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

Revision 1.0, October 2003

Part Number
82-000035-02

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

ANALOG DEVICES

Copyright Information

Disclaimer

Trademark and Service Mark Notice

CONTENTS

VCSE MRESULT CODES

VCSE UTILITIES

PREFACE

Thank you for purchasing Analog Devices (ADI) development software for Digital Signal Processor (DSP) applications.

Purpose of This Manual

The *VisualDSP++ 3.5 Component Software Engineering User's Guide* describes development tools and programming guidelines for creating VisualDSP++™ reusable software components and building embedded DSP applications that exploit such components.

VisualDSP++ Component Software Engineering (VCSE) is designed for effective operations on Analog Devices processor architectures: ADSP-218x, ADSP-219x, and ADSP-BF53x Blackfin® processors.

The majority of the information in this manual is generic. Information applicable to only a particular target processor, or to a particular processor family, is provided in Appendix A, "VCSE Assembler Macros" on page A-1.

This manual is designed so that you can quickly learn about the VCSE internal structure and operation.

Intended Audience

The primary audience for this manual is programmers who are familiar with Analog Devices DSPs. This manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices DSPs can use this manual but should supplement it with other texts, such as *Hardware Reference* and *Programming Reference* manuals, that describe your target architecture.

Manual Contents

The manual consists of:

- Chapter 1, "Introduction to VCSE"

  Concentrates on concepts, evolution, and general architectural principals of VisualDSP++ Component Software Engineering.

- Chapter 2, "Developing and Using VCSE Components"

  Demonstrates how a VCSE component, which provides an implementation of a typical DSP algorithm, is defined and developed and how an application incorporates such components.

- Chapter 3, "Standard Interfaces"

  Describes VCSE standard interfaces, which provide a set of standard services for components' developers and users. Additional interfaces and their usage are also described.

- Chapter 4, "VIDL Language Reference"

  Provides reference information about the syntax and semantics of the VisualDSP++ Interface Definition Language (VIDL), a descriptive notation used to specify VCSE components and interfaces.

- Chapter 5, "VIDL Compiler Command Line Interface"

  Explains the operation of the VIDL compiler as it is invoked from the command line to process a VIDL specification. The various types of generated files and switches, which are used to tailor the compiler operation, are also described in this chapter. This software release also provides a description of generated text shells

- Chapter 6, "VCSE Rules and Guidelines"

  Documents the rules, guidelines, and best programming practices associated with the software components' successful development and inclusion into DSP applications.

- Appendix A, "VCSE Assembler Macros"

  Documents the processor-specific information, such as assembly macros, for ADSP-BF53x Blackfin and ADSP-21xx DSP processors.

- Appendix B, "VCSE MRESULT Codes"

  Documents the `MSRESULT` codes.

- Appendix C, "VCSE Utilities"

  Describes the VCSE utilities in detail.

- Appendix D, "PCC — An Example of VCSE Interface Design"

  Introduces individual interfaces and describes how components that implement the interfaces are built and used.

What's New in This Manual

This revision of the *VisualDSP++ Component Software Engineering User's Guide* documents the VCSE support for the new ADSP-BF531, ADSP-BF533, DM102, and AD6532 Blackfin processors, in addition to the existing processors, ADSP-BF532 and ADSP-BF535. Note that the older part numbers, "ADSP-21532" and "ADSP-21535", are deprecated and replaced with "ADSP-BF532" and "ADSP-BF2155", respectively.

The Blackfin processors are embedded processors that sport a Media Instruction Set Computing (MISC) architecture. This architecture is the natural merging of RISC, media functions, and digital signal processing (DSP) characteristics towards delivering signal processing performance in a microprocessor-like environment.

The manual describes the current release of the VCSE software. Future releases may include support for additional Analog Devices DSP architectures.

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.
DSP Division
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA
```

Supported Processors

VisualDSP++ 3.x Component Software Engineering currently supports the following Analog Devices processors.

- ADSP-BF531, ADSP-BF532 (formerly ADSP-21532), ADSP-BF533, ADSP-BF535 (formerly ADSP-21535), DM102, and AD6532
- ADSP-2191, ADSP-2192-12, ADSP-2195, and ADSP-2196

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 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, data sheets, 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) or **+49 (0) 89 76903-157** (Europe)
- Access the Digital Signal Processing Division's FTP website at `ftp.analog.com` or **ftp 137.71.23.21** `or ftp://ftp.analog.com`

Related Documents

For information on product related development software, see the following publications for the appropriate processor family.

VisualDSP++ 3.x Getting Started Guide

VisualDSP++ 3.x User's Guide

VisualDSP++ 3.x C/C++ Compiler and Library Manual

VisualDSP++ 3.x Assembler and Preprocessor Manual

VisualDSP++ 3.x Linker and Utilities Manual

VisualDSP++ 3.x Kernel (VDK) User's Guide

Quick Installation Reference Card

For hardware information, refer to your processor's *Hardware Reference*, *Programming Reference*, and data sheet.

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.

File	Description
`.PDF`	Manuals and data sheets in Portable Documentation Format are located in the installation CD's `Docs` folder. Viewing and printing VisualDSP++ 3.5 Component Software Engineering User's Guide 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 shortcuts you may construct, Windows provides many ways to open VisualDSP++ online **Help** or the supplementary documentation.

Help system files (`.CHM`) are located in the `VisualDSP\Help` folder. Manuals and data sheets in PDF format are located in the `Docs` folder of the installation CD. The installation CD also contains the Dinkum Abridged C++ library and FlexLM network license manager software documentation in the `\Reference` folder.

Using Windows Explorer

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

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 by:

- Emailing `dsp.techpubs@analog.com`
- Filling in and returning the attached Reader's Comments Card found in our manuals

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 TO VCSE

This chapter concentrates on concepts, evolution, and general architectural principals of component software engineering. It also provides an overview of the benefits of using VisualDSP++ Component Software Engineering (VCSE) on a DSP.

This chapter contains the following sections.

Origin of Components

The idea of creating programs from reusable parts is not new and can be traced back to the earliest days of computing. The original objective was to provide additions to the user's program to allow it to execute on a particular computer. Typically, each program was supplemented with a fixed set of routines for interfacing to hardware and operating system kernels or providing support for early programming languages.

Interestingly, most of the programming devices that we associate with reusable code were invented almost half a century ago. Callable subroutines were present in the Fortran language designed by John Backus in 1954, though the idea had been implemented in assembly language even earlier. Subroutines had evolved in stack-based procedures by the time Algol 60 was introduced a few years later.

Most remarkably, a construct called a *class* and a related mechanism called *inheritance* were developed in the mid-sixties and incorporated into the language Simula 67 by O.-J. Dahl and co-workers at the Norwegian Computer Center. Classes languished in obscurity for twenty years until a Danish computer scientist Bjarne Stroustrup developed a variant of C called “C with Classes”, which subsequently evolved into the C++ language.

Two other advances that enabled software to be reused were the emergence of libraries of useful subroutines and the related development of relocatable linkers, which allowed the precompiled versions to be combined with a user’s program. Many important scientific applications were created in this way and distributed as library packages for use on mainframe computers.

Despite these very early innovations, there was little attempt to apply *reuse* in the way that we aspire to today. The early days of computing were dominated by large mainframes shared by many users. Application programs were small—a few hundred lines—mostly because they were stored on physical media like cards or tapes. Programs were written in proprietary assembly languages or fairly primitive programming languages for very locale-specific purposes.

Consequently, there was very little need for portability (beyond the requirement to carry a tray of cards from one building to another!). If there was any demand for reusability, then it usually arose within a single company or organization. However, by the mid-sixties, certain groups of users—particularly, researchers in universities and government agencies—started to develop requirements for exchanging and moving software from one computer to another.

The current interest in reusable software components derives from a number of important developments in computer hardware and software that have occurred over the last thirty years. These developments include:

1. The use of digital media for secondary storage, allowing programs to grow dramatically in size.

2. The development of computer networks and mini-computers, which led to greater demand for program portability. It also increased the use of high level languages with "standard" definitions distinguishing implementation-dependent and portable features.

3. The emergence of platforms, such as PC/Windows and Unix/WorkStations, which created two distinct markets for application developers using the C programming language. The possibilities for building interoperating applications that straddled process or platform boundaries began to be explored.

4. Finally, the emergence of the public internet and the world wide web, which revived the fortunes of Oak, a little-known language invented by James Gosling at Sun MicroSystems. The language, now called Java, carries the "write once – run anywhere" marketing claim.

Embedded systems, by their very nature, have been insulated from many of the developments described above. But next generation systems are growing now in size and complexity. For example, they may have multi-function capability or are required to run on multiple platforms or processor families. In certain market sectors, requirements are beginning to emerge for applications to function in networked environments; or to be downloaded or dynamically modified and reconfigured. In turn, this has led to the gradual adoption of high level programming languages like C or C++, where the compiler effectively automates code generation and where assembly code can be reinserted to match performance requirements.

Increases in size and complexity are also leading software developers to reconsider how embedded systems should be developed in the future. In particular, how long will the "build from scratch" approach remain viable? Equally, are newer component-based approaches relevant—and what are components anyway?

Software Components

Modern software components usually conform to one of two industry platforms: Microsoft Component Object Model (COM) or the Object Management Group's Common Object Request Broker Architecture (CORBA). Both standards promote an "object-based" approach to reusable software that embraces certain key aspects of object-oriented software development without committing to any particular programming language. The same approach has been incorporated into the design of VCSE.

Now let us look at components in greater detail. First, a software component is designed to function as a reusable part of a larger program. Usually, it is not the whole program, but at the same time, it is larger and more powerful than a single subroutine. It is also useful to bear in mind that component developers and component users are usually different groups of people.

A component provides a service that is specified through a set of function declarations called an *interface*; the functions are called the *methods* of the interface. The algorithms and implementation details employed by the methods are hidden from the component user and are said to be *encapsulated* by the component.

A user interacts with a component by calling the methods of its interface and passing in parameters. The way in which the call is implemented must allow the component user and the component developer to use different programming languages. Because components may be written in C, C++, or assembly, their interfaces are specified using a special notation called *Interface Definition Language,* or IDL. IDL resembles the declarative parts of C, but it is not a full programming language.

In addition to its methods, a component contains a set of private variables that hold its *state*. For example, a component implementing a time-of-day clock stores the current time as part of its state. The variables that comprise a component's state normally hold values that must be preserved across calls to its method functions.

A user can create multiple *instances* of a component. Each instance shares the same methods but has a distinct state. This is arranged by storing the state variables for each instance in a separate region of memory. For example, to build an application recognizing international time zones, we might create several instances of the clock component whose separate states store different regional times. The memory used to store the state of a component is sometimes referred to as *instance storage*.

Component instances are created and destroyed by special factory functions called `Create` and `Destroy`. When an instance is created, the factory function ensures that storage is allocated and returns a handle to the component. The user must retain the handler for as long as the component is required. When a component instance is destroyed, the instance storage is released by passing the handle to the `Destroy` function.

In the case of workstations and PCs, components are usually distributed in a standard format and installed by creating an entry in a component database on the host machine. Systems that support the interaction of components across networks use the database to activate the component when a request to create a new instance is received.

Benefits of Components

Software components offer a number of benefits that derive directly from the properties described in "Software Components" on page 1-4.

1. Components are easy to maintain because they hide all their implementation details. Consequently, a developer can make internal changes to a component provided its external interface stays the same. Many component based applications on PCs and workstations access components using dynamic link and call mechanisms. These mechanisms allow new component versions to be installed without requiring the application programs to be reinstalled.

2. Components are flexible and reusable because they are language neutral. Both the component user and component implementor are free to choose the most appropriate language for development. In addition, there is no difference, other than in performance, between using a component locally (on the same machine) and remotely (on a different machine).

3. Components are extensible because they may provide new methods that are packaged as an extension of an older interface. When an extended version of a component is deployed, users access the new methods by requesting access to the extended interface. Note that the component still provides the non-extended version of the interface, so that existing applications continue to work with the new component.

VCSE Components

Components developed with VisualDSP++ share a common set of attributes that are determined by the VCSE Component Model. These include:

1. Interfaces and encapsulation. Components provide encapsulated implementations of one or more interfaces.
2. Instance creation. Component instances are created and destroyed dynamically. A component that is created dynamically may be supplied with memory that is allocated statically.
3. Flexibility. Components can be implemented and deployed using any combination of C, C++, and assembly programming languages.
4. Automation. VisualDSP++ provides support for semiautomatic generation of component and interface specifications and for the deployment, installation, and documentation of completed components.
5. Interoperability. Components from different vendors can interoperate without the risk of resource issues, such as name clashes or memory management conflicts.

The VCSE Component Model also ensures that components are tailored for embedded DSP applications, in particular:

- The overhead associated with components—particularly code size and execution time—is minimized. The overhead in learning how to develop and use components is minimized by the VCSE development tools provided with VisualDSP++.
- There is no dependence on any particular run-time environment. VCSE components may be used in standalone applications or in conjunction with a variety of multithreaded kernels.

- Components delegate the allocation of resources, such as memory, to the application framework in which they are deployed. Applications can supply statically allocated memory to a component rather than rely on the less efficient heap-based mechanisms that are invoked from C or C++.
- The Component Model specifies a hierarchical namespace that enables all components and their related files to be identified. Each organization may reserve a portion of the namespace by registering a unique namespace tag. The management of names within a tagged namespace is delegated to the organization registering the tag. See "Company Namespace Registration" on page 2-87 for more information on registering namespaces.
- VCSE allows the eventual deployment of components on simple homogeneous multiprocessor systems. Interprocessor communication is provided in a way that is transparent to both the developer and user of a component.

Component Software Engineering Concepts

The two key concepts provided by the VCSE Component Model are interfaces and components. Broadly speaking, an interface specifies what is to be done, while a component determines how it is to be done. More formally, we say the component provides an implementation of the interface.

VCSE Interfaces

An interface is a collection of functionally related operations that provide a service. The operations are specified by a list of functions called methods that an application may invoke. The methods by themselves may not provide a complete definition of the service and may require supplemental documentation, which specifies additional operational details, such as the order in which methods are to be invoked or the range of values a parameter is permitted to take.

An interface is completely abstract—it is not tied to any particular implementation. For example, you can define a sorting interface that specifies methods for entering and retrieving data, as well as for triggering the sort, but which does not contain any elements that oblige the sort to be performed by a particular algorithm.

An interface must not be changed once it has been published (made available to users). However, it is possible to define a new interface as an extension of an existing interface by supplying a list of additional methods. For example, we might extend a "sort" interface into an "ordered sort" interface by adding a new method that controls the order (ascending or descending) of the sort. The "sort" interface continues to exist as a part of the "ordered sort" interface.

In VCSE, an interface name must start with an 'I'. Thus, a sorting interface is called `ISort` rather than `Sort`.

Interface Example

Interfaces are specified using a notation called the VCSE Interface Definition Language (VIDL). A simplified version of the VIDL definition of an interface supporting image compression is as follows.

