VISUALDSP++™ 3.5 Product Release Bulletin for 16-Bit Processors

Revision 1.0, October 2003

Part Number
82-000035-01

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106

ANALOG DEVICES

CONTENTS

PREFACE

Thank you for purchasing VisualDSP++™, Analog Devices development software for digital signal processor (DSP) applications.

Purpose of This Document

This document briefly describes the new features and enhancements provided by VisualDSP++ 3.5 for 16-bit digital signal processors. It also describes the differences (obsolete features and functions) between VisualDSP++ 3.5 and previous VisualDSP++ releases.

For details, refer to the VisualDSP++ 3.5 manuals listed in "Related Documents" and online Help.

Intended Audience

This publication is primarily intended for programmers who are upgrading from the previous releases of VisualDSP++ development software and who want an overview of the changes to VisualDSP++ 3.5.

Manual Contents

This manual consists of:

- Chapter 1, "Introduction"
  Describes VisualDSP++ 3.5 and its benefits, provides the minimal system requirements for running the product, and lists the supported processors.

- Chapter 2, "VisualDSP++ 3.5 Major Changes"
  Describes major changes in VisualDSP++ 3.5 compared to VisualDSP++ 3.0 and VisualDSP++ 3.1 (for Blackfin processors) releases.

- Chapter 3, "New Features and Enhancements"
  Describes what is new in the VisualDSP++ 3.5 IDDE, assembler, compiler, linker, loader, and documentation. Also describes the new features in the Expert Linker (EL), VisualDSP++ Component Software Engineering (VCSE), and the VisualDSP++ Kernel (VDK).

- Chapter 4, "Obsolete or Removed Features"
  Describes the removed/obsolete features in VisualDSP++ 3.5 (compared to the previous VisualDSP++ software release as they pertain to code generation tool chain: commands, switches, operators, directives, pragmas, keywords, macros, and library functions.

Technical or Customer Support

You can reach DSP Tools Support in the following ways.

- Visit the DSP Development Tools website at

  `www.analog.com/technology/dsp/developmentTools/index.html`

- Email questions to

  `dsptools.support@analog.com`

- Phone questions to 1-800-ANALOGD

- Contact your ADI local sales office or authorized distributor

- Send questions by mail to

  ```
  Analog Devices, Inc.
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA
  ```

Supported Processors

VisualDSP++ 3.5 release for 16-bit processors supports Blackfin, ADSP-219x, and ADSP-218x processors. For more information, refer to "Platform and Processor Support" on page 1-3.

Product Information

You can obtain product information from the Analog Devices website, from the product CD-ROM, or from the printed publications (manuals).

Analog Devices is online at `www.analog.com`. Our website provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

MyAnalog.com

MyAnalog.com is a free feature of the Analog Devices website that allows customization of a webpage to display only the latest information on products you are interested in. You can also choose to receive weekly email notification containing updates to the webpages that meet your interests. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Registration:

Visit `www.myanalog.com` to sign up. Click Register to use MyAnalog.com. Registration takes about five minutes and serves as a means for you to select the information you want to receive.

If you are already a registered user, just log on. Your user name is your email address.

DSP Product Information

For information on digital signal processors, visit our website at `www.analog.com/dsp`, which provides access to technical publications, datasheets, application notes, product overviews, and product announcements.

You may also obtain additional information about Analog Devices and its products in any of the following ways.

- Email questions or requests for information to `dsp.support@analog.com`
- Fax questions or requests for information to
  1-781-461-3010 (North America)
  +49 (0) 89 76903-157 (Europe))
- Access the Digital Signal Processing Division's FTP website at
  `ftp ftp.analog.com` or ftp 137.71.23.21
  `ftp://ftp.analog.com`

Related Documents

For information on product related development software, see the following publications:

VisualDSP++ 3.5 Getting Started Guide for 16-Bit Processors

VisualDSP++ 3.5 User's Guide for 16-Bit Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for Blackfin Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors

VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs

VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x DSPs

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for ADSP-219x DSPs

VisualDSP++ 3.5 Loader Manual for 16-Bit Processors

VisualDSP++ 3.5 Kernel (VDK) User's Guide for 16-Bit Processors

VisualDSP++ 3.5 Component Software Engineering User's Guide for 16-Bit Processors

Quick Installation Reference Card

For hardware information, refer to your DSP's *Hardware Reference* manual and datasheet. All documentation is available online. Most documentation is available in printed form.

Online Documentation

Online documentation comprises Microsoft HTML Help (`.CHM`), Adobe Portable Documentation Format (`.PDF`), and HTML (`.HTM` and `.HTML`) files. A description of each file type is as follows.

File	Description
`.CHM`	VisualDSP++ online Help system files and VisualDSP++ manuals are provided in Microsoft HTML Help format. Installing VisualDSP++ automatically copies these files to the `VisualDSP\Help` folder. Online Help is ideal for searching the entire tools manual set. Invoke Help from the VisualDSP++ Help menu or via the Windows `Start` button.
`.PDF`	Manuals and data sheets in Portable Documentation Format are located in the installation CD's `Docs` folder. Viewing and printing a `.PDF` file requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher). Running `setup.exe` on the installation CD provides easy access to these documents. You can also copy `.PDF` files from the installation CD onto another disk.
`.HTM` or `.HTML`	Dinkum Abridged C++ library and FlexLM network license manager software documentation is located on the installation CD in the `Docs\Reference` folder. Viewing or printing these files requires a browser, such as Internet Explorer 4.0 (or higher). You can copy these files from the installation CD onto another disk.

Access the online documentation from the VisualDSP++ environment, Windows Explorer, or Analog Devices website.

From VisualDSP++

VisualDSP++ provides access to online Help. It does not provide access to `.PDF` files or the supplemental reference documentation (Dinkum Abridged C++ library and FlexLM network licence). Access Help by:

- Choosing Contents, Search, or Index from the VisualDSP++ Help menu
- Invoking context-sensitive Help on a user interface item (toolbar button, menu command, or window)

From Windows

In addition to any shortcuts you may have constructed, there are many ways to open VisualDSP++ online Help or the supplementary documentation from Windows.

Help system files (`.CHM` files) are located in the `Help` folder, and `.PDF` files are located in the `Docs` folder of your VisualDSP++ installation. The `Docs` folder also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation.

Using Windows Explorer

- Double-click any file that is part of the VisualDSP++ documentation set.
- Double-click the `vdsp-help.chm` file, which is the master Help system, to access all the other `.CHM` files.

Using the Windows Start Button

Access the VisualDSP++ online Help by clicking the Start button and choosing Programs, Analog Devices, VisualDSP++ for 16-bit processors, and VisualDSP++ Documentation.

From the Web

To download the tools manuals, point your browser at `www.analog.com/technology/dsp/developmentTools/gen_purpose.html`.

Select a DSP family and book title. Download archive (`.ZIP`) files, one for each manual. Use any archive management software, such as WinZip, to decompress downloaded files.

Printed Manuals

For general questions regarding literature ordering, call the Literature Center at 1-800-ANALOGD (1-800-262-5643) and follow the prompts.

VisualDSP++ Documentation Set

Printed copies of VisualDSP++ manuals may be purchased through Analog Devices Customer Service at 1-781-329-4700; ask for a Customer Service representative. The manuals can be purchased only as a kit. For additional information, call 1-603-883-2430.

If you do not have an account with Analog Devices, you will be referred to Analog Devices distributors. To get information on our distributors, log onto `www.analog.com/salesdir/continent.asp`.

Hardware Manuals

Printed copies of hardware reference and instruction set reference manuals can be ordered through the Literature Center or downloaded from the Analog Devices website. The phone number is 1-800-ANALOGD (1-800-262-5643). The manuals can be ordered by a title or by product number located on the back cover of each manual.

Data Sheets

All data sheets can be downloaded from the Analog Devices website. As a general rule, printed copies of data sheets with a letter suffix (L, M, N, S) can be obtained from the Literature Center at 1-800-ANALOGD (1-800-262-5643) or downloaded from the website. Data sheets without the suffix can be downloaded from the website only—no hard copies are available. You can ask for the data sheet by part name or by product number.

If you want to have a data sheet faxed to you, the phone number for that service is 1-800-446-6212. Follow the prompts and a list of data sheet code numbers will be faxed to you. Call the Literature Center first to find out if requested data sheets are available.

Contacting DSP Publications

Please send your comments and recommendations on how to improve our manuals and online Help. You can contact us at `dsp.techpubs@analog.com`.

Notation Conventions

The following table identifies and describes text conventions used in this manual. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
Close command (**File** menu) or **OK**	Text in **bold** style indicates the location of an item within the VisualDSP++ environment's menu system and user interface items.
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets separated by vertical bars; read the example as `this` or `that`.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this,…]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipsis; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, code examples, and feature names are in text with `letter gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
ⓘ	A note providing information of special interest or identifying a related topic. In the online version of this book, the word **Note** appears instead of this symbol.
🛇	A caution providing information about critical design or programming issues that influence operation of a product. In the online version of this book, the word **Caution** appears instead of this symbol.

1 INTRODUCTION

This chapter describes the product, VisualDSP++, and the requirements for running its latest revision, VisualDSP++ 3.5. It also lists the supported processors and some of the benefits provided by this release.

The information is organized as follows.

- "Product Release Description" on page 1-2
- "VisualDSP++ 3.5 System Requirements" on page 1-2
- "Platform and Processor Support" on page 1-3
- "VisualDSP++ 3.5 Product Highlights" on page 1-4

Product Release Description

VisualDSP++ is Analog Devices project management and development environment for Digital Signal Processor (DSP) applications. VisualDSP++ 3.5 successfully integrates a graphical user interface and code generation and debugging tools, enabling programmers to move easily between editing, building, debugging, and deployment of final products.

The code generation tool chain is comprised of the processor-specific software necessary for completing a DSP-based project: simulator, assembler, C/C++ compiler (no C++ support for ADSP-218x processors) and libraries, linker, loader, splitter, and utilities. Analog Devices also provides VisualDSP++ Component Software Engineering (VCSE) with VIDL compiler and VisualDSP++ Kernel (VDK),

The product CD-ROM also includes an evaluation suite of the EZ-KIT Lite® software, which provides an easy method for initial evaluation of a target processor system and allows application prototyping.

The successor to VisualDSP++ 3.0 (and VisualDSP++ 3.1 for Blackfin processors), this software release incorporates a number of new features and enhancements, as described in "New Features and Enhancements".

VisualDSP++ 3.5 System Requirements

To install and run VisualDSP++ 3.5, your PC must provide the following software, configuration, and system resources.

- Windows 98 SR2 (or greater)/NT 4.0 SP3/2000/ME/XP
- At least 100 MB of available hard drive space
- At least 32 MB of RAM
- CD-ROM drive
- Internet Explorer 4.01 or later

Platform and Processor Support

Blackfin Processors

The name “*Blackfin*” refers to a family of Analog Devices 16-bit, fixed-point embedded processors. VisualDSP++ currently supports all Analog Devices Blackfin processors:

- ADSP-BF532 (formerly ADSP-21532)
- ADSP-BF535 (formerly ADSP-21535)
- ADSP-BF531
- ADSP-BF533
- ADSP-BF561
- AD6532

The ADSP-BF531 and ADSP-BF533 processors are memory variants of the ADSP-BF532 processor. The ADSP-BF561 processor is a dual-core processor using the same core as the ADSP-BF532 processor.

ADSP-218x and ADSP-219x DSPs

The name “*ADSP-21xx*” refers to two families of Analog Devices 16-bit, fixed-point processors. VisualDSP++ currently supports the following processors:

- ADSP-218x DSPs: ADSP-2181, ADSP-2183, ADSP-2184/84L/84N, ADSP-2185/85L/85M/85N, ADSP-2186/86L/86M/86N, ADSP-2187L/87N, ADSP-2188L/88N, and ADSP-2189M/89N

- ADSP-219x DSPs: ADSP-2191, ADSP-2192-12, ADSP-2195, ADSP-2196, ADSP-21990, ADSP-21991, and ADSP-21992

VisualDSP++ 3.5 Product Highlights

Major product highlights and benefits include:

- Platform and Processor Support.
  VisualDSP++ 3.5 for 16-bit processors supports all Analog Devices Blackfin processors, ADSP-219x processors and ADSP-218x DSPs on Windows® 98, Windows ME, Windows NT 4.0, Windows 2000, and Windows XP.

- Robust and Flexible Project Management
  The VisualDSP++ 3.5 Integrated Development and Development Environment (IDDE) provides robust and flexible project management for the development of applications and includes access to all the activities necessary to create and debug projects. It enables users to open and switch between multiple projects in the same session.

- Time-Saving Debugger
  The VisualDSP++ debugger has a user-friendly, common interface to simulators and emulators available from Analog Devices and participating third-parties. On top of that, the debugger has many features that greatly reduce debugging time. Users can view C/C++ source code interspersed with the resulting assembly code, profile execution of a range of instructions in a program, set watchpoints on hardware, view program and data memory, and trace instruction execution and memory accesses. These time-saving features enable users to quickly correct coding errors, identify bottlenecks, and examine signal processor performance all within the debugger. Also, when used with the simulator, the debugger can generate inputs, outputs, and interrupts to simulate real world application conditions and give users better insight in tuning the performance of their code.

- VisualDSP++ Kernel
  The VisualDSP++ Kernel (VDK) is a real-time operating system that provides scheduling and resource allocation services applicable to embedded programming. VDK applications are configured within the IDDE and initial source files are generated from templates, where appropriate, to provide a working project framework within which users can insert their own code.
- Automation API and Aware Scripting Engine
  The Automation Aware Scripting Engine using the ActiveX script host framework allows the use of multiple popular scripting languages such as VBScript and JavaScript to access the Automation API. Now you are able to interface with the IDDE using either a single command or a script file.
- Multiple Processor (MP) Support
  The VisualDSP++ multiple processor (MP) support provides a single seamless interface for debugging multiple processors on the same physical hardware. Users can easily issue parallel step, run, and halt commands to all of the applicable processors. Developers can easily pick and choose individual processor registers, or memory sets of interest, by pinning those that should be updated between runs, halts and steps. This feature also eliminates screen clutter in multiple processor debugging.
- Background Telemetry Channel Support
  The Background Telemetry Channel (BTC) feature is a mechanism for exchanging data between a host and target application, with minimal intrusion on the target system's "real-time" characteristics and minimal addition on a development and debugging time. BTC enables real-time data collection and status messaging, eliminating the overhead involved with halting the target application, getting the desired information, and then restarting the target application. VisualDSP++ 3.5 extends BTC support on Blackfin and ADSP-219x processors, so that users will be able to directly benefit

from BTC in the IDDE plot window. In this case, the plot window will read the target's memory contents on a user-defined time interval and upon receipt of the data, convert them to the desired data type and update the plot display for users to view and analyze immediately.

- Statistical Profiling
  Statistical profiling allows a more generalized form of profiling of which JTAG emulator debug targets can take advantage. The debugger has the ability to unobtrusively and statically sample the target processors and then present a graphical display of the resultant samples for review. This enables to easily and effortlessly identify where your application is spending more of its time.

- Graphical Plotting
  VisualDSP++ includes numerous graphical plotting options, including Line, Constellation, Eye Diagrams, and 3D waterfall plots that help users to better visualize, analyze, and understand their data. The plotting engine is also capable of doing some simple data processing such as Fast Fourier Transform, 2-D Fast Fourier Transform, and Convert to Decibels on the data before it is displayed.

- VisualDSP++ Component Software Engineering (VCSE)
  VCSE supports an Interface Definition Language (IDL) and compiler that allow developers to create and reuse components without having to become familiar with the detail of the model and the mechanisms it involves. Components can easily be integrated into an application and are reusable. VCSE dramatically simplifies the process of incorporating and utilizing components from a variety of developers. The VIDL compiler can automatically generate a test shell for each component that can be used to monitor a component, to validate its actions and to measure the resources used by it.

