAndCheck | ParameterList ',' ParameterAndCheck |
Parameters : '(' ParameterList ')' DfNodeCtor ';'
ReferencedNameList : ReferencedNameList MemberHelp | MemberHelp
Safe : '`SafeDelete`' Cpp
Size : '[' *integer* ']' | '[' *integer* '-' *integer* ']' |
StateEmit : '`StateEmit`' Cpp
Timing : '`Timing`' Cpp | '`Timing`' '(' Name ',' Name ')' Cpp |
  '`Timing`' '(' ',' Name ')' Cpp | '`Timing`' '(' Name ',' ')' Cpp |

Figure 4: ObjectPro++ node syntax

Symbol	Meaning	Section	Page
BaseDfNodeCtor	constructor for primary base function		
BaseDescription	describes the purpose of a base class		
Body	parts needed only for an executable node		
Comment	C++ style comments can end any line		
Cpp	C++ code delimited with '{' and '}'		
Ctor	`Ctor` constructor code		
Declaration	constructor and base class		
DeclarationCode	C++ class declarations	2	19
Default	marks first default	1.1	2
DefaultList	default values for an array parameter		
DfNode	node definition		
DfNodeCtor	base and member constructors		
Dtor	destructor code	3.2	25
FullDescription	long description of a construct		
Header	first part of node definition		
Help	default help file	6.1	45
Includes	include files	1.1	1
InteractiveClass	class of classes for node	4.1	28
Kernel	kernel code	3.1	21
MemberName	member function name		
Member	member object or function	2	19
MenuLine	synopsis of object, parameter or function		
NodeDescription	documentation of node		
NodeName	class name of node being defined		
NumParamValue	numeric parameter value		
NumberList	list of numbers separated by a ','		
OtherBaseList	member or base function constructor		
ParamValue	numeric (or `enum` parameter value		
Parameter	name, limits and documentation		
ParameterAndCheck	full parameter definition	2.2	20
ParameterList	for constructor or member function		
Parameters	constructor parameters and base		
Safe	code to test if node can be deleted	3.3	25
Size	allowed size of array parameter		
StateEmit	emit state code	3.1.2	22
StaticDeclare	members to preinitialize	3.6	27
StaticInit	target preinitialization	3.6	27
Timing	code for timing	3.1.1	22
Wait	suspends user input until complete		

Table 4: Symbols in ObjectPro++ syntax

Base class	Used for	Parent	header file
BufferDescript	buffering	UserEntity	buffer.h
DfNode	all DSP nodes	TargetNode	dfnode.h
DisplayNode	plotting	DisplayNodeStr	display.h
DisplayNodeStr	DSP output	Node	dsplstr.h
GenericBlockPlot	two dimensional plotting	GenericPlot	blockplt.h
GenericBlockPlotStr	plotting	GenericPlotStr	blkpltstr.h
GenericPlot	plotting	GenericPlotStr	genplot.h
GenericPlotStr	plotting	PlotNode	gpltstr.h
Miscellaneous	miscellaneous	UserEntity	miscel.h
NetControl	network control	UserEntity	netcnt.h
NetworkSystem	network systems	Miscellaneous	netsys.h
Node	all DSP nodes	DfNode	node.h
PlotNode	plotting	DisplayNodeStr	plotnd.h
ProcessNet	networks	UserEntity	network.h
ProcessNode	DSP processing	ProcessNodeStr	procnode.h
ProcessNodeStr	DSP processing	Node	procstr.h
Signal	DSP signals	SignalStr	sigbase.h
SignalStr	DSP signals	Node	signode.h

Table 5: ObjectPro++ base classes

their type, default value and bounds. For most parameters these characteristics are the same for every base class that uses the parameter. If this is not true the parameter entry is repeated for each base class that uses the parameter. These contain the name of the class and the limits and defaults for that class. If there is no class name then the entry applys to all base classes that use the parameter. The table is split both horizontally and vertically into several parts across pages.

1.4.2 Inherited member functions

Base class constructors define the generic characteristics of a node and also determine what common interactive member functions they inherit. Separate

Base class	Constructor parameters
BufferDescript	SIZE TYPE
DfNode	DELAY_IN DELTA_IN DELTA_OUT IN NODE_DELAY OUT OVERLAP STREAM_IN STREAM_OUT TIMING_TYPE
DisplayNode	ARITH_TYPE_IN BLOCK_SIZE DELTA_IN ELEMENT_SIZE IN TIMING_TYPE
DisplayNodeStr	DELTA_IN IN STREAM_IN TIMING_TYPE
GenericBlockPlot	BLOCK_SIZE CAPTION ELEMENT_SIZE MAXIMUM_X MAXIMUM_Y MINIMUM_X MINIMUM_Y NUMBER_BLOCKS SCALE_FLAG TIMING_TYPE
GenericBlockPlotStr	CAPTION DYNAMIC_TYPE MAXIMUM_X MAXIMUM_Y MINIMUM_X MINIMUM_Y NUMBER_BLOCKS SCALE_FLAG STREAM_IN TIMING_TYPE
GenericPlot	BLOCK_SIZE CAPTION ELEMENT_SIZE IN MAXIMUM_X MAXIMUM_Y MINIMUM_X MINIMUM_Y NUMBER_BLOCKS PLOTTING_STREAM_TYPE SCALE_FLAG TIMING_TYPE XY_SAMPLES_PER_PLOT
GenericPlotStr	CAPTION DYNAMIC_TYPE IN MAXIMUM_X MAXIMUM_Y MINIMUM_X MINIMUM_Y NUMBER_BLOCKS PLOTTING_STREAM_TYPE SCALE_FLAG STREAM_IN TIMING_TYPE XY_SAMPLES_PER_PLOT
Miscellaneous	
NetControl	NETWORK
NetworkSystem	
Node	DELAY_IN DELTA_IN DELTA_OUT IN NODE_DELAY OUT OVERLAP STREAM_IN STREAM_OUT TIMING_TYPE
PlotNode	IN SCALE_FLAG STREAM_IN TIMING_TYPE
ProcessNet	
ProcessNode	ARITH_TYPE_IN ARITH_TYPE_OUT BLOCK_SIZE DELAY_IN DELTA_IN DELTA_OUT ELEMENT_SIZE ELEMENT_SIZE_OUT IN NODE_DELAY OUT OVERLAP TIMING_TYPE
ProcessNodeStr	DELAY_IN DELTA_IN DELTA_OUT IN NODE_DELAY OUT OVERLAP STREAM_IN STREAM_OUT TIMING_TYPE
Signal	ARITH_TYPE_OUT BLOCK_SIZE DELTA_OUT ELEMENT_SIZE OUT TIMING_TYPE
SignalStr	DELTA_OUT OUT STREAM_OUT TIMING_TYPE

Table 6: ObjectPro++ base class parameters

Parameter	Description	Class
`ARITH_TYPE_IN`	input arithmetic type[1]	
`ARITH_TYPE_OUT`	output arithmetic type[1]	
`BLOCK_SIZE`	number of samples in a block	
`CAPTION`	plot or listing caption	
`DELAY_IN`	not used	
`DELTA_IN`	input samples for `DELTA_OUT` outputs	
`DELTA_OUT`	output samples for `DELTA_IN` inputs[2]	
`DYNAMIC_TYPE`	fixed or dynamic input data type	
`ELEMENT_SIZE`	input sample size in words	
`ELEMENT_SIZE_OUT`	output sample size in words	
`IN`	number of input channels	
`MAXIMUM_X`	maximum X plot value[3]	GenericBlockPlot
`MAXIMUM_X`	maximum X plot value[3]	GenericBlockPlotStr
`MAXIMUM_X`	maximum X plot value[3]	GenericPlot
`MAXIMUM_X`	maximum X plot value[3]	GenericPlotStr
`MAXIMUM_Y`	maximum Y plot value[3]	
`MINIMUM_X`	minimum X plot value[3]	GenericBlockPlot
`MINIMUM_X`	minimum X plot value[3]	GenericBlockPlotStr
`MINIMUM_X`	minimum X plot value[3]	GenericPlot
`MINIMUM_X`	minimum X plot value[3]	GenericPlotStr
`MINIMUM_Y`	minimum Y plot value[3]	
`NETWORK`	class `ProcessNet`	
`NODE_DELAY`	sampless generated with *no* input	
`NUMBER_BLOCKS`	number of blocks in one display[4]	GenericBlockPlot
`NUMBER_BLOCKS`	number of blocks in one display[4]	GenericBlockPlotStr

Table 7: ObjectPro`++` base class constructor parameters (part 1 of 4)

[1]The arithmetic types are given in Table 3.7 on page 27. The default and Min `enum` values are prefixed with `ArithType`. This was removed to save space in the table.

[2]`DELTA_OUT` is used to set the output link parameter `IncrementOut`. Sorry!

[3]Ordinarily plots are dynamically scaled and these limits are set to allow this.

[4]The current plotting nodes either fix the number of samples in a plot at the block size or allow the user to change the number of samples if the data is not blocked.In some cases these can be overridden with options to fix the samples of blocks per plot.