```
[iid("a988bd82-e306064b-a9938513-3ced0fa8")]
interface IImageCmp extends IBase {
   MRESULT SetSNR(
      [in]        int      snr );
   MRESULT CompressImage(
      [in]        int     length,
      [out]       int     CompressedLength,
      [in, out]   int     image[256] );
   MRESULT DecompressImage(
```

```
        [in]       int    Compressedlength,
        [out]      int    Length,
        [in, out]  int    image[256] );
};
```

The IImageCmp interface consists of three methods: SetSNR, CompressImage, and DecompressImage. Collectively, they provide the *functional specification* of the image compression service. Each method is described by a declaration specifying the types of parameters and return result. Various attributes, supplied to each parameter, describe how the parameter is used.

SetSNR takes an "in parameter" snr, which supplies the minimum acceptable signal to noise ratio. CompressImage takes an "in parameter" length, which specifies the number of supplied image elements, and returns an "out parameter" CompressedLength, which holds the corresponding number of elements in the compressed image. The array image is an "in-out parameter" that supplies the uncompressed elements and returns the compressed elements to the caller.

The description of an image compression service provided by a *particular* implementation of the IImageCmp interface may require extra information concerning usability, performance, and quality of service. This information, which is referred to as the *operational specification* of the interface, is provided by inserting special comments at appropriate points in the VIDL. As described later in this manual, VCSE provides a feature called *auto-doc*, which allows the contents of these comments to be extracted and converted into HTML.

IImageCmp is defined as an extension of a predefined interface IBase, which provides a single method called GetInterface. This method allows an application to request an interface by specifying its iid (interface identifier). If the component implements the interface, the request returns a pointer that allows the interface's methods to be called. If the interface is not implemented, GetInterface returns an error. VCSE requires every interface to be extended directly or indirectly from IBase, so GetInter-

`face` is always available as a method. This means that an application may use `GetInterface` to navigate through all the interfaces provided by each component.

VCSE Components

A component provides the implementation of one or more interfaces by supplying the code for their method functions. However, the methods are encapsulated within the component, so their internal working variables and utility procedures cannot be accessed from outside the component. In fact, the only way an application can interact with the component is by calling its interface methods. These constraints help to protect components from misuse and improve their ability to be deployed in different operational contexts.

VCSE allows components the freedom to reuse or leverage other component implementations. However, a component must document its dependencies, so that installation may be managed consistently. The VIDL notation allows the dependencies between components to be recorded without revealing the nature of the interactions between them.

Interfaces make it easy to exchange and upgrade the components installed in an application. If the new version of a component continues to provide the same interfaces, no changes to the application code are required. If the new version provides additional methods in an extension to a previous interface, applications can choose whether to use the extended or original interface. If the new interface is required, the application must be modified accordingly, recompiled, and linked with the component. But if the old interface is still adequate, the application needs only be relinked to the component. Interface extension is a very useful way of providing new functionality while preserving existing interfaces.

Components conform to naming conventions to make them easy to deploy without risk of name clashes with other components already in use. For more information, see “File Names” on page 5-23.

An application may create one or more instances of components, each with a private set of *instance variables*. The methods of the component may store and retrieve the values of the instance variables, so that collectively they represent the state of the instance. In the case of the clock component referred to earlier, the state may be represented by a single instance variable that contains the current time. Component instances provide a very convenient way to model real-world objects. For example, an application that uses multiple data channels may represent each channel by an instance of a "channel component". Each instance holds the state of its channel privately, so there is no possibility of interference between them.

Applications may create and destroy component instances dynamically during execution. When an instance is created, an area of memory called instance storage is allocated for the instance variables and retained until the instance is destroyed. VCSE allows considerable flexibility in the way in which instance storage is managed. Components may choose to allocate memory internally or to acquire it from an external memory manager. Memory managers may themselves supply memory using static or dynamic allocation strategies.

It is worth noting that the idea of instances helps distinguish components from other reusable software entities, such as program libraries. Although the functions within a library may require state to be preserved, it is the responsibility of the library user to preserve the state information and to supply it explicitly on each call. In addition, components allow more than one implementation of an interface or service within one program, whereas there can be only one version of a library per program.

It is unusual to find true dynamic linking in embedded DSP applications because of the run-time overhead involved. In VCSE, components are statically linked to programs, and the run-time cost of instantiation is minimized.

Component Example

The following example shows a slightly simplified VIDL description of two components offering different implementations of the generic IImageCmp interface.

```
[iid("a988bd82-e306064b-a9938513-3ced0fa8")]
interface IImageCmp extends IBase
{
   MRESULT SetSNR(
      [in]       int     snr );
   MRESULT CompressImage(
      [in]       int    length,
      [out]      int    CompressedLength,
      [in, out]  int    image[256] );
   MRESULT DecompressImage(
      [in]       int    Compressedlength,
      [out]      int    Length,
      [in, out]  int    image[256] );
};
component  CJpeg  implements IImageCmp;
component  CGif   implements IImageCmp;
```

The CJpeg component provides support for JPEG compression, which is most effective for images with smooth color changes, while the second component CGif uses GIF compression, which is much more effective for images with sharp edges. The relative effectiveness of the two components, therefore, depends on the type of image to be compressed, although both offer the same functional interface. The performance of the two components is also quite different since they use distinct algorithms. The user of either component, therefore, relies on its operational specification to choose a suitable component implementation of the IImageCmp interface for a particular task.

Applications using the IImageCmp interface can switch between the two implementations simply by invoking the Create functions of one or other of the components. The method calls required to invoke compression or decompression do not need to change because each component provides the same interface. Consequently, switching between components only requires a small change to the name of the function used to create the component instance. This makes it easy to evaluate and select the component that is best suited to the image processing required.

The second example shows the VIDL description of two components offering different implementations of a generic sorting interface ISort. The ISort interface consists of three methods: SetData, GetData, and Sort. Collectively, they provide the functional specification of a sorting service. Each method is described by a declaration specifying the types of parameters and the result returned. The description is sufficiently general to permit several possible implementations. For instance, GetData and SetData may physically copy the data or may note the address of the data, so that sorting is performed "in place".