- Cache Visualization (Blackfin processors only)
  Cache statistics, such as Total Cache Accesses, Cache Hits, and Cache Misses, are associated with both the PC/Source Line and the Cache Line/Set and are collected by the simulator. Once these statistics are collected, you have the option to easily view and analyze them in the following forms: Histogram by PC/Source Line, Cache Line Display where hit/miss data is associated by Cache Line/Set(way), and Summary Display of totals for hits/misses by cache.

- Pipeline Viewer
  The Pipeline Viewer allows to easily view the instruction flow through the sequencer's pipeline. Stalls, aborts, and other pipeline events are graphically represented in an easy-to-read format for the developer. Visualization of the pipeline and of the events, which occur with it, allow you to better understand where and why latencies and stalls are being introduced into an executable. Armed with this knowledge, you can effectively and efficiently optimize an executable's instruction sequence to minimize the number of undesirable pipeline events.

- Compiled Simulation (Blackfin processors only)
  Traditionally, a standard simulator fetches, decodes and then simulates each instruction that an application executes. This approach is inefficient and costly as each time an instruction is executed, it has to be first decoded. With Compiled Simulation, the simulation compiler automatically examines the whole application once and generates C code for each instruction in the application, generating a native Win32 executable tailored to run your application. As a result, the generated application can be used to simulate that one application very efficiently (at speed of 100 to 1000 times faster than the ordinary simulator).

- C/C++ Compiler
  The best-in-class C/C++ compiler is a time-saver for developers who use it for application code generation. It generates efficient application code that is optimized for both code density and execution time. C/C++ code modules can be easily interfaced with assembly code modules, allowing users to program in C/C++ and still use assembly for time-critical loops. Beyond that, developers can realize an additional significant decrease in their time to market with the ability to efficiently work with complex data types and take advantage of specialized operations without having to understand the underlying architecture. C++ is supported on Blackfin and ADSP-219x processors; ADSP-218x DSPs do not have C++ support.

  Among other notable features, the VisualDSP++ 3.5 compiler offers 64-bit integer data types support; C++ standard exception handling as defined by the ISO/IEC 14882:1998 standard on Blackfin processors; and improved external interface on ADSP-219x processors.

- Profile-Guided Optimization (Blackfin processors only)
  Profile-Guided Optimization (PGO) is an iterative compilation approach which uses information from previous compilations to improve the optimizer's decisions on the code being compiled. Traditionally, a compiler only compiles each function once and attempts to generate code that will perform optimally in most cases by making reasonable default assumptions in the behavior of that code. With PGO, the compiler makes educated assumptions based on data collected during previous executions of the generated code and subsequently makes decisions about the relative importance of parts of the application rather than simply using the default behavior.

This technique can enable large gains to be realized in the run-time performance and code density of the program automatically, without additional effort by the users.

- Expert Linker
  The Expert Linker creates a graphical utility that makes it easier for users to produce Linker Description File (LDF) without having to learn the LDF syntax. The graphical representation of commands in an LDF file also allows the engineer to easily make changes or to generate a new LDF file. In VisualDSP++ 3.5, the Expert Linker also allows users to easily profile object sections of their program, identify "hotspots" graphically, and optimize their placement of code in one single step with minimal additional effort.

- Integrated Source Code Control
  The Source Code Control (SCC) integration in the IDDE enables users to easily connect to SCC applications that are installed on their machines through the Microsoft Common Source Code Control (MCSCC) interface that is widely supported by leading SCC vendors. Using the plug-in, you can also access commonly used features (such as getting the latest version, checking out, and removing a selected file from source code control) of these SCC applications, launch the SCC applications, and view a file's source control status in a project window quickly and conveniently without leaving the IDDE.

2 VISUALDSP++ 3.5 MAJOR CHANGES

This chapter summarizes major changes in VisualDSP++ 3.5 compared to VisualDSP++ 3.0 and VisualDSP++ 3.1 (for Blackfin processors) releases.

The chapter details:

Please note that all obsolete/removed features are listed in Chapter 4.

Changes to Installer

Discrete Installer

VisualDSP++ 3.5 and all future releases of VisualDSP++ will install discretely from other versions of VisualDSP++. Historically, releases installed into a common, shared location. The new installer allows you to switch between different releases of VisualDSP++ more efficiently. Each installation of VisualDSP++ will have its own default installation directory and Start menu icons.

ⓘ Installing VisualDSP++ into the same location as another version of VisualDSP++ is not supported and highly discouraged.

Discrete installation relies on certain facilities within Microsoft Windows that are not supported on older versions of the operating system. VisualDSP++ will not be fully discrete under Windows 95, Windows 98, and Window NT 4.0. Users of those operating systems are highly encouraged to migrate to Windows XP. On the named, older versions of Windows, the debug targets to which VisualDSP++ IDDE can connect will not be handled discretely. VisualDSP++ will be able to connect to another installation's targets; however, the build tools will be managed discretely.

When using multiple versions of VisualDSP++ on the same host PC, be aware of:

1. Multiple product installations have no visibility into each other (by design) to ensure that each installation behaves exactly as it does on a dedicated host PC. One implication of this is that user settings, such as JTAG platforms, debug session, and preferences need to be set under each installation.

2. License management is likewise discrete. In most cases, licenses already installed and validated for a past release of VisualDSP++ can be imported into VisualDSP++ 3.5 by simply copying the `…\System\license.dat` file from the historical release of VisualDSP++ to the `…\System` directory of the new installation. Alternatively, the license can be reregistered from within VisualDSP++ 3.5.

3. If building from the command line, make sure that the `PATH` environment variable is correctly set. The VisualDSP++ installer will place a helper file, `VDSP3_5.BAT`, in the VisualDSP++ directory that will correctly set the `PATH` variable to point to that installation of VisualDSP++ 3.5.

4. Very old releases of the tools had used an environment variable, `ADI_DSP`. This variable should not be set at all when using VisualDSP++ 3.5. Third-party debug targets under older releases can be selected from the New Session dialog box by selecting the Show All Targets checkbox.

5. If a VisualDSP++ connection is via Automation (for example, from a VBScript), the last registered version of `IDDE.EXE` is invoked. To change the version of VisualDSP++, run the desired version of `IDDE.EXE` with the `/RegServer` switch.

6. VisualDSP++ project file (`.DPJ`) association is a Windows-wide setting and, thus, is not handled discretely; it is generally made to be associated with whichever version of VisualDSP++ was installed most recently. Association can be changed as desired within Windows. The exact procedure depends on a particular Windows version. For example, under Windows XP, select a .DPJ file, chose Properties from the context (right-click) menu, and click the Change… button next to the Open with field.

Emulation Tools Included in Installer

The emulation tools now are included in the base VisualDSP++ installation. There is no longer a second ICE software installation procedure. To install emulation support, select the appropriate components in the installation wizard. The installer handles the installation and/or update of the emulator hardware device driver.

Changes to Blackfin Compiler

This section summarizes changes to the Blackfin compiler environment since VisualDSP++ 3.1. As with any major upgrade, some of the improvements have necessitated changes to how the compiler is invoked. The compiler's optimizing technology has undergone extensive revision as well, but most of these changes are purely internal.

In general, the changes can be grouped into:

- "Command-Line Switches" (some have been removed, some renamed)
- "Linker Description Files" (the default LDFs have been reorganized to reflect the new libraries)
- "Startup Code and Libraries" (the new libraries require different initialization)
- "Miscellaneous changes" (other miscellaneous changes)

Command-Line Switches

Some command-line switches that have been removed, as they are no longer appropriate or enable archaic language dialects no longer supported by the compiler. Other switches have been removed because they historically supported facilities useful for UNIX-targeted compilers but have became inappropriate for compilers targeted at signal processing and embedded platforms.

-csync, -avoid-dag1, -isr-imask-check, -isr-ssync
These switches instruct the compiler to work around particular hardware anomalies. They have all been replaced by a uniform mechanism under the `-workaround` switch. Instead of invoking:

```
ccblkfn -csync prog.c
```

you now invoke:

```
ccblkfn -workaround csync prog.c
```

The other anomaly switches are also passed to `-workaround`, in a similar fashion. This change has been made to create a uniform approach to hardware `-workaround` switches across Analog Devices compilers.

-Ox, -Oz
The `-Ox` and `-Oz` switches have been removed.

The `-Ox` switch specified that the code being compiled contained arithmetic that stayed within the confines of 16 bits. The intention was that DSP code, often expressed via short `ints`, could be evaluated without first expanding expressions to 32-bit `ints`, as required by strict adherence to the ANSI C Standard. Unfortunately, this assertion was difficult to make reliable and led to incorrect program behavior, as comparisons and other operations worked on incorrect data. Now the compiler removes all sign- and zero-extensions it can provably discard and leaves all others in. The `-Ox` switch is accepted by the compiler but has no effect. Use `-O` instead.

The `-Oz` switch specified that the entire program and data resided in the address range `0x00000000` to `0x0000FFFF`, with the intention of only needing to load the low halves of pointer registers. The idea was to reduce program size. Blackfin processors place `L1`, MMR space, and (where available) `L2` into the address space above this range. Furthermore, the low 64 Kbytes of memory space is occupied by slow external memory, so there were no gains to be made from this option. The `-Oz` switch is accepted by the compiler but has no effect.

-circbuf
The old `-circbuf` switch has been renamed to `-force-circbuf` to better indicate that it is causing the creation of circular buffers that would not otherwise happen. The `-no-circbuf` switch exists to prevent the automatic creation of circular buffers. Explicit circular buffer usage, via the corresponding intrinsic functions, is still honored.

-alttok

The `-alttok` switch, which directs the compiler to allow digraph sequences in C and C++ source files, is no longer on by default as it was in previous releases.

-analog is renamed to -c89

The `-analog` switch is renamed to `-c89` to reference a particular revision of the ANSI C Standard.

-2153{1|2|3|5}, -BLACKFIN

These switches, which were used to determine which target to compile for, are removed. Use `-proc` *`processor`* (where *`processor`* is `ADSP-BF53{1|2|3|5}`) instead. The `-BLACKFIN` switch is ignored.

-BF53{1|2|3|5}, -AD6532

These switches have been deprecated in favor of `-proc` *`processor`* (`ADSP-BF53{1|2|3|5}`) and `-proc AD6532`.

-inline, -no-inline

These switches are no longer supported because the inliner is integrated into the optimizer. When optimization is on, the inliner operates. When optimization is off, the inliner is disabled. See also `-Oa` (auto-inlining).

-xml

This switch specified that map files generated by the linker should be in XML format. In VisualDSP++ 3.5, the linker always generates XML-format map files, so the switch has been removed. The `-xml` switch is still accepted but has no effect.

-no-bss

In VisualDSP++ 3.1, the default mode was `-no-bss`. In VisualDSP++ 3.5, the default mode is `-bss`. Having `-bss` as the default has ramifications for the Linker Description Files, as the "`bsz`" sections must be consumed by the linker.

-mem-bsz
This switch invoked the Memory Initializer to process the "`bsz`" sections. In VisualDSP++ 3.5, the Memory Initializer is invoked with `-mem` and defaults to processing all sections that are marked as `ZERO_INIT` and `RUNTIME_INIT`. The "`bsz`" section is marked as `ZERO_INIT` in the default LDFs, so `-mem-bsz` now is redundant. The switch is accepted but ignored.

Other switches
The following switches have been removed because they correspond to language options no longer supported in VisualDSP++ 3.5.
`-traditional`, `-dollar`, `-no-dollar`, `-J`, `-force`, `-instantall`, `-instantlocal`, `-instantused`, `-suppress`, `-tpautooff`.

The following switches have been removed because they were used to select between two compilation modes; only the default mode is available in VisualDSP++ 3.5.
`-bool`, `-explicit`, `-namespace`, `-newforinit`, `-newvec`, `-std`, `-typename`, and `-wchar`. These switches now are always on.

The `-no-bool`, `-no-explicit`, `-no-namespace`, `-trdforinit`, `-no-newvec`, `-no-std`, and `-no-wchar` switches are no longer available because their corresponding elections are not supported in VisualDSP++ 3.5.

Linker Description Files

There are many differences between the LDFs in VisualDSP++3.5 and preceding VisualDSP++ releases. Projects that use customized LDFs derived from earlier VisualDSP++ releases almost certainly require some modification in order to link successfully with VisualDSP++ 3.5.

BSS sections created by default
The libraries now separate global zero-initialized data into the "`bsz`" section, which must be consumed by the LDFs; the compiler will follow suit in later releases. Consequently, the "`bsz`" input sections must be mapped to output sections in data memory space by the LDF.

cplbtab*x*.doj now linked directly by LDF
In VisualDSP++ 3.1, each Blackfin core used a generic memory configuration table. ADSP-BF531/BF532/5333 processors used `cplbtab532.doj` from `libc532.dlb`, and the ADSP-BF535 processor used `cplbtab535.doj` from `libc535.dlb`. Now that each processor has its own specific memory table, the LDF must include this object in the `$OBJECTS` link macro. For example, ADSP-BF533 processor projects should be linked with `cplbtab533.doj`.

Voldata has moved
In VisualDSP++ 3.1, the `voldata` section was mapped into the `L1` Scratchpad area. This is incorrect because `voldata` may contain initialized data, and `L1` Scratchpad may not have initialized data mapped to it. In VisualDSP++ 3.5, the `voldata` section is mapped to normal data space, which is safe by default as the run-time library flushes its `voldata` items from data cache before referencing them. If your application has any additional data items that are mapped into `voldata`, you must map the `voldata` to a suitable, non-cached space or flush the data from cache before referring to it.

USERMODE no longer supported, MEM_SYSSTACK section removed
The optional `USERMODE` configuration of VisualDSP++ 3.1 is no longer supported in VisualDSP++ 3.5. Consequently, there is no need for two stack sections. The space occupied by the `MEM_SYSSTACK` section has been reallocated to the `MEM_STACK` section.

Farther modifications to your application program can take advantage of additional run-time library facilities.

USE_CACHE macro for enabling cache and external memory
The default LDFs allow building in two configurations:

1. The first configuration—internal memories only; no space reserved for caching.
2. The second configuration—internal and external memories; some internal space reserved for caching.

The second configuration is selected by the `USE_CACHE` linker macro.

"Guard" symbols to determine cache availability
The default LDFs define symbols, such as `___l1_cache_data_a`, setting its value to be `1` to indicate that the space is available for cache, or to be `0` to indicate that its space has been used for program/data. The run-time library can use these guard symbols to detect invalid cache configurations.

C++ exception support
The compiler's C++ exception support relies on a support library (`libx*.dlb`) and exceptions-enabled versions of the run-time libraries (named with an "`x`" suffix). In addition, the exceptions support requires several new input sections (`.frt`, `.edt`, `.cht`, `.gdt`) that must be mapped to data areas.

Memory Initializer
The Memory Initializer utility makes use of a special "`.meminit`" section to determine where initialization tables should be placed.

-no-std-lib support
The `-no-std-lib` compiler option indicates that the LDF should not use the default search path to locate libraries by defining linker macro `__NO_STD_LIB` to guard the `SEARCH_DIR` directive.

Startup Code and Libraries

Changes to the run-time library cause a small number of changes to the startup code in `basiccrt.s`.

Clock-speed changes
On ADSP-BF533 processors, the default clock speed is set to be approximately 600 MHz.

No User mode support
As the User mode operation has been discontinued, only one stack is configured, the Supervisor stack. The User stack pointer is set to point to the Supervisor stack. However, the default startup code only goes into User mode long enough to return to Supervisor mode at the lowest Supervisor priority. The startup code does not make use of the User mode stack during this process.

Disposition functions for CPLB failure
After (optionally) enabling CPLBs and any requested caching, the startup code calls `__install_default_handlers`, a stub function that just returns. This routine may be replaced with an alternative routine that installs additional handlers. This is useful if you wish to install handlers prior to `main()` being invoked.