Parameter	Type	Default	Min	Max
ARITH_TYPE_IN	UserEntity	ArithTypeUndefined	ArithTypeUndefined	MaxArithTypes
ARITH_TYPE_OUT	UserEntity	ArithTypeUndefined	ArithTypeUndefined	MaxArithTypes
BLOCK_SIZE	int32	0	0	2147483647
CAPTION	char *	*NULL*		
DELAY_IN	int32	0	0	2147483647
DELTA_IN	int32	1	1	2147483647
DELTA_OUT	int32	1	1	2147483647
DYNAMIC_TYPE	UserEntity	PlotDynStatic	PlotDynStatic	PlotDynDyn
ELEMENT_SIZE	int32	1	1	2147483647
ELEMENT_SIZE_OUT	int32	0	0	2147483647
IN	int16	1	0	32767
MAXIMUM_X	double	15	-1e+100	1e+100
MAXIMUM_X	double	0	-1e+100	1e+100
MAXIMUM_X	double	0	-1e+100	1e+100
MAXIMUM_X	double	0	-1e+100	1e+100
MAXIMUM_Y	double	0	-1e+100	1e+100
MINIMUM_X	double	-16	-1e+100	1e+100
MINIMUM_X	double	0	-1e+100	1e+100
MINIMUM_X	double	0	-1e+100	1e+100
MINIMUM_X	double	0	-1e+100	1e+100
MINIMUM_Y	double	0	-1e+100	1e+100
NETWORK	ProcessNet	DefProcessNet		
NODE_DELAY	int32	0	-2147483647	2147483647
NUMBER_BLOCKS	int16	1	1	32767
NUMBER_BLOCKS	int16	1	1	32767

Table 8: ObjectPro++ base class constructor parameters (part 2 of 4)

Parameter	Description	Class
`NUMBER_BLOCKS`	number of blocks in one display[4]	GenericPlot
`NUMBER_BLOCKS`	number of blocks in one display[4]	GenericPlotStr
`OUT`	number of output channels	
`OVERLAP`	overlapped input samples [5]	
`PLOTTING_STREAM_TYPE`	eye plot or linear plot	
`SCALE_FLAG`	not used	
`SIZE`	buffer size	
`STREAM_IN`	input data stream description[6]	DfNode
`STREAM_IN`	input data stream description[6]	DisplayNodeStr
`STREAM_IN`	input data stream description[6]	GenericBlockPlotStr
`STREAM_IN`	input data stream description[6]	GenericPlotStr
`STREAM_IN`	input data stream description[6]	Node
`STREAM_IN`	input data stream description[6]	PlotNode
`STREAM_IN`	input data stream description[6]	ProcessNodeStr
`STREAM_OUT`	output data stream description[6]	DfNode
`STREAM_OUT`	output data stream description[6]	Node
`STREAM_OUT`	output data stream description[6]	ProcessNodeStr
`STREAM_OUT`	output data stream description[6]	SignalStr
`TIMING_TYPE`	linear or undefined timing[7]	
`TYPE`	type of buffering[8]	
`XY_SAMPLES_PER_PLOT`	number of samples in an eye plot[4]	

Table 9: ObjectPro++ base class constructor parameters (part 3 of 4)

[4]The current plotting nodes either fix the number of samples in a plot at the block size or allow the user to change the number of samples if the data is not blocked.In some cases these can be overridden with options to fix the samples of blocks per plot.

[5]Any overlapped data must be stored internally. The node will not be called until sufficient input is availible for the first execution. After that the overlapped data may be overwritten on the input buffer.

[6]The stucture that describes data streams is documented in Section 1.5 on page 17. The default value for streams is `new StreamStr(`*name in table*`)`. The constructor was removed from the table (only the parameter was left) to save space and `StreamNotInitialize` was shortened tp `NotInitialized`.

[7]The `TIMING_TYPE` default, min and max `enum` values have `TimingType` as a prefix. This was deleted from the table to save space.

[8]Currently only circular buffers (`CircBufDes`) are supported.

Parameter	Type	Default	Min	Max
OUT	int16	1	0	32767
OVERLAP	int32	0	0	2147483647
PLOTTING_STREAM_TYPE	UserEntity	PlotYs	PlotYs	PlotPairs
SCALE_FLAG	int16	0	0	1
SIZE	int32	512	1	2147483647
STREAM_IN	StreamStr	NotInitialized		
STREAM_IN	StreamStr	StreamNotSet		
STREAM_IN	StreamStr	StreamNotSet		
STREAM_IN	StreamStr	StreamNotSet		
STREAM_IN	StreamStr	NotInitialized		
STREAM_IN	StreamStr	StreamNotSet		
STREAM_IN	StreamStr	StreamNotSet		
STREAM_OUT	StreamStr	NotInitialized		
STREAM_OUT	StreamStr	NotInitialized		
STREAM_OUT	StreamStr	StreamNotSet		
STREAM_OUT	StreamStr	StreamReal		
TIMING_TYPE	UserEntity	Linear	Linear	Random
TYPE	int	1	0	1
XY_SAMPLES_PER_PLOT	double	0	0	1e+100

Table 10: ObjectPro++ base class constructor parameters (part 4 of 4)

base classes are provided for signal creation, DSP processing and display or output nodes. The list of these classes the nodes that belong to them and references to their member functions are in the chapter on 'Class hierarchy' in *ObjectProDSP User's Reference*[2].

1.5 Data stream description

The class `StreamStr` defined in file
`$OPD_ROOT/src/include/ObjProGui/ObjProGui/strmstr.h`
gives the `ElementSize`, `BlockSize` and `StreamArithType` for each channel. The constructor for a node creates one copy of this structure for the input channels and another for the output channels. Whenever practical it is best to write a node that works on as wide variety of data types and that adjusts itself to the data it receives. For example the `Plot` node can operate on real and complex input. It can also tell it its input is from an FFT (using the `BlockSize` parameter) and generate a frequency axis. One does not need to set parameters for these inputs. The `Plot` node reads the information from the `StreamStr` object for its input channel.

The `STREAM_IN` and `STREAM_OUT` parameters for a node can be set to specific values. To do this use the constructor for `StreamStr` that explicitly sets the values. There are predefined objects for unblocked real (`StreamReal`) and unblocked complex `StreamComplex` data. To use these use the constructor that operates on an existing `StreamStr` object, i. e. write
`STREAM_IN = new StreamStr(StreamComplex)`
to for a node with a complex input stream. For a node that adjust to the input block and sample size use the predefined object `StreamNotSet`.

Signal generation nodes that read disk files (such as `InputNode`) may not be able to set the block and sample size until after they read the file that contains this information. Use the predefined object `StreamNotInitialized` for these.

Most of the predefined objects set the `ArithType` to the default type of the simulator. A few allow you to set the output type, (`Block`) or do type conversions(`ToInteger` and `ToMach`). Only the 32 bit floating point version of the simulator supports both integer and floating point data streams.

```
// Dummy class to limit name space polution
class ArithType {
public:
    enum ArithTypes {ArithTypeUndefined=0,ArithDouble=1, ArithInt16=2,
        ArithInt32=3, ArithFloat=4, MaxArithTypes=4};
    enum ArithCapabilities {
        ArithCapabilityFixed=    MaxArithTypes+1,
        ArithCapabilityAnyInt=    MaxArithTypes+2,
        ArithCapabilityAny=    MaxArithTypes+3,
        MaxArithCapabilities=    MaxArithTypes+3};
    static const char * CppNames[] ;
    static const double Accuracy[MaxArithTypes+1];
    static const char * target_sub_dir[MaxArithTypes+1];
    static const int SizeInBytes[MaxArithTypes+1];
    static const char * CapabilityNames[MaxArithCapabilities+2] ;
    static const char * ClassSuffix[MaxArithCapabilities+2] ;
    ArithType();
};
```

Figure 5: `ArithTypes` class for data stream arithmetic types

It is possible to write nodes that will operate on either 32 bit fixed point or 32 bit floating point data. This is done using the arithmetic types defined in Figrure 5 on page 18.

The arithmetic type of an output data stream must be defined before any node is linked on one of its output channels. The input arithmetic type can be defined any time before the network is executed. There are two virtual functions that are used to set these types. `input_linked` is called after the first input channel to a node is linked. `InitAfterLinked` is called after the network has been completely linked but before it is executed. If you need to propagate an arithmetic type from the input of a node to the output you must do this in `input_linked`. If you only need to adjust the input type based on the data being input you can use `InitAfterLinked`. There is a standard base function `DfNode::propagate_arith_type` that will propagate the input arithmetic type from the first channel linked to all other input channels and all output channels. Figure 1 on page 3 is an example of how these functions are declared.

Information on the arithmetic type of channels is only available for the interactive nodes and networks. Flags that determine the type should be set in the above functions and these flags should be declared `Static` as described in Section 3.6 on page 27 so the values set by the interactive code will be preinitialized in the target code.

2 Member objects and functions

Member objects and functions can be declared in three ways. Members declared in the `Class` construct are standard C++ members. They are not accessible interactively and passed unchanged by ObjectPro++ to C++.

Members objects declared inside the `StaticDeclare` construct are treated similarly for interactive execution. It is assumed that these are initialized in the interactive simulator prior to execution. These are then initialized to those value when code is generated for stand alone execution as explained in Section 3.6 on page 27.

2.1 Interactive member functions

The declaration for an interactive member function is similar to the definition of an interactive class constructor. Entries for each member function will appear in the menu data base under the class and under each instance of the class. The functions can be called from the `instance` menus and can be called in DSP++ statements entered directly.

Member objects can be declared at the same place interactive member functions are as the `Member` syntactic construct. Currently there is little difference between these and member functions declared in the `Class` construct. If a variable *name* is declared in this way then a function to access the variable called `Get`*name*`()` is generated. This is currently only a C++ and not an interactive function.

```
ScaledMachWord * Coeff [ 3 - 1024 ] : -30000 <= {
      1.00018552e-01,
      3.70747893e-01,
      4.46594395e-01,
     -5.36969535e-01
} <= 30000
```

Figure 6: Declaring an interactive array parameter

2.2 Parameter checking

The most common form of parameter checking specifies upper and lower inclusive bounds. The key words `Min` and `Max` can be used to specify the limits for a particular data type.

For array parameters the limits apply to each value in the array. One must also specify lower and upper limits on the array size as shown in Figure 6 on page 20. The array parameter `Coeff` in this example must contain at least 3 and no more than 1024 elements. Each element can be between -30000 and 30000.

The type `ScaledMachineWord` is provided to allow values to be scaled as appropriate for a given simulator arithmetic. For the floating point simulator the values will be as shown. For the 16 bit simulator 1.0 is multiplied by `NornToOneMachWord` or 32768. This is defined in file
`ObjProArith/normone.h`.
The values inside the braces define the default array which has four values.

You can include more general checks on parameters by placing code after the `CheckParameter` key word in the parameter definition. Typically this is a subroutine call and not the code to do the check. Examples are in
`$OPD_ROOT/src/dsp/nodes/sigdsk/import.usr`. The body of the called subroutines is in
`$OPD_ROOT/src/dsp/nodes/sigdsk/common/importi.C`.

3 Node semantics

There are several constructs that specify the semantics of the node all of these (except the `Kernel`) are optional. These all consist of a key word followed by C++ code enclosed in braces.