```
[iid("dfa1bd82-e306064b-a9938513-de440fa8")]
interface ISort extends IBase {
   MRESULT SetData(
      [in]              long int N,
      [in, size_is(N)]  float data[] );
   MRESULT GetData(
      [in]              long int N,
      [out, size_is(N)] float data[] );
   MRESULT Sort(void)
};
component CBubbleSort implements ISort;
component CQuickSort implements ISort;
```

The CBubbleSort component uses the bubble-sort algorithm and, therefore, has the performance characteristics typical for that method of sorting. The CQuickSort component uses the quick-sort algorithm, which is usually faster but may require additional memory to achieve the increased performance. Naturally, each instance of `CBubbleSort` and `CQuickSort` applies the appropriate sorting method to the instance data supplied by `SetData`.

Once again, the provision of a common interface makes it easy for applications to switch between the two components and to evaluate them with appropriate test data.

Binary Standard Interface

The VCSE Component Model defines a binary standard that specifies a mechanism for invoking interface methods. The standard is independent of the language in which the component or its application environment is written. The two most important features of the standard are:

- The methods of an interface and the application environment in which they are invoked must support the C language run-time model for function calls.
- The methods of an interface are called indirectly through a binary structure called a *method table*.

A component provides a method table for each supplied interface; each entry in the table contains the address of the component function that implements the method. Figure 1-1 on page 1-16 shows the method table for the `ISort` interface as implemented by the `CQuickSort` component. The table has an entry for each interface method (including those in its base interfaces) to reference the corresponding function in the `CQuickSort` component implementation.

Each instance of an interface is represented by an interface pointer, which refers to a structure containing the address of the interface method table. The method table, in conjunction with the use of the C run-time model, provides a standard mechanism to ensure VCSE components and applications work together, irrespective of the language in which they are written.

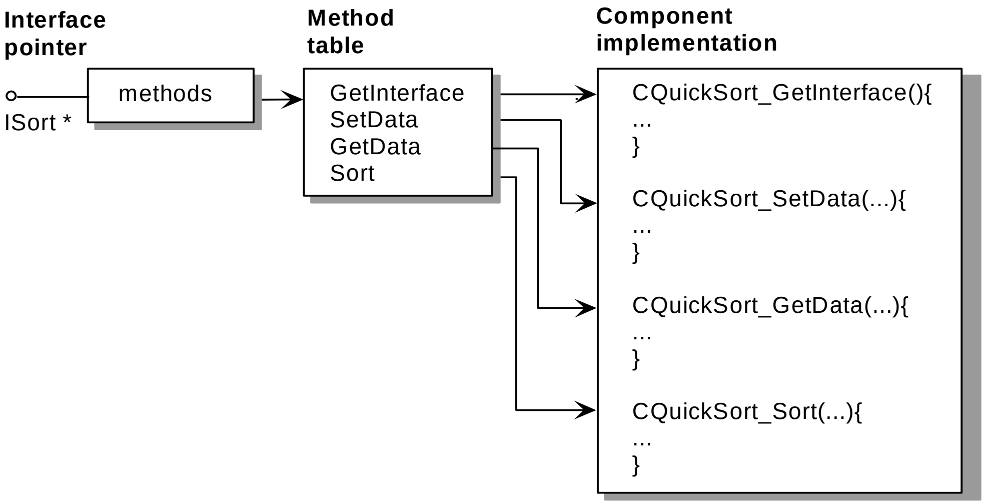


Figure 1-1. ISort Interface Method Table

It also allows interfaces to be decoupled from specific implementations. For example, we can provide access to an `ISort` implemented by a `CBubbleSort` component by creating a method table whose entries reference the corresponding method functions in `CBubbleSort`.

Method tables allow different implementations of the same interface to coexist within the same application. In the previous example, the `ISort` interface pointers returned by `CQuickSort` and `CBubbleSort` refer to the separate method tables provided by these components. It follows, calling the `Sort` method with an `ISort` pointer returned by `CQuickSort` will invoke the function `CQuickSort_Sort`, while calling the `Sort` method with an `ISort` pointer provided by `CBubbleSort` will invoke the function `CBubbleSort_Sort`.

Separate instances of the same component return different interface pointers, which nevertheless refer to the same method table. In general, all instances of a component share the same method code and method tables.

Interface Definition Language and Compiler

The VisualDSP++ Interface Definition Language allows you to specify interfaces and components that conform to the VCSE Component Model.

VIDL specifications are contained in text files, which are created with an editor or by invoking the dialog-driven VCSE wizards within the VisualDSP++ Integrated Development and Debugging Environment (IDDE). The VIDL files are processed by a translator called the VIDL compiler, which generates a framework or implementation *shell* for each component using C, C++, or assembly language. The shell is normally completed by the component developer before submitting it to the language compiler or assembler.

VIDL is a language-neutral way to specify components and interfaces. It favors neither C, C++ or assembly and, therefore, allows developers to choose between implementation languages. For information on the VIDL syntax, see "VIDL Language Reference" on page 4-1; for information on how to create interfaces, see "Developing and Using VCSE Components" on page 2-1.

Figure 1-2 on page 1-18 illustrates how a VIDL specification is transformed by the VIDL compiler into sets of program source files. Note that this is a simplified example since the number of generated files and their names normally depend upon the entities defined in the `.IDL` file being processed and not on the name of this file.

If the specification for an interface and a component is held in the file `example.idl`, the VIDL compiler generates a header file `example.h` for the interface together with corresponding C, C++, or assembly component implementation files, depending upon the setting of a command line switch.

The header file contains the declarations of the method functions for the interfaces defined in the VIDL file, and the `.C`, `.CPP`, or `.ASM` files contain the shells for the components. Each shell contains a set of method func-

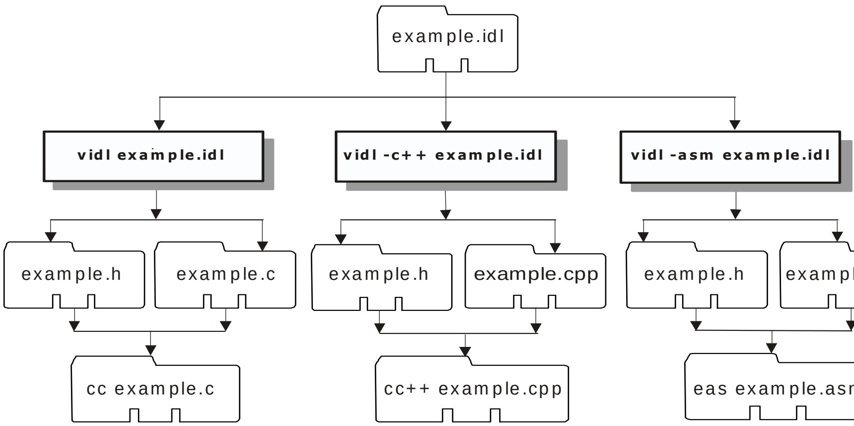


Figure 1-2. VIDL Compiler Operation

tion "stubs" that are completed by the component developer. The operation of the VIDL compiler is described in detail in "VIDL Compiler Command Line Interface" on page 5-1.

Integration With VisualDSP++

The VisualDSP++ IDDE provides comprehensive support for creating and using VCSE components, which includes the following elements.

- Wizards to create initial VIDL descriptions for interfaces and components using intuitive, dialog-driven interfaces.
- A VisualDSP++ project type to develop VCSE components and to incorporate the VIDL compiler into the build process.
- A VisualDSP++ Component Manager to maintain a database of VCSE components. The component manager supports installing new components, browsing for existing components and importing them into development projects, and uninstalling obsolete components.

- A wizard to manage the process of packaging component files into a compressed file for distribution.

The following sections provide summary descriptions of each IDDE facility. For detailed instructions on how to use them, see the VisualDSP++ online **Help**.

Component Projects

A component project automatically incorporates the extra steps required to manage the development of a VCSE component within a VisualDSP++ project.

In the first step of the build process, the VIDL compiler processes the VIDL file and generates the implementation and header files. If the implementation files already exist, the VIDL compiler preserves all the code in user supplied areas, such as the bodies of interface method functions. In addition, if a method has been removed, the user supplied method body is still kept and accumulated in a holding area at the end of each file.

The C/C++ compiler or assembler is then invoked on the project's source files, and a library is created. Additional source files can be added to the project to be compiled and included into the library as part of the component implementation.

New Interface and Component Wizards

The New Interface Wizard guides you, step by step, through the process of generating a VIDL interface specification.

In the first step, supply the name of the interface, the namespace in which it is defined, and the interface it extends. Also provide a short description of the service that the interface provides. In the following steps, specify and describe the methods and supply the names and types of their parameters. The wizard propagates the interface and method descriptions into auto-doc comments that are generated in the VIDL file.

The process of specifying a new component and creating a VisualDSP++ component project is managed by the New Component Project Wizard. The wizard allows you to specify the name of the project and the location of its development directory. Then you supply the component's company tag, name, title, and category and set its attributes. When all the information is gathered, the wizard creates the component's development project and generates a VIDL file containing the component definition.

Component Packaging Wizard

Once a component is fully developed, it must be packaged into a compressed VisualDSP++ component package file (`.VCP`) for distribution. The packaging is primarily controlled by the component manifest file (`.XML`), which is created by the VIDL compiler. The New Component Package Wizard combines information from the manifest file with information from the wizard and generates the `.VCP` file for distribution.

First, the wizard requests the name for the `.XML` manifest files. If you initiate the wizard while a component development project is active, the wizard defaults to suggesting the `.XML` file for the project. In the next step, the wizard shows various attributes of the component and allows specification of the version number and status. The distributed component can be, for example, the full version, a demonstration version, or may only contain the documentation.

The package wizard allows the addition or removal of files from the list of files in the manifest, enabling complete control over the distributed file contents. You can also specify which files are to be automatically added to a project when you add the component to that project. Finally, the wizard enables you to specify the directory in which the packaged file is to be stored.

Component Manager

The Component Manager provides a comprehensive set of facilities enabling you to browse, download, and install components onto your system. Once installed, the components can be easily added to VisualDSP++ projects.

View either the list of components installed on your system or those that are available from the Analog Devices web site. Each component is displayed with a brief description of its function and application domain. The list of components can be sorted by various properties, such as the component name, supported interface, component category, status, and the target processor.

Once you have identified a component that meets your needs, the Component Manager can download and install it on your system, making the component available for your development projects.

The Component Manager also can be used to uninstall components from your system.

> Adding a component to a VisualDSP++ project does not copy the component files into the project's directory but adds references to the installed files. Installing a new version of a component, therefore, impacts all projects using that component.

Software Architecture

The VCSE software architecture, which controls the interaction between the application and its components, is based on the client-server model, where the application is the client and the component is the server providing the client with certain well-defined services.

The VCSE architecture is platform independent and does not specify any particular run-time environment. Components can be used in a single threaded or multithreaded environment, although VCSE itself provides no support for the interactions between threads. The architecture assumes resource synchronization is handled directly by the application client and its components.

The VCSE architecture has also been designed to cater for multiprocessor systems, where a component and its client application may execute on different processors. Multiprocessor support will be available in future versions of VCSE.

On a typical system, a client application may use components from more than one vendor. The structure of an application, where the client and its components execute on the same processor, is shown in Figure 1-3.

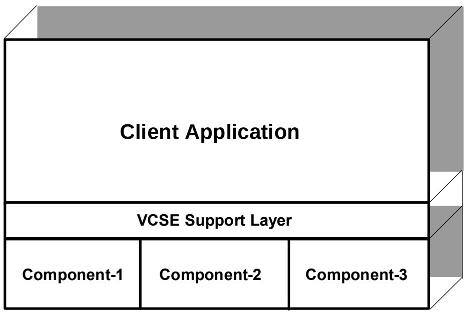


Figure 1-3. Simple Application Model

When the client application and the server components reside on the same processor, VCSE forms a very thin layer that provides essential services for creating and destroying component instances and for acquiring component interfaces. The application interacts directly with the component whenever it calls an interface method.

When the application and its components reside on different processors, the VCSE architecture allows them to remain unaware of their relative separation. In this case, the VCSE layer is responsible for providing a remote method invocation mechanism that enables the method calls to be transported from the application to the target component's processor. The VIDL attributes attached to the declaration of each method parameter ensure that their values are passed correctly.

Rules and Guidelines

The VCSE Component Model specifies how re-usable components may be constructed for applications running on Analog Devices DSP processors. The VCSE development tools provided within VisualDSP++ help to create application frameworks in which components operate irrespective of the implementation language. Although the Component Model ensures interoperability between applications and components can be met, it cannot guarantee this will always be the case, particularly when assembly language is involved. For this reason, the Component Model and the development tools are supplemented with a set of rules and guidelines, which are designed to ensure that VCSE components will interoperate successfully.

The rules and guidelines cover two broad areas—programming and packaging—although these two sometimes overlap. Issues concerning the correct operation of a component, considered in isolation, come under programming, while issues concerning a component's inclusion in an application that may use other components come under packaging.

The rules and guidelines for VCSE components and interfaces are described in "VCSE Rules and Guidelines" on page 6-1.

Rules and guidelines are grouped in two sets: a core set applicable to all components and a set applicable to components that implement VCSE *algorithms*. A VCSE algorithm is a component supporting an interface that is extended from the standard interface `VCSE::IAlgorithm`.

The rules describe mandatory actions or practices that application and component developers must follow. Applications may fail to build or run properly if they, or any component they include, fail to obey a rule.

The guidelines describe actions or practices that Analog Devices strongly recommends application and component developers to follow. Applications may build or run if a guideline is not heeded, but they may be harder to debug or deploy. In addition to the rules and guidelines, "VCSE Rules and Guidelines" includes notes and tips regarding the VCSE component software.

2 DEVELOPING AND USING VCSE COMPONENTS

In this chapter, you will learn how a component that provides an implementation of the μ-law encoding scheme from the ITU recommendation G.711 is created. The algorithm needed to effect encoding or decoding is very straightforward and enables to concentrate on the process of definition and creation of the component that implements the algorithm.

The chapter contains:

The first and most important step in developing a component is to decide on the functionality that the component is to provide; in particular:

- What services the component is to provide
- How the user of the component should request the services provided by the component
- How the available services are to be implemented and tested

The VCSE Component Model can help to structure and guide these decisions since a key part of developing a component is deciding the interfaces that it will offer. Each interface specifies a service provided by a set of programming language functions called "methods".

The interface itself does not contain the body or definition of the method function; it only contains the declaration or description of the methods. The interfaces provided by a component represent a contract between the component and the applications in which it is used, and they should not be changed once an interface has been issued for use.

VCSE interfaces are expressed using the VCSE Interface Definition Language (VIDL). The creation of a suitable description of the interface using VIDL can clarify the specification of the offered services. Structuring the development into one or more interfaces can also create the appropriate structure for the implementation even though the details of the implementation should remain hidden behind the published interfaces.

VCSE defines some standard interfaces, which component developers may consider providing support for. The standard interfaces, documented in "Standard Interfaces" on page 3-1, define service interfaces for capabilities that many components may wish to provide. The two main standard interfaces are: `IAlgorithm`, which defines a common subset of methods supported by algorithms and `IMemory`, which defines the standard mechanism for allocating memory resources.

Defining an Interface

The μ-law compression algorithm converts a 16-bit value in the range –8192 to 8191 to a more compact 8-bit value in the range 0 to 255; the decompress function reverses the compression.

To minimize the overhead arising from invoking the interface methods, the interface is designed to ensure the most commonly used methods have a reasonable amount of processing on each method call. In the case of G.711, the interface design guarantees an array of data elements is passed into and returned from each invocation of both the compress and decompress methods.

The prototypes for the compress and decompress functions in C language are shown in Listing 2-1.

Listing 2-1. G.711 Function Prototypes

```
int Compress(int N, const short *inData, short *outData);
int Decompress(int N, const short *inData, short *outData);
```

where the first parameter `N` specifies the number of values supplied in the input arrays `inData` and returned in the output arrays `outData`. The array is to be either compressed or decompressed, and the processed values are returned in the corresponding elements of the output array `outData`. The return value indicates whether the whole operation is successful.

To provide this μ-law encoding service as a VCSE component interface, specify the corresponding interface as follows.

```
[iid("e42dec41-1936ff4e-9b392d02-7d5f3731")]
interface IG711 extends IBase
{
  MRESULT  Compress(
     [in]   unsigned N,
     [in]   short   inData[256],
```

```
        [out]  short   outData[256]);
      MRESULT  Decompress(
        [in]   unsigned N,
        [in]   short    inData[256],
        [out]  short    outData[256]);
   };
```

The VIDL definition defines the name of the interface to be `IG711` and specifies that this interface extends the `IBase` interface. Every interface name must start with an '`I`' and must extend directly or indirectly from `IBase`, the VCSE root interface. Every VCSE interface must also be given a totally unique identifier, so different interfaces can be distinguished while executing. The `[iid("e42dec41-1936ff4e-9b392d02-7d5f3731")]` attribute specifies the unique interface identifier that allows the avoidance of name clashes with other interfaces. VisualDSP++ provides a tool to generate a unique identifier in the above format ready for incorporation in the specification of an interface.

The `IG711` interface has two methods, `Compress` and `Decompress`, which correspond to the two C function prototypes in Listing 2-1 on page 2-3. Every VCSE method must return a value of type `MRESULT`, a short integer (16-bit) indicating if the method call is successful or not.

The VIDL method definitions provide more information about each of the parameters than their prototypes. This additional information is provided in the form of attributes, which are enclosed in square brackets and precede the definition of each parameter.

In `IG711`, the attribute `[in]` specifies that the value of the parameter `N` is being passed into each of the methods. In the case of the `inData` parameter, the VIDL explicitly specifies the parameter is an array of 256 short integers whose values are passed into the method. Similarly, the `[out]` attribute specifies the `outData` parameter is an array of 256 short integers whose values are returned from the method.

Although the VIDL specification provides the information about the number of elements in the array and the direction the data values are transferred, only a pointer to the start of the array is actually passed when a method is being invoked.

If the `[out]` attribute is specified for a scalar parameter, such as an `int`, a pointer to an `int` is actually passed when the method is invoked. A parameter can also be qualified with the attribute combination `[in, out]`, which implies the value is passed into the method and a possibly different value is returned. A scalar parameter qualified with the `[in, out]` attribute is also actually passed as a pointer when the method is invoked.

For more information about the VIDL syntax, see "VIDL Language Reference" on page 4-1.

The interface specification restricts to 256 elements the maximum number of elements that can be passed to either method. Any implementation must obey the restriction that it can only access a maximum of 256 elements in any of the passed arrays.

When an array parameter is qualified with the attribute `[in]`, the corresponding parameter of a C or C++ method is qualified with the `const` keyword since a parameter marked as `[in]` should not be changed by the invoked method. In addition, the use of the `const` qualifier gives the C and C++ compiler optimizer a better opportunity to optimize access to the array within the method since the optimizer knows the values of the array cannot be changed.

Since the number of elements to be processed is passed as the first parameter, the interface definition can be rewritten to provide a much greater degree of flexibility to the users of the interface. The number of elements of a passed array can be specified dynamically by rewriting the interface as follows.

```
[iid("e42dec41-1936ff4e-9b392d02-7d5f3731")]
interface IG711 extends IBase
{
```

```
    MRESULT  Compress(
       [in]  unsigned N,
       [in, size_is(N)]   short inData[],
       [out, size_is(N)]  short outData[]);
    MRESULT Decompress(
       [in] unsigned N,
       [in, size_is(N)]   short inData[],
       [out, size_is(N)]  short outData[]);
  };
```

The size_is attribute specifies that the number of array elements for the qualified array is determined by the value of the passed parameter N. The size_is attribute allows the caller to control the maximum size of the array that is allocated and to notify the method of the number of elements of the array that it can access. An interface that handles arrays of different sizes is generally much more useful than an interface that only handles arrays with a fixed size.

On ADSP-BF53x DSPs, the C/C++ compiler optimizer cannot vectorize access to arrays of short integers unless it ensures that such arrays are word aligned rather than half-word aligned as required by the C or C++ language. The VIDL language allows you to specify such a requirement for the parameters being passed to the interface. When you do so, the VIDL compiler generates the appropriate C and C++ language structures to notify the optimizer that the parameters are in fact aligned. Adding the necessary align attribute for each of the short arrays provides the improved version of the interface definition for the IG711 interface. For example,

```
[iid("e42dec41-1936ff4e-9b392d02-7d5f3731")]
interface IG711 extends IBase
{
  MRESULT Compress(
    [in]  unsigned N,
    [in,  size_is(N), align(4)] short inData[],
```

```
    [out, size_is(N), align(4)] short outData[]);
  MRESULT  Decompress(
    [in]  unsigned N,
    [in,  size_is(N), align(4)] short inData[],
    [out, size_is(N), align(4)] short outData[]);
};
```

The `align(4)` attribute attached to each of the array parameters specifies that the arguments passed to these methods must have at least word alignment on ADSP-BF53x processors.

On ADSP-BF53x processors, the data layout generated by the C and C++ compilers for static data normally satisfies this word alignment requirement.

On ADSP-218x and ADSP-219x processors, the `align` attribute is not normally required since these processors are word-addressed architectures. However, on ADSP-218x DSPs, the `align` attribute may be used where arrays are to be accessed as circular buffers since these arrays must be correctly aligned to correspond to the size of the buffer.

The name of the interface, `IG711`, is derived from a reference standard for voice compression and decompression, so it is possible that another developer might choose the same name but define the interface differently. In order to avoid name clashes, VCSE provides namespaces that allow identically named interfaces and components to be distinguished. Namespaces are themselves assigned names that identify the company or organization that owns the names that it contains.

For example, Analog Devices, Inc. has reserved the `ADI` namespace for its components. The `EXAMPLES` namespace has been reserved for ADI's example interfaces and components used in the VCSE documentation and tutorials. The `LOCAL` namespace has also been reserved for interfaces and components that will not be distributed outside of creating environment.

Analog Devices maintains a registry of namespace names to ensure they are unique. See "Company Namespace Registration" on page 2-87 for more information on registering namespaces.

To define the above IG711 interface within the EXAMPLES namespace, the full definition of the interface is re-written as follows.