Default exception handler
The default exception handler installed for CPLB management now invokes stub functions when the event is unrecognized or when the replacement operation fails:

```
__unknown_exception_occurred()
__cplb_miss_all_locked()
__cplb_miss_without_replacement()
```

These stub functions merely loop on `IDLE`, since there is no sensible way for the application program to continue, but your program may choose to provide alternative behavior.

Redefined csqu_fr16() intrinsic
The `csqu_fr16()` intrinsic was incorrect, in that it did not sensibly produce the square of a complex 16-bit fractional number. This intrinsic now performs as required.

arg_fr16 function

The DSP run-time library function `arg_fr16` returns the phase that is associated with a Cartesian number. In previous releases of VisualDSP++, the phase returned by the function would be in the range [`-1.0`, `1.0`) corresponding to [`0`, `2ð`). This range was not consistent with the range used by the `polar_fr16` library function, which converts a polar coordinate into a Cartesian number. As a consequence, the following identity was not true:

```
polar (cabs(a),arg(a)) == a
```

In VisualDSP++ 3.5, this conflict has been addressed by modifying the range of the phase returned by the `arg_fr16` function to be [`0`, `1.0`). The mentioned identify now holds true. Furthermore, the following identity,

```
cartesian (polar(mag,phase),&phase) == mag
```

is also true for the new library function `cartesian_fr16`, which has been added for VisualDSP++ 3.5.

Window functions

The definition of the windowing functions (defined in the `window.h` header file) includes the argument *`N`*, which in VisualDSP++ 3.1 and earlier releases is defined as the size of the output vector. Thus, for a stride argument *`a`* other than `1`, the functions would return an incomplete window of *`N/a`* samples.

In VisualDSP++ 3.5, the argument *`N`* has been redefined as the size of the window required. The windowing functions now generate a complete window of *`N`* samples, but the size of the output vector must be *`N * a`*. Note that this change does not affect applications which use a stride value of `1`.

Miscellaneous changes

Interprocedural Analysis (IPA)
The IPA optimizer, `ipa.exe`, has been entirely redesigned for VisualDSP++ 3.5. Although there are no user-visible changes, you should be aware that there are likely to be significant changes to the assembler generated by an IPA-optimized function. You should also ensure that full rebuilds are done, cleaning out any residual `.opa` or `.ipa` files.

Others

- The compiler reorders functions in assembly files (`.s`). Functions do not necessarily appear in the assembly file in the same order in which they are defined in the source file.
- `MEM_PCI_IO` and `pci_io` are no longer needed. The `MEM_PCI_IO` memory space and the "`pci_io`" input sections only need to be included in the LDF when using a PCI-based IO system. This PCI IO system is used for some third-party Blackfin boards and is not part of the standard VisualDSP++ distribution.
- The `defbf532.h` header defined `LOOP` as a macro, which conflicted with assembler syntax. Now it is defined as `LOOP_ENA`.
- The `CHIPID` macro is no longer defined in the `defBF531.h`, `defBF532.h`, or `defBF533.h` include files. The corresponding C macro is not defined in the `cdefBF531.h`, `cdefBF532.h`, or `cdefBF533.h` files.

Changes to ADSP-219x Compiler

This section summarizes changes to the ADSP-219x compiler environment since VisualDSP++ 3.0. As with any major upgrade, some of the improvements have necessitated changes to how the compiler is invoked. The compiler's optimizing technology has undergone extensive revision for VisualDSP++ 3.5, but most of these changes are purely internal.

In general, the changes can be grouped into:

- "Command-Line Switches" (some have been removed, some renamed)
- "Linker Description Files" (some minor changes to support new features)
- "Miscellaneous Changes" (other miscellaneous changes)

Command-Line Switches

Some command-line switches have been removed, as they are no longer appropriate and enable archaic language dialects no longer supported by the compiler. Other switches have been removed because they historically supported facilities useful for UNIX-targeted compilers and became inappropriate for compilers targeted at signal processing and embedded platforms.

-2191, -2192-12, -219x, -219x
These switches have been deprecated in favor of `-proc ADSP-2191`, `-proc ADSP-2192-12`, and `-proc ADSP-219x`.

-alttok is no longer a default
The `-alttok` (alternative tokens) switch directs the compiler to allow digraph sequences in C and C++ source files. This switch is no longer on by default as it was in previous releases.

-analog is renamed to -c89
This switch is renamed to reference a particular revision of the ANSI C Standard.

-inline, -no-inline (deprecated switches)
These switches are no longer supported because the inliner is integrated into the optimizer. When optimization is on, the inliner operates. When optimization is off, the inliner is disabled. See also `-Oa` (auto-inlining).

-no-alttok (now default)
The `-no-alttok` switch directed the compiler not to accept alternative operator keywords and digraph sequences in the source files. In VisualDSP++ 3.5, this is done by default.

Other switches
The following switches have been removed because they correspond to language options no longer supported in VisualDSP++ 3.5.
`-traditional`, `-dollar`, `-no-dollar`, `-J`, `-force`, `-instantall`, `-instantlocal`, `-instantused`, `-suppress`, `-tpautooff`.

The following switches have been removed because they were used to select between two compilation modes; only the default mode is available in VisualDSP++ 3.5.
`-bool`, `-explicit`, `-namespace`, `-newforinit`, `-newvec`, `-std`, `-typename`, and `-wchar`. These switches now are always on.

The `-no-bool`, `-no-explicit`, `-no-namespace`, `-trdforinit`, `-no-newvec`, `-no-std`, and `-no-wchar` switches are no longer available because their corresponding elections are not supported in VisualDSP++ 3.5.

Linker Description Files

Default LDFs support -no-std-libs
The compiler has a `-no-std-lib` switch, which directs the compiler not to search the standard library directory for libraries to resolve symbols. This is achieved by defining the linker preprocessing macro `__NO_STD_LIB`, which guards the `SEARCH_DIR` linker directive. Without this guarding macro, the LDF still directs the linker to search the standard directory.

Default LDFs include .meminit
The memory initializer uses a special "`.meminit`" section to determine placement of initialization tables.

Libraries

arg_fr16 function
The DSP run-time library function `arg_fr16` returns the phase that is associated with a Cartesian number. In previous releases of VisualDSP++, the phase returned by the function would be in the range [`-1.0`, `1.0`) corresponding to [`0`, `2ð`). This range was not consistent with the range used by the `polar_fr16` library function, which converts a polar coordinate into a Cartesian number. As a consequence, the following identity was not true:

```
    polar (cabs(a),arg(a)) == a
```

In VisualDSP++ 3.5, this conflict has been addressed by modifying the range of the phase returned by the `arg_fr16` function to be [`0`, `1.0`). The mentioned identify now holds true. Furthermore, the following identity,

```
    cartesian (polar(mag,phase),&phase) == mag
```

is also true for the new library function `cartesian_fr16`, which has been added for VisualDSP++ 3.5.

Window functions
The definition of the windowing functions (defined in the `window.h` header file) includes the argument *`N`*, which in VisualDSP++ 3.0 and earlier releases, is defined as the size of the output vector. Thus, for a stride argument *`a`* other than `1`, the functions return an incomplete window of *`N/a`* samples.

In VisualDSP++ 3.5, the argument *`N`* has been redefined as the size of the window required. The windowing functions now generate a complete window of *`N`* samples, but the size of the output vector must now be *`N *a`*. Note that this change does not affect applications that use a stride value of `1`.

Miscellaneous Changes

Structure return values when the structure size is one or two words
The compiler and libraries of VisualDSP++ 3.5 have been modified to fix a long-standing problem concerning one- and two-word structures. In the previous releases, one- and two-word structures were returned through the aggregate return mechanism using callee-allocated memory with the address passed in registers `I0`. The C run-time description states that structures of size one (word) should be returned by the `AX1` register, and size two should be returned by `SR1:0`. Assembly code that corresponds to the previous behavior (use of `I0`) needs to be modified to use `AX1` or `SR1:0`. All C code needs to be recompiled with the new compiler to ensure compatibility.

Global interrupt enable
The `signal()` and `interrupt()` functions now are no longer enable interrupts; therefore, it might be necessary to globally enable interrupts where it might not have been required before.

Changes to ADSP-218x Compiler

This section summarizes changes to the ADSP-218x compiler environment since VisualDSP++ 3.0. As with any major upgrade, some of the improvements have necessitated changes to how the compiler is invoked. The compiler's optimizing technology has undergone extensive revision for VisualDSP++ 3.5, but most of these changes are purely internal.

In general, the changes to the ADSP-218x compiler can be grouped into:

- "Command-Line Switches" (some have been removed, some renamed)
- "Linker Description Files" (some minor changes to support new features)
- "Miscellaneous Changes" (other miscellaneous changes)

Command-Line Switches

Some ADSP-219x compiler switches that have been removed, as they are no longer appropriate or enable archaic language dialects no longer supported by the compiler. Other switches have been removed because they historically supported facilities useful for UNIX-targeted compilers but inappropriate for compilers targeted at signal-processing and embedded platforms.

-21{8x|81|83|84|85|86|87|88|89}
These switches have been deprecated in favor of
`-proc ADSP-21{81|83|84|85|86|87|88|89}`.

-analog is renamed to -c89
The `-c89` switch is renamed to reference a particular revision of the ANSI C Standard.

-inline, -no-inline (deprecated switches)
These switches are no longer supported because the inliner is integrated into the optimizer. When optimization is on, the inliner operates. When optimization is off, the inliner is disabled. See also `-Oa` (auto-inlining).

Other switches
The following switches have been removed because they correspond to language options no longer supported in VisualDSP++ 3.5.
`-traditional`, `-dollar`, `-no-dollar`, `-J`.

Linker Description Files

Default LDFs support -no-std-libs
The compiler has a `-no-std-lib` switch, which directs the compiler not to search the standard library directory for libraries to resolve symbols. This is achieved by defining the linker preprocessing macro `__NO_STD_LIB`, which guards the `SEARCH_DIR` linker directive. Without this guarding macro, the LDF still directs the linker to search the standard directory.

Default LDFs include .meminit
The memory initializer makes use of a special "`.meminit`" section to determine the placement of initialization tables.

Miscellaneous Changes

Structure return values when the structure size is one or two words
The compiler and libraries of VisualDSP++ 3.5 have been modified to fix a long-standing problem concerning one- and two-word structures. In the previous releases, one- and two-word structures were returned through the aggregate return mechanism using callee-allocated memory with the address passed in registers `I0`. The C run-time description states that structures of size one (word) should be returned by the `AX1` register, and size two should be returned by `SR1:0`. Assembly code that corresponds to the previous behavior (use of `I0`) needs to be modified to use `AX1` or

`SR1:0`. All C code needs to be recompiled with the new compiler to ensure compatibility.

Changes to ADSP-BF535 Simulator

The ADSP-BF535 simulator has been entirely redesigned to support cycle-accurate simulations. The new simulator models core and memory events in order to forecast latencies and feed the pipeline viewer with appropriate information. This replaces the post-pass instruction analysis used by the functional simulator to detect latencies.

The new simulator also allows cache events to be tracked with the cache viewer.

3 NEW FEATURES AND ENHANCEMENTS

VisualDSP++ 3.5 has a number of new features and enhancements designed to increase productivity and shorten application development cycles. This chapter describes the new features and enhancements introduced in VisualDSP++ 3.5.

The information is presented as follows.

VisualDSP++ IDDE

The VisualDSP++ 3.5 Integrated Development and Debugging Environment (IDDE) introduces:

- “New Processor Support” on page 3-3
- “Multiple Project Support” on page 3-3
- “XML Project File Format” on page 3-3
- “Project Migration” on page 3-3
- “License Management” on page 3-4
- “Data Streaming and Logging” on page 3-4
- “Profile-Guided Optimization” on page 3-4
- “Integrated Source Code Control” on page 3-4
- “Automation Aware Scripting Engine” on page 3-5
- “Address Bar in Memory and Disassembly Windows” on page 3-5
- “Menus with Icons” on page 3-5
- “Enhanced Compiled Simulation Support” on page 3-5

For more information about VisualDSP++ IDDE, refer to the *VisualDSP++ 3.5 User’s Guide for 16-Bit Processors* and online Help.

New Processor Support

The following new processor is supported by VisualDSP++ 3.5.

`ADSP-BF561`

Refer to the *ADSP-BF561 Blackfin Processor Hardware Reference* and data sheet for details.

Multiple Project Support

VisualDSP++ provides the ability to switch among multiple open projects in the same IDDE session. The **Project** window displays active projects.

XML Project File Format

The binary project file format used in previous releases of VisualDSP++ has been replaced with a new text-based XML format. The XML format has several benefits:

- Forward and backward compatibility in successive releases
- Better version control and comparison in source code control environments
- Better readability

Project Migration

Projects migrated to VisualDSP++ 3.5 cannot be opened in previous versions. However, a backup copy of the project is automatically made before conversion that can be opened for backwards compatibility.

License Management

License installation and validation has been integrated into the VisualDSP++ IDDE. Installing a FlexLM license server is still handled by the separate installation application. Two licensing options are available: *single-user* and *client*. A *server* license is required before you can install a client license.

Data Streaming and Logging

VisualDSP++ now offers the ability to stream from a target DSP without halting the DSP. The IDDE takes advantage of this capability in plot windows. If the target supports BTC, the plot window is updated without halting the target.

Profile-Guided Optimization

The VisualDSP++ IDDE includes facilities to run common Profile-Guided Optimization (PGO) scenarios simply and also provides a mechanism for advanced users who require more control over the profiling process via scripting. The technique relies on setting up and executing data sets to produce an optimized application. See "Optimization Control" on page 3-16 for more information.

Integrated Source Code Control

VisualDSP++ provides integration between the IDDE and Integrated Source Code Control (SCC) applications (such as Visual SourceSafe, PVCS Version Manager, and Concurrent Versions System (CVS)) installed on your machine through Microsoft Common Source Code Control (MCSCC) interface. You can conveniently access commonly-used SCC features from VisualDSP++ without leaving the IDDE. Application-specific and advanced SCC features not available from the IDDE must be run directly from the SCC applications.

Automation Aware Scripting Engine

VisualDSP++ includes a scripting engine that utilizes the Microsoft ActiveX script host framework. The engine allows you to use multiple scripting languages, such as VBScript, JavaScript and others, to access the VisualDSP++ Automation API.

You can interact with the IDDE using a single command or a script file similar to the Tcl scripting functionality, which was available in previous versions of VisualDSP++.

Address Bar in Memory and Disassembly Windows

When enabled, Disassembly and Memory windows display an address bar. Use the address bar to navigate by address, symbol, or expression. The address bar maintains a most recently used history of visited locations.

Menus with Icons

Icons now appear beside menu commands that have equivalent toolbar buttons.

Enhanced Compiled Simulation Support

Compiled simulation has been enhanced for the VisualDSP++ 3.5 release. Compiled simulation now supports the following functions:

- “Memory Streams”
- “Overlays”
- “PGO”
- “Linear Profiling”
- “ADSP-BF535 SPORT”

Memory Streams

The 8-, 16-, and 32-bit streams are supported to any suitably aligned memory address (excepting the core MMRs).

Overlays

Code overlays are now supported. Debugging is available if your memory manager includes the debug labels `_ov_start` and `_ov_end`, as for normal simulation.

PGO

The compiler's Profile-Guided Optimization is supported by compiled simulation to provide a faster simulation speed.

Linear Profiling

Larger programs can be now profiled due to the faster simulation speed. Note that the cycle counts are not fully cycle accurate, they reflect an instruction count plus core program control stalls.

ADSP-BF535 SPORT

The ADSP-BF535 processor now includes the SPORT peripheral.

Assembler for Blackfin Processors

For Blackfin processors, the assembler's most notable new features and enhancements are:

- "Feature Macro" on page 3-7
- "VCSE Optimization Directives" on page 3-7
- "Assembler Command-Line Switch" on page 3-8
- "Preprocessor Macros" on page 3-8
- "Preprocessor Command-Line Switches" on page 3-8
- "Include Path Search Algorithm Matches Compiler" on page 3-8

For more information, refer to the *VisualDSP++ 3.5 Assembler and Preprocessor Manual for Blackfin Processors* and online Help.