- `Kernel` Code to process data. This section is required.
- `Timing` Time of first sample output.
- `StateEmit` Non standard state definition.
- `SafeDelete` Test if this object can be deleted.
- `Destructor` Delete an instance of the node.

3.1 Node kernel

The code following the key word `Kernel` is used to define the body of a member function `DoNode` with the single `int32` parameter, `k`. This is the number of 'chunks' to process. The chunk size is the product of the `BlockSize`, the `ElementSize` and `IncrementOut`. (`IncrementOut` is set by the `DELTA_OUT` parameter described in Table 7 on page 13. The other values are set in the same way and the other names are more consistent.

`DoNode` will ordinarily not be called unless there is enough available input data and output buffer space to process `k` blocks. If there is not a standard relationship between the input samples available and the output space needed, `TimingTypeRandom` (see Section 3.1.1 on page 22), then it may not be possible to process `k` blocks. It is never necessary to process this much data for interactive execution or for target execution with dynamic scheduling. With fixed scheduling a node must process exactly the data it is requested to process. It is always necessary to update the buffer pointers to reflect the data processed and output written.

The simplest way to process data is to use `ReadWord` to read an input sample and `WriteWord` to write and output sample. These adjust the buffer pointers and report an error if you wrap the buffers. This is inefficient. Data can be accessed and updated more efficiently using the routines that access data via pointers. The declarations for these routines are given in 7 on page 23 and their purpose given in Table 11 on page 24. These are all member functions of `TargetNode` (defined in `ObjProDSPcom/tarnod.h`) and thus available in both interactive and stand alone execution. Many of these routines come in versions for `IntegerMachWord` and `MachWord` data. This difference is only meaningful in the floating point simulators. There is also an `UnsignedIntegerMachWord` data type.

Classes `ReadWriteSingleChannel` and `ReadWriteBlock` defined in `ObjProDSPcom/blckwrt.h` can simplify the use of these routines for some applications.

When the node completes execution it should return the `enum OK` if no problems were encountered and `FatalError` if a error sufficient to halt network execution was encountered. Other possible return values as defined in file `ObjProGen/errcode.h` are `OutputBuffersFull`, `Warning`, `EndOfData` and `ExecutionComplete`. These other options are primarily for internal use.

3.1.1 Node timing

Ordinarily initial node timing is computed from base function constructor parameters. These define the relationship between input and output data as described in *ObjectProDSP User's Reference*[2] in the 'Data flow model' subsection of the chapter on 'The DSP++ language". You can override this with the code in the `Timing` construct or you can directly define the virtual function
`double` *class_name*`::TimeFirst(DfNodeInLink *In, DfNodeOutLink * Out)` that this construct emits. With the former you do not need to use conditionals to keep the declaration out of the target code. This routine returns the absolute sample time of the first sample output.

3.1.2 Emitting the state

When the state of ObjectProDSP is saved a file is written that contains a call to the constructor for each object currently defined. The parameters to

```
void UpdateRead(int32 size, int chan=0) ;
void UpdateWrite(int32 size, int chan=0) ;

const MachWord * GetReadPtr(int chan=0) const ;
const MachWord * GetReadBase(int chan=0) const ;
const MachWord * GetReadEnd(int chan=0) const ;

MachWord * GetWritePtr(int chan=0) const ;
MachWord * GetWriteBase(int chan=0) const ;
MachWord * GetWriteEnd(int chan=0) const ;

const UnsignedIntegerMachWord * GetBinReadPtr(int chan=0) const ;
const UnsignedIntegerMachWord * GetBinReadBase(int chan=0) const ;
const UnsignedIntegerMachWord * GetBinReadEnd(int chan=0) const ;

UnsignedIntegerMachWord * GetBinWritePtr(int chan=0) const  ;
UnsignedIntegerMachWord * GetBinWriteBase(int chan=0) const ;
UnsignedIntegerMachWord * GetBinWriteEnd(int chan=0) const ;

void WriteInteger(IntegerMachWord Data, int chan=0) ;
void WriteWord(MachWord Data, int chan=0) ;

IntegerMachWord ReadBinary(int chan=0)  ;
MachWord ReadWord(int chan=0)  ;

int32 GetAvailableData(int chan=0) const ;
int32 GetSpace(int chan=0) const ;
int32 GetContiguousAvailableData(int chan=0) const ;
int32 GetContiguousSpace(int chan=0) const ;

void WriteCxWord(CxMachWord Data, int chan=0) ;
CxMachWord ReadCxWord(int chan=0) ;
```

Figure 7: Data access routines, also see Table 11

Routine	Operation
`UpdateRead`	update read pointer on `chan` `size` words
`UpdateWrite`	update write pointer on `chan` `size` words
`GetReadPtr`	get read pointer on `chan`
`GetReadBase`	get base of input buffer on `chan`
`GetReadEnd`	get address past end of input buffer on `chan`
`GetWritePtr`	get write pointer on `chan`
`GetWriteBase`	get base of output buffer on `chan`
`GetWriteEnd`	get address past end of output buffer on `chan`
`GetBinReadPtr`	same as `GetReadPtr`, unsigned integer pointer
`GetBinReadBase`	same as `GetReadBase`, unsigned integer pointer
`GetBinReadEnd`	same as `GetReadEnd`, unsigned integer pointer
`GetBinWritePtr`	same as `GetWritePtr`, unsigned integer pointer
`GetBinWriteBase`	same as `GetWriteBase`, unsigned integer pointer
`GetBinWriteEnd`	same as `GetWriteEnd`, unsigned integer pointer
`WriteInteger`	write one integer word, update pointers
`WriteWord`	write one `MachWord`, update pointers
`ReadInteger`	read one integer word, update pointers
`ReadWord`	read one `MachWord`, update pointers
`GetAvailableData`	get number of words available on input `chan`
`GetSpace`	get words of space on output `chan`
`GetContiguousAvailableData`	get number of contiguous input words on `chan`
`GetContiguousSpace`	get words of contiguous space on output `chan`
`WriteCxWord`	write one `CxMachWord`, update pointers
`ReadCxWord`	read one `CxMachWord`, update pointers

Table 11: Data access routines, also see Figure 7

this call are the current values for them in the object. This should work for any DSP nodes but does not work for other objects such as networks. For these the `StateEmit` construct defines a virtual function
`const char *` *class_name*::EmitState(OutTokens& Out)
that emits the state. `OutTokens` is a class used to output text for various purposes. It is not usually needed in writing nodes. It is defined in file `ObjProGen/outtok.h`.

3.2 Node destructor

If the node allocates dynamic storage or can be referenced by other objects in some way other then the standard network interface you need to write a C++ destructor for it using the ObjectPro++ `Ctor` construct. The should `delete` any dynamically allocated objects and remove any external references to them or to the node being deleted.

Base class constructors insure that all references through the network are deleted and that any menu references including those currently displayed are removed. Deleting an object can cause displayed menus to disappear.

3.3 Check if node is deletable

Interactively deleting objects can create problems. For example one cannot delete a network without removing any nodes linked in that network. Otherwise those nodes will reference a deleted object. Most nodes are only referenced through the standard network interface. If this is true they can be deleted at any time. However if other objects can reference a node in non standard ways these references must be removed as part of the destructor for the object (see Section 3.2 on page 25) or the object must be flagged as not interactively deletable.

Before the constructor for a node is called the virtual function
`int CheckSafeDelete()` is called. If this returns 0 the destructor is not called and the object is not deleted. The ObjectPro++ construct `SafeDelete` defines this virtual function. The first two lines of the emitted code are:

```
int Safe_Check_Return = UserEntity::CheckSafeDelete();
if (!Safe_Check_Return) return 0;
```

The code you write comes after this and can include additional tests.

3.4 Interactive code

The code you generate is used for both target and interactive execution. There are some things you can do to make the interactive code more user friendly. For example if an error condition is encountered you should report it with the a call to `State.Error()`. There must be one parameter and can be up to 8 of type `const char *`. When concatenated these parameters should produce a meaningful error message.

You can direct messages to the user using the C++ stream like construct `HelpOut`.

Debugging messages can be directed to log file `dsp.messages` using the stream `LogOut`.

All of the above can be used for both stand alone and interactive code. For example `HelpOut` is defined is `cerr` for stand alone code.

If you need to write code that is only used for interactive execution you can use the C++ define `INTERACTIVE`. This is only defined for code compiled for interactive execution.

3.5 Stand alone target code

There are many differences between code compiled for stand alone target execution and that compiled for interactive execution. All stand alone nodes have the same base class, `TargetNode`, with a simple fixed set of constructor parameters. The interactive node base classes have `TargetNode` as a parent. All the interactive code such as code to describe parameters or set them is stripped from the target code.

As much as possible values are precomputed and initialized as part of the process of emitting the target C++ code. This can minimize both code space and execution time. There are facilities to allow user defined variables and arrays to be initialized during interactive execution and those values emitted as static initializers for the target code. These are described in the next section.

3.6 Preinitialized target code

Objects and arrays declared in the ObjectPro++ construct, `StaticDeclare`, are initialized during interactive execution and those values are used as static initializers in generating target stand alone code. The initialization is most commonly done in two places. Code in the `StaticInit` construct is only output in the interactive code and occurs in the node constructor before the code output in the `Ctor` construct. You can put initialization code here that will not be needed on the target. The code in the `Ctor` section can assume this initialization has been done before it starts regardless of whether the code is executing on interactively or as a stand alone target.

In some cases static initialization cannot be completed until the node is linked into a network. For example this may be necessary for a node that can operate on different input data types. This can be done in virtual function `InitAfterLinked`. This function should only be defined interactively by using C++ conditional compilation, `#ifdef INTERACTIVE`
or by placing it in a `.C` file in the same directory that the `.usr` file is.

3.7 Target arithmetic

For the floating point simulator target arithmetic using the types `MachWord` and `AccMachWord` is done with single precision floating point arithmetic.

For the 16 bit simulator these two data types represent 16 and 32 bit accuracy that might be available on an integer DSP processor. Multiply for this type is defined as if the number was a binary faction with the decimal point to the left of the most significant non sign bit. Thus fractions between 1 - 1/32768 to

Name	Type of objects
`InteractiveNode`	DSP processing nodes (default)
`InteractiveNet`	networks
`InteractiveBuffer`	buffers between nodes
`InteractiveScheduler`	network controllers
`InteractiveSignal`	input or signal generation nodes
`InteractiveDisplay`	output or display nodes
`InteractiveMiscellaneous`	other types of objects

Table 12: Groups of classes for placement in data base

-1 are represented. Multiplication by two full scale values will never produce an overflow although the product of -1 $\times$ -1 will be 1 - 1/32768 and not 1 which cannot be represented.