```
namespace EXAMPLES {
     [iid("e42dec41-1936ff4e-9b392d02-7d5f3731")]
     interface IG711 extends IBase
     {
     MRESULT Compress
         [in]  unsigned N,
         [in,  size_is(N), align(4)] short inData[],
         [out, size_is(N), align(4)] short outData[]);
     MRESULT Decompress(
         [in]  unsigned N,
         [in,  size_is(N), align(4)] short inData[],
         [out, size_is(N), align(4)] short outData[]);
     };
 };
```

The VIDL compiler does not accept any definitions that are placed outside of a namespace. Since the IG711 interface is defined within the EXAMPLES namespace, the fully qualified name for the interface is EXAMPLES::IG711. The full name of the interface includes the namespace prefix to ensure its uniqueness. For example, a different interface called IG711 may be defined within the ADI namespace and identified by its full name ADI::IG711.

Although the previous VIDL definition incorporates significantly more information than a C or C++ prototype, the interface definition by itself is not sufficient to use the interface. To use the services offered by an interface, further information, such as the operational specification of the interface, is needed. The operational specification covers such aspects as the order in which the methods of the interface are called or ranges of values that are valid for each parameter.

The VIDL language supports a formalized comment notation, called auto-doc comments, which allows specification of operational details along with the formal definition of the interface and its methods. Auto-doc comments are translated into HTML text and can contain any HTML constructs necessary to format the translated text. For more information, see "Auto-doc Comments" on page 4-72. Listing 2-2 shows the definition of the interface completed with the auto-doc comments.

Listing 2-2. EXAMPLES::IG.711 VIDL Specification

```
namespace EXAMPLES {
/**
 * G.711 is the international standard for encoding telephone
 * audio on a 64 Kbps channel. It is a pulse code modulation
 * (PCM) scheme operating at a 8 kHz sample rate, with 8 bits
 * per sample. There are two different variants of G.711:
 * A-law and mu-law. A-law is the standard for international
 * circuits.
 * <p>
 * The IG711 interface defines a service that allows values to
 * be compressed or de-compressed using either variant.
 */
 [iid("e42dec41-1936ff4e-9b392d02-7d5f3731")]
 interface IG711 extends IBase
 {
  /**
   * The Compress function is used to compress a block of
   * data. The function compresses each of the values supplied
   * in the inData array and stores the 8-bit compressed
   * value in the corresponding element of the outData array.
   * @param N is the number of values held in the inData
```

```
    *        array.
    * @param inData is the array of input values each of which
    *        is to be compressed.
    * @param outData is the array which will receive the
    *        compressed values.
    */
    MRESULT  Compress(
        [in]  unsigned N,
        [in,  size_is(N), align(4)] short inData[],
        [out, size_is(N), align(4)] short outData[]);
    /**
    * The Decompress function is used to de-compress a block of
    * data. The function de-compresses each of the 8-bit values
    * supplied in the inData array and stores the uncompressed
    * 16-bit value in the corresponding element of the
    * outData array.
    *
    * @param N is the number of values held in the inData array.
    * @param inData is the array of input values each of which is
    *        to be de-compressed.
    * @param outData is the array which will receive the
    *        de-compressed values.
    */
    MRESULT  Decompress(
       [in]  unsigned N,
       [in,  size_is(N), align(4)] short inData[],
       [out, size_is(N), align(4)] short outData[]);
     };
};
```

An application cannot directly use the VIDL specification for the service offered by the `IG711` interface. The VIDL compiler does, however, process the VIDL specification and generate a header file `EXAMPLES_IG711.h`. The header provides definitions of the interface that can be used in C, C++, or assembly language source modules to access the interface.

Assuming the VIDL specification (Listing 2-2 on page 2-9) is held in a file ig711.idl, invoke the VIDL compiler that is appropriate for your target processor:

```
vidlblkfn ig711.idl
vidl218x ig711.idl
vidl219x ig711.idl
vidlts ig711.idl
vidl21k ig711.idl
```

The compiler generates the EXAMPLES_IG711.h file. The generated header file can be included by any application or component that wishes to use the interface.

Although "::" separates the namespace and simple name parts of a full interface name, you must use an underscore to separate the same elements in file names.

The VIDL compiler also produces a set of .HTML files, which document the interface and combine information from the VIDL statements and any auto-doc comments. These files are stored in an html subdirectory. If you open the html\EXAMPLES_IG711.html file, a page similar to that in Figure 2-1 on page 2-18 appears.

Properties

In some cases, component developers need to enable a component to have certain publicly visible properties that can be accessed directly via an interface pointer. The VIDL language provides interface data members to be specified to allow those properties to be defined and to control the access allowed to such properties. The values of these properties can then be read or written using automatically-generated access methods inlined by the C or C++ compiler to minimize any overheads.

Consider the following interface definition:

```
interface IMultiRateFilter extends IBase
{
        // Rate is a read/write property
        [ get, set ] int Rate

// ErrorCode is a read-only property.
[ get ] MRESULT ErrorCode;

// Statistics is a read-only array
[ get ] short Statistics[3];
};
```

The use of the `[get]` attribute informs the VIDL compiler that an access method to obtain the value of the property should be generated; and the use of the `[set]` attributes informs the VIDL compiler that an access method to change the value of the property should be generated.

The C/C++ mapping for properties is an *inlined* method of the interface with `get` or `set` prefixed to the property's name. In the case of `array-type` properties, an index is included as a parameter of the access method. For assembly language the mapping of properties is a convenience macro that indexes into the interface to access the specified property.

For example the C++ mapping of the above interface is:

```
interface IMultiRateFilter : public IBase
{
     protected:
            int m_Rate;
            MRESULT m_ErrorCode;
            short m_ Statistics [3];
      public:
            inline int getRate() { return m_Rate; }
            inline void setRate( int _val ) { m_Rate = val; }
```

```
            inline MRESULT getErrorCode() { return m_ErrorCode; }
            inline short getStatistics ( int _index) { return
m_Statistics[ _index ]; }
};
```

Accessing the properties of an interface is syntactically the same as invoking any of the methods of the interface. In C++ the value of the `Rate` property can be obtained using an expression such as `imrf->getAccess()`, whereas in C or assembler the corresponding expression to obtain value of the property is `ADI_IMultiRateFilter_getRate(imrf);`

The significant difference between accessing the value of properties and invoking the methods of an interface is that the access to the properties can be inlined automatically by the C/C++ compiler and assembler. Therefore, the properties of an interface provide an efficient means to communicate configuration and status type data between a component and its client.

By allowing protected data members in an interface, client use of the access methods result in a single memory access in application code rather than calling a function. Compared to a method call, access to a property is faster and results in using less lines of code in both the component and the application.

As a result, using properties can significantly reduce the costs of programming techniques, such as event polling. At the same time, they provide very low cost access to status information. The setting of the value of a property can also provides a very inexpensive way of providing configuration information to a component. By definition a property provides only read and/or write access to an interface data member. If for any reason accessing a property may need to trigger some action or side effect within the component, then a standard method should be used rather than a property.

Since properties are protected data members of an interface they become protected members of any component that implements the interface, either directly or indirectly. Unlike interface methods properties are not declared by an interface but are defined as properties of the interface.

It is possible for a component to have more that one property with the same name since properties are defined as an element of an interface. For example, if interfaces `IB` and `IC` both extend `IA` and `IA` defines the property `PropA`, then a component implementing `IB` and `IC` will have two `PropA` properties (one for its implementation of `IB` and one for `IC`).

Depending on the component under consideration, the use of properties may or may not be the best way to communicate values to or from the component. In any case the component developer should ensure that the intended behavior of the interface properties is fully documented.

Interface Properties

It may useful to enable a component to have certain publicly visible properties that can be accessed directly via an interface pointer. The VIDL language provides interface data members which can be specified to allow those properties to be defined and to control the access allowed to such properties. These values of these properties can then be read or written using automatically-generated access methods that are inlined by the C or C++ compiler to minimize any overhead.

Consider the following interface definition:

```
interface IMultiRateFilter extends IBase
{
            // Rate is a read/write property
            [ get, set ] int Rate;

            // Rate is a read/write property
            [ get, set ] int Rate;
```

```
            // ErrorCode is a read-only property.
            [ get ] MRESULT ErrorCode;

             // Statistics is a read-only array
             [ get ] short Statistics[3];
};
```

The use of the `[get]` attribute informs the VIDL compiler that an access method to obtain the value of the property should be generated. The use of the `[set]` attributes informs the VIDL compiler that an access method to change the value of the property should be generated.

The C/C++ mapping for properties is simply an inlined method of the interface with `get` or `set` prefixed to the properties name. In the case of `array-type` properties, an index is included as a parameter of the access method. For assembly language the mapping of properties is simply a convenience macro that indexes into the interface to access the specified property.

For example the C++ mapping of the above interface is:

```
interface IMultiRateFilter : public IBase
{
   protected:
        int m_Rate;
        MRESULT m_ErrorCode;
        short m_ Statistics [3];
   public:
        inline int getRate() { return m_Rate; }
        inline void setRate( int _val ) { m_Rate = val; }
        inline MRESULT getErrorCode() { return m_ErrorCode; }
        inline short getStatistics ( int _index) { return
m_Statistics[ _index ]; }
};
```

Accessing the properties of an interface is syntactically the same as invoking any of the methods of the interface. Hence in C++ the value of the `Rate` property can be obtained using an expression such as `imrf->getAccess()`, whereas in C or assembler the corresponding expression to obtain value of the property is `ADI_IMultiRateFilter_getRate(imrf);`

The significant difference between accessing the value of properties and invoking the methods of an interface is that the access to the properties can be inlined automatically by the C/C++ compiler and assembler. The properties of an interface therefore provide a very efficient means to communicate configuration and status type data between a component and its client.

By allowing protected data members in an interface, client use of the access methods result in a single memory access in application code rather than calling a function. Compared to a method call, access to a property is faster and results in less code in both the component and the application.

Properties can therefore significantly reduce the costs of programming techniques such as event polling and also provide very low cost access to status information. The setting of the value of a property can also provides a very inexpensive way of providing configuration information to a component. By definition a property provides only read and/or write access to an interface data member. If there is a possibility that accessing a property may need to trigger some action or side effect within the component then a standard method should be used rather than a property.

Since properties are protected data members of an interface they become protected members of any component that implements the interface, either directly or indirectly. Unlike interface methods properties are not declared by an interface but are defined as properties of the interface.

It is possible for a component to have more that one property with the same name since properties are defined as an element of an interface. For example, if interfaces IB and IC both extend IA and IA defines the property PropA, then a component implementing IB and IC has two PropA properties (one for its implementation of IB and one for IC).

Depending on the component under consideration, the use of properties may or may not be the most desirable way to communicate values to or from the component. In any case the component developer should ensure that the intended behavior of the interface properties is fully documented.

Creating Interface Implementation

Once we have created the VIDL interface definition and generated the interface header file, VCSE can automatically create the framework for a component that can be used to implement the interface. The VIDL needed to create the framework of an implementation of the EXAMPLES::IG711 interface is shown in Listing 2-3 on page 2-17.

Listing 2-3. Component Implementing EXAMPLES::IG711 Interface

```
#include “ig711.idl”
namespace EXAMPLES {
   /**
    * The CULaw component provides an implementation of the
    * EXAMPLES::IG711 interface and implements the mu-law
    * encoding as specified in the ITU B.711 specification.
    */
[
    category(“Examples\Telephony”),
    company(“Analog Devices, Inc”),
    title(“Example component for G711 which implements mu-law
    encoding”)]
```

Figure 2-1. Examples::IG711 Interface Documentation Files

```
    component CULaw implements IG711;
};
```

To generate the framework needed to implement this component in C, issue a command that corresponds to your target processor family:

```
vidlblkfn g711.idl
vidl218x g711.idl
vidl219x g711.idl
vidlts g711.idl
vidl21k g711.idl
```

The VIDL compiler processes the supplied VIDL and generates a set of C files needed to create the component. A C based implementation is the default. To generate a set of C++ files, add the `-c++` switch to the command line; add the `-asm` switch to generate the methods of the component in assembly language. The set of generated C files is outlined in Table 2-1.

Table 2-1. EXAMPLES::IG711 Interface Implementation Files

File Name	Description
`EXAMPLES_CULaw.c`	Contains the code needed to create and destroy the component.
`EXAMPLES_CULaw.h`	Contains the definition of the structure that holds the instance data for the component.
`EXAMPLES_CULaw.rbld`	Deleting this file triggers a complete rebuild within a VisualDSP++ project that creates the component.
`EXAMPLES_CULaw.xml`	Controls the packaging of a component when it is being prepared for distribution.
`EXAMPLES_CULaw_factory.h`	Contains the prototypes for the `Create` and `Destroy` functions for the component.
`EXAMPLES_CULaw_methods.c`	Contains the method functions used to actually implement the interfaces.

A component is built as a library of objects; the name of the library should be EXAMPLES_CULaw.dlb to avoid clashes with other components. The generated files are complete and ready to be compiled. The library can be created by issuing a command that corresponds to your design processor family, such as:

```
ccblkfn -build-lib -o EXAMPLES_CULaw.dlb EXAMPLES_CULaw.c
EXAMPLES_CULaw_methods.c
```

```
cc219x -build-lib -o EXAMPLES_CULaw.dlb EXAMPLES_CULaw.c
EXAMPLES_CULaw_methods.c
```

```
cc218x -2184 -build-lib -o EXAMPLES_CULaw.dlb EXAMPLES_CULaw.c
EXAMPLES_CULaw_methods.c
```

The command creates a library containing the executable code for the component. A distributed component is expected to provide header files, documentation files, and other files along with the actual library. The packaging of a component is primarily controlled by the contents of the .XML file and the Component Packaging Wizard used to create the distribution package.

In order to effect the actual implementation of the component, modify two of the files, EXAMPLES_CULaw.h and EXAMPLES_CULaw_methods.c.

C Component Instance Structure

The VCSE Component Model is designed to enable each component to have more than one instance simultaneously. The data associated with each instance of the component is known as the *instance data* and is held in a structure that is defined by the VIDL compiler. Each instance of the component has its own copy of the instance data. When the implementation language is C, the file `EXAMPLES_CULaw.h` contains the definition of the instance structure as follows.