Feature Macro

The new feature macros is `-D__ADSPBF561__=1`. It is present when running `easmblkfn -proc ADSP-BF561`.

VCSE Optimization Directives

The `.VCSE_` directives are the optimization directives for VCSE components. The `.VCSE_METHODCALL_START` and `.VCSE_METHODCALL_END` directives mark VCSE methods for linker code/data elimination. The linker is provided with the interface name and actual offset of the corresponding entry in the method table. The `.VCSE_RETURNS` directive is used for marking VCSE constant methods.

Assembler Command-Line Switch

The new assembler command line switch `-si-revision` *`version`* specifies a silicon revision of the named processor.

Preprocessor Macros

The new preprocessor predefined macro, `__LASTSUFFIX__`, specifies the last value of suffix that was used to build preprocessor generated labels.

The new preprocessor feature macro is `__ADSPBF561__`.

Preprocessor Command-Line Switches

VisualDSP++ 3.5 introduces some new preprocessor command-line switches:

Switch Name	Description
`-cstring`	Enables the stringization operator and provides “C compiler” style preprocessor behavior
`-tokenize-dot`	Treats “.” (dot) as an operator when parsing identifiers
`-v[erbose]`	Displays information about each preprocessing phase
`-version`	Displays version information for preprocessor
`-w`	Removes all preprocessor-generated warnings
-`W`*`number`*	Suppresses any report of the specified warning
`-warn`	Prints warning messages (default)

Include Path Search Algorithm Matches Compiler

In VisualDSP++ 3.5, the `include` path semantics used by the linker/assembler preprocessor and the compiler for user and system header files are the same.

The `#include <file>` (system header file) search order is:

1. include path specified by the `-I` switch
2. `...\VisualDSP\processor\include` folders

The `#include “file”` (user header file) search order is:

1. local directory —the directory in which the source file resides
2. include path specified by the `-I` switch
3. `...VisualDSP\processor\include` folders

When both `-I` and `-I-` appear on the command line, the system search path (`#include < >`) is modified in such a manner that search directories specified with the `-I` switch that appear before the directory specified with the `-I-` switch are ignored.

Assembler for ADSP-218x and ADSP-219x DSPs

For ADSP-218x and ADSP-219x DSPs, the assemblers' most notable new features and enhancements are:

- "VCSE Optimization Directives" on page 3-10
- "Assembler Command-Line Switch" on page 3-10
- "Preprocessor Macro" on page 3-11
- "Preprocessor Command-Line Switches" on page 3-11
- "Include Path Search Algorithm Matches Compiler" on page 3-11

For more information, refer to the *VisualDSP++ 3.5 Assembler and Preprocessor Manual for ADSP-218x and ADSP-219x DSPs* and online Help.

VCSE Optimization Directives

The `.VCSE_` directives are the optimization directives for VCSE components. The `.VCSE_METHODCALL_START` and `.VCSE_METHODCALL_END` directives mark VCSE methods for linker code/data elimination. The linker is provided with the interface name and actual offset of the corresponding entry in the method table. The `.VCSE_RETURNS` directive is used for marking VCSE constant methods.

Assembler Command-Line Switch

The new assembler command line switch `-si-revision` *`version`* specifies a silicon revision of the named processor.

Preprocessor Macro

The new preprocessor predefined macro `__LASTSUFFIX__` specifies the last value of suffix that was used to build preprocessor generated labels.

Preprocessor Command-Line Switches

The following preprocessor command-line switches have been introduced in VisualDSP++ 3.5.

Switch Name	Description
`-cstring`	Enables the stringization operator and provides "C compiler" style preprocessor behavior
`-tokenize-dot`	Treats "." (dot) as an operator when parsing identifiers
`-v[erbose]`	Displays information about each preprocessing phase
`-version`	Displays version information for preprocessor
`-w`	Removes all preprocessor-generated warnings
`-Wnumber`	Suppresses any report of the specified warning
`-warn`	Prints warning messages (default)

Include Path Search Algorithm Matches Compiler

In VisualDSP++ 3.5, the `include` path semantics used by the linker/assembler preprocessor and the compiler for user and system header files are the same.

The `#include <file>` (system header file) search order is:

1. include path specified by the `-I` switch
2. `...\VisualDSP\processor\include` folders

The `#include "file"` (user header file) search order is:

1. local directory —the directory in which the source file resides
2. include path specified by the `-I` switch
3. `...VisualDSP\processor\include` folders

When both `-I` and `-I-` appear on the command line, the system search path (`#include < >`) is modified in such a manner that search directories specified with the `-I` switch that appear before the directory specified with the `-I-` switch are ignored.

Compiler and Library for Blackfin Processors

For Blackfin processors, the most notable new features and enhancements of the C/C++ compiler are:

- "File Extensions" on page 3-14
- "Compiler Command-Line Switches" on page 3-14
- "Optimization Control" on page 3-16
- "Bank Type Qualifiers" on page 3-17
- "ETSI Support" on page 3-18
- "Video Operation Built-In Functions" on page 3-18
- "Pragmas" on page 3-19
- "GCC Compatibility Extensions" on page 3-21
- "Caching and Memory Protection" on page 3-22
- "Predefined Compiler Macros" on page 3-23
- "Data Storage Formats" on page 3-24
- "C/C++ Libraries and Startup Files" on page 3-24
- "DSP Run-Time Library" on page 3-26

For detailed information on these features, refer to the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors* and online Help.

File Extensions

The compiler supports new file extensions.

File Extension	Description
`.cc`	C++ source code
`.idl`	Interface definition language files for VCSE
`.pgo`	Execution profile generated by Profile-Guided Optimization
`.pch`	Precompiled header file
`.xml`	Processor memory map file output

Compiler Command-Line Switches

This section summarizes C/C++ compiler command-line switches introduced or enhanced in VisualDSP++ 3.5. Table 3-1 through Table 3-3 list and briefly describe each switch:

- Table 3-1, “C or C++ Mode Selection Switches” on page 3-14
- Table 3-2, “C/C++ Compiler Common Switches” on page 3-15
- Table 3-3, “C++ Mode Compiler Switches” on page 3-16.

Table 3-1. C or C++ Mode Selection Switches

Switch Name	Description
`-c89`	Supports programs that conform to the ISO/IEC 9899:1990 standard

Table 3-2. C/C++ Compiler Common Switches

Switch Name	Description
`-default-linkage-<asm|C|C++>`	Sets the default linkage type.
`-ED`	Preprocesses and sends all output to a file.
`-flags-meminit`	Passes each comma-separated option to the Memory Initializer utility.
`-force-circbuf`	Treats array references of the form `array[i%n]` as circular buffer operations.
`-force-link`	Forces stack frame creation for leaf functions. Always creates a new stack frame for leaf functions (defaults to ON with `-g` option set, enforced for the `-p` option).
`-I-`	Specifies the point in the include directory list where the search for header files enclosed in angle brackets should begin.
`-i`	Outputs only header details or makefile dependencies for `include` files specified in double quotes.
`-MD`	Generates `make` rule, compiles, and prints to a file.
`-Mo` *filename*	Writes dependency information to *filename*. This switch is used in conjunction with the `-ED` or `-MD` options.
`-no-circbuf`	Disables the automatic generation of circular buffering code.
`-no-force-link`	Does not create a new stack frame for leaf functions, if one can be omitted. Overrides the default for -g.
`-Oa`	Enables automatic function inlining.
`-path-`*tool*	Enhanced. Uses the specified directory as the location of the build *tool*.
`-pch`	Enables automatic generation and use of precompiled header files.
`-pch` *directory*	Specifies an alternative directory to `PCHRepository` in which to store precompiled header files.
`-pguide`	Add instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization.
`-signed-bitfield`	Makes the default type for `int bitfields` signed.

Table 3-2. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-si-revision` *`version`*	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision.
`-sysdefs`	Instructs the driver to define preprocessor macros that describe the current user and machine.
`-unsigned-bitfield`	Makes the default type for plain int bitfields unsigned.
`-val-global` *`name-list`*	Adds global names.
`-workaround` *`workaround`*	Enables code generator workaround for specific hardware errata.

Table 3-3. C++ Mode Compiler Switches

Switch Name	Description
`-anach`	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use.
`-eh`	Enables exception handling.
`-no-anach`	Disallows the use of anachronisms that are prohibited by the C++ standard.
`-no-eh`	Disables exception-handling.
`-no-rtti`	Disables run-time type information.
`-rtti`	Enables run-time type information.

Optimization Control

The following list identifies several new optimization levels. Refer to Chapter 2, *Achieving Optimal Performance from C/C++ Source Code*, of the compiler manual for detailed information on how to obtain maximal code performance from the compiler.

The new and enhanced optimization features are:

- Profile-Guided Optimizations
  The compiler performs advanced aggressive optimizations using profiler statistics generated from running the application using representative training data. PGO can be used in conjunction with interprocedural optimizations (IPA) and automatic inlining. The PGO operation is handled via a new PGO submenu added to the top-level Tools menu: Tools -> PGO -> Manage Data Sets.

- Automatic Inlining
  The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code. The compiler determines when the inlining will reduce execution time. How aggressively the compiler performs automatic inlining is controlled by the `-Ov` switch. Automatic inlining is enabled using the `-Oa` switch and additionally enables Procedural Optimizations (`-O`).

- C++ Standard Exceptions Support
  When in C++ mode, the compiler supports run-time type information (RTTI) and exception handling as defined by the ISO/IEC 14882:1998 standard and modified by Technical Corrigendum 1. Exceptions are enabled using the `-eh` switch, and the RTTI is enabled using the `-rtti` switch.

- 64-bit Integer Support
  The C/C++ compiler fully supports 64-bit integer types: `long long` and `unsigned long long`.

Bank Type Qualifiers

The new `bank("string")` keyword can be used in data declarations to indicate that the data resides in a particular memory bank.

ETSI Support

VisualDSP++ 3.5 for Blackfin processors provides European Telecommunications Standards Institute (ETSI) support routines in `libetsi*.dlb`. The library contains routines for manipulating the `fract16` and `fract32` data types. The routines provide bit-accurate calculations for common operations and conversions between `fract16` and `fract32` data. To use the ETSI routines, the header file `libetsi.h` must be included, and all source code must be compiled with the `ETSI_SOURCE` macro defined.

The "*ETSI Support*" section of the *VisualDSP++ 3.5 C/C++ Compiler Manual for Blackfin Processors* provides descriptions of all 16- and 32-bit fractional ETSI routines.

Video Operation Built-In Functions

The VisualDSP++ 3.5 C/C++ compiler provides new built-in functions for the Blackfin processor's video pixel operations. You should include the video.h header file before using the builtins. For further information regarding the underlying Blackfin processor instructions that implement the video operations, refer to the *Blackfin DSP Instruction Set Reference*.

Pragmas

The VisualDSP++ 3.5 C/C++ compiler supports a number of new pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. The new and enhanced pragmas are described briefly in Table 3-4.

Table 3-4. ADSP-BF53x Compiler Pragmas

Pragma	Function
`#pragma all_aligned`	Asserts that all pointers are initially aligned on the most desirable boundary; applies to the subsequent loop.
`#pragma no_vectorization`	Turns off all vectorization for the loop on which it is specified.
`#pragma different_banks`	Allows the compiler to assume that groups of memory accesses based on different pointers within a loop reside in different memory banks.
`#pragma loop_count(`*`min, max, modulo`*`)`	Asserts that the loop will iterate at least *`min`* times, no more than *`max`* times, and a multiple of *`modulo`* times. This information enables the optimizer to omit loop guards, to decide whether the loop is worth completely unrolling, and whether code need be generated for odd iterations.
`#pragma optimize_as_cmd_line`	Resets the optimization settings to be those specified on the `ccblkfn` command line when the compiler was invoked.
`#pragma alloc`	Tells the compiler that the function behaves like the library function "`malloc`", returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call.

Table 3-4. ADSP-BF53x Compiler Pragmas (Cont'd)

Pragma	Function
`#pragma pure`	Tells the compiler that the function does not write to any global variables, and does not read or write any volatile variables.
`#pragma const`	This is a more restrictive form of the `pure` pragma. It tells the compiler that the function does not read from global variables as well as not writing to them or reading or writing volatile variables. The result of the function is therefore a function of its parameters.
`#pragma regs_clobbered` *`string`*	Used with a function declaration or definition to specify which registers are modified (or clobbered) by that function.
`#pragma result_alignment (`*`n`*`)`	Asserts that the pointer or integer returned by the function has a value that is a multiple of *`n`*.
`#pragma instantiate` *`instance`*	Requests the compiler to instantiate *`instance`* in the current compilation.
`#pragma do_not_instantiate` *`instance`*	Directs the compiler not to instantiate *`instance`* in the current compilation.
`#pragma can_instantiate` *`instance`*	Tells the compiler that if instance is required anywhere in the program, it should be instantiated in this compilation and has the same effect as `#pragma instantiate`.
`#pragma hdrstop`	Used in conjunction with the `-pch` (precompiled header) switch. The switch tells the compiler to look for a precompiled header (`.pch` file) and, if it cannot find one, to generate a file for use on a later compilation.
`#pragma no_pch`	Overrides the `-pch` (precompiled headers) switch for a particular source file. It directs the compiler not to look for a `.pch` file and not to generate one for the specified source file.

Table 3-4. ADSP-BF53x Compiler Pragmas (Cont'd)

Pragma	Function
`#pragma once`	Appears at the beginning of a header file and tells the compiler that the header is written in such a way that including it several times has the same effect as including it once.
`#pragma system_header`	Appears in a header file and identifies the file as one that is supplied with VisualDSP++.

GCC Compatibility Extensions

The compiler provides compatibility with the C dialect accepted by version 3.2 of the GNU C Compiler. Many of these extensions are available in the C99 ANSI Standard. A brief description of each extension is included in the compiler manual (see Chapter 1). The following topics are described:

- Statement expressions
- Type reference support keyword (`Typeof`)
- GCC generalized `Lvalues`
- Hexadecimal floating-point numbers
- Arithmetic on pointers to `void` and pointers to functions
- Cast to `union`
- Ranges in `case` labels
- Declarations mixed with code
- Conditional expressions with missing operands
- Zero length arrays
- Variable argument macros

- Line breaks in string literals
- Escape character constant
- Alignment inquiry keyword (`__alignof__`)
- Keyword for specifying names in generated assembler (`asm`)
- Function, variable and type attribute keyword (`__attribute__`)

Caching and Memory Protection

Blackfin processors support caching of external memory or L2 SRAM into L1 SRAM, for both Instruction and Data memory. The cache configuration is defined through the memory protection hardware, using tables that define Cache Protection Lookaside Buffers (CPLBs).

The new and/or enhanced features are:

- Code and Data CPLBs can be enabled independently through new bits in the `__cplb_ctrl` control variable.
- Write Back cache mode is now supported in addition to Write Through cache mode.
- There is a complete CPLB table specific to each Blackfin processor in the appropriate `cplbtabx.doj`.
- The routine for handling CPLB exceptions now calls stub functions when some error condition is detected. These stubs can be replaced to provide application-specific error handling.

Predefined Compiler Macros

The new predefined compiler macros are:

- The `__ADSPLPBLACKFIN__` macro is defined as 1 when the target processor (set with the `-proc` switch) is one of ADSP-BF531, ADSP-BF532, ADSP-BF533, or ADSP-BF561 processors.
- The `__ADSPBF561__` macro is defined as 1 when building for the ADSP-BF561 processor target with `-proc ADSP-BF561`.
- The `__DOUBLES_ARE_FLOATS__` macro is always defined as 1. This macro indicates that the `double` type is supported as a single-precision type, the same as type `float`.
- The `__EXCEPTIONS` macro is defined as 1 when C++ exception handling is enabled (using the `-eh` switch).
- The `__LANGUAGE_C` macro is always defined as 1. This macro is present when used for C compiler calls to specify headers.
- The `__RTTI` macro is defined as 1 when C++ run-time type information is enabled (using the -rtti switch).
- The `__SIGNED_CHARS__` macro is defined as 1, unless you compile with the `-unsigned-char` command-line switch.