You can write nodes that will work with both simulators but you must be careful with normalization when using integer arithmetic.

You can define arithmetic operations that exactly match the hardware of your target processor and create a new simulator for that arithmetic. That is outside the scope of this manual. The files to do this for the existing simulators are in directory
`$OPD_ROOT/src/dsp/arith`
and subdirectories of this directory. The header files are in directories under `ObjProFlt` and `ObjProInt16` which are subdirectories of
`$OPD_ROOT/src/include`.

4 Integrating a new node

4.1 Integration with the menu data base

ObjectPro++ outputs a `.nod` file into directory
`$OPD_ROOT/src/menus/nodes`
for integration of the node with the menu data base. You must specify where

this documentation is to be placed. If you do nothing special it will treated as a DSP processing node. You can specify the menu it is to be added to as described in Section 1.2 on page 6.

You can use the `InteractiveEntityList` construct to define the class of classes the node belongs to. If you did not give a specific menu for it this will determine where it is linked. The possible classes are listed in Table 12 on page 28.

4.2 Integrating with the documentation

The two `.tex` files are written to subdirectories `tex` and `texs` these are used to integrate the node into the ObjectProDSP manuals. The `texs` files are used in *ObjectProDSP Overview and Tutorial*[1]. The `tex` files are used in *ObjectProDSP Library Reference*[3]. The `.nod` files are used to create the `.tex` documentation of the data base in *ObjectProDSP User's Reference*[2].

4.3 Integrating a node into ObjectProDSP

The simplest way to integrate a node is to place it in a directory in which there are already `.usr` files. The standard directories are listed in Table 13 on page 30. You should manually execute routine
`$OPD_ROOT/bin/mknode`
with the new `.usr` file as a single argument. This will create output `.C` and `.h` files. Then go to directory
`$OPD_ROOT/src/dsp/build/fltgui` or
`$OPD_ROOT/src/dsp/build/int16gui`
for the floating point or 16 bit simulators and execute the command `domake-make`. Then go back to the directory with the new `.usr` file and `touch` that file so it will be processed again with `mknode` using the correct parameters for creating the documentation and target files. Then you can go to the appropriate subdirectory (`fltgui` or `int16gui`) and do a `make` to run `mknode` and the C++ compiler for your new node.

To rebuild ObjectProDSP with your new node go back to the master build

Directory under `$OPD_ROOT/src/dsp/nodes`	Type of nodes
`proc`	DSP processing nodes
`proc16`	DSP nodes only for 16 bit simulator
`proc32`	DSP nodes only for 32 bit simulator
`sigdsk`	input nodes that read disk files
`signal`	other signal nodes
`ionode`	input/output nodes supported on all targets
`display`	output nodes

Table 13: Directories to place new nodes in

directory for the version you are working with (`$OPD_ROOT/src/dsp/build/fltgui` or `$OPD_ROOT/src/dsp/build/int16gui`) and do a `make`. To include your new node in target libraries go to the master build directory (`$OPD_ROOT/build`) and do a `make TOUCH_DOMAKEMAKE` followed with a `make EXE`. Remember to do the latter using `make_both.sh` if you want to update both the 16 bit integer and floating point versions. Otherwise only the floating point versions will be updated. After ObjectProDSP has been built with you new node you can update the documentation by doing `make DOC` in the master build directory.

4.4 Removing a node from ObjectProDSP

To remove a node you must remove the node source file and all the files created by `mknode` from the source file when `mknode` is invoked with the options specified by `makemake`. These files are shown in Table 14 on page 31. After that do `make TOUCH_DOMAKEMAKE` and `make EXE` in the master build directory (`$OPD_ROOT/build`). Note file `$OPD_ROOT/src/include/initinc.h` references file *node_name*`I.h` for each node. `initinc.h` is created when `domenus` is run based on the `.nod` files in `$OPD_ROOT/menus/nodes`. You must remove the `.nod` file before running `domenus` from `make`. Do not worry

File	Purpose	Directory (. is relative to node others are relative to `$OPD_ROOT/src`)
name`.C`	interactive source	.
name`.C`	target source	`./target`
name`.h`	interactive source	`include/ObjProDSPint/ObjProUsr`
name`.h`	target source	`include/ObjProDSPtar/ObjProUsr`
name`.tex`	full documentation	`./tex`
name`.tex`	synopsis documentation	`./texs`
name`I.h`	default objects	.
name`.nod`	interactive documentation	`menus/nodes`
name`.e`	text for spelling check	subdierecroty `mknode` is executed from

Table 14: Files created by `mknode` (with `makemake` options) from *name*`.usr`

if you get an undefined reference to the `I.h` file when you run `domakemake`. This reference will be absent when `initinc.h` is recreated when `domenus` is run.

You will also need to rebuild the files in `$OPD_ROOT/overview/doc` that reference the `.tex` files. Do a `fgrep` in this directory on `*.tex` for the base name of the file *name*`.usr` file you are deleting. Make sure the two files that reference *name*`.tex` are created by `make` and them delete them so they will be remade the next time the documentation is rebuilt.

5 Modifying the menu data base

The menu data base is defined by a text file `opd.menu` in directory `$OPD_ROOT/menus` and by the `.nod` text files created by ObjectPro++. In this section we describe the syntax and semantics of these files.

5.1 Menu data base menus

Figure 8 on page 34 is the start of the definition of the main menu from that file. The keyword `Menu` followed by '{' indicates the start of the menu. The items in the menu are delimited by the opening and closing braces. The closing brace for the main menu is not in the figure because this is only a part of the meu. Next is the menu name `MainCgi`. This id used in higher level menus to specify a submenu. For this main menu it is referenced at the start of the file to define the root of the menu tree. (The syntax there that supports multiple main menus is obsolete and not documented but you do need to copy that structure to specify the root menu.)

Following the menu name is a '`:`' and text that describes the menu. This text is displayed above the buttons for the menu when the cursor is not positioned over any of those buttons. Next is a sequence of menu items each of which corresponds to a button in the menu. If there are more items then there is space for buttons submenus under '`other`' are automatically created.

5.2 Menu items

The first menu item starts with the command or label for the button, `help`, followed by a '`,`' and the name of the structure corresponding to that command the character '`=`' and the type of the command. In this example all three menu items reference submenus. Next there is a '`:`' and text describing the menu item. This is the text displayed above the menus when the cursor is positioned over the button for this item. Next is optional text enclosed in braces that provides a more complete description of this item. Any of this text enclosed with '`$`' characters should refer to user commands.

```
Menu { MainCgi            : ObjectProDSP menu data base

help,HelpMenu=Menu        : Main help menu {
    The $help$ menu contains information organized by topics.
    It covers the ObjectProDSP language and describes the use
    of this program. It has information for the new user and is
    an on-line reference manual.  You can control the amount of
    automatic help information from
    this menu.  }
HelpFile : help

objects,AllCls=Menu    : Display and describe existing objects {
    The $objects$ menu provides tree structured access to the
    definitions and descriptions of objects. It allows objects
    to be created and destroyed.  }
HelpFile : objects

setup,FilesMenu=Menu        : Read state and plot files, debugging {
    From the $setup$ menu
    you may read and execute a ObjectProDSP state file created
    in a previous session or created manually. You may also read
    a plot file created in a previous session and control debugging
    options.  }
HelpFile : setup
```

Figure 8: First part of the definition of the main menu

These are set in `typescript font` in the documentation and index entries are created for them. Such text is enclosed in single quotes when the text is displayed interactively. This description of the item is displayed in the `help information` window when the left or right mouse button is *released* with the cursor over this button. This can be disabled by changing the help levels.

If the same text is used in several menu items it can be defined once with the `HelpDefinition` { *definition_name help text* }
construct. The text can then be referenced many times with the '*definition_name*=`HelpText`' option in the syntax for a menu item.

5.3 Help files

Finally is the keyword `HelpFile` a '`:`' and the base name for the help file. This is optional. See Section 6.1 on page 45 for more information on these files. Instead of a help file base name the key word '`Default`' may be used. This refers to the last help file specified with the `HelpFileDefault` construct.

5.4 Menu qualifiers

There are several qualifiers that can precede the key word menu.

- `History` The action taken may be a function of previous selections using action parameters as described in 5.7 on page 36.
- `Dynamic` The menu may have items added to it after executions starts.
- `Multiple Use` The menu may occur at multiple points in the tree with different initializations (obsolete).
- `Orphan` A menu that has no predefined place in the tree. This is used for member functions of a base class. The class member functions are not included in the tree through the class itself but through other classes derived from the base class. The `Orphan` menu generated for such a class is merged with the member function menus of all derived classes that occur in the tree.

- `Select` The actions are initialized at execution time based on a `Template` menu item. See Section 5.7 on page 36.

5.5 Simplified menu syntax

Figure 9 on page 37 gives a simplified `yacc` (or `bison`) syntax for the menus. The rules are in alphabetical order. The top level symbol is ObjectList. Literal terminal symbols are in `typewriter` font and enclosed in single quotes. Standard terminal symbols like *name* are in *italicized font*. If a rule ends with '|' it can be reduced to the null symbol.

Table 15 on page 38 gives a brief explanation for each nonterminal symbol and references for some. The full `yacc` (or `bison`) syntax is in file `$OPD_ROOT/src/util/mkmenu/menu_b.y`.
Note some constructs in the full syntax are for future expansion and others are obsolete but retained for backwards compatibility.

5.6 Writing menu action code

A menu item can select another menu display a help file or perform some action. These actions can be `Local` (executed on the user interface process) `Remote` (executed on the DSP process) or `LocalRemote` (executed on both). The name for an action (that precedes the character '`ActionType`') should be the name of a C++ subroutine in the appropriate executable. Dummy stubs for all actions are generated unless the action name is listed in the `DefinedAction` construct. After you have written the code for an action you must add its name to this list or you will get a double definition of the subroutine when you link.