```
component  EXAMPLES_CULaw {
  struct EXAMPLES_IG711_methods *EXAMPLES_IG711;

  VCSE_IBase_ptr m_penv;
  VCSE_HANDLE m_token;
  VCSE_ADDRESS m_addr;
  //##########################################################
  //####SCF Start of component private members, EXAMPLES_CULaw

  // Any user specific members for instance data of the compo-
  // nent, CULaw, should be inserted here

  //####ECF End of component private members, EXAMPLES_CULaw
  //#######################################################
};
```

Note that `component` is a macro defined as `struct` when the VIDL compiler target language is C. This increases the readability of generated code by making it as close as possible to the VIDL component definition.

The name of the structure is generated from the combination of the namespace and the component name separated by an underscore. This is the standard way to make C names unique. Normally, a name is also appended to indicate the function or use of the name, such as the name of a method function. The VCSE framework controls and uses the fields at the start of the component structure. Any data needing different values for each instance is defined by replacing the comment

```
  // Any user specific members for instance data of the compo-
  // nent, CULaw, should be inserted here
```

with the data definitions.

Any changes made within the

```
//##################################################################
```

markers are automatically preserved by the VIDL compiler and restored when it regenerates the implementation shell. For example, if you want to add an `int` field to hold the count of the number of times the `Compress` method is called, change the above block as follows.

```
component  EXAMPLES_CULaw {
  struct EXAMPLES_IG711_methods *EXAMPLES_IG711;

  VCSE_IBase_ptr m_penv;
  VCSE_HANDLE m_token;
  VCSE_ADDRESS m_addr;
  //##################################################################
  //####SCF Start of component private members, EXAMPLES_CULaw

  /* count the no. of times Compress is invoked */
  int m_CompressCt;
  /* count the no. of times Decompress is invoked */
  int m_DecompressCt;

  //####ECF End of component private members, EXAMPLES_CULaw
  //##################################################################
};
```

The factory functions, which are used to create and destroy instances of the component, are generated in the file `EXAMPLES_CULaw.c` and are described in "Component Factory Header File" on page 2-59.

When an instance of a component is created, an instance of the component structure has to be allocated and initialized. The component `Create` function generated by the VIDL compiler uses the passed `IMemory` interface to allocate this instance structure, using a request for instance memory with default alignment and of any type and any lifetime. For details of the memory allocation requests, see “IMemory Interface” on page 3-2.

C Interface Method Functions

The file `EXAMPLES_CULaw_methods.c` contains the definitions of the two methods defined in the interface `IG711` (see Listing 2-2 on page 2-9). The principal modification is to provide the actual body of these two functions. The code generated by the compiler for the `Compress` method is as follows.

```
static __VCSEMETHOD VCSE_MRESULT EXAMPLES_CULaw_Compress(
  VCSE_IBase_ptr base,
  unsigned       int N,
  const short    inData[N],
  short          outData[N])
{
  __ASSIGN_THIS_POINTER(__this,EXAMPLES_CULaw);
  __builtin_aligned(inData,4);
  __builtin_aligned(outData,4);
  //###############################################################
  //####SCF Start of interface member function, EXAMPLES_CULaw_Compress
  {
  // Any user specific code needed within the interface member
  // function, EXAMPLES_CULaw_Compress, should be inserted here.
  return (VCSE_MRESULT)MR_OK;
  }
  //####ECF End of interface member function, EXAMPLES_CULaw_Compress
  //###############################################################
}
```

The main points to be noticed:

- The first parameter passed to each method is a pointer to the component instance data structure. By convention, it is assigned to a variable called `__this`, whose type is a pointer to the component structure (using the macro `__ASSIGN_THIS_POINTER`).
- When the interface definition marks the array parameters with the `align` attribute, this information is supplied to the compiler using the `__builtin_aligned` intrinsic.
- The actual body of the method is placed within the user-modifiable block markers.
- The method function is defined as `static` and, therefore, cannot be directly referenced outside this file. Access to the methods of an interface is always indirect via the interface instance pointer.

A possible implementation of the `Compress` method is as follows.

```
static __VCSEMETHOD VCSE_MRESULT EXAMPLES_CULaw_Compress(
  VCSE_IBase_ptr base,
  unsigned       int N,
  const short    inData[N],
  short out      Data[N])
{
  __ASSIGN_THIS_POINTER(__this,EXAMPLES_CULaw);
  __builtin_aligned(inData,4);
  __builtin_aligned(outData,4);
  //##################################################################
  //####SCF Start of interface member function, EXAMPLES_CULaw_Compress
  {
    int      calcVal;
    int      seg;
    unsigned i;
    short    inVal;
    /* Increment the count in the inst.data */
```

```
    __this->m_CompressCt++;

    for(i = 0; i < N; ++i)

    {
      /* Handle negative input with sign bit below */
      inVal = inData[i];
      calcVal = abs_(inVal);
      calcVal += 33;
      calcVal = min_(calcVal, 8159); /* bound input */
      seg = signbits_(calcVal);
      calcVal <<= seg;                /* normalize input        */
      calcVal ^= 0x4000;              /* strip off the high bit */
      calcVal >>= 10;                 /* get the position       */
      if (inVal < 0)          /* add the sign bit to the output */
         calcVal |= 0x80;
      seg = 9 - seg;            /* we need to change segment    */
      calcVal |= (seg << 4);    /* add the segment ID           */
      outData[i] = ~calcVal;    /* invert the output            */
    }
    return (VCSE_MRESULT)MR_OK;
  }
  //####ECF End of interface member function, EXAMPLES_CULaw_Compress
  //#################################################################
}
```

A component must always access its instance data via the `__this` pointer. It is valid for the component to read global data, but all normal data updates should be via the instance pointer.

C++ Interface Methods

When a C++ implementation is selected, the component is created as a C++ class whose members are the instance data, and the interface methods are the methods of the class. The component class is defined in a C++

namespace, which has the same name as the component name. This namespace is further embedded in C++ namespaces with the same name as the VIDL namespaces that the component is defined in. Hence, the component class for the `CULaw` component is effectively defined in C++ as:

```
namespace EXAMPLES {
     namespace CULaw {
          component CULaw {
          }
     }
}
```

Note that `component` is a macro defined as `class` when the VIDL compiler target language is C++. This increases the readability of generated code by making it as close as possible to the VIDL component definition.

The component `CULaw` is defined within an enclosing `EXAMPLES::CULaw` namespace to ensure that any global variables are defined in the `EXAMPLES::CULaw` namespace and, thereby, guarantee uniqueness between components. The factory functions that `Create` and `Destroy` the component are declared as friends of the component class in a C++ source file.

If a C++ implementation shell is generated using a command line that corresponds to your design processor family[1],

```
vidlblkfn -c++ g711.idl
vidl219x -c++ g711.idl
vidlts -c++ g711.idl
vidl21k -c++ g711.idl
```

then the interface `IG711` is defined as an abstract class derived from the `::VCSE::IBase` class as follows.

[1] There is no C++ support for ADSP-218x DSPs.

```
interface IG711 :
    public ::VCSE::IBase
{
public:
    virtual __VCSEMETHOD VCSE::MRESULT GetInterface(
            const VCSE::RefIID iid,
            VCSE::IBase_ptr *iptr) = 0;
    virtual __VCSEMETHOD VCSE::MRESULT Decompress(
            unsigned int N,
            const short *inData,
            short *outData) = 0;
    virtual __VCSEMETHOD VCSE::MRESULT Compress(
            unsigned int N,
            const short *inData,
            short *outData) = 0;
};
```

Note that `interface` is a macro defined as `class` when the VIDL compiler target language is C++. This increases the readability of generated code by making it as close as possible to the VIDL interface definition.

The component instance data is then defined as a class derived from the abstract classes, which represent the interfaces supported by the component, as follows.

```
namespace EXAMPLES {
      namespace CULaw {
            component CULaw:
                public::EXAMPLES::IG711
            {
            ...
            }
      }
}
```

The shell generated for the `Compress` method is:

```
namespace EXAMPLES {
  namespace CULaw {
  __VCSEMETHOD VCSE::MRESULT CULaw::Compress(
          unsigned int  N,
          const short   *inData,
          short         *outData)
 {
 __builtin_aligned(inData,4);
 __builtin_aligned(outData,4);
 //#############################################################
 //####SCF Start of interface member function, EXAMPLES::CULaw::Compress
 {
 //Any user specific code needed within the interface member
 //function, EXAMPLES::CULaw::Compress, should be inserted here.
 return (VCSE_MRESULT)MR_OK;
 }
 //####ECF End of interface member function, EXAMPLES::CULaw::Compress
 //#############################################################
}
…
```

The instance data within the method can be accessed directly since C++ uses the `this` pointer implicitly. For example,

```
  m_CompressCt++;  /* increment the count in the instance data */
```

Assembly Interface Methods

When an assembly implementation is selected, the methods are created as assembly based shells, while the factory functions that create and destroy the component are created as C functions that do not use the C run-time library. The names of the functions for the methods are the same as those used in the C implementation, but the assembly functions are defined as global since the method table is created within the C factory function file.

If an assembly implementation shell is generated using one of the following command lines,

```
vidlblkfn -asm -trace g711.idl
vidl219x -asm -trace g711.idl
vidl218x -asm -trace g711.idl
vidlts -asm -trace g711.idl
vidl21k -asm -trace g711.idl
```

then the shell generated for the `Compress` method is:

```
////////////////////////////////////////////////////////////////
// VCSE_MRESULT EXAMPLES_CULaw_Compress(
//        VCSE_IBase_ptr base,
//        unsigned int N,
//        const short inData[N],
//        short outData[N])

__STARTFUNC(_EXAMPLES_CULaw_Compress, __GLOBAL)

//##############################################################
//####SCF Start of interface member function, EXAMPLES_CULaw_Compress
__LINK(0)
__DEBUG_TRACE_ENTRY('EXAMPLES_CULaw_Compress')

//Any user specific code needed within the interface member
//function,EXAMPLES_CULaw_Compress, should be inserted here.

__DEBUG_TRACE_EXIT('EXAMPLES_CULaw_Compress')
__RETURN(MR_OK)

//####SCF End of interface member function, EXAMPLES_CULaw_Compress
```

```
//#############################################################################
__ENDFUNC(_EXAMPLES_CULaw_Compress)
```

It should be noted that the function entry trace `__DEBUG_TRACE_ENTRY('EXAMPLES_CULaw_Compress')` and corresponding exit trace are only generated if `-trace` is used when the assembler source is generated for the first time because the line is within the user-modifiable block.

VCSE provides various macros for use within an assembly source file by `#include <vcse.h>`, as described in "VCSE Assembler Macros" on page A-1.

Advanced Component Construction

Method Language Selection

Suppose you have a fast and efficient Assembler routine for an algorithm which you would like to form the core of a VCSE component. However, along with the algorithmic core there are a number of configuration tasks for which, as they amount to a small fraction of the processing time taken by the algorithmic core itself, it would be more cost efficient to write in C.

The `language` statement, which can appear within the component declaration block, enables the target language of one or more interface methods to be different from the default, as dictated by the command line switches.

For example, if the majority of the component `CAlg`'s methods are to be implemented in C, and it is required that `foo` is implemented in Assembler then the VIDL syntax

```
component CAlg implements IBar {
      language foo is asm;
};
```

could be used to override the default behavior of the compilation.

The only limitation to using this approach is that at present it is confined to mixing C and Assembler code; the option to mix C or Assembler with C++ is not currently supported.

For a detailed description of the syntax for the `language` statements see "language statement" on page 4-66.

Method Placement

On occasion, developers need to control the placement of methods within memory in order to obtain the best possible performance for an application. To achieve this it is necessary that developers specify non-default section names for these methods. In line with the C/C++ compilers, the VIDL compiler provides the place statement to allow a user-defined section name to be supplied for any method.

This is achieved through the introduction of the `place` statement, which can be added to the component declaration body of the VIDL syntax. For example, the following syntax

```
component CAlg implements IBar {
      place foo in l2_cache;
};
```

instructs the VIDL compiler to add `__VCSESECTION_DEF(l2_cache)` to the declaration of the method `foo`:

```
VCSE_MRESULT __VCSESECTION_DEF(l2_cache) ACME_Calg_foo( … ) {  …
}
```

The `__VCSESECTION_DEF(S)` macro simply maps to `section(s)`, the syntax used by the C/C++ compilers. A similar approach is applied to Assembler methods.

The `place` statement *must* be used in conjunction with an `LDF` file tailored for your application's needs.

For a detailed description of the syntax for the `place` statements see "place statement" on page 4-65.

Distinct Components

When generating the source code for a component that implements multiple interfaces the default behavior of the VIDL compiler is to generate a single implementation of a method from one interface whose name and parameter list are identical to those in another interface.

For example, if component `C1` implements `IA` and `IB`, both of which have a method `foo` that takes no parameters, then the same function will be called irrespective of the interface used.

The VIDL compiler provides the `distinct` component attribute, which will override the behavior described above, for example:

```
[distinct, …] component C1 implements IA, IB;
```

Applied to component `C1` above, it would result in two implementations of `foo` one that would be invoked via the `IA` interface and another via `IB`. The distinct implementation of `foo` is transparent to the user of a component `C1`.

In addition to separate implementations of a method, the component developer can determine whether the separate implementations will have access to the same or different instance data. The default behavior is that they have access to separate instance data. This feature can be overridden by using of the `common` component attribute,

```
[distinct,common, …] component C1 implements IA, IB;
```

and in which case the two different implementations of `foo` share the same instance data.

For further details on the use of the `distinct` and `common` component attributes, see pages "common Attribute" on page 4-55 and "distinct Attribute" on page 4-56.

The use of the `distinct` attribute results in all methods of all interfaces being given distinct implementations. However, there may be times when this is not the desirable behavior: maybe only one or two methods are required to have distinct implementations. Furthermore, the distinct implementation may not be required for all interfaces.

For example, extending the above example so that `C1` implements a further interface `IC` that also supplies an identical method `foo`, and all interfaces have an identical bar method, we may choose to have a separate implementation of `foo` for interface `IC`, and yet retain a single implementation for interfaces `IA` and `IB`. Additionally, we may choose a single implementation of bar for all three interfaces.

The `distinct` statement enables the arbitrary selection of which interface methods are to be given a distinct implementation. For example, the revised behavior in the preceding paragraph can be implemented using the following VIDL syntax:

```
component CAlg implements IBar {
      distinct IC::foo;
};
```

As it applies to data access, all distinct methods operate on distinct data while all common methods operates on the same common data. To force all implementations to operate on the same data, the `common` component attribute can be used as previously.

Documenting Components

The use of VIDL to define component interfaces that each component supports allows a lot of information about a component to be specified formally. However, a component also needs to provide information that describes how it is to be used and the kind of operating environment that it expects. This is the “operational specification” for the component, and it is provided by auto-doc comments embedded in the VIDL.

In the example below, the auto-doc comment provides a description of the `CDSM2150F5V` component. Notice that the auto-doc comment body contains HTML items, such as paragraph tags and an HTML link.