Data Storage Formats

The new data types and data formats used by Blackfin processors are.

Table 3-5. New Data Storage Formats and Data Type Sizes

Type	Bit Size	Number Representation	`sizeof` returns
`long long`	64 bits signed	64-bit two's complement	8
`unsigned long long`	64 bits unsigned	64-bit unsigned magnitude	8

Refer to "*Using Data Storage Formats*" in Chapter 1 of the compiler manual for more information.

C/C++ Libraries and Startup Files

The new C/C++ run-time libraries and startup files are:

Library\Startup File	Description
`cplbtab*.doj`	Memory protection and caching attributes for each Blackfin memory map. Default cache configuration table.
`idle*.doj`	Normal "termination" code that enters IDLE loop after "end" of the application.
`__initsbsz*.doj`	Memory initializer support files.
`libx*.dlb`	C++ exception handling support library.

ⓘ All startup files and run-time libraries are located in `Blackfin\lib` subdirectory of your VisualDSP++ installation directory.

The new and updated filename suffixes (`*`) are.

Filename Suffix	Description
`531`	Compiled only for ADSP-BF531 processor
`532`	Compiled for execution on a ADSP-BF531, ADSP-BF532, ADSP-BF533, or ADSP-BF561 processor
`533`	Compiled only for ADSP-BF533 processor
`535`	Compiled for execution on a ADSP-BF535 or AD6532 processor
`561`	Compiled only for ADSP-BF561 processor
`a`	Compiled only for ADSP-BF561 Core A processor
`b`	Compiled only for ADSP-BF561 Core B processor
`x`	Compiled with C++ exception handling enabled

C Library Functions

The C run-time library has been extended with the addition of some new functions and enhanced functionality.

The formatted input/output functions defined in `stdio.h` (i.e. `printf`, `scanf`, `fprintf`, `...`) now support the *`%a`* conversion specifier. The *`%a`* specifier is similar both in form and meaning to the *`%e`* specifier, with the exception that the `%e` specifier is used to input and output decimal floating-point numbers, while the *`%a`* specifier is used to input and output hexadecimal floating-point numbers.

Additional functions defined in the `stdio.h` header file are supported in the VisualDSP++ 3.5 release. These functions are:
`fgetpos`, `fseek`, `fsetpos`, `ftell`, `remove`, `rename`, `rewind`

ⓘ The C standard `stdio.h` functions `tmpfile` and `tmpnam` are not supported in this release. The `isinf` and `isnan` functions existed in VisualDSP++ 3.1 but were not documented.

The complete list of supported library functions in the `stdio.h` header file is as follows.

`clearerr`	`fclose`	`feof`
`ferror`	`fflush`	`fgetc`
`fgetpos`	`fgets`	`fprintf`
`fputc`	`fputs`	`fopen`
`fread`	`freopen`	`fscanf`
`fseek`	`fsetpos`	`ftell`
`fwrite`	`getc`	`getchar`
`gets`	`perror`	`putc`
`putchar`	`puts`	`printf`
`remove`	`rename`	`rewind`
`scanf`	`setbuf`	`setvbuf`
`sprintf`	`sscanf`	`ungetc`
`vfprintf`	`vprintf`	`vsprintf`

The `atof` and `strtod` functions have been modified to support hexadecimal floating-point numbers. The function documentation has been updated to reflect the new functionality.

DSP Run-Time Library

Some new functions have been added to the DSP run-time library. These functions are identified in the following table.

Library Function	Description
`alog`	calculates the natural (base e) anti-log of its argument
`alog10`	calculates the base 10 anti-log of its argument
`cartesian`	transforms a complex number from Cartesian notation to polar notation

Library Function	Description
`cfftf_fr16`	fast N-point radix-4 complex input FFT
`twidfftf_fr16`	generate FFT twiddle factors for cfftf_fr16

ⓘ The new functions are fully documented in the compiler manual.

Compiler and Library for ADSP-219x Processors

For ADSP-219x processors, the most notable new features and enhancements of the C/C++ compiler are:

- "File Extensions" on page 3-28
- "Compiler Command-Line Switches" on page 3-29
- "Optimization Control" on page 3-31
- "ETSI Support" on page 3-35
- "Pragmas" on page 3-36
- "GCC Compatibility Extensions" on page 3-38
- "Predefined Compiler Macro" on page 3-39
- "File IO Support" on page 3-39
- "C Library Functions" on page 3-40
- "DSP Run-Time Library Functions" on page 3-41

For more information about these features, refer to the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual for ADSP-219x DSPs* and online Help.

File Extensions

The compiler supports new file extensions:

File Extension	Description
`.cc`	C++ source code
`.ipa, .opa`	IPA files used internally by the `.pch` header file

File Extension	Description
`.pch`	Precompiled header file
`.xml`	Processor memory map file output

Compiler Command-Line Switches

This section summarizes C/C++ compiler command-line switches introduced or enhanced in VisualDSP++ 3.5. Table 3-6 through Table 3-8 list and briefly describe each switch:

- Table 3-6, "C or C++ Mode Selection Switches" on page 3-29
- Table 3-7, "C/C++ Compiler Common Switches" on page 3-29
- Table 3-8, "C++ Mode Compiler Switches" on page 3-31

Table 3-6. C or C++ Mode Selection Switches

Switch Name	Description
`-c89`	Supports programs that conform to the ISO/IEC 9899:1990 standard.

Table 3-7. C/C++ Compiler Common Switches

Switch Name	Description
`-bss`	Causes the compiler to put global zero-initialized data into a separate BSS-style section.
`-const-read-write`	Specifies that data accessed via a pointer to `const` data may be modified elsewhere.
`-ED`	Preprocesses and sends all output to a file.
`-flags-meminit`	Passes each comma-separated option to the Memory Initializer utility.

Table 3-7. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-fp-associative`	Treats floating-point multiplication and addition as associative.
`-i`	Outputs only header details or makefile dependencies for `include` files specified in double quotes.
`-ipa`	Specifies that interprocedural analysis should be performed for optimization between translation units.
`-jump-{pm|dm|same}`	Specifies where the compiler should place jump tables in memory.
`-MD`	Generates `make` rule, compiles, and prints to a file.
`-Mo` *`filename`*	Writes dependency information to *`filename`*. This switch is used in conjunction with the `-ED` or `-MD` options.
`-mem`	Causes the compiler to invoke the Memory Initializer after linking the executable file.
`-no-bss`	Causes the compiler to group global zero-initialized data into the same section as global data with non-zero initializers.
`-no-circbuf`	Disables the automatic generation of circular buffering code.
`-no-fp-associative`	Does not treat floating-point multiply and addition as an associative.
`-no-mem`	Causes the compiler to not invoke the Memory Initializer after linking; set by default.
`-Oa`	Enables automatic function inlining.
`-Ov` *`num`*	Controls speed vs. size optimizations.
`-oldasmcall-{csp|8x}`	Switches the operation of the `OldAsmCall` linkage specifier between compatibility call for the ADSP-21csp01 and legacy ADSP-218x processors (`-oldasmcall-csp` is default).

Table 3-7. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
`-path-`*`tool`*	Enhanced. Uses the specified directory as the location of the specified compilation *`tool`* (assembler, compiler, library builder, linker, or memory initializer).
`-signed-bitfield`	Makes the default type for `int bitfields` signed.
`-si-revision` *`version`*	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision.
`-unsigned-bitfield`	Makes the default type for plain int bitfields unsigned.

Table 3-8. C++ Mode Compiler Switches

Switch Name	Description
`-anach`	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use.
`-no-anach`	Disallows the use of anachronisms that are prohibited by the C++ standard.

Optimization Control

The following list identifies several new optimization levels. Refer to Chapter 2, *Achieving Optimal Performance from C/C++ Source Code*, of the compiler manual for detailed information on how to obtain maximal code performance from the compiler. The new and enhanced optimization features are:

- Automatic Inlining
  The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code. The compiler determines when the inlining will reduce execution time. How

aggressively the compiler performs automatic inlining is controlled using the `-Ov` switch. Automatic inlining is enabled using the `-Oa` switch and additionally enables procedural optimizations (`-O`).

- Interprocedural Optimizations
  The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. IPA is enabled using the `-ipa` switch and additionally enables Procedural Optimizations (`-O`).

Assembly Construct Operands

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. To know how to assign registers to operands, the compiler needs information on operand constraints. A detailed description of how to convey this information to the compiler can be found in the compiler manual.

New asm constraints for MAC and SHIFTER long result targets
Two new `asm()` statement constraints, “`M`” and “`S`”, have been added to allow users to map `MAC` and `SHIFTER` outputs to C/C++ long (double-word) variables. Example usage:

```
static long f1(int a, int b) {
      long ret;
      asm volatile("%0 = %1 * %2 (SS);"
           :"=M"(ret)
           : "b"(a),"B"(b) : );
      return ret;
}

static long f2(int a, int b) {
      long ret;
      asm volatile("SE=%1; %0 = LSHIFT %2 (lo);"
           :"=S"(ret)
           : "e"(b), "d"(a) : "SE" );
      return ret;
}
```

Hard-register support in asm statements
It is now possible to claim registers directly for use as `asm()` constraints, instead of requesting a register from a certain class using the constraint letters. This is done by simply naming the register in the location where the class letter would previously have been given. For example,

```
    asm("%0 = %1 + %2;"
                   :"=ar"(sum)      /* output */
                   :"g"(x), "G"(y)  /* input */
       );
```

loads `x` into ALU-X register, `y` into ALU-Y register, and calculated `sum` in register `AR`.

Expanded list of asm constraints: *+symbol*, ?*symbol*, and #*symbol*
The `+symbol` operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C/C++ expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C/C++ expression. Therefore, as with pure outputs, the C/C++ expression must be one that is valid on the left-hand side of an assignment.

The `?symbol` operand is temporary. It must appear as part of the third argument to the `asm()` construct. A register is allocated as working space for the duration of the `asm()` template execution. The register's initial value is undefined, and the register's final value is discarded. The corresponding C/C++ expression is not loaded into the register but must be present. This expression is normally specified using a literal zero.

The `#symbol` operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. The operand must appear as part of the second argument to the `asm()` construct.

Builtins for Non-literal Address Inputs

The `o_space_read` and `io_space_write` `sysreg.h` builtins now support non-literal address inputs.

The instruction set limits the IO memory load and store instructions to using literal addresses only. Previously, the compiler-supported builtins, which used the `io_space_read` and `io_space_write` instructions, asserted or caused an assembler error when non-literal addresses or an out-of-range literal were used. The VisualDSP++ 3.5 compiler has been enhanced to call a support subroutine defined in the C run-time library to implement the unsupported cases by generating the IO access instruction using values loaded programatically. The support will not be most efficient and `io_space_read` and `io_space_write` address parameters should be passed as literals when this is possible.

System Control Register Set

New `sysreg.h` support for non-mapped system registers has been added to the `sysreg` enumeration. The register definitions added are `CACTL`, `DBGC-TRL`, `DBGSTAT`, `CNT0`, `CNT1`, `CNT2`, and `CNT3`. The new compiler is incompatible with previous versions of `sysreg.h`.

Near and Far Type Qualifiers

The ADSP-219x processors can have external memory, which by default, for reasons of efficiency, be addressable in 16-bits from C/C++ source. The compiler provides an extension to support access to external memory, which allows using external memory in C applications without degrading performance when accessing internal memory. This extension is enabled using C type qualifiers, “`far`” and “`near`”.

Circular Buffer Built-in Functions

The C/C++ compiler provides built-in support for the ADSP-219x processor's circular buffer mechanisms. The compiler can automatically detect situations where circular buffers would be appropriate and generate code to use the buffers.

Circular buffers may also be specified explicitly using built-in functions. The builtins are:

- Circular buffer increment of an index (`__builtin_circindex()`)
- Circular buffer increment of a pointer (`__builtin_circptr()`)

ETSI Support

The ETSI support for ADSP-219x processors is a collection of functions that provide high-performance implementations for operations commonly required by DSP applications. The operations, provided by the ETSI library (`libetsi.dlb`) and built-in functions defined in `ETSI_fract_arith.h`, support fractional or fixed-point arithmetic.

The "*ETSI Support*" section of the *VisualDSP++ 3.5 C/C++ Compiler Manual for Blackfin Processors* provides descriptions of all 16- and 32-bit fractional ETSI routines.

Pragmas

The VisualDSP ++ 3.5 C/C++ compiler supports a number of new pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. The new and enhanced pragmas are described briefly in Table 3-9.

Table 3-9. ADSP-219x C/C++ Compiler Pragmas

Pragma	Function
`#pragma align` *`num`*	May be used before variable and field declarations. It applies to the variable or field declaration that immediately follows the pragma. Use of this pragma causes the compiler to generate the next variable or field declaration aligned on a boundary specified by *`num`*.
`#pragma pad (`*`alignopt`*`)`	May be applied to struct definitions. It applies to struct definitions that follow, until the default alignment is restored by omitting *`alignopt`*, for example, by `#pragma pad()` with empty parentheses.
`#pragma vector_for`	Notifies the optimizer that it is safe to execute two iterations of the loop in parallel. The pragma does not force the compiler to vectorize the loop; the optimizer checks various properties of the loop and does not vectorize it if it believes it is unsafe.
`#pragma loop_count(`*`min, max, modulo`*`)`	Asserts that the loop will iterate at least *`min`* times, no more than *`max`* times, and a multiple of *`modulo`* times. This information enables the optimizer to omit loop guards, to decide whether the loop is worth completely unrolling, and whether code need be generated for odd iterations.
`#pragma optimize_as_cmd_line`	Resets the optimization settings to be those specified on the `cc219x` command line when the compiler was invoked.

Table 3-9. ADSP-219x C/C++ Compiler Pragmas (Cont'd)

Pragma	Function
`#pragma no_alias`	Tells the compiler the following has no loads or stores that conflict due to references to the same location through different pointers, known as "aliases".
`#pragma alloc`	Tells the compiler that the function behaves like the library function "`malloc`", returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call.
`#pragma pure`	Tells the compiler that the function does not write to any global variables, and does not read or write any volatile variables.
`#pragma const`	This is a more restrictive form of the `pure` pragma. It tells the compiler that the function does not read from global variables as well as not writing to them or reading or writing volatile variables. The result of the function is therefore a function of its parameters.
`#pragma regs_clobbered` *`string`*	Used with a function declaration or definition to specify which registers are modified (or clobbered) by that function.
`#pragma result_alignment (`*`n`*`)`	Asserts that the pointer or integer returned by the function has a value that is a multiple of *`n`*.
`#pragma instantiate` *`instance`*	Requests the compiler to instantiate *`instance`* in the current compilation.
`#pragma do_not_instantiate` *`instance`*	Directs the compiler not to instantiate *`instance`* in the current compilation.
`#pragma can_instantiate` *`instance`*	Tells the compiler that if instance is required anywhere in the program, it should be instantiated in this compilation, and has the same effect as `#pragma instantiate`.

Table 3-9. ADSP-219x C/C++ Compiler Pragmas (Cont'd)

Pragma	Function
`#pragma hdrstop`	Used in conjunction with the `-pch` (precompiled header) switch. The switch tells the compiler to look for a precompiled header (`.pch` file), and, if it cannot find one, to generate a file for use on a later compilation.
`#pragma no_pch`	Overrides the `-pch` (precompiled headers) switch for a particular source file. It directs the compiler not to look for a `.pch` file and not to generate one for the specified source file.
`#pragma once`	Appears at the beginning of a header file and tells the compiler that the header is written in such a way that including it several times has the same effect as including it once.
`#pragma system_header`	Appears in a header file and identifies the file as one that is supplied with VisualDSP++.