5.7 Examples menu and action parameters

Actions can have parameters which refer to previous menu selections. This is most commonly used in menus generated by ObjectPro++ described in

Action : Name Parameters | Name Parameters '`Wait`' | '`Orphan`'
ActionType : '`=Local`' | '`=Remote`' | '`=LocalRemote`' | '`=RemoteOptions`' | '`=Menu`' | '`=DynamicMenu`' | '`HelpFile`'
ClassRelation : '{' *name name* '}'
Command : WordString
Comment : '`//`' *text to end of line*
DefinedActionList : '`DefinedAction`' '{' NameList '}'
Help : '{' *text* '}'
HelpDef : '`HelpDefinition`' '{' *name help text* '}'
HelpDefaultFile : '`HelpDefaultFile`' *quoted string*
HelpFile : `=HelpFile` '`base_file_name` | '`Default`' |
HelpRef : '`,`' *name* '`=HelpText`'
InitEntry : '`Init`' '{' *C++ code* '}' *name* '=' PriorNameList ';'
| '`Init`' '{' *C++ code* '}' *name* ';'
Menu : MenuId '{' Name ':' MenuTitle MenuBody '}'
| '`Add To Menu`' '{' *name* MenuBody '}'
MenuBody : MenuBody MenuItem |
MenuId : Qualifier '`Menu`'
MenuItem : Command '`,`' Action ActionType HelpRef ':' *text* Help HelpFile
| Command '`,`' Action ActionType HelpRef ':' *text* HelpFile
| Command '`,`' Action ActionType ':' *text* Help HelpFile
| Command '`,`' Action ActionType ':' *text* HelpFile
| '`Template`' '`,`' Action ActionType ':' *text* HelpFile
| Command '`=Reference`'
MenuStackReference : '`[`' *nonnegative integer* '`]`' MenuTitle : *text*
NameList : *name* | NameList '`,`' *name*
Object : Menu | HelpDef | HelpDefaultFile | DefinedActionList |
InitEntry | ClassRelation
ObjectList : Object | ObjectList Object
Parameter : *number* | *quoted string* | MenuStackReference
ParameterList : ParameterList '`,`' Parameter | Parameter
Parameters : '(' ParameterList ')' |
PriorNameList : *name* | PriorNameList '`,`' *name*
Qualifier : Qualifier '`Dynamic`' | Qualifier '`Multiple Use`'
| Qualifier '`History`' | Qualifier '`Select`' | Qualifier '`Orphan`' |
WordString : *text*

Figure 9: Menu data base syntax

Symbol	Meaning	Section	Page
Action	what to do if selected	5.6	36
ActionType	what type of action	5.6	36
ClassRelation	class and base class	5.8	39
Command	user command name	5.2	33
Comment	C++ style comments can end any line		
DefinedActionList	actions implemented	5.6	36
Help	help text for an item	5.2	33
HelpDef	help text used more than once	5.2	33
HelpDefaultFile	default help file name	5.3	35
HelpFile	help file action	5.6	36
HelpRef	reference to help text	5.8	39
InitEntry	initialization of default objects	5.8	39
Menu	complete menu	5.1	33
MenuBody	items in a menu	5.2	33
MenuId	menu type and internal name	5.1	33
MenuItem	one entry under a menu	5.2	33
MenuStackReference	previous menu reference	5.6	36
MenuTitle	displayed title of menu	5.1	33
NameList	list of defined actions	5.6	36
Object	menu or other major syntactic element		
ObjectList	top level syntactic object		
Parameter	action parameter	5.7	36
ParameterList	list of parameters	5.7	36
Parameters	parameters for an action	5.7	36
PriorNameList	required objects	5.8	39
Qualifier	type of menu	5.4	35
Wait	disables input until command completes		
WordString	command name	5.2	33

Table 15: Symbols in ObjectPro++ syntax

Section 5.8 on page 39. Action parameters are also used in the examples menu. The top level menu for examples is a `Select` menu that has a single `Template` menu item. This template will be copied for every item under this menu. This is done by reading a directory where examples are stored and generating an entry for every file with a `.xml` suffix in that directory. The code to update the menus is in class `DynamicMenuServer` defined in files `dynmnu.h` and `dynmnug.C` in directory
`$OPD_ROOT/src/dsp_gui/gui`
The code that calls the `DunamicMenuServer` with the action names for the menu items is in the constructor of `ExamplesDspPP` in file
`$OPD_ROOT/src/gui/lib/examp.C`.

The definition of the '`Example networks`' menu subtree from `opd.menu` is shown in Figure 10 on page 40. The first top level menu contains a single `Template` menu item that will be duplicated for each example found as just explained. The rest of the figure contains the menu items in the '`Describe or execute this example`' menu. Each of these actions has at least one parameter that refers to the selection in the previous menu. In the first item the action is `DescribeExample([1])`. When this item is selected subroutine `DescrubeExample` will be called with a single '`const char *`' parameter that will be the label on the button selected. This is denoted by '`[1]`' in the menu definition. The integer one refers to the most recent menu selection. Earlier selections can are denoted by larger integers.

The `ExecutExample` action has a second parameter: `"no"`. An action can have any number of parameters and they can be literal strings integers or previous menu selections.

5.8 Menus generated by ObjectPro++

Figures 11 on page 41 and 12 on page 43 show the first and last parts of the menus generated by ObjectPro++ for the node defined in `const.usr`. This starts with a line that designates `ConstantData` as a derived class from `Signal`. This will insure any member functions of `Signal` will also appear in the menus under `ConstantData`. This is followed by `Init` and C++ code enclosed in braces to initialize default instances of this class. This code is

```
Multiple Use Dynamic Select Menu { ExampleMenu       : Example networks
Template,ExampOptMenu=Menu    : Select this network
}

History Menu { ExampOptMenu    : Describe or execute this example

desc,DescribeExample([1])=Local    : Describe this example

HelpFile:examp

execute,ExecuteExample([1],"no")=Local : Execute this example {
    This executes the selected example. First a graphical display
    of the DSP network will appear. This will be followed by
    windows for each plot or listing object in the network. When
    execution completes you can edit the example network. You can
    only execute an example once because after that all the objects
    in the example will be defined and cannot be redefined
    with the execute command.}
HelpFile:examp

execute over,ExecuteExample([1],"over")=Local
    : Execute this example and overwrites existing objects {
    This executes the selected example. First a graphical display
    of the DSP network will appear. This will be followed by windows
    for each plot or listing object in the network. When execution
    completes you can edit the example network. $execute over$
    overwrites any objects in the example that are already defined.
    You can execute this command as many times as you want but it can
    also overwrite objects you have defined.
    Be careful in using this option.}
HelpFile:examp
}
```

Figure 10: ‘Example networks’ menu tree definition

```
{ ConstantData Signal }
Init {
#include "ObjProUsr/const.h"
    ConstantDataDef = new ConstantData("ConstantDataDef", 1024);
} ConstantDataDef ;
HelpDefinition { ConstantDataMainHelpDefinition
      'ConstantData'   writes parameter   'Value'   to the output stream
    repeatedly. It is written as a binary integer constant.
}
Add To Menu { SignalNodesMenu
ConstantData,ConstantDataNodeOptMenu=Menu,
    ConstantDataMainHelpDefinition=HelpText :
    generate a 'MachWord' constant
HelpFile : signalT
}
```

Figure 11: First part of `ConstantData` menus generated by ObjectPro++

collected in file `meninit.C` in routine `InitAllMenuRoutines` under directory `$OPD_ROOT/src/menus/dsp`.

Next is statement`Add To Menu` which provides the connection to the full menu tree. It causes an item to be added under the menu named `Signal-NodesMenu`. This connection is specified by giving the menu name after the key word `MenuLine` for the node. In `opd.menu` menu `SignalNodesMenu` is defined without any entries for commands or submenus.

Next is a `Dynamic` menu to select an instance of `ConstantData`. This menu has no predefined entries. It is a `Template` used to add entries to the menu as new instances of `ConstantData` are created. Next is the submenu
`ConstantDataInstancesAccessMenu`
that operates on the selected instance. This is done by referring to previous selections in the action. For example the action
`DescribeNodeInstance("ConstantData",[1])=Remote`
calls the function `DescribeNodeInstance` in the DSP process with parameters `"ConstantData"` and a string giving the name of the selected instance

of `ConstantData`.

5.9 Menus documented at the top level

The menus are automatically translated to documentation for the *Object-ProDSP User's Reference*. The table of contents of this document has a hierarchy that mirrors the menu hierarchy. To prevent too many levels the menu for all classes is forced to the top level i. e. `\section` level. The name of this menu, `AllCls` is hardwired into program
`$OPD_ROOT/src/util/mkmenu/menmain.C`.
It is also used in
`$OPD_ROOT/doc/userman/menu.tex`.
If you change the name of this menu you wiil need to edit those files.

However to keep the table of contents from getting t going too many levels deep some menus are forced

```
Multiple Use Dynamic Select Menu { ConstantDataInstancesMenu :
        Select an Instance of 'ConstantData' t
Template,ConstantDataInstanceAccessMenu=Menu   :
        Select this instance of 'ConstantData'
}

History Menu { ConstantDataInstanceAccessMenu:
        Describe or delete an instance of  object 'ConstantData'
desc,DescribeNodeInstance("ConstantData",[1])=Remote,
    NodeInstanceDescribeHelp=HelpText   :
        Describe this instance of 'ConstantData'
HelpFile : Default
param,ConstantDataInstanceParamMenu=Menu,
    ParamInstanceDescribeHelp=HelpText  :
        Describe parameters of this 'ConstantData'
HelpFile : Default
exec, Orphan=Menu,
    HelpMemberExecute=HelpText  :
        Select a member of 'ConstantData' to execute
HelpFile : Default
variables,ConstantDataInstanceVariableMenu=Menu,
    VariableInstanceDescribeHelp=HelpText   :
        Describe variables of this 'ConstantData'
HelpFile : Default
set,ConstantDataSetInstanceVariableMenu=Menu,
    VariableInstanceDescribeHelp=HelpText   :
        Set variable values of this 'ConstantData'
HelpFile : Default
delete,DeleteNodeInteractiveEntity("ConstantData", [1])=Remote,
    NodeDeleteHelp=HelpText : Delete this 'ConstantData'
HelpFile : Default
}
```

Figure 12: Last part of `ConstantData` menus generated by ObjectPro++

List file	Directory $OPD_ROOT/doc/	Manual	Manual file
examp_list	examptex	*ObjectProDSP User's Reference*	exampmn.tex
node_list	nodetex	*ObjectProDSP Library Reference*	ovnodlst.tex
help_list	helptex	*ObjectProDSP User's Reference*	mnroffmn.tex

Table 16: Lists of help files

6 Updating the documentation

6.1 Help files

The help files referenced in the menu data base and the pull down menus are also used in the printed documentation. They are written in a very restricted `groff` format using the `mm` macro package. They are translated to LaTeX format for inclusion in the manuals. The originals are in directory `$OPD_ROOT/doc/roff`. A help file name `help.hlp` comes from a file *X*`help.roff` or *X*`helpT.roff`. '*X*' can be any single character. The optional '`T`' indicates the roff file must be processed with `tbl`. Such files must be displayed with a fixed font or the tables will be skewed. The 'T' is used to control this.