```
namespace EXAMPLES {
  /**
  * This component is an ADSP-BF535 implementation of the
  * EXAMPLES::IFlash interface for the ST DSM2150F5V DSP System
  * Memory device. This device is used on the ADSP-BF535 EZ-Kit
  * board, but it is suitable for many other ADI processors too.
  *  Consult the
  * <a href="http://us.st.com/stonline/books/pdf/docs/8461.pdf">
  * ST data sheet</a> for full details of this part <p>.
  **/

[
  title("Flash Programmer for ST DSM2150F5V DSP System Memory"),
  info("www.analog.com"),
  category("Example\Non-algorithm"),
  company("Analog Devices, Inc"),
  version(1.0.0),
```

```
  aggregatable
]
  component CDSM2150F5V implements EXAMPLES::IFlash {};
```

Auto-doc supports tags that allow specific features of an interface (or a method) to be clearly documented and tabulated in HTML. Each tag is prefixed with an `@` character. The supported tags include `@param`, `@return`, `@example`, and `@keyword`. In the following example, the auto-doc comment provides a summary description of the `SetDeviceAddresses` method from the `IFlash` interface and additional information on each of its parameters.

```
/**
* Tells the IFlash component how the flash device's sectors
* are mapped into the general address space. It also ensures
* the device is in its default state. This method must be
* invoked before any others and may only be invoked once.
*
* @param NumRanges Specifies the number of address ranges in
*        the address range table. Must be a positive value.
*
* @param AddressRanges Array of address range descriptors.
*        Each descriptor specifies a starting address in the
*        DSP's memory map, the length in bytes of the address
*        range andinformation about the substructure (if any)
*        of the range.The elements of the array need not be in
*        any particular order.
*
* @return MRESULT MR_IFLASH_BAD_RANGES, MR_IFLASH_INCOMPLETE_RANGES,
*       MR_IFLASH_RANGES_ALREADY_SET or MR_OK (see IFlash_Results).
**/
MRESULT SetDeviceAddresses(
   [in] int NumRanges,
   [in,size_is(NumRanges)] AddressRange AddressRanges[] );
```

When you compile the VIDL definition for a component, the compiler generates a set of `.HTML` files that document the component and all the referenced interfaces. The generated HTML documentation for the component and all its supported interfaces includes a table of contents as well as an automatically generated index.

If you open the file `html\EXAMPLES_CDSM2150F5V.html` in a browser, click on the **Index** button, select the `SetDeviceAddresses` method entry, and then the item `EXAMPLES::IFlash interface — SetDevicesAdresses`, a screen similar to that in Figure 2-2 appears.

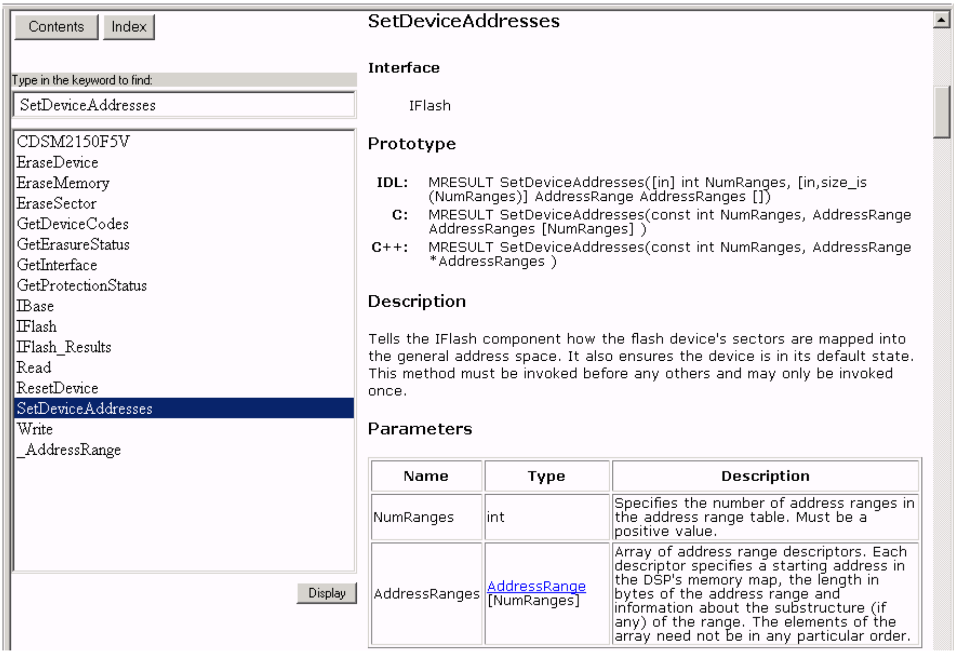


Figure 2-2. Examples::IG711 Interface Documentation Files

Testing Components

The VIDL compiler provides aids for testing a component in two separate ways. It can generate a test harness program that will invoke every method of every interface. This test harness can be modified to effect testing, as needed. In addition the VIDL compiler can generate a test shell component that can be used automatically to validate the actions of the component or of the application that invokes the component.

Testing Harnesses

When generating the implementation shell of a component, you can request some tracing code be added to the generated methods by specifying the `-trace` switch to the VIDL compiler. For example, if you specify one of the following commands,

```
vidlblkfn -trace g711.idl
vidl219x -trace g711.idl
vidl218x -trace g711.idl
vidlts -trace g711.idl
vidl21k -trace g711.idl
```

the generated shell for the `Compress` method contains the code:

```
static __VCSEMETHOD VCSE_MRESULT EXAMPLES_CULaw_Compress(
   VCSE_IBase_ptr base,
   unsigned       int N,
   const short    inData[N], short    outData[N]
{
   __ASSIGN_THIS_POINTER(__this,EXAMPLES_CULaw);

   __builtin_aligned(inData,4);
   __builtin_aligned(outData,4);
   __DEBUG_TRACE_ENTRY("EXAMPLES_CULaw_Compress");
   //##################################################################
```

```
    //####SCF Start of interface member function, EXAMPLES_CULaw_Compress
    {
       // Any user specific code needed within the interface
       // member function, EXAMPLES_CULaw_Compress, should be
       // inserted here.

       __DEBUG_TRACE_EXIT("EXAMPLES_CULaw_Compress");
       return (VCSE_MRESULT)MR_OK;

    }
    //####SCF End of interface member function, EXAMPLES_CULaw_Compress
    //####################################################################
}
```

The default __DEBUG_TRACE_ENTRY and __DEBUG_TRACE_EXIT macros use the VCSE_printf function to display a message on entry to and exit from the method.

If you generate an initial set of component shell source files without specifying -trace but specify -trace in a subsequent call on the compiler, the compiler only adds the entry trace macro since the exit trace is within a user defined block.

The VCSE support library contains a specialized version of printf called VCSE_printf (and a corresponding VCSE_fprintf), which supports a limited number of format specifications but can be used independently of the standard C/C++ run-time library. The only format specifications supported are %s, %x, %p, %c, %d, and %i. There is no support for field widths or padding either.

The VIDL compiler also generates a simple test program for a component if you supply the -harness switch. The generated test program for the CULaw component would be EXAMPLES_CULaw_test.c. The test program creates an instance of the component and then invokes each of its methods before destroying the component instance.

A VCSE component is not expected to allocate memory directly itself; instead, it uses an `IMemory` interface instance supplied by the application to allocate memory on request. The generated test program includes the source for a simple memory allocation component `VCSE::CSimpleMemory`, which allocates memory from the system heap.

Once you have created the component library, compile and build the test program using the appropriate command for your target processor family:

```
ccblkfn EXAMPLES_CULaw_test.c -L. EXAMPLES_CULaw.dlb -lvcseBF532
cc219x EXAMPLES_CULaw_test.c -L. EXAMPLES_CULaw.dlb -lvcse219x
cc218x -2184 EXAMPLES_CULaw_test.c -L. EXAMPLES_CULaw.dlb -lvcse218x
```

The `-L.` switch specifies that the current directory is to be searched for the specified libraries, such as the component library `EXAMPLES_CULaw.dlb`. The `-lvcseBF532`, `-lvcse219x`, or `-lvcse218x` is needed to enable the linker to find the VCSE support library for the respective processor family, as summarized in Table 2-2.

Table 2-2. VCSE Support Libraries

Processor Family	Switch	VCSE Library
ADSP-218x[1]	`-lvcse218x`	`libvcse218x.dlb`
ADSP-219x[2]	`-lvcse219x`	`libvcse219x.dlb`
ADSP-BF531	`-lvcseBF532`	`libvcseBF532.dlb`
ADSP-BF532	`-lvcseBF532`	`libvcseBF532.dlb`
ADSP-BF533	`-lvcseBF532`	`libvcseBF532.dlb`
ADSP-BF535	`-lvcseBF535`	`libvcseBF535.dlb`

1 If the component must avoid registers reserved for auto-buffering, use `-lvcse218xab.dlb`.
2 If the component is compiled for the ADSP-2192-12 DSP, use `-lvcse219x_type32aworkaround` to avoid ADSP-2192-12 DSP hardware anomalies.

When adding components to projects using the VisualDSP++ Component Manager, the necessary libraries are automatically added to the project by the Component Manager.

Test Shell Components

The VIDL test shell support provides for the nonintrusive testing of a component that is either under development, or being used in its targeted application. To achieve this the VIDL compiler can be instructed to generate a component that is identical to the component under test except that the interface methods of the test shell component contain code prior to and following a call to the actual method of the component under test. The content of this code is determined by the test attributes specified in the VIDL test shell syntax. User-modifiable blocks are also provided to allow additional testing to be performed.

VCSE provides automatically-generated test shell support for validation of method parameters and for monitoring of resources used, both in terms of cycles and memory. This can be achieved without adding a single line of code to the component under test.

The VIDL Compiler generates a test shell component that can be wrapped around the actual component and supplies identical methods on the same interfaces. The name of the test shell component is formed by adding the `_VCSETEST` suffix to the name of the actual component. The only other differences are that the `Create` function creates an instance of the actual component as well and the test shell methods simply forward the call to the required method of the actual component. Validation checks and performance analysis are performed prior to and following the forwarded call as detailed in the VIDL syntax for test shell components.

To use the test shell component simply replace the actual component in the client code. Macros are defined in the test shell's factory header so that the test component can act as the component itself with minimal change to the client code. Consider the following simplified example:

```
#ifdef __USE_TEST_COMPONENT__
#include "ADI_C1_VCSETEST_factory.h"
#else
#include "ADI_C1_factory.h"
#endif

:
ADI_C1_Create(NULL,ADI_TOOLS_IA_IID,&pIA,… )
:
int n=3;
short data[3], output[3];
pIA->foo(n,m,data, output);

:
ADI_C1_Destroy(pIA)
```

Compiling the above with default options results in the actual component only being used, while adding `-D__USE_TEST_COMPONENT__` to the C++ compiler command line results in the test shell component being used as a wrapper around the actual component. This is the approach used in the test harness code, generated with the `-harness` command line option.

To ensure that `n` and `m` are limited to the range 0-10, for example, and to check that the bounds for `data` are not exceeded, as well as perform timing analysis on the `foo` method, involves using the following VIDL file:

```
#include "ADI_C1.idl"
namespace ADI {
      testing ADI::TOOLS::C1 {
[            timing,in_assert(n>=0 && n<=10)]
            foo( [array_check] data );
```

```
        };
};
```

where `timing`, `in_assert` and `array_check` are termed test shell attributes. For a more detailed description of the VIDL syntax for test shell components and the various attributes that can be used see "Generated Test Shells" on page 4-77.

Test shell components are only generated in C or C++ depending on the command line options. For components partially or fully implemented in Assembler, C is used for the test shell.

Since the test shell components use the same interfaces as the actual components, method call optimizations are inhibited in the executable for the interface methods under test.

Description of Generated Test Shell Component Files

The files generated by the VIDL Compiler when test shell syntax is present are similar to those generated for the actual component, except that no Assembler files are generated. The names of the files are listed in Table 2-3:

Table 2-3. Generated Test Shell Files Names

Type	Actual Component	Test Shell Component
Component header	`ADI_C1.h`	`ADI_C1_VCSETEST.h`
Component Management		
C	`ADI_C1.c`	`ADI_C1_VCSETEST.c`
C++	`ADI_C1.cpp`	`ADI_C1_VCSETEST.cpp`
Component Methods		
C	`ADI_C1_methods.c`	`ADI_C1_VCSETEST_methods.c`

Table 2-3. Generated Test Shell Files Names

Type	Actual Component	Test Shell Component
C++	ADI_C1_methods.cpp	ADI_C1_VCSETEST_methods.cpp
ASM	ADI_C1_methods_asm.asm	ADI_C1_VCSETEST_methods.c
Component Factory Header	ADI_C1_factory.h	ADI_C1_VCSETEST_factory.h

Supporting Tools for Test Shell Components

VCSE provides built in support within both a test shell component and the test harness source files to enable the various validation checks to be carried out, and to communicate diagnostic messages to the component tester, with little or no user intervention. This support is provided in three ways:

- Macros and library methods,
- A simple standard report interface, `VCSE::ITestReport`, and
- A simple lightweight component. `VCSE::CTestReport`, that implements `ITestReport`.

Macros and Library Methods for Component Validation

The generated code in a test shell component makes use of a number of macros that, for the most part, are mapped directly to function calls within the VCSE library. The prototypes for these functions are provided in `VCSE.h`:

```
char* VCSE_TestAlgoModel(int* state, int cur_operation);
int* VCSE_GetStackEnd();
VCSE_MRESULT VCSE_FillStack(int* end);
VCSE_MRESULT VCSE_MeasureStack(int* end, int* size);
```

And the following is declared for 21xx timing calculations only:

```
unsigned long VCSE_emuclk();
```

All macros are defined within the “Component Global Settings” user-modifiable block at the top of the test shell component methods file, so that they can be tailored to the individual needs of the component tester. Calls to these methods and others are automatically inserted by the VIDL compiler in response to the test attributes specified in the VIDL file defining the test shell component.

The definition and intended use of these macros are as follows.

__VCSE_STACKVARS(E,U)

The `__VCSE_STACKVARS` macro is provided when the `stack_usage(custom)` attribute is used. It is always empty. In the body of a test shell method it is passed two parameters, `__vcse_stack_end` and `__vcse_stack_use`, which must be declared within the macro definition. These variables are declared without the use of a macro when the `stack_usage(default)` attribute is used.

__VCSE_STACKFILL(E)

The `__VCSE_STACKFILL` macro provides a means of filling the stack with a pattern prior to the forwarding call to the interface method of the component under test. It is passed the `__vcse_stack_end` variable.

The macro is empty when the `stack_usage(custom)` attribute is used, otherwise it is defined so as to invoke the `VCSE_FillStack` function within the VCSE library:

```
#define __VCSE_STACKFILL(E)  VCSE_FillStack(E)
```

__VCSE_STACKUSE(E,ADDR))

The __VCSE_STACKUSE macro provides a means of measuring the amount of stack used during the forwarding call to the interface method of the component under test. It is passed the __vcse_stack_end variable. It is also passed the address of the __vcse_stack_use variable to enable a value for the amount of stack used to be assigned.

The macro is empty when the stack_usage(custom) attribute is used, otherwise it is defined so as to invoke the VCSE_MeasureStack function within the VCSE library:

```
#define __VCSE_STACKUSE(E,U) VCSE_MeasureStack(E,U)
```

Macros and Library Methods for Reporting Messages to the User

Diagnostic and error messages are reported to the component tester via a single macro, __VCSE_SHELL_REPORT, which can either be used as generated by the VIDL Compiler or can be customized to suit the individual requirements of the component tester. The __VCSE_SHELL_REPORT macro generated by the VIDL Compiler is placed in the "Component Global Settings" user-modifiable block and is defined as:

```
#define __VCSE_SHELL_REPORT(RPTNO,...) \
   if ( p_VCSE_ITestReport != NULL ) { \
       char *__VCSE_line; \
       if ( RPTNO ) \
         __VCSE_line = VCSE_ShellMessage(RPTNO,__VA_ARGS__);  \
   else \
         __VCSE_line = VCSE_sprintf(__VA_ARGS__);  \
   VCSE_ITestReport_AddString(p_VCSE_ITestReport,__VCSE_line); \
   } else {      \
       if ( RPTNO ) \
         VCSE_PrintShellMessage(RPTNO,__VA_ARGS__);  \
   else \
```

```
            VCSE_printf(__VA_ARGS__);    \
}
```

This macro provides for one of four approaches to report the required information to the user, depending on whether use is made of the RPTNO parameter and or the VCSE::ITestReport standard interface. For a full description of the VCSE::ITestReport interface see "ITestReport Interface" on page 3-39.

The RPTNO parameter is an enum VCSE_Shell_MsgNo_e value, which identifies the type of the message to be reported. In addition a variable number of parameters can be passed to the macro to qualify the messages' content.

At is most complex the macro invokes the VCSE_ShellMessage library function to generate a predefined message based on the enum VCSE_Shell_MsgNo_e value and the appropriate number of arguments corresponding to the required message. This message is then passed to the AddString method of the VCSE::ITestReport interface. The following table describes the predefined message format strings and the arguments required for the appropriate enum VCSE_Shell_MsgNo_e value:

Figure 2-3. Predefined Shell Messages

enum VCSE_Shell_MsgNo_e value	**Message**
VCSE_Shell_Msg_UNKNOWN	"this is an unknown - uninitialized error message.\n"
VCSE_Shell_Msg_ALIASING_CHECK	"%s : error : Parameter \"%s\" is aliased to parameter \"%s\".\n"
VCSE_Shell_Msg_ARRAY_CHECK	"%s : error : Input array parameter \"%s\" modified.\n"
VCSE_Shell_Msg_INVALID_BOUNDS	"%s : error : Array parameter \"%s\" has invalid bounds.\n"
VCSE_Shell_Msg_BOUNDS_EXCEEDED	"%s : error : Array parameter \"%s\" written beyond bounds.\n"
VCSE_Shell_Msg_IN_ASSERT	"%s : error : Assert expression failed upon entry to \"%s\".\n"

Figure 2-3. Predefined Shell Messages (Cont'd)

enum VCSE_Shell_MsgNo_e value	Message
VCSE_Shell_Msg_OUT_ASSERT	"%s : error : Assert expression failed upon exit from \"%s\".\n"
VCSE_Shell_Msg_USE_STATES	"%s : error : Component \"%s\" does not use required set of states.\n"
VCSE_Shell_Msg_INIT_STATE	"%s : error : Initial state not set for component \"%s\".\n"
VCSE_Shell_Msg_REQUIRED_STATE	"%s : error : Component \"%s\" not in required state \"%s\".\n"
VCSE_Shell_Msg_SET_STATE	"%s : error : Does not set required component state.\n"
VCSE_Shell_Msg_RESET_TO_STATE	"%s : error : Does not reset to required component state.\n"
VCSE_Shell_Msg_CLEAR_STATE	"%s : error : Does not clear component state.\n"
VCSE_Shell_Msg_TIMING	"%s : Cycles taken = %d.\n"
VCSE_Shell_Msg_STACK_USAGE	"%s : Stack used = %d.\n"
VCSE_Shell_Msg_MEM_USAGE	"%s\n"
VCSE_Shell_Msg_MEM_INVALID	"%s : error : Memory at 0x%x has corrupted \"%s\"\n"
VCSE_Shell_Msg_USER_DEFINED	""

The simplest use of the macro involves using neither a enum `VCSE_Shell_MsgNo_e` value or the `VCSE::ITestReport` interface, and simply passes the variable list of arguments to the `VCSE_printf` library function for immediate output to the console.

The prototypes for the various library methods invoked by the `__VCSE_SHELL_REPORT` macro are defined in `VCSE.h`:

```
char *VCSE_sprintf(char *fmt,...);
char *VCSE_ShellMessage(enum VCSE_Shell_MsgNo_e msgno, ...);
void VCSE_PrintShellMessage(enum VCSE_Shell_MsgNo_e msgno, ...);
```

The VCSE_PrintShellMessage function behaves in a similar fashion to VCSE_ShellMessage except that instead of returning a string it immediately outputs the message to the console.

An example use of the __VCSE_SHELL_REPORT macro, for the reporting of the number of cycles used by an interface method, is

```
__VCSE_SHELL_REPORT(
      VCSE_Shell_Msg_TIMING,
      "EXAMPLES_CULawc::Compress",
      __vcse_timing_result
      );
```

where it can be seen that two arguments are passed to the macro, in addition to the enum VCSE_Shell_MsgNo_e value, to identify the interface method and the number of cycles taken. The resulting message for a cycle count of 7980 is:

```
EXAMPLES_CULawc::Compress :  Cycles taken = 7980.
```

To make use of the VCSE::ITestReport interface the user can either use the simple VCSE::CTestReport component, described below, or develop their own. In either case a VCSE_WEAK pointer to the VCSE::ITestReport interface is declared immediately prior to the __VCSE_SHELL_REPORT macro and is set to NULL:

```
VCSE_WEAK VCSE_ITestReport_ptr p_VCSE_ITestReport = NULL;
```

To make use of a reporting component, the value of p_VCSE_ITestReport is required to be set in another module.

The VCSE::CTestReport Component

A simple implementation of the `VCSE::ITestReport` interface is provided in the VCSE library as the `VCSE_CTestReport` lightweight component. This component only implements the `AddString` and `AddStringWithNumber` methods to generate a linked list of messages. These messages are then written to the console by the `DumpAllMessages` method.

This component can be used in conjunction with the generated test shell components as a first step to validating a component under development. The following code listing details its use within a test application as given in the generated test harness program.

Listing 2-4. Example Use of the VCSE_CTestReport Component

```
// include test report component factory header:
#include "VCSE_CTestReport_factory.h"

// declare the VCSE::ITestReport interface pointer:
VCSE_WEAK VCSE_ITestReport_ptr p_VCSE_ITestReport;
:
// Assign the VCSE::ITestReport interface pointerl:
VCSE_CTestReport_Create(0,VCSE_ITestReport_IID,(::VCSE::IBase_pt
r *)&p_VCSE_ITestReport,0,0);
:
// perform tests using an unmodified generated test shell
component
:
// Output the results
p_VCSE_ITestReport->DumpAllMessages();
```

The `VCSE::CTestReport` interface pointer is a lightweight component and as such is only suitable for use with a single test shell component.

Packaging Components

Once a component has been developed and tested, it needs to be packaged in a standard format ready for distribution. The component package manifest is generated by the VIDL compiler. The package contains the essential information needed to describe the component and to specify the files that are to be incorporated in the packaged component.

The distributed package normally contains at least the following files.

- The library containing the component implementation, in the previous example, `EXAMPLES_CULaw.dlb`. This file is added to the VisualDSP++ project when the component is added to the project.
- The header file containing the declarations of the component `Create` and `Destroy` functions, in the previous example, `EXAMPLES_CULaw_factory.h`. This file is added to the VisualDSP++ project when the component is added to the project.
- The header files for any interface that has been defined or is referenced in the VIDL specification. The C representation for the `IG711` interface is generated in `EXAMPLES_IG711.h`. These files are added to the VisualDSP++ project when the component is added to the project.
- The set of `.HTML` documentation files for the component, which are all contained in the `html` directory. The main component file, `EXAPLES_CULaw.html` in the previous example, is the only file added to the VisualDSP++ project when the component is added to the project.

The VIDL compiler automatically includes these files in the manifest list. You can add further files to the manifest, such as data files or images referenced from the updated `.HTML` documentation files. The manifest `.XML` file has a section where additional files can be specified and which are preserved when the VIDL compiler is re-run. The user-modifiable section of the manifest is as follows.

```
<!--
##############################################################
//####SCF Start of package manifest, EXAMPLES_CULaw
-->
<!--
Any User specific manifests should be added here.
-->
<!--
//####ECF End of package manifest, EXAMPLES_CULaw
//############################################################
-->
```

Start the New Component Package Wizard by clicking the **Tools** menu and choosing **VCSE**, **New Component Package.** The step by step wizard guides you through the process of preparing a component for distribution. See "Component Packaging Wizard" on page 1-20 as well as the online Help for detailed descriptions of the wizard.

Manifest Commands

For each component specified in the VIDL input file, the VIDL compiler produces an XML based manifest file. Use this file to control the packaging wizard when the component is being packaged for distribution. The name of the packaging file is <NS>_<C>.xml. The packaging wizard combines the contents of the .XML file with information derived from the wizard steps to complete a component package.

The manifest file includes two user-modifiable blocks that can be used to either add files to the package or to override the part of the manifest that is generated by the VIDL compiler.

Within the user-modifiable block that is located within the <manifest> section the following tokens can be used to

- Add additional files to the package
- Override the file actions automatically generated by the VIDL compiler
- Add licensing terms that a user must accept before the component can be installed

Adding a File to the Package

To add an additional file to the package the `<file>` token is used, which has the format below

```
<file [addtoproject="yes"] [target="processor-name"] > filename
</file>
```

where the `addtoproject` and `target` attributes are optional items. *filename* is the name of a file included in the package and installed when the component is installed. If no path is specified or the supplied path is for a subdirectory, then installed file is installed in the same relative path when the component is installed. If the specified path specifies a directory that is not a subdirectory, then the actual file is installed in the component directory. In the examples below

```
<file>header.h</file>
<file>include\hdr.h</file>
<file>..\src\example.c</file>
```

The file `header.h` is installed in the `component` directory, the file `header.h` is placed in a subdirectory called `include`. The file `example.c` is also located in the `component` directory since the source directory is not a subdirectory.

If the `addtoproject="yes"` is specified then the file in the `component` directory is also added to the project when the component is added to a project. The default value for the `addtoproject` attribute is "no", which means that the file will not be added to a project.

When the `target` attribute is specified then the component file is only added to a project if the specified processor (processor-name) matches the target processor for the project.

Overriding File Actions

The `manifest-command` token is used to override the file actions that are generated by the VIDL compiler. The general format of this token is as shown below

```
<manifest-command
   type="command-name"
   source="source-element"
   destination="destination-element"/>
```

where `command-name` is the name of the command, and source-element/destination-element are defined based on the type of command.

The `file-ignore` command specifies that the file named in the command should be ignored when packaging a component. The following example

```
<manifest-command type="file-ignore"
source="ADI_CMyComponent.idl"/>
```

specifies that any `<file>` token referencing the file `ADI_CMyComponent.idl` should be ignored when packaging the component.

The file-rename command can be used to control the placement of a file within the component directory and sub-directories. In the case of a <file> token such as

```
<file>..\src\example.c</file>
```

then the default location of the file `example.c` will be in the `component` directory. However if you wish to locate the file in a source subdirectory then you would supply a `file-rename` command such as

```
<manifest-command type="file-rename" source="..\src\example.c"
destination="source\example.c" />
```

The destination specified by a `file-rename` command must be the `component` directory or a subdirectory of the component directory.

The file -addtoproject command can be used to change the default VIDL compiler generated action on whether a file should be added to a project when the component is added to a project. The format of the file-addto-project command is as shown below

```
<manifest-command type="file-addtoproject" source="filename"
value="yesorno"/>
```

where filename is the name of the file as it appears in the corresponding <file> token and yesorno can have the values "yes" or "no". For example if you want the file testing.h to be added to a project when the component is added to a project then you can override the default action using the command