GCC Compatibility Extensions

The compiler provides compatibility with the C dialect accepted by version 3.2 of the GNU C Compiler. Many of these extensions are available in the C99 ANSI Standard. A brief description of the following extensions is included in the compiler manual (see Chapter 1):

- Statement expressions
- Type reference support keyword (`Typeof`)
- GCC generalized `Lvalues`
- Conditional expressions with missing operands
- Hexadecimal floating-point numbers
- Arithmetic on pointers to `void` and pointers to functions

- Cast to `union`
- Ranges in `case` labels
- Declarations mixed with code
- Zero-length arrays
- Variable argument macros
- Line breaks in string literals
- Escape character constant
- Alignment inquiry keyword (`__alignof__`)
- Keyword for specifying names in generated assembler (`asm`)
- Function, variable and type attribute keyword (`__attribute__`)

Predefined Compiler Macro

The new macro `__VERSION__` defines a string constant, giving the version number of the compiler used to compile this module.

File IO Support

In earlier VisualDSP++ releases, the implementation of the functions defined in the header file `stdio.h` was based on a device driver, known as `primIO`, provided by the VisualDSP++ simulator and EZ-KIT Lites. This device driver, however, only provides access to the host file system.

VisualDSP++ 3.5 permits alternative device drivers to be registered, which can then be used through the normal `stdio` functions and, therefore, enable the routines to interact with a device other than the host file system. By default, the `stdio` functions will continue to use the device driver provided by the VisualDSP++ simulator and EZ-KIT Lites, and users should not notice any difference in functionality. Full details of the new extensible driver mechanism are available in the C/C++ compiler manual.

C Library Functions

The C run-time library has been extended with the addition of some new functions and enhanced functionality.

The formatted input/output functions defined in `stdio.h` (i.e. `printf`, `scanf`, `fprintf`, `...`) now support the *`%a`* conversion specifier. The *`%a`* specifier is similar both in form and meaning to the *`%e`* specifier, with the exception that the *`%e`* specifier is used to input and output decimal floating-point numbers, while the *`%a`* specifier is used to input and output hexadecimal floating-point numbers.

Additional functions defined in the `stdio.h` header file are supported in the VisualDSP++ 3.5 release. These functions are:
`fgetpos`, `fseek`, `fsetpos`, `ftell`, `remove`, `rename`, `rewind`

ⓘ The C standard `stdio.h` functions `tmpfile` and `tmpnam` are not supported in this release. The `isinf` and `isnan` functions existed in VisualDSP++ 3.1 but were not documented.

The complete list of supported library functions in the `stdio.h` header file is as follows.

`clearerr`	`fclose`	`feof`
`ferror`	`fflush`	`fgetc`
`fgetpos`	`fgets`	`fprintf`
`fputc`	`fputs`	`fopen`
`fread`	`freopen`	`fscanf`
`fseek`	`fsetpos`	`ftell`
`fwrite`	`getc`	`getchar`
`gets`	`perror`	`putc`
`putchar`	`puts`	`printf`
`remove`	`rename`	`rewind`
`scanf`	`setbuf`	`setvbuf`
`sprintf`	`sscanf`	`ungetc`
`vfprintf`	`vprintf`	`vsprintf`

The `atof`, `strtod`, and `strtodf` functions have been modified to support hexadecimal floating-point numbers. The function documentation has been updated to reflect the new functionality.

DSP Run-Time Library Functions

Some new functions have been added to the DSP run-time library; these functions are identified as follows.

Library Function	Description
`alog`	calculates the natural (base e) anti-log of its argument
`alog10`	calculates the base 10 anti-log of its argument
`cartesian`	transforms a complex number from Cartesian notation to polar notation

The new functions are fully documented in the compiler manual.

Compiler and Library for ADSP-218x DSPs

For ADSP-218x DSPs, the most notable new compiler's features and enhancements of the C compiler are in the following areas:

For information on these features, refer to the *VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x DSPs* and online Help.

Input and Output File Extensions

The compiler supports new file extensions.

File Extension	Description
`.ipa, .opa`	IPA files used internally by the `.pch` header file
`.pch`	Precompiled header files
`.xml`	Processor memory map file output

C Compiler Command-Line Switches

This section summarizes C/C++ compiler command-line switches introduced or enhanced in VisualDSP++ 3.5. Table 3-10 lists and briefly describes each switch.

Table 3-10. C Compiler Common Switches

Switch Name	Description
`-bss`	Causes the compiler to put global zero-initialized data into a separate BSS-style section.
`-const-read-write`	Specifies that `const` data accessed via a pointer may be modified elsewhere.
`-ED`	Preprocesses and sends all output to a file.
`-flags-meminit`	Passes each comma-separated option to the Memory Initializer utility.
`-fp-associative`	Treats floating-point multiplication and addition as associative.
`-full-version`	Displays version information for build tools.
`-i`	Outputs only header details or makefile dependencies for `include` files specified in double quotes.
`-ipa`	Specifies that interprocedural analysis should be performed for optimization between translation units.
`-jump-{pm|dm|same}`	Specifies that the compiler should place jump tables in data memory (`-jump-dm`), program memory (`-jump-pm`) or the same memory section as the function to which it applies (`-jump-same`).
`-MD`	Generates `make` rule, compiles, and prints to a file.
`-Mo` *`filename`*	Writes dependency information to *`filename`*. This switch is used in conjunction with the `-ED` or `-MD` options.
`-mem`	Causes the compiler to invoke the Memory Initializer after linking the executable file.
`-no-bss`	Causes the compiler to group global zero-initialized data into the same section as global data with non-zero initializers.

Table 3-10. C Compiler Common Switches (Cont'd)

Switch Name	Description
`-no-fp-associative`	Does not treat floating-point multiply and addition as an associative.
`-no-mem`	Causes the compiler not to invoke the Memory Initializer after linking; set by default.
`-Oa`	Enables automatic function inlining.
`-Ov` *`num`*	Controls speed vs. size optimizations.
`-P`	Preprocesses, but does not compile, the source file. Omits line numbers in the preprocessor output.
`-PP`	Similar to `-P`, but does not halt compilation after preprocessing.
`-path-`*`tool`*	Enhanced. Uses the specified directory as the location of the specified compilation tool.
`-signed-bitfield`	Makes the default type for `int bitfields` signed.
`-si-revision` *`version`*	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision.
`-unsigned-bitfield`	Makes the default type for plain int bitfields unsigned.

Optimization Control

The following list identifies several new optimization levels. Refer to Chapter 2, *Achieving Optimal Performance from C Source Code*, of the compiler manual for detailed information on how to obtain maximal code performance from the compiler. The new and enhanced optimization features are:

- Automatic Inlining
  The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code. The compiler determines when the inlining will reduce execution time. How

aggressively the compiler performs automatic inlining is controlled using the `-Ov` switch. Automatic inlining is enabled using the `-Oa` switch and additionally enables procedural optimizations (`-O`).

- Interprocedural Optimizations
  The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. IPA is enabled using the `-ipa` switch and additionally enables Procedural Optimizations (`-O`).

Assembly Construct Operands

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. To know how to assign registers to operands, the compiler needs information on operand constraints. A more detailed description of how to convey this information to the compiler can be found in the compiler manual.

New asm constraints for MAC and SHIFTER long result targets
Two new `asm()` statement constraints, “`M`” and “`S`”, have been added to allow users to map `MAC` and `SHIFTER` outputs to C/C++ long (double-word) variables. Example usage:

```
static long f1(int a, int b) {
      long ret;
      asm volatile("%0 = %1 * %2 (SS);"
           :"=M"(ret)
           : "b"(a),"B"(b) : );
      return ret;
}

static long f2(int a, int b) {
      long ret;
      asm volatile("SE=%1; %0 = LSHIFT %2 (lo);"
           :"=S"(ret)
           : "e"(b), "d"(a) : "SE" );
      return ret;
}
```

Hard-register support in asm statements

It is now possible to claim registers directly for use as `asm()` constraints, instead of requesting a register from a certain class using the constraint letters. This is done by simply naming the register in the location where the class letter would previously have been given. For example,

```
    asm("%0 = %1 + %2;"
               :"=ar"(sum)       /* output */
               :"g"(x), "G"(y)  /* input */
       );
```

loads `x` into ALU-X register, `y` into ALU-Y register, and calculated `sum` in register `AR`.

Expanded list of asm constraints: *+symbol*, *?symbol*, and *#symbol*

The `+`*`symbol`* operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C/C++ expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C/C++ expression. Therefore, as with pure outputs, the C/C++ expression must be one that is valid on the left-hand side of an assignment.

The `?`*`symbol`* operand is temporary. It must appear as part of the third argument to the `asm()` construct. A register is allocated as working space for the duration of the `asm()` template execution. The register's initial value is undefined, and the register's final value is discarded. The corresponding C/C++ expression is not loaded into the register but must be present. This expression is normally specified using a literal zero.

The `#`*`symbol`* operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. The operand must appear as part of the second argument to the `asm()` construct.

Builtins for Non-literal Address Inputs

The `o_space_read` and `io_space_write` `sysreg.h` builtins now support non-literal address inputs.

The instruction set limits the IO memory load and store instructions to using literal addresses only. Previously, the compiler-supported builtins, which used the `io_space_read` and `io_space_write` instructions, asserted or caused an assembler error when non-literal addresses or an out-of-range literal were used. The VisualDSP++ 3.5 compiler has been enhanced to call a support subroutine defined in the C run-time library to implement the unsupported cases by generating the IO access instruction using values loaded programatically. The support will not be most efficient and `io_space_read` and `io_space_write` address parameters should be passed as literals when this is possible.

ETSI Support

The ETSI support for ADSP-218x processors is a collection of functions that provides high-performance implementations for operations commonly required by DSP applications. These operations, provided by the ETSI library `libetsi.dlb` and built-in functions defined in `ETSI_fract_arith.h`, support fractional or fixed-point arithmetic.

The ADSP-218x compiler manual provides descriptions of all 16- and 32-bit fractional ETSI routines.

Pragmas

The ADSP-218x C compiler supports a number of new pragmas. Pragmas are implementation-specific directives that modify the compiler's behavior. The new and enhanced pragmas are described in Table 3-11.

Table 3-11. ADSP-218x Compiler Pragmas

Pragma	Function
`#pragma aligned` *`num`*	May be used before variable and field declarations. It applies to the variable or field declaration that immediately follows the pragma. Use of this pragma causes the compiler to generate the next variable or field declaration aligned on a boundary specified by *`num`*.
`#pragma pad (`*`alignopt`*`)`	May be applied to `struct` definitions. It applies to `struct` definitions that follow, until the default alignment is restored by omitting *`alignopt`*, for example, by `#pragma pad()` with empty parentheses.
`#pragma vector_for`	Notifies the optimizer that it is safe to execute two iterations of the loop in parallel. The pragma does not force the compiler to vectorize the loop; the optimizer checks various properties of the loop and does not vectorize it if it believes it is unsafe.
`#pragma loop_count(`*`min, max, modulo`*`)`	Asserts that the loop will iterate at least *`min`* times, no more than *`max`* times, and a multiple of *`modulo`* times. This information enables the optimizer to omit loop guards, to decide whether the loop is worth completely unrolling, and whether code need be generated for odd iterations.
`#pragma optimize_as_cmd_line`	Resets the optimization settings to be those specified on the `cc218x` compiler's command line when the compiler was invoked.
`#pragma no_alias`	Tells the compiler the following has no loads or stores that conflict due to references to the same location through different pointers, known as "aliases".

Table 3-11. ADSP-218x Compiler Pragmas (Cont'd)

Pragma	Function
`#pragma alloc`	Tells the compiler that the function behaves like the library function "`malloc`", returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call.
`#pragma pure`	Tells the compiler that the function does not write to any global variables and does not read or write any volatile variables.
`#pragma const`	This is a more restrictive form of the `pure` pragma. It tells the compiler that the function does not read from global variables as well as not writing to them or reading or writing volatile variables. The result of the function is therefore a function of its parameters.
`#pragma regs_clobbered` *`string`*	Used with a function declaration or definition to specify which registers are modified (or clobbered) by that function.
`#pragma result_alignment (`*`n`*`)`	Asserts that the pointer or integer returned by the function has a value that is a multiple of *`n`*.
`#pragma hdrstop`	Used in conjunction with the `-pch` (precompiled header) switch. The switch tells the compiler to look for a precompiled header (`.pch` file) and, if it cannot find one, to generate a file for use on a later compilation.
`#pragma no_pch`	Overrides the `-pch` (precompiled headers) switch for a particular source file. It directs the compiler not to look for a `.pch` file and to not generate one for the specified source file.
`#pragma once`	Appears at the beginning of a header file and tells the compiler that the header is written in such a way that including it several times has the same effect as including it once.
`#pragma system_header`	Appears in a header file and identifies the file as one that is supplied with VisualDSP++.

GCC Compatibility Extensions

The compiler provides compatibility with the C dialect accepted by version 3.2 of the GNU C Compiler. Many of these extensions are available in the C99 ANSI Standard. A brief description of each extension is included in the compiler manual (see Chapter 1). The following topics are covered in the manual.

- Statement expressions
- Type reference support keyword (`Typeof`)
- GCC generalized `Lvalues`
- Conditional expressions with missing operands
- Hexadecimal floating-point numbers
- Arithmetic on pointers to `void` and pointers to functions
- Cast to `union`
- Ranges in `case` labels
- Zero length arrays
- Variable argument macros
- Line breaks in string literals
- Escape character constant
- Alignment inquiry keyword (`__alignof__`)
- Keyword for specifying names in generated assembler (`asm`)
- Function, variable and type attribute keyword (`__attribute__`)

Predefined Compiler Macro

The new macro `__VERSION__` defines a string constant, giving the version number of the compiler used to compile this module.

File IO Support

In earlier VisualDSP++ releases, the implementation of the functions defined in the header file `stdio.h` was based on a device driver, known as `primIO`, provided by the VisualDSP++ simulator and EZ-KIT Lites. This device driver, however, only provides access to the host file system.

VisualDSP++ 3.5 permits alternative device drivers to be registered, which can then be used through the normal `stdio` functions and, therefore, enable the routines to interact with a device other than the host file system. By default, the `stdio` functions will continue to use the device driver provided by the VisualDSP++ simulator and EZ-KIT Lites, and users should not notice any difference in functionality. Full details of the new extensible driver mechanism are available in the C/C++ compiler manual.

C Library Functions

The C run-time library has been extended with the addition of some new functions and enhanced functionality.

The formatted input/output functions defined in `stdio.h` (i.e. `printf`, `scanf`, `fprintf`, `...`) now support the *`%a`* conversion specifier. The *`%a`* specifier is similar both in form and meaning to the *`%e`* specifier, with the exception that the *`%e`* specifier is used to input and output decimal floating-point numbers, while the *`%a`* specifier is used to input and output hexadecimal floating-point numbers.

Additional functions defined in the `stdio.h` header file are supported in the VisualDSP++ 3.5 release. These functions are:
`fgetpos`, `fseek`, `fsetpos`, `ftell`, `remove`, `rename`, `rewind`

ⓘ The C standard `stdio.h` functions `tmpfile` and `tmpnam` are not supported in this release. The `isinf` and `isnan` functions existed in VisualDSP++ 3.1 but were not documented.

The complete list of supported library functions in the `stdio.h` header file is as follows.

`clearerr`	`fclose`	`feof`
`ferror`	`fflush`	`fgetc`
`fgetpos`	`fgets`	`fprintf`
`fputc`	`fputs`	`fopen`
`fread`	`freopen`	`fscanf`
`fseek`	`fsetpos`	`ftell`
`fwrite`	`getc`	`getchar`
`gets`	`perror`	`putc`
`putchar`	`puts`	`printf`
`remove`	`rename`	`rewind`
`scanf`	`setbuf`	`setvbuf`
`sprintf`	`sscanf`	`ungetc`
`vfprintf`	`vprintf`	`vsprintf`

The functions `atof`, `strtod`, and `strtodf` have been enhanced to support hexadecimal floating-point numbers. The description of these functions in the manual has been updated to reflect this new functionality.