6.1.1 `Makefile` to process help files

A subdirectory `mmake` under the roff directory contains tools for creating the `Makefile` to process the help files with `groff` and LaTeX. Each help file name is listed in one of three files shown in Figure 16 on page 45. These are read by `mmake` (source file `mmake.C`) to create the `Makefile` in the parent directory. If you add an example or help file you need to update these list files and do a `make` in the directory they occur to create the `Makefile` with your changes in the parent directory.

Table 16 on page 45 also shows the the directory (under `$OPD_ROOT/doc`) that the generated LaTeX files are written to, the manual those files are incorporated in and the file for that manual that references these files. `exampmn.tex` is created with the names of all the files in `examp_list` and similarly `ovnodlst.tex` is created with all the names of all the files in `node_list` when you `make` the manuals. To add a new entry to the documentation you only need to add it to the appropriate list and do the required `make`'s.

`mnroffmn.tex` includes the files from `help_list` indirectly through several files that are created automatically. This is to allow different introductory comments for different groups of files. Comments in `mnroffmn.tex` describe these groups and are used to automatically create the indirect include files. You can add a new help file to any of these sections by adding it at the appropriate place in `help_list` and doing the required makes.

6.1.2 Help file format

Figure 13 on page 47 is an example `groff` help file. It begins with a section name at level 1. All help files should start this way. This will be translated to a `\subsection` LaTeX command. Lower level sections will be translated consistently.

The `groff` comment starting with '`\".LINE`' is a way of providing different text to LaTeX and `groff`. The remainder of the text on this line is *only* processed by LaTeX. It it is a comment in `groff` and the '`\".LINE`' is removed from the LaTeX file. The next line is not copied to the LaTeX files and is thus *only* processed by `groff`.

The `groff` comment `\".CAPTION` starts a LaTeX table. It provides a caption and label for the table as two strings. Preceding these strings is a list of integers indicating columns in the table. The text in those columns is enclosed in the `\Index` macro. If the column number is negative the `\DppNm` macro is used instead. The first macro includes the text as an index entry. The second macro also sets the text in `typewriter font`. The remainder of the table is specified in `groff` format and translated to LaTeX. Figure 14 on page 48 shows the LaTeX file generated from the `groff` file in Figure 13 on page 47. Note any words in single quotes in the `.roff` file are made arguments of the `\DppNm` macro in the `.tex` file.

```
\"  ssignalT.roff from ObjectProDSP 0.1
\"  Copyright (C) 1994, Mountain Math Software, All rights reserved.
\"  Licensed for free use and distribution under version 2 of the Gnu General
\"  Public License. Please see file COPYING for details and restrictions.
\"
\"  ObjectProDSP is a trademark of Mountain Math Software.
\"
.H 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.

\".LINE Signal generation nodes are shown in Table~\PageRef{Tbl:sig_gen}.
Signal generation nodes include the following:

\".CAPTION -1 "Signal generation nodes\index{signal}" "Tbl:sig_gen"
.TS
center;
ll.
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
.TE

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

Figure 13: Example groff help file

```
%  ssignalT.tex from ObjectProDSP 0.1
%  Copyright (C) 1994, Mountain Math Software, All rights reserved.
%  Licensed for free use and distribution under version 2 of the Gnu General
%  Public License. Please see file COPYING for details and restrictions.
%
%  ObjectProDSP is a trademark of Mountain Math Software.
%
\subsection{Signal nodes }
\index{ Signal nodes }

The nodes under \DppNm{objects} and \DppNm{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~\PageRef{Tbl:sig_gen}.
\begin{table}
\begin{center}
\begin{tabular}{|ll|} \hline
Name & Signal type \\ \hline
\DppNm{ConstantData} & constant level \\
\DppNm{Cos} & real cosine \\
\DppNm{CxCos} & complex cosine \\
\DppNm{CxImp} & complex impulse or square wave \\
\DppNm{Normal} & Gaussian distributed noise \\
\DppNm{Ramp} & ramp function \\
\DppNm{UniformNoise} & uniformly distributed noise \\
\hline
\end{tabular}
\end{center}
\caption{Signal generation nodes\index{signal}}
\label{Tbl:sig_gen}
\end{table}

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

Figure 14: LaTeX file generated from groff file

Directory under `$OPD_ROOT/src/dsp`	File	Type of file
`nodes/proc`	`node.tex`	processing nodes
`nodes/proc32`	`node.tex`	only used in 32 bit simulator
`nodes/signal`	`signal.tex`	signal generation nodes
`nodes/sigdsk`	`signal.tex`	read signal from disk
`nodes/display`	`display.tex`	output and display
selected files	`basenod.tex`	shared member functions
`lib/control`	`auxfuncs.tex`	other than node classes
`lib/network`	`auxfuncs.tex`	other than node classes

Table 17: Where nodes are documented

6.2 Overview and tutorial

Appendix A in *ObjectProDSP Overview and Tutorial* is generated automatically from the `.tex` files created in subdirectory `texs` from the directory in which the node `.usr` file is created. Lists of these files are created when you `make` the *ObjectProDSP Overview and Tutorial*. These files are sorted alphabetically based on the class name and occur in the Appendix in that order and grouped in sections as shown in Table 17 on page 49.

As long as you are adding nodes to an existing directory listed in Table 17 you do not need to do anything except remake this manual to include new nodes in the Appendix. The names under 'Files' in this table are for the *ObjectProDSP Library Reference*. A file '*name*`.tex`' in the table will be '*name*`s.tex`' in the `Makefile` for creating the Appendix.

No help files are used in this manual.

6.3 Library manual

Nodes in *ObjectProDSP Library Reference* are updated just as they are for *ObjectProDSP Overview and Tutorial* as described in the previous section. The .tex files in subdirectory `tex` (not `texs`) are used. Table 17 on page 49 shows where different kinds of nodes occur in the manual and what directories are searched for these nodes.

Help files that describe nodes or classes of nodes are included in `ovnode.tex`. If you add a help file for a node or group of nodes you should add it to `ovnode.tex` and edit the `Makefile`.

6.4 User's manual

The nodes are not directly documented in the *ObjectProDSP User's Reference*. However the entire menu tree is including the portions of it that are built from the node definitions. The LaTeX files for this are created by the `Makefile` in directory `$OPD_ROOT/doc/build`. You should run this `make` whenever you add nodes, update existing ones or make other changes to the menu data base. This will update LaTeX files for the *ObjectProDSP User's Reference*. These updated `.tex` files are built from `opd.menu` and the `.nod` files in subdirectory `nodes` under `$OPD_ROOT/src/menus`".

All help files other than those documenting nodes or classes of nodes are included in this manual from file `mnroffmn.tex`. You should update this file and the `Makefile` for this manual if you add help files like this.

6.5 Developers manual

The developers manual (this manual) does not ordinarily need to be updated as a result of changing nodes or menus. However if you need to define new base classes with new constructor parameters you will need to update the structures defined in
`$OPD_ROOT/src/util/mknode/ctorinit.C`.
You should update the documentation included in these data structures. That documentation is used in creating several tables in Section 1.4.1 on page 8.

It is outside the current scope of this manual to explain how to do this but if necessary you should be able to figure it out from the code in `ctorinit.h` and `ctorinit.C` and the `Makefile` for the developer's manual.

File under $OPD_ROOT	LaTeX file	Figure	Page
src/dsp/nodes/proc/block.usr	blockusr	1	3
src/include/ObjProDSP/arthtyp.h	artyp	5	18
src/menus/opd.menu	mmenu	8	34
src/menus/opd.menu	exampmenu	10	40
doc/roff/ssignalT.roff	sighlp	13	47
doc/nodetex/ssignalT.tex	sigtex	14	48

Table 18: Source files used as examples in this manual

Several files are used as examples in creating this manual. If you change any of these files you should check to insure this does not mess up their use in the documentation. These files are shown in Table 18 on page 51.

7 Building ObjectProDSP and `makemake`

The `Makefile`s used to generate utilities and ObjectProDSP executables are created by a custom `makemake` utility. This utility is less flexible then `imake` but very much faster. It knows about much of the structure of ObjectProDSP such as the relationship between `.usr` files, `.C` files and menu files. It does not use the C preprocessor to expand source so it is sometimes necessary to create dummy headers to prevent warning messages about missing files that are conditionally included.

It is outside the current scope of this manual to fully document this utility. Figure 15 on page 54 is a terse synopsis of its options. Some of these are obsolete or for future expansion.

In this section we will describe how you invoke `makemake`, where its gets its input from and what you need to do add or delete a directory or `.usr` file to the existing structure. To add a `.C` or `.h` file to a directory in which such files already exist you only need to add them and regenerate the `Makefile`s.

7.1 Project description

In each directory in which an executable is constructed there is a file `domakemake` which invokes `makemake` to create all needed `Makefiles`. The `Makefile` created in this main directory is made dependent on `domakemake` so subsequent `Makefile`'s will be updated if you change `domakemake`.

The master directory is the first argument for the `-m` option in `makemake`. Subsequent arguments indicate additional directories with `.C` files. Directories with `.C` files can also be designated with the `-l` option. Libraries are constructed in these directories.