```
<manifest-command type="file-addtoproject" source="testing.h"
value="yes" />
```

Adding Licensing Terms

If a <license-text> token appears in the manifest section then the packaged component can only be installed if the user accepts the terms of the license specified in the license-text token. The text in the license-text field is rendered as HTML and any text that is legal input to a web browser (plain-text, html, xml, and so on) can be used. When the user attempts to install the component, the license text is displayed in a dialog and by default the I disagree radio button is selected and the OK button is disabled. The user is only able to select OK if the I agree radio button is selected. Canceling the dialog results in no action being taken.

An example of the use of the subdirectory field is shown below

```
<license-text><![CDATA[The license terms should be
included here. The field can consist of anything that is
acceptable to a web browser and can consist of multiple
lines of input. ]]></description>
```

Using Modifiable Sections

The VIDL compiler automatically inserts several user-modifiable sections in each component it generates. All the changes that you make to the generated files must be confined to these sections. If you add material outside a section, it will be lost next time the VIDL compiler regenerates the source file.

Component Factory Source File

Table 2-4 summarizes user-modifiable sections that may be used in the component factory source file.

Table 2-4. Component Factory Source File

Modifiable Section	Description
Component Global Settings	Provide global definitions required for component implementation or any definitions required by the factory functions. For example, you may redefine macros, such as `__VCSE_malloc`, to provide private memory allocation procedures that the component may use if no `IMemory` implementation is passed to the `Create` function. If the component factory functions require access to any library functions, this is the appropriate place to include the necessary header files.
Component Class Factory-Create	Provides an opportunity in the component `Create` function to either modify or replace the standard generated code used to allocate the component instance data using the `VCSE::IMemory` interface.
Component Class Factory-Create.1	Provides an opportunity in the component `Create` function to initialize any private instance data fields irrespective of how the instance data itself is allocated. If the component also requires any working storage to be available throughout its lifetime, this is a suitable point to arrange for its allocation.
Component Class Sizeof	Provides the opportunity to override the size of the component returned by the component `SizeOf` function.

Table 2-4. Component Factory Source File

Component Class Factory-Destroy	Provides the opportunity in the component `Destroy` function to either modify or replace the standard generated code used to free the component instance data using the `VCSE::IMemory` interface. If changes were made in the `Create` function, the appropriate changes to effect the freeing of the allocated memory should be made here.
Component Class Factory-Cleanup	Provides the opportunity in the component `Destroy` function to release any resources that the instances owns or to carry out any other tidy up action before the instance memory is freed.

Component Methods Source File

Table 2-5 summarizes user-modifiable sections that may be used in the component methods source file.

Table 2-5. Method Source File

Modifiable Section	Description
Component Global Settings	Provide any specific global definitions required for the component implementation or any definitions required by the method functions. If the method functions require access to any library functions, then this is the appropriate place to include the necessary header files.
Component Global Settings.1	Provides an opportunity to redefine the automatically generated macros that can be used to trace function entry and exit and so on. For example, the component may wish to use a `VCSE::IError` interface that has been provided for error reporting
Interface Member Function	Provides an opportunity to supply the actual body of each member function. Each member function of every interface supported by the component will have such a section.

Component Instance Header File for C/Assembly

Table 2-6 summarizes user-modifiable sections that may be used in the component instance header file.

Table 2-6. C Component Instance Header

Modifiable Section	Description
Component Global Settings	Provide any specific global declarations or preprocessor definitions required for component implementation.
Component Private Members	Specify any component specific private members for the instance data structure of the component.

Component Instance Header File for C++

Table 2-7 summarizes user-modifiable sections that may be used in the component instance header file.

Table 2-7. C++ Component Instance Header

Modifiable Section	Description
Component Global Settings-Include	Provides an opportunity to include any standard header files that may be required by the component implementation
Component Global Settings	Provide any specific global declarations or preprocessor definitions required for component implementation. This section occurs within a nested namespace that ensures uniqueness across all component definitions.
Component Private Members	Specify any component specific private members for the instance data class of the component.

Component Factory Header File

Table 2-8 summarizes user-modifiable sections that may be used the component factory header file.

Table 2-8. Component Factory Header

Modifiable Section	Description
Component Size Definition	Allows you to specify a preprocessor macro that defines the maximum amount of memory that the component instance data may require.

Component Package Manifest File

Table 2-9 summarizes user-modifiable sections that may be used in the component package manifest file (.XML).

Table 2-9. Component Manifest File

Modifiable Section	Description
Package Manifest	Provides the opportunity to specify which additional files should be packaged with the component by the packaging wizard.
Package Component Details	Provides the opportunity to override the subsequent automatically generated definitions.

Test Shell Component User-Modifiable Sections

The user-modifiable sections for a test shell component are exactly the same as for an actual component. The VIDL compiler inserts macro definitions, validation and reporting code inside the user-modifiable sections in the test shell component methods source file to provide the user with complete control over how the validation is to be performed. The contents of this code is summarize below.

Component Global Settings

The supporting macros and the `VCSE::ITestReport` interface pointer declaration are placed in this section, allowing the user to redefine them as required. For more information, see "Supporting Tools for Test Shell Components" on page 2-43.

Interface Member Function

Both the call to an actual interface method of the component under test and the supporting validation and reporting code, generated by the VIDL Compiler, prior to and following this method call are enclosed within the "Interface Member Function" user-modifiable section of each interface method within the test shell component methods source file.

Important note: Since the method call to the actual interface method is enclosed within a UMB, any changes to the signature of this method are not be reflected in subsequent passes of the VIDL compiler to update the test shell component.

Using Components

Once a component is installed on a system, the VCSE Component Manager is used to add the component to a VisualDSP++ project. After a component is added to the project, you can access the component's header file and HTML documentation. The libraries needed for the component to use at link time are also automatically added to the project. Once the component is added to the project, you are ready to create an instance of the component and use the services offered by its supported interfaces.

Creating Component Instances

To create an instance of the component, use the component's Create function. The function prototype of the Create function for each component expects the same parameters. The Create function for the EXAMPLES_CULaw component (Listing 2-3 on page 2-17) has the following prototype.

```
VCSE_MRESULT EXAMPLES_CULaw_Create(
            const  VCSE_IBase_ptr outer,
            const  VCSE_RefIID    iid,
            VCSE_IBase_ptr        *iptr,
            const  VCSE_IBase_ptr ienvp,
            const  VCSE_HANDLE    token);
```

The main points to be noticed:

- The name of the Create function is obtained by prefixing _Create with the concatenation of the defining namespace, an underscore, and the component name.
- The first parameter (outer) is normally a NULL pointer. If the component instance is being aggregated into an existing component, this parameter is an IBase interface pointer for the aggregating component.

  Aggregation and its effects are described in "Aggregating Components" on page 2-80.
- The second parameter (iid) specifies the unique interface identifier for an interface supported by the new component. The corresponding interface pointer is returned via the third parameter (*iptr) if the component instance is created successfully.

- Components do not allocate memory and other resources; instead, they request resources from the application. The fourth parameter (`ienvp`) is used to pass an interface pointer to the component. The pointer allows the component to request needed resources, including memory from a resource allocator component.

  The standard interface for allocating memory for use by a component is called `VCSE::IMemory` and is described in "IMemory Interface" on page 3-2. Examples of memory allocator components that implement this interface are provided; one example offers simple allocation from the standard heap, another provides some additional debugging and statistical support.

  If the fourth parameter is `NULL`, the instance creation fails unless the component has been designed to employ its own memory allocation in such a situation.

- The final parameter that can be passed into the `Create` function is a `token`, which the component simply passes back to any resource allocation interface it invokes. The component is not expected to directly use or understand the significance of the `token` but simply pass it back when it is allocating or freeing a resource.

The `Create` function prototype of a component is defined in the factory header file distributed with this component. The name of the factory header file for the `EXAMPLES_CULaw` component is `EXAMPLES_CULaw_factory.h`.

Assuming there is a pointer `p_VCSE_IMemory` to the `VCSE::IMemory` interface (described in "IMemory Interface" on page 3-2), an instance of the `EXAMPLES_CULaw` component (defined in Listing 2-3 on page 2-17) can be created as follows.

Listing 2-5. Instantiating EXAMPLES_CULaw Component

```
mr = EXAMPLES_CULaw_Create(NULL, VCSE_IBase_IID, &p_VCSE_IBase,
                           p_VCSE_IMemory, NULL);
if ( MR_FAILURE(mr) ) {
...  /* if the instantiation fails */
}
```

Access to the simple memory allocator `VCSE::CSimpleMemory` is obtained by including the header file `VCSE_CSimpleMemory.h`. Internally, the `VCSE::CsimpleMemory` component always uses the heap for memory allocation. An instance of the `VCSE::IMemory` interface can be obtained as follows.

Listing 2-6. Examples_CULaw_Create Function

```
VCSE_IBase_ptr p_VCSE_IMemory;

mr = VCSE_CSimpleMemory_Create(NULL, VCSE_IMemory_IID,
                               &p_VCSE_IMemory, NULL, NULL);
if ( MR_FAILURE(mr) ) {
...  /* if the instantiation fails */
}
```

A component instance is accessed by invoking a method function on one of the interfaces that the component supports. The `Create` function of a component returns one specified interface pointer for the created instance. Each interface provides the `GetInterface` method to obtain an interface pointer for any other interface that the component supports.

In the `EXAMPLES_CULaw_Create` function, the initial interface pointer obtained is the `VCSE::IBase` pointer. A `VCSE::IBase` pointer is available for any component or interface since the interface of the same name is provided by all components. The interface pointer returned in the third parameter is specific to the component instance that has been created.

If you create two instances of CSimpleMemory and specify VCSE_IMemory_IID in each, then the interface pointers returned by each call will be different. Each interface provides a method called GetInterface that can be used to obtain an interface pointer for any other interface that the component supports.

The allocator VCSE::CSimpleMemory uses the system heap to allocate memory. That may be acceptable during the early stages of development, but a more application-specific approach is likely to be required for production purposes. The IMemory allocation interface is capable of supporting a wide variety of memory allocation strategies.

VCSE Memory Allocators

Components do not allocate memory and other resources; instead, they request resources from the application. The fourth parameter (ienvp) of a component's Create function is used to pass an interface pointer to the component. The pointer allows the component to request needed resources, including memory from a resource allocator component.

The standard interface for allocating memory for use by a component is called VCSE::IMemory and is described in "IMemory Interface" on page 3-2. VCSE provides two memory allocator components that implement this interface; one component offers simple allocation from the standard heap, another a pre-allocation (most likely static) scheme.

VCSE::CSimpleMemory

The VCSE::CSimpleMemory component is a lightweight component for the management of component memory on the heap, using the malloc and free functions in the C library.

Access to the simple memory allocator `VCSE::CSimpleMemory` is obtained by including the header file `VCSE_CSimpleMemory.h`. Internally, the `VCSE::CSimpleMemory` component always uses the heap for memory management. An instance of the component `VCSE::CSimpleMemory` can be created as follows.

Listing 2-7. Creating VCSE::CSimpleMemory

```
VCSE_IBase_ptr p_Allocator;
mr = VCSE_CSimpleMemory_Create( NULL, VCSE_IBase_IID,
                                 &p_Allocator, NULL);
if( MR_FAILURE( mr ) ) {
           …  /* if the instantiation fails*/
}
```

Once the application has an instance of `CSimpleMemory`, an interface pointer from `CSimpleMemory` can be passed as the `ienvp` parameter of a component's `Create` function. Below is an example of creating an instance of the `CULaw` component with the above `p_Allocator` interface pointer.

Listing 2-8. Using VCSE::CSimpleMemory

```
/*p_Allocator set in above listing.*/
mr = Examples_CULaw_Create(NULL, VCSE_IBase_IID, &p_VCSE_IBase,
                                 p_Allocator, NULL);
if( MR_FAILURE( mr ) ) {
…  /* if the instantiation fails*/
}
```

VCSE::CInstMemory

The `VCSE::CInstMemory` component is a lightweight component for the management of component instance memory that makes use of a portion of previously (most likely, statically) allocated memory.

Access to the instance memory allocator `VCSE:: CInstMemory` is obtained by including the header file `VCSE_CInstMemory_factory.h`. An instance of the `VCSE::CInstMemory` component can be created as follows.

Listing 2-9. Creating VCSE::CInstMemory

```
VCSE_IBase_ptr p_Allocator;
mr = VCSE_CInstMemory_Create( NULL, VCSE_IBase_IID,
                                &p_Allocator, NULL);

if( MR_FAILURE( mr ) ) {
        …  /* if the instantiation fails*/
}
```

Once the application has an instance of `CInstMemory`, an interface pointer from `CInstMemory` can be passed as the `ienvp` parameter of a component's `Create` function and a `CInstMemory_Allocation` pointer as the `token` parameter.

Listing 2-10. CInstMemory_Allocation structure

```
struct CInstMemory_Allocation {
      unsigned int Length;
      void* Memory;
};
```

Below is an example of creating an instance of the CULaw component with the above `p_Allocator` interface pointer.

Listing 2-11. Using VCSE::CInstMemory

```
static int data[ INSTANCE_SIZE ];
VCSE_IBase_ptr pIBase;
static VCSE_CInstMemory_Allocation alloc = {
      sizeof(data),
```

```
      (void*)data};
Examples_CULaw_Create(0, VCSE_IBase_IID, &pIBase,
        p_Allocator_IBase, (VCSE_HANDLE)&alloc );
```

`CInstMemory` requires a `CInstMemory_Allocation` pointer to be passed as the `Token` parameter of the `VCSE::IMemory::Allocate` method. As seen in Listing 2-11, the application indirectly sets the `Token` parameter by passing the `CInstMemory_Allocation` pointer as the `token` parameter of the component's `Create` function.

Inside `VCSE::IMemory::Allocate`, `CInstMemory` checks that the length of the allocation request is less than or equal to the length of the data in the `VCSE_CInstMemory_Allocation` structure. If the request is for more data than available `MR_NO_MEMORY` is returned. On a successful request, the `Allocation` parameter of `VCSE::IMemory::Allocate` is set to the `Memory` field of the `CInstMemory_Allocation` structure.

`VCSE::IMemory::Free` provides no implementation and simply returns `MR_OK`.

`VCSE_CInstMemory` is suitable for components that require an allocation for