Linker and Utilities

The VisualDSP++ 3.5 linker and utility programs are upgraded to operate more efficiently on 16-bit fixed-point Blackfin and ADSP-21xx processors.

For the linker and utilities, the most notable new features and enhancements are:

- "Modified Link Page in Project Options Dialog Box" on page 3-53
- "Migrating LDFs from Previous Installations" on page 3-55
- "Linker Command-Line Switches" on page 3-56
- "Updated List of LDF Keywords" on page 3-57
- "Modifications to LDF Commands" on page 3-58
- "Breakpoints on Overlays" on page 3-60
- "Expert Linker" on page 3-61
- "Memory Map File (.XML)" on page 3-62
- "Archiver" on page 3-63

For more information, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* and online Help.

Modified Link Page in Project Options Dialog Box

Within VisualDSP++, the **Link** page of the **Project Options** dialog box is modified to have more flexibility in specifying tool settings for project builds. Choosing a **Category** from the pull-down list at the top of the **Link** page presents several different pages of options.

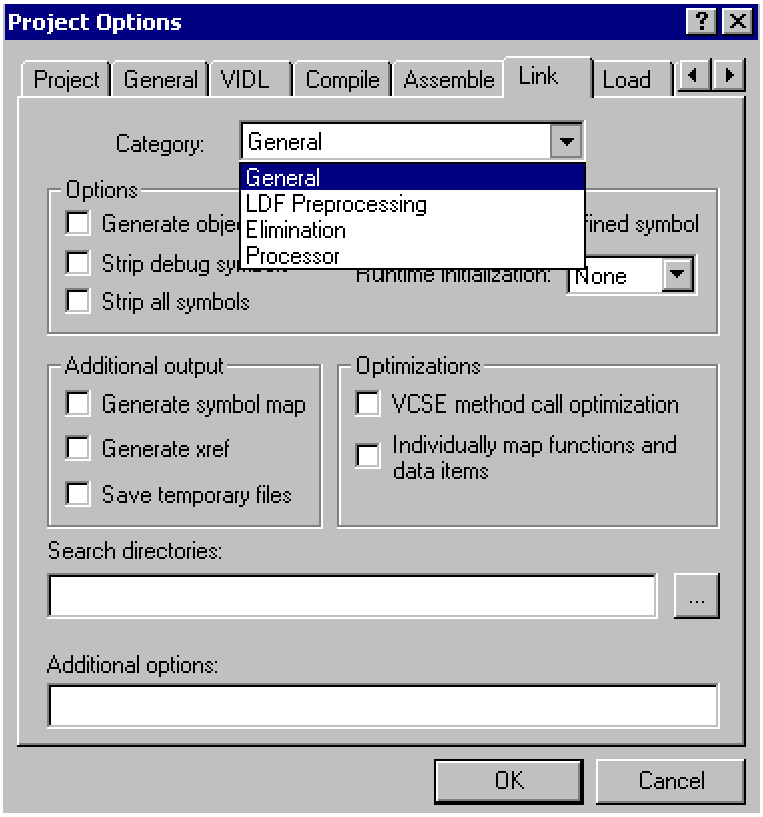

Figure 3-1. Main Link Tab with Category Selections

There are four sub-pages you can access—**General**, **LDF Preprocessing**, **Elimination**, and **Processor**. Almost every setting option has a corresponding compiler command-line switch. For more information, refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* and online Help.

Migrating LDFs from Previous Installations

Migrating LDFs from previous VisualDSP++ installations includes linking CPLB configuration tables directly, adding guard symbols, and supporting run-time initialization.

For example, an `.LDF` file of VisualDSP++ 3.1 has the following code to support the run-time initialization.

```
bsz
{
INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS($OBJECTS(bsz) $LIBRARIES(bsz))
} >MEM_DATA

bsz_init
{
INPUT_SECTION_ALIGN(4)
  INPUT_SECTIONS($OBJECTS(bsz_init) $LIBRARIES(bsz_init))
} >MEM_DATA2
```

In VisualDSP++ 3.5, this needs to be replaced by the following.

```
bsz ZERO_INIT
{
INPUT_SECTION_ALIGN(4)
    INPUT_SECTIONS($OBJECTS(bsz) $LIBRARIES(bsz))
} >MEM_L1_DATA_A

bsz_init
{
INPUT_SECTION_ALIGN(4)
  INPUT_SECTIONS($OBJECTS(bsz_init) $LIBRARIES(bsz_init))
} >MEM_L1_DATA_A
.meminit {} >MEM_L1_DATA_A
```

The key points to note are:

- The output section "`bsz`" now has the `ZERO_INIT` flag specified.
- The special output section named "`.meminit`" is included.
- The size reduction, which was used only when the `-mem-bsz` switch was set, is now always done.
- The initialization of a section flagged with `ZERO_INIT` is done by the appropriate utility (the IDDE, loader, etc.) instead of at run-time.

Refer to "SECTIONS{} LDF Command" on page 3-59 for more detail on the changed and new functionality for `ZERO_INIT` and `RUNTIME_INIT`.

Linker Command-Line Switches

Table 3-12 lists the new or modified linker command-line switches.

Table 3-12. New Linker Command-Line Switches

Switch	Description
`-D`*`processorID`*	Modified syntax and description. Specifies the target processor ID. The use of the `-proc` *`processor`* switch is recommended.
`-Ovcse`	Enables VCSE method call optimization.
`-Wwarn` *`number`*	Demotes the specified error message to a warning.
`-flags-meminit`	Passes each comma-separated option to the Memory Initializer utility.
`-flags-pp`	Passes each comma-separated option to the preprocessor.
`-ip`	Modified syntax and description. Fills fragmented memory with individual data objects that fit and requires that objects have been assembled with the assembler's `-ip` switch. Note: ADSP-21xx DSPs only.

Table 3-12. New Linker Command-Line Switches (Cont'd)

Switch	Description
`-jcs21`	Modified syntax and description. Converts out-of-range short calls and jumps to the longer form. Note: Blackfin processors and ADSP-219x DSPs only.
`-jcs21+`	Modified syntax and description. Enables `-jcs21` and allows the linker to convert out-of-range branches to indirect calls and jumps sequences Note: Blackfin processors only.
`-meminit`	Directs the linker to post-process the `.DXE` file through the Memory Initializer utility. This will cause the sections specified in the `.LDF` file to be "run-time" initialized by the C run-time library. By default, if this flag is not specified, all sections are initialized at "load" time (for example, via the VisualDSP++ IDDE or the boot loader).
`-si-revision` *`version`*	Specifies a silicon revision of the specified processor.
`-v|-verbose`	Verbose—outputs status information.

Jump/Call Expansions

The mechanism, which converts out-of-range short calls and jumps to the longer or indirect form on Blackfin and ADSP-219x processors has been modified for VisualDSP++ 3.5.

Refer to the Linker manual for more information about jump and call expansions. Refer to the *ADSP-BF53x Instruction Set Reference* for more information about jump and call instructions.

Updated List of LDF Keywords

Table 3-13 lists `.LDF` file keywords that apply to all 16-bit processors (Blackfin, ADSP-218x, and ADSP-219x).

Table 3-13. LDF File Keywords Summary

ABSOLUTE	ADDR	ALGORITHM
ALIGN	ALL_FIT	ARCHITECTURE
BEST_FIT	BM[1]	BOOT
DEFINED	DM[2]	ELIMINATE
ELIMINATE_SECTIONS	END	FALSE
FILL	FIRST_FIT	INCLUDE
INPUT_SECTION_ALIGN	INPUT_SECTIONS	KEEP
LENGTH	LINK_AGAINST	MAP
MEMORY	MEMORY_SIZEOF	MPMEMORY
NUMBER_OF_OVERLAYS	OUTPUT	OVERLAY_GROUP
OVERLAY_ID	OVERLAY_INPUT	OVERLAY_OUTPUT
PACKING	PAGE_INPUT[2]	PAGE_OUTPUT[2]
PLIT	PLIT_SYMBOL_ADDRESS	
PLIT_SYMBOL_OVERLAYID	PM[2]	PROCESSOR
RAM	RESOLVE	RESOLVE_LOCALLY
ROM	SEARCH_DIR	SECTIONS
SHARED_MEMORY	SHT_NOBITS	SIZE
SIZEOF	START	TYPE
VERBOSE	WIDTH	XREF

1 Supported on ADSP-218x DSPs only.
2 These keywords apply only to ADSP-218x/9x LDFs.

Modifications to LDF Commands

Some LDF commands are enhanced to support better linking and memory management in VisualDSP++ 3.5.

SECTIONS{} LDF Command

The `SECTIONS{}` command uses memory segments (defined by `MEMORY{}` commands) to specify the placement of output sections in memory.

The `section_declaration` of the `SECTIONS{}` command is enhanced to use a special section name `.MEMINIT` that indicates where to place the “run-time” initialization structures to be used by the C run-time library. The linker will “place” this section into the largest available unused memory at the specified memory segment. The Memory Initializer post-processor will fill this space with the data needed by the C run-time library for run-time initialization. The `.MEMINIT` section should be placed in non-overlay memory.

The *`init_qualifier`* specifies run-time initialization type (optional). The qualifiers are:

- `NO_INIT` – The section type contains un-initialized data. There is no data stored in the `.DXE` file for this section (equivalent to the `SHT_NOBITS` legacy qualifier).
- `ZERO_INIT` – The section type contains only “zero-initialized” data. If invoked with the `-meminit` switch, the “zeroing” of the section is done at runtime by the C run-time library. If `-meminit` is not specified, the “zeroing” is done at “load” time.
- `RUNTIME_INIT` – If the linker is invoked with the `-meminit` switch, this section will be filled at runtime. If `-meminit` is not specified, the section will be filled at “load” time.

Refer to the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* for more information.

OVERLAY_GROUP{} Command

The `OVERLAY_GROUP{}` LDF command is deprecated and is not recommended for use. Though in VisualDSP++ 3.5, the `OVERLAY_GROUP{}`

can still be used to group overlays, it is recommend that you revise any LDF files which include the command. The linker of the current release processes all overlay groups and explicit `OVERLAY_GROUP{}` commands, producing a warning. The preferable way to manage overlays is to create a separate output section for each overlay group.

Breakpoints on Overlays

You may require an overlay manager to perform other specialized tasks to satisfy the special needs of a given application. Overlay managers for Blackfin processors must be developed by the user.

One overlay enhancement is how the overlay manager handles breakpoints on overlays. The debugger relies on the presence of the `__ov_start` and`__ov_end` symbols to support breakpoints on overlays.The symbol manager will set a silent breakpoint at each symbol.

The more important of the two symbols is the breakpoint at `_ov_end`. Code execution in the overlay manager should pass through this location once an overlay has been fully swapped in. At this point, the debugger may probe the target to determine which overlays are in context. The symbol manager will now set any breakpoints requested on the overlays and resume execution.

The second breakpoint is at `_ov_start`. The label `_ov_start` should be defined in the overlay manager, in code always executed immediately before the transfer of a new overlay begins. The breakpoint disables all of the overlays in the debugger—while the target is running in the overlay manager, the target is “unstable” in the sense that the debugger should not rely on the overlay information it may gather since the target is “in flux”. The debugger will still function without this breakpoint, but there may be some inconsistencies while overlays are being moved in and out.

Expert Linker

The Expert Linker provides a GUI-based means of supplying the link commands currently supplied to the linker in a Linker Description File (`.LDF`). The tool also provides graphical and hypertext representations of text output for cross-reference, the linker map, and `ELFDUMP`. For more information on the Expert Linker, refer to the corresponding chapter of the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors* and online Help.

Menu Updates

The VisualDSP++ 3.5 Expert Linker GUI and context menus are enhanced to better process graphical data. The most significant menu changes are as follows.

In the Input Sections context menu, new selections are:

- Remove – Removes an LDF macro from another LDF macro but does not delete the input section mappings that contain the removed macro. The difference between Delete and Remove is that Delete deletes the input section macros that contain the deleted macro. The Remove option becomes available only if you right-click on an LDF macro that is part of another LDF macro.
- Expand All LDF Macros – Expands all the LDF macros in the input sections pane to display the contents of all the LDF macros.

In the Memory Map context menu, a new selection is:

- Expand All – Expands all items in the memory map tree so that their contents are visible.

Profiling Object Sections

The Expert Linker is enhanced to give you the option of profiling object sections. If this feature is enabled, the Expert Linker uses the profiler to collect profiling information while your program is running. When the program halts, the Expert Linker graphically displays how much time was spent in each object section so that you can see "hotspots" in the code and move that code to faster internal memory.

To profile a program with the Expert Linker:

7. Enable profiling in the Global Properties dialog box.
8. Load the program into the current project. After the program is loaded, the Expert Linker sets up the profiling bins to collect the profiling information.
9. Run the program. When the program halts, the Expert Linker will color each object section with a different shade of red to indicate how much time was spent executing that section.

From the Expert Linker, you can view PC sample counts for object sections. To view an actual PC sample count, move the mouse pointer over an object section and view the PC sample count. To view sample counts for functions located within an object section, double-click on the object section. You can view detailed profile information, such as the sample counts, for each line in the function.

Memory Map File (.XML)

The linker can output memory map files that contain memory and symbol information for your executable file(s). The map contains a summary of memory defined with `MEMORY{}` commands in the `.LDF` file and provides a list of the absolute addresses of all symbols. For VisualDSP++ 3.5, the file format has been changed from plain text to XML, and the extension has been changed from `.MAP` to `.XML`.

Archiver

For the VisualDSP++ 3.5 release, the archiver (`elfar`) provides several improvements. For more information about the archiver, refer to "*Archiver*" chapter in the *VisualDSP++ 3.5 Linker and Utilities Manual for 16-Bit Processors*.

Archiver Switches

Table 3-14 summarizes new archiver command-line switches.

Table 3-14. New Archiver Command-Line Switches

Item	Description
`-anv`	Appends one or more object files and clears version information
`-dnv`	Removes the listed object file(s) from the specified library file and clears version information
`-pv`	Prints only version information in library to standard output
`-pva`	Prints all version information in library to standard output
`-t` *`verno`*	Tags the library with version information in string
`-tx` *`filename`*	Tags the library with version information in the file
`-twc` *`ver`*	Tags the library with version information in the *`num.num.num`* form
`-tnv`	Clears version information from a library
`-version`	Prints the archiver (`elfar`) version to standard output
`-w`	Removes all archiver-generated warnings
`-Wnnnn`	Selectively disables warnings specified by one or more message numbers. For example, `-W0023` disables warning message `ar0023`.

Warnings for Duplicate Library Entries

The archiver will warn if two objects with the same name are put into a library. In VisualDSP++ 3.5, the archiver produces the warning:

```
[Warning ea0079] An object file named "<fname>" already exists in
this library
```

Improved Support for File Specifications

In VisualDSP++ 3.5, the archiver accepts command lines with wildcard specification of the files for inclusion:

```
elfar -c mylib.dlb *.doj
```

The archiver now accepts UNC filename specification:

```
elfar -r fruit.dlb \\c\tests\strawberry.doj
```

or

```
elfar -r fruit.dlb //c/tests/strawberry.doj
```

Tagging an Archive with Version Information

The archiver supports embedding version information into a library built with `elfar`. The following is a list of version information tagging features provided by the archiver.

Basic Version Information

You can “tag” an archive with a version. The easiest way to tag an archive is using the `-t` switch. The `-t` switch can be used in addition to any other `elfar` switch. To highlight the version information, precede it with “`::`”.

User-Defined Version Information

Any number of user-defined version values can be provided by supplying a text with those values. Each line in the text file begins with a name, followed by a space, and then the value associated with that name.

Printing Version Information

Use the `-p` switch to print version information. The `-pv` switch prints only version information and does not print the contents of the archive. The `-pva` switch prints all version information. Version names without values are not be printed with `-p` or `-pv` but are shown with `-pva`.

Removing Version Information from an Archive

Adding "`nv`" to a switch strips version information. In addition, a special form of the `-t` switch, which takes no argument, can be used for stripping version information from an archive.