The `Makefile`s created are independent. The master `Makefile` for the main directory will by default invoke all other `Makefile`s. If you do `make Target` then none of the other `make`s will be done. If you change a single file you only need to do a `make` for the directory in which that file occurs and a `make Target` in the main directory. Of course if you change a header file it is

```
Usage is:
[ -arith ] | [ -c cdir1 cdir2 ... cdirn ] | [
    -check_make_lst dir1 dir2 ... dirn ] | [ -ckusrc ] | [
    -coll ] | [ -dbXtra ] | [ -dir_space N ] | [
    -ext_lib_make dir lib ] | [ -f command_line_file ] | [
    -gf ] | [ -gfb ] | [ -gfs ] | [ -h hdir1 hdir2 ...
hdirn ] | [ -inc make include file ] | [ -incm main make
include file ] | [ -int_lib_make dir lib ] | [ -l libdir1
libdir2 ... libdirn ] | [ -Lib N ] | [ -libcolldirlst
dir_lib dir1 dir2 ... dirn ] | [ -libdirlst dir1 dir2 ...
dirn ] | [ -liblst dir1/file1 dir2/file2 ... dirn/filen ] |
    [ -libnm ] | [ -libs [dir1/]file1 [dir2/]file2 ...
[dirn/]filen ] | [ -libupdirlst dir1 dir2 ... dirn ] | [
    -list_files ] | [ -lkf ] | [ -m cdir_link cdir2 ...
cdirn ] | [ -makemake command [file1 file2 ... filen] ] | [
    -menu MenuDir MenuInput MenuIncDir file1 file2 ...
filen ] | [ -menuflag flag ] | [ -nc file1 file2 ... filen
    ] | [ -ncoll dir1 dir2 ... dirn ] | [ -NE ] | [
    -no_source_list dir1 dir2 ... dirn ] | [ -o executable
    ] | [ -obj obj_suffix ] | [ -og ] | [ -rl ] | [ -root
root directory name ] | [ -sc subdirectory name ] | [ -scu
usr subdirectory name ] | [ -sourcelist ] | [ -tex ] | [
    -tic30 ] | [ -user_copyright ] | [ -usr usrdir1 usrdir2
... usrdirn ]
The interpretation of these options is:
 -arith
Specifies that C files supporting multiple arithmetic
models are supported.
 -c cdir1 cdir2 ... cdirn
Specifies the list of directories with '.C' program files.
 -check_make_lst dir1 dir2 ... dirn
Specifies a list of directories to connect to and do a make
before anything else.
 -ckusrc
Specifies insure that all -usr files are also -c file.
```

Figure 15: makemake command line options (part 1 of 5)

```
 -coll
Specifies the collection of all object files not in library
directories in a gloabl library.
 -dbXtra
Specifies create file dbXtra.in to give list of source
directories.
 -dir_space N
Specifies the extra space for directories (default 32
should be adequate).
 -ext_lib_make dir lib
Specifies go to directory ‘dir’ to ‘make’ library ‘lib’.
 -f command_line_file
Specifies that all other options are to be read from the
file argument.
 -gf
Specifies that the global macro lists are to be output to
files.
 -gfb
Specifies each line of the global macro files will end with
a back slash (\).
 -gfs
Specifies no newlines in the global macro files.
 -h hdir1 hdir2 ... hdirn
Specifies the list of directories with ‘.h’ header files.
 -inc make include file
Specifies a file to be included in all Makefiles generated.
 -incm main make include file
Specifies a file to be included in the master Makefile.
 -int_lib_make dir lib
Specifies move library created in (use -l)  ‘dir’ to  ‘lib’.
 -l libdir1 libdir2 ... libdirn
Specifies the list of library directories with ‘.C’ program
files.
 -Lib N
Specifies N repetitions of the library references in the
```

Figure 16: `makemake` command line options (part 2 of 5)

```
final link (this may be needed with some linkers to pull in
all referenced files).
 -libcolldirlst dir_lib dir1 dir2 ... dirn
Specifies a list of directories from which files will be
linked in a single library directory ‘dir_lib’.
 -libdirlst dir1 dir2 ... dirn
Specifies the list of directorys where librarys with names
Libdiri are contained.
 -liblst dir1/file1 dir2/file2 ... dirn/filen
Specifies the list of library names.
 -libnm
Specifies the use of the parent directory name to generate
the library name.
 -libs [dir1/]file1 [dir2/]file2 ... [dirn/]filen
Specifies the list of librarys to be searched in building
the executable.
 -libupdirlst dir1 dir2 ... dirn
Specifies the list of directorys where librarys with name
LibUpDiri are contained where UpDiri is the parent
directory of diri.
 -list_files
Specifies write lists of various file types in ‘ALL_type’.
 -lkf
Specifies that the final link is to be done using file
‘makemake_link’.
 -m cdir_link cdir2 ... cdirn
Specifies the list of directories with ‘.C’ program files
beginning with directory for global (across directory)
linking.
 -makemake command [file1 file2 ... filen]
Specifies command to make ‘Makefile’ and list of
dependencies.
 -menu MenuDir MenuInput MenuIncDir file1 file2 ...
filen
Specifies the directory for creating menu files, the menu
```

Figure 17: makemake command line options (part 3 of 5)

```
input file, the menu includes directory and the list of
files created by the menu generator.
 -menuflag flag
Specifies pass the parameter to the 'domenus' command
(requires -menu).
 -nc file1 file2 ... filen
Specifies the '.C' files for which no CC command will be
created (usually the files are compiled with special
options using 'Makefile.tail').
 -ncoll dir1 dir2 ... dirn
Specifies the directories to skip in collecting object
files (the -coll option must be set).
 -NE
Specifies creation of librarys only with no executable.
 -no_source_list dir1 dir2 ... dirn
Specifies the list of library directories for which source
code is not available. If '-dbXtra' or '-sourcelist' is
set, any library directory not containing file
'source.list' and not in the list specified by this command
gives a warning.
 -o executable
Specifies name of executable to create (default is 'a.out').
 -obj obj_suffix
Specifies the suffix for object files (default is o).
 -og
Specifies no global macros in the 'Makefile' (if these
macros are too long, some versions of make abort with a
core dump).
 -rl
Specifies use ranlib.
 -root root directory name
Specifies set root name (the default is:
/usr/local/lib/opd_root).
 -sc subdirectory name
Specifies the directory name (as a subdirectory of the C
```

Figure 18: `makemake` command line options (part 4 of 5)

```
and library directories) to put the object and library
generated files (if this subdirectory does not exist it
will be created).
 -scu usr subdirectory name
Specifies set usr subdirectory name (by default it is set
to the C subdirectory name this overides that default).
 -sourcelist
Specifies creation in each library and executable directory
of file 'source.lst' containing all directories that
provide source code.
 -tex
Specifies output command line options in tex format.
 -tic30
Specifies create line file names for tic30.
 -user_copyright
Specifies add user copyright notice to each Makefile
generated.
 -usr usrdir1 usrdir2 ... usrdirn
Specifies the list of directories with '.usr' program files
(usually each '.usr' directory must also be a header (-h)
directory and either a library (-l) or C (-c) directory).
```

Figure 19: makemake command line options (part 5 of 5)

generally necessary to run `make` on all the `Makefile`s.

Most often the `-sc` option is used so the compiles are done in a subdirectory off the directory containing the source code. This allows the same source to be used to generate multiple libraries and executables with different compile time flags. The master `domakemake` is in this subdirectory of the directory specified by the first argument to `-m`. A single `makemake` can generate `Makefile`s for many different directories. If `-sc` is used a subdirectory will be created for every directory under the `-m` and `-l` options.

Typically a number of files are read by `domakemake`. These include script files in `$OPD_ROOT/scripts` and various `head`, `inc` and `tail` files in various directories. File `SYSTEM` in the `scripts` directory determines the operating system the make is for. This influences what `head` and `tail` files will be incorporated in the `Makefile` by defining a suffix for this search. Comments in the `Makefile` show what files are included and at what point.

The algorithm for selecting these files is as follows.

1. `Makefile_head*` — include all with searched suffixes in searched directories at the start of the `Makefile`.
2. `Makefile_inc*` — include all with searched suffixes in searched directories at the end of the `Makefile`.
3. `Makefile_tail*` — include exactly one file. If multiple files with searched suffixes are present choose the first one in the search sequence. This is the last part of the `Makefile`.

The search sequence looks in generic places (such as the `scripts` directory) first and in specific places last. This sequence is *inverted* when only a single file is included for `Makefile_tail`. In all other cases all matching files are included.

The serach sequence for directories is as follows.

1. `$OPD_ROOT/scripts`.
2. Directory that is the first argument to `-m` (main directory).

Directory $OPD_ROOT/doc/	Object $OPD_ROOT/doc/	Purpose
build	*all* .dvi *files*	
devman	devman/devman.dvi	*ObjectProDSP Developer's Reference*
overview	overview/overview.dvi	*ObjectProDSP Overview and Tutorial*
	overview/nodeman.dvi	*ObjectProDSP Library Reference*
roff	.hlp files	groff to .hlp and .tex format
roff/mmake	doc/roff/Makefile	create Makefile from lists of help files
userman	userman/userman.dvi	*ObjectProDSP User's Reference*

Table 19: Directories for creating documentation

Directory $OPD_ROOT/src/	Object $OPD_ROOT/bin/	Purpose
gui/build/iv	opd_gui_exe	user interface process
dsp/build/fltgui	opd_dsp_exe	floating point DSP process
dsp/build/int16gui	opd_dsp_exe	16 bit integer DSP process
start_up	opd	executable to start both processes
genlib/opdgen/opdgeno	$OPD_ROOT/lib/ Libopdgen	generic library
$OPD_ROOT/build	*everything*	

Table 20: Directories for creating interactive libraries and executables

3. -sc specified subdirectory of main directory.

4. The directory for which this Makefile is being generated (from a -m or -l argument).

5. The -sc subdirectory of the directory for which this Makefile is being generated. (The Makefile is placed in this subdirectory.)

The serach sequences for suffixes is as follows.

1. NULL suffix.

2. .*name* where *name* is the first line in file $OPD_ROOT/scripts/SYSTEM.