Checking Version Number

The `-twc` switch causes the archiver to raise a warning if the version number is not provided. The check ensures that the version number starts with a number in that format.

Adding Text to Version Information

You can add additional text to the end of the version information.

Loaders

The loader program (`elfloader.exe`) for Blackfin and ADSP-219x processors has been modified as follows.

Blackfin Loader Features

The Blackfin loader modifications are:

- The loader has been updated to support ADSP-BF561 processors. Refer to the *VisualDSP++ 3.5 Loader Manual for 16-Bit Processors* for details.
- The `-si-revision` switch has been added to provide a silicon revision number to the loader.
- The `-init` *`filename`* switch has been added to provide an initialization file name to the loader. This switch is for ADSP-BF531/BF532/BF533 and ADSP-BF561 processors only.
- The `-GHK #` switch has been added to provide a 4-bit global header cookie value. This switch is for the processors which have a global header with their loader files.
- The SPI boot mode now supports Intel HEX format.

ADSP-219x Loader Features

In VisualDSP++ 3.5, the ADSP-219x loader supports multiple `.DXE` booting in parallel EPROM boot mode. New switches, such as `-pd` *`addr inputfile`*, allow you to append another application program at the specified address.

For details, see the *VisualDSP++ 3.5 Loader Manual for 16-Bit Processors*.

VCSE

VCSE is a combination of tools and guidelines that simplify the process of developing reusable components and help to document and validate such components. These tools and guidelines:

- Enable applications to incorporate and use software algorithm components from other developers easily and with confidence
- Ensure that components from multiple vendors do not interact with each other in unpredictable ways or have resource clashes
- Allow components to be developed in assembly, C, or C++ and be used from applications developed in any of these languages
- Allow components to be reused easily
- Allow comparison of algorithms that offer the same functionality
- Encourage third party developers to provide the implementation of algorithms as easily used components
- Automatically generate a set of HTML pages that document a component and the interfaces it provides or requires. The generated documentation include a table of contents and an index.

VCSE supports an Interface Definition Language (IDL) and a VIDL compiler that enable developers to specify and then create and use components without having to become familiar with the detail of the model and its mechanisms.

The VIDL compiler can automatically generate a test shell component from the VIDL definition of a component. The generated test can include code to validate the argument values passed to and from the component, carry out array bound checking of arguments, ensure that methods of an interface are called in the correct sequence, and measure the resources used by a component.The level of checking effected by the test shell can be controlled by the developer. The generated test shells can be used by the developer of a component while testing his component but can also be used by the user of a component to check that he is using a component in a valid way.

VCSE Peripheral Control Components

A collection of VCSE components for Blackfin processors is available though the Tools -> VCSE -> Manage components for download using the VisualDSP++ VCSE component manager. The available components are:

- Components that provide support for DMA, the SPORT, SPI and the codecs on the Blackfin EZ-KIT Lite evaluation systems and example talk-through applications using the components either stand alone or with VDK.
- Components that allow a host application to communicate with a DSP application over the background telemetry channel, and support to allow standard file system input/output to be effected over BTC. File input/output over BTC is much faster than standard file input/output.
- Components that provide a TCP/IP stack which provide the capability of supporting different network interfaces.

VDK

The following VisualDSP++ Kernel (VDK) features have been added to the VisualDSP++ 3.5 release.

- Support for configuring and using multiple heaps.
- Ability to specify, at build time, the heap from which each object (such as semaphores, device flags, etc.) is allocated.
- Ability to specify, at build time, the heap from which the stack and thread structure for each thread type are allocated.
- Ability to specify the size of the stack and the heap for the idle thread.
- Support for inter-processor messaging, which uses the same messaging API as intra-processor messaging.
- Out-of-the-box support for transporting messages between the two cores of the ADSP-BF561 processor is provided using DMA.
- Support for marshalling the payloads of inter-processor messages which have been allocated from heaps and memory pools.
- Support for user-defined marshalling of message payloads.
- Routing between processors can be configured in the Kernel tab to allow messages to be passed between processors that are not directly connected.
- Support for importing projects that will be used on other processors in order to simplify the configuring of multi-processor messaging.

Object Protection

The VisualDSP++ 3.5 tool chain now supports the encryption and decryption of object files. This feature enables algorithm providers to distribute objects and libraries under license protection.

Documentation Changes

This section describes the changes to online Help and online manuals.

Compiler Manuals

In VisualDSP++ 3.5, each compiler manual has a new chapter, called "*Achieving Optimal Performance from C/C++ Source Code*". The text focuses on how to fine-tune your program code to obtain maximal code performance from the compiler while optimizing for minimum code size.

The chapter starts with discussing some general optimization principles and explains how the compiler can lend the most help to your optimization effort. Optimal coding styles are described in detail. Special features, such as compiler switches, built-in functions, and pragmas are also discussed. The chapter ends with a short example to demonstrate how the optimizer works.

The new chapter (Chapter 2) is available in the following manuals:

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors

VisualDSP++ 3.5 C/C++ Compiler and Library Manual for ADSP-219x DSPs

VisualDSP++ 3.5 C Compiler and Library Manual for ADSP-218x DSPs

In addition to the new chapter, each compiler manual in the VisualDSP++ 3.5 documentation set has been enhanced with the addition of the new or existing run-time libraries, header files, and function descriptions. Notably, the `stdio.h` header file's description is updated to provide additional information about alternative device drivers, software restrictions, and IO support using `_primIO()`. Other examples are: `iso646.h`, `def218x.h`, `locale.h`, `setjump.h`, `stdarg.h`, `stdlib.h`, `fgetpos()`, `fseek()`, `fset-pos()`, `ftell()`, `remove()`, `rename()`, `rewind()`, `isinf()`, `isnan()`, and `cartesian_fr16()`.

Many library function descriptions have been updated to reflect the changed functionality. Some examples are:
`atof()`, `cos()`, `io_space_read()`, `io_space_write ()`, `setjmp()`, `sin()`, `strtod()`, `strtodf()`, `sysreg_read()`, `sysreg_write()`, `va_arg()`, `va_start()`, `arg()`, `histogram()`, and `polar()`.

VisualDSP++ 3.5 User's Guide

The legacy Tcl documentation is removed from the User's Guide but made available through the technical support as requested.

Online Help

An ongoing effort to enhance the VisualDSP++ online Help yields a more detailed error message presentation, a unified index for all online manuals in the documentation set, and the ability to search for keywords across the documentation set.

Error Messages

To view an explanation of the error message, select the six-character error identifier (for example, `cc0251`) in the **Output** window's **Build** page with the mouse or keyboard. Press the F1 key. Error message details appear in the Help window.

Merged Index

Index entries from all the VisualDSP++ 3.5 manuals have been merged into one unified index, which you can access by clicking the Index tab in the online Help. You can use the Search function in the online Help to search for keywords and instructions across all VisualDSP++ 3.5 publications for the target processor family. The Help window lists each occurrence and its location (manual).

Online Manuals

All manuals in the VisualDSP++ 3.5 documentation set have been converted to HTML Help and merged to facilitate searches in the online Help. Each book is still available in PDF format.

The Docs directory of your VisualDSP++ 3.5 installation CD contains a complete set of documentation for processors that are supported by your development tools suite. The set is comprised of VisualDSP++ tools and EZ-KIT Lite manuals, hardware manuals, and data sheets placed in the appropriate folders:

- Datasheets directory contains one `.PDF` file for each data sheet.
- Tools Manuals and EZ-KIT Lite Manuals directories contains one `.PDF` file for each manual.
- Hardware Manuals directory contains one `.PDF` file for each book or for each chapter in the manual.

4 OBSOLETE OR REMOVED FEATURES

This chapter describes the features that have been deprecated or removed since VisualDSP++ 3.0 (VisualDSP++ 3.1 for Blackfin processors). Read this chapter if upgrading from the previous software release.

Existing project files (`.DPJ`) can be imported into the new release. However, once the project file is imported, you are not able to bring the project back into VisualDSP++ 3.0. Similarly, new projects created using VisualDSP++ 3.5 cannot be used by earlier versions of the tools.

This chapter contains listings of obsolete and/or removed features:

You may want to consult the cover letter that accompanies the product installation CD for the last-minute information concerning this release.

Assembler and Preprocessor for Blackfin Processors

For VisualDSP++ 3.5 assembler/preprocessor for Blackfin processors, the deprecated and/or removed features are grouped as follows.

- "Directives and Keywords" on page 4-2
- "Assembly Operator" on page 4-2
- "Feature Macros" on page 4-3
- "Preprocessor Command-Line Switch" on page 4-3

Directives and Keywords

The following assembly directives and keywords have been removed or deprecated from VisualDSP++ 3.5.

Table 3-1. Obsolete/Deprecated Assembly Directives

Directive	Description
`.ASCII`	Deprecated; replaced with the `.BYTE` directive
`.ORG`	Removed

Assembly Operator

Operator	Description
`pageof()`	Removed

Feature Macros

In both the assembler and preprocessor, the following feature macros are removed.

Macro Definition	Description
`-D__ADSPDM102__=1`	In assembler. Removed. ADSP-DM102 DSP is not supported.
`__ADSPDM102__`	In preprocessor. Removed. ADSP-DM102 DSP is not supported.

Preprocessor Command-Line Switch

Switch	Description
`-cpredef`	Deprecated; replaced with the `-cstring` switch

Assembler and Preprocessor for ADSP-21xx DSPs

For VisualDSP++ 3.5 assembler/preprocessor for ADSP-21xx DSPs, the deprecated and/or removed features are grouped as follows.

- "Assembly Input Section Names" on page 4-4
- "Directives and Keywords" on page 4-4
- "Assembler Command-Line Switches" on page 4-5
- "Preprocessor Command-Line Switch" on page 4-5

Assembly Input Section Names

Input section names that you can use in assembly source files are `data1` and `program`. The `data2` section name (for a section that holds data) is not used in VisualDSP++ 3.5.

Directives and Keywords

The following assembly directive has been removed from VisualDSP++ 3.5.

Directive	Description
`.ORG`	Removed - no longer supported

Assembler Command-Line Switches

Switch	Description
`-ip`	Removed
`-jcs2l`	Removed
`-no-ip`	Removed

Preprocessor Command-Line Switch

Switch	Description
`-cpredef`	Removed; replaced with the `-cstring` switch.

Compiler and Library for Blackfin Processors

The compiler features that were either renamed, removed, or became obsolete in VisualDSP++ 3.5 as compared to VisualDSP++ 3.1 are grouped as follows.

- "C/C++ Compiler Command-Line Switches" on page 4-6
- "Predefined Macro" on page 4-8
- "C/C++ Run-Time Library" on page 4-8

Refer to Chapter 3, "*New Features and Enhancements*" of this manual for more information about the changes to the C/C++ compiler and run-time libraries.

C/C++ Compiler Command-Line Switches

The following switches are obsolete or deprecated in VisualDSP++ 3.5.

Table 4-2. Obsolete/Deprecated C or C++ Mode Selection Switches

Switch Name	Description
`-analog`	Renamed to `-c89`
`-traditional`	Removed

Table 4-3. Obsolete/Deprecated C/C++ Compiler Common Switches

Switch Name	Description
`-csync`	Deprecated, use `-workaround` instead
`-expert-linker`	Removed
`-inline`	Removed

Table 4-3. Obsolete/Deprecated C/C++ Compiler Common Switches

Switch Name	Description
`-mem-bsz`	Accepted but ignored; replaced with `-mem`.
`-no-dir-warnings`	Removed
`-no-inline`	Removed
`-no-restrict`	Removed; use `-no-extra-keywords` instead
`-restrict`	Removed; now the default mode
`-traditional`	Removed
`-xml`	Removed

Table 4-4. Obsolete/Deprecated C++ Mode Compiler Switches

Switch Name	Description
`-explicit`	Removed
`-instant[all|local|used]`	Removed
`-namespace`	Removed
`-newforinit`	Removed
`-newvec`	Removed
`-no-explicit`	Removed
`-no-namespace`	Removed
`-no-newvec`	Removed
`-notstrict`	Removed; now is the default mode
`-no-wchar`	Removed
`-Ox`	Removed; replaced with the `-O` switch
`-Oz`	Removed
`-strict`	Removed; use `-pedantic-errors` instead
`-strictwarn`	Removed; use `-pedantic` instead

Table 4-4. Obsolete/Deprecated C++ Mode Compiler Switches (Cont'd)

Switch Name	Description
`-tpautooff`	Removed
`-trdforinit`	Removed
`-typename`	Removed
`-wchar`	Removed

Predefined Macro

Prior to VisualDSP++ 3.5, the long long integer data types were not supported, causing the macro `__NO_LONGLONG` to be always defined. VisualDSP++ 3.5 adds support for the `long long int` data types; therefore, the `__NO_LONGLONG` macro is no longer predefined.

C/C++ Run-Time Library

The `bootup*.doj`, the C/C++ run-time startup file used to define jump-to-start symbols, is removed.

Compiler and Library for ADSP-219x DSPs

The compiler features that were either renamed or became obsolete in VisualDSP++ 3.5 as compared to VisualDSP++ 3.0 are described in Table 4-5 through Table 4-7.

Table 4-5. Obsolete/Deprecated C or C++ Mode Selection Switches

Switch Name	Description
`-analog`	Renamed to `-c89`
`-traditional`	Removed

Table 4-6. Obsolete/Deprecated C Compiler Switches

Switch Name	Description
`-21{9x|91|92-12}`	Removed; use `-proc` *`processor`* instead
`-dollar`	Removed
`-J`	Removed
`-no-dollar`	Removed
`-no-dir-warnings`	Removed
`-no-inline`	Removed

Table 4-7. Obsolete/Deprecated C++ Mode Compiler Switches

Switch Name	Description
`-no-restrict`	Removed; use `-no-extra-keywords` instead
`-restrict`	Removed; now the default mode

Compiler and Library for ADSP-218x DSPs

The compiler features that were either removed or became obsolete in VisualDSP++ 3.5 as compared to VisualDSP++ 3.0 are described in Table 4-8.

Table 4-8. Obsolete/Deprecated C Compiler Switches

Switch Name	Description
`-21{81|83|84|85|86|87|88|89}`	Removed; use `-proc` *`processor`* instead
`-analog`	Renamed to `-c89`
`-dollar`	Removed
`-J`	Removed
`-no-dir-warnings`	Removed
`-no-dollar`	Removed
`-no-inline`	Removed
`-traditional`	Removed

Linker

For VisualDSP++ 3.5 linker, no Linker Description File, command-line switches, LDF commands, and utilities were removed or considered obsolete. The only feature to be considered is the use of the `OVERLAY_GROUP{}` legacy LDF command.

In VisualDSP++ 3.5, the `OVERLAY_GROUP{}` is still used to group overlays, allowing each overlay to run from a different start address in run-time memory. The linker still processes all overlay groups and explicit `OVERLAY_GROUP{}` commands, producing a warning. However, the preferable way to manage overlays is to create a separate output section for each overlay group.

Silicon Part Number

The ADSP-DM102 part, supported in VisualDSP++ 3.1 has been removed from VisualDSP++ 3.5. All designs should migrate toward the ADSP-BF533 processor.

ADSP-BF535 Simulator

Compared to VisualDSP++ 3.1, in VisualDSP++ 3.5 for ADSP-BF535 processors, the following simulator menu options are not available:

Settings -> Simulator -> Exceptions

Settings -> Simulator -> ARGV/ARGC

Tcl Scripting Engine

The existing Tcl scripting engine within the IDDE is removed in favor of the more generic Automation scripting approach, which is the preferred mechanism for VisualDSP++ scripting.

The new approach is backward-compatible in nearly all respects to the legacy Tcl functionality. Only four Tcl commands no longer available in VisualDSP++ 3.5. If any of these commands are invoked, a message will be printed stating that the command is no longer supported:

```
dspplotwin
dspplotrotate
dspmemorywin
dspregisterwin
```