Directory $OPD_ROOT/src/dsp/lib/Target/	Object $OPD_ROOT/lib/	Purpose
tarflt	LibTarLnxFlt	stand alone float libraries
	TarnobenchmFlt	
tarint	LibTarLnxInt16	stand alone 16 bit libraries
	TarnobenchmInt16	

Table 21: Directories for creating target libraries

Directory $OPD_ROOT/src/util/	Object $OPD_ROOT/bin/	Purpose
hyphen	cp_part	copy selected part of file
	hyphen	add hyphenation for LaTeX
	toc_depth	LaTeX table of contents filter
indextex/ix	indextex	create index for LaTeX
makemake/mk	makemake	Makefile generator
maketex/mktex	maketex	create lists of files in a directory
mkmenu/menuo	domenus	create menu data base code
mknode	mknode/mkndoeo	translate ObjectPro++ source
nametrans	nametrns	adds conditional include to header used in $OPD_ROOT/bin/headcnv
rofftotex	rofftotex	limited roff mm to LaTeX translator

Table 22: Directories for creating utilities

The search first looks for all suffixes in a directory and then moves on to the next directory.

7.2 Building ObjectProDSP

README in $OPD_ROOT/build contains instructions for building ObjectProDSP from the source distribution. Tables 19 on page 60 through 22 on page 61 gives the directories for building specific components of ObjectProDSP. You should only need to go to one of these directories and enter make to update one of these components.

7.3 Validation

7.3.1 ObjectProDSP directory structure

Table 23 on page 63 gives the top level ObjectProDSP directory structure.

7.3.2 Validation

The validation shell script
`$OPD_ROOT/scripts/master_validate.sh`
controls the creation of base line test data and the execution of tests against that data.

Directory under `$OPD_ROOT`	Contents
`bin`	executables
`build`	master build directory for everything
`doc`	documentation
`doc/build`	master build directory for documentation
`doc/mac`	documentation macros
`doc/scripts`	documentation script files
`doc/trademarks`	restricted use macros
`examp`	ObjectProDSP examples
`help`	interactive help files
`lib`	libraries
`src`	source code
`src/dsp`	DSP source code
`src/dsp/build`	master directory for DSP process
`src/dsp/lib`	libraries for interactive and stand alone DSP code
`src/dsp/lib/target`	root for stand alone code
`src/dsp/lib/target/build`	master directory for stand alone libraries
`src/dsp_gui`	classes common to both processes
`src/gui`	user interface process source code
`src/gui/build`	master directory for user interface process
`src/include`	include files for writing nodes
`src/menus`	menus and menu builds for both processes
`src/genlib`	generic library source code
`src/start_up`	start up code that initiates both processes
`src/util`	utilities source code
`validate`	validation log, action files, and DSP++ programs

Table 23: Top level ObjectProDSP directory structure

Feature	Affect
changing menu data base structure	affects everything lower in tree
new menu data base items or menus	no affect
delete menu data base items	only affects item deleted
changing menu bar key codes	affects items changed
new menu bar items	no affect
new menu bar top level menus	no affect
deleting menu bar items and menus	affects only items deleted
adding new classes or member functions	no affect
changing class names	affects classes changed

Table 24: User interface changes that break and do not break validation

8 Regression tests

Regression tests can assure that under controlled inputs a program produces exactly the output it had previously. The original output that the test is run against must be manually checked. One difficulty is insuring the correctness of this base line test output. Another difficulty is making the regression tests 'orthogonal' so a small change in the program will have a known small change in the test output. Regression tests allow one to make code changes and have a reasonable assurance that one has not fixed one bug only to introduce several new ones. However this is only possible if the tests can be made sufficiently independent of each other. This can be particularly difficult in user interface code.

Table 24 on page 65 shows what elements of the user interface can and cannot be changed without invalidating the existing regression test suite. Of course these lists are not exhaustive. However they describe important areas where you cannot make changes unless you are willing to recreate or heavily edit the regression tests. They describe other important areas where you can change the user interface without breaking the existing validation suite.

8.1 Running and creating regression tests

Script `$OPD_ROOT/scripts/master_validate.sh` creates and runs the standard suite of regression tests. This is most conveniently used with the `Makefile` in `$OPD_ROOT/build`. For instructions execute the shell script or do a `make` with no arguments. The rest of this section describes how to add tests to the standard suite created and run by this script.

The standard tests are determined by the files in `$OPD_ROOT/validate` of the forms `make_*_validate.rec`,
`make_*_validate_float.rec` and
`make_*_validate_int16.rec`.
Files of the first type are used to generate tests for both simulators. The other types generate tests for the floating point and 16 bit simulators individually. To add new tests create an action file with the appropriate name and add it to this directory. This action file must construct a network and do a `TargetValidate` on the network. It should then exit (preferably without saving the state). Make sure your network does not conflict with any of the networks already used. (For each network there is directory
`$OPD_ROOT/validate/test_nodes/val_`*network_name*`_float`
and/or a directory
`$OPD_ROOT/validate/test_nodes/val_`*network_name*`_int16`
created. Make sure your new network is not the same as *network_name* in any of these directory names.) (To assign a name to a network create a nondefault instance of a network. You will be prompted for the name to use. You must do this before you create any instances of objects or a network will be created with a default name and the objects associated with that network. You can move objects between networks, if they are not linked, but you cannot change the name of a network once it has been created.)

8.2 Regression tests created with `TargetValidate`

The `Network` member function `TargetValidate` generates ObjectProDSP.dpp files and shell scripts to create and execute regression tests. In these scripts the output of all `Listing` and `Plot` nodes is automatically tested. Two networks are generated from the original. In the first each occurrence of `Listing` or Plot is replaced with an OutputNode. In the second the same nodes are replaced with a CompareDisk node. CompareDisk reads the data generated by the OutputNode and compares it with the current input. The scripts created by TargetValidate are used for both target code and interactive code validation.)

8.3 Writing and reading a file in different tests

There are two ways to test output files (other than those created by OutputNode). You can write a file in one test and then read it in a subsequent test. The second test should send the data to a Plot or Listing node. so the data will be verified. You can also do byte by byte comparison against base line validation files by writing a file that ends with suffix .cmp.

If you write a file in one test and read it in another you must be sure the two tests are done in the correct order. Tests are done in alphabetical order (as defined by the sort command). They are alphabetized by network name and by the make*validate.rec name. To have one test generate output and a second read it you must choose these names so the tests that creates the data runs first.

Different tests are run in different directories. You must specify an absolute path name if different tests are to access the same file. The standard place to put the data is directory $OPD_ROOT/validate/test_data. (You can use environmental variables when entering file names.) If this directory does not exist it will be created when validation starts. You may need to create the directory manually for debugging your tests.

8.4 Byte by byte comparison files

Some files cannot be easily tested by writing and then reading a file. Such files can be tested by doing a byte by byte comparison against a base line version of the file. To test files in this way write them in the current directory and give them a name ending with .cmp. When you do a make VAL_DATA to integrate your tests (as described below) a base line version of these files will be created with suffix `.cmpb`. Subsequent `make VALIDATE` executions will test the newly generated `.cmp` files against the `.cmpb` files by doing a comparison on every byte in both files.

All tests must contain at least one `Plot` or `Listing` node and must include the `MakeTarget` operation. If no output is generated the test will abort. `MakeTarget` is necessary to create the test scripts.

8.5 Documenting tests

Each `make_*_validate.rec` file must start with documentation of the tests. If a file does not contain a line similar to the following the validation script will abort:

```
########################################### END OF VALIDATION LIST
```

This indicates the end of the information that annotates the tests. Program `$OPD_ROOT/report_test` reads the information above this line in each test file and uses it to print a summary of what nodes and other features were tested and how completely these features were tested. If `report_test` does not find the line *exactly* as it expects, it will abort with an error message. Copy the line from an existing file to make sure it is exactly correct.

Above this line you should include a single line for each node that occurs in your test of the form:

```
# node_name tests performed or other comments
```

Do not include `Listing` or `Plot` nodes in the above list as these are replaced with other nodes in generating the tests. Make sure that every thread ends with either a `Plot` or `Listing` node so at least the final output of the thread will be verified.

Following all nodes you can list other features tested one per line. These lines should start with two sharp signs: `##`. See any of the standard validation files, `make_*_validate*.rec`, for examples. In these features list include the name of any `.cmp` files that the test generates. This will make it easy to go from the `.cmp` file name to the validation script that generated the file.

8.6 Creating base line test data

To create base line test data run `$OPD_ROOT/scripts/master_validate.sh` with argument `VAL_DATA`. You can temporarily move all of the existing `make*validate.rec` files in `$OPD_ROOT/validate` to a different directory to just do a `make VAL_DATA` with your new validation files. This will not change the base line test data for the existing tests. (If you just add your files and then do a `make VAL_DATA` all base line test data will be recreated.) You can then transfer the original `make*validate.rec` files back to `$OPD_ROOT/validate` and the next `make VALIDATE` will run the full suite including those tests you added. Make sure you manually check the correctness of the base line test data you have added.

When you next do `make VALIDATE` the validation log should report no errors but additional test cases not in the previous log. Copy this log to the appropriate base line validation log to complete the update. File `base_log` is the log for floating point and 16 bit integer tests. `base_float_log` is for floating point tests only. (There is no capability to run only 16 bit integer tests.) If your update includes floating point tests you should update both base line log files.

8.7 Make many mistakes in recording your test

These are tests of both the DSP process and the GUI since the networks are built interactively. It is good to make mistakes in construction the actions files for these tests. It makes the GUI test more effective.

APPENDIXES

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.

References

[1] P. Budnik, *ObjectProDSP Overview and Tutorial* Mountain Math Software, September 1994

[2] P. Budnik, DppUser, Mountain Math Software, September 1994

[3] P. Budnik, *ObjectProDSP Library Reference*, Mountain Math Software, September 1994

[4] P. Budnik, *ObjectProDSP Developer's Reference*, Mountain Math Software, September 1994

[5] Brian W. Kernighan and Dennis R. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[6] The ANSI X3J11 committe and Herbert Schildt *The Annotated ANSI C Standard, ANSI/ISO 9899-1990*, Osborne McGraw-Hill, 1990.

[7] Edward A. Lee and David G. Messerschmitt, Synchronous Data Flow, *Proceedings of the IEEE*, Vol 75, No. 9, September 1987.

[8] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

B–2

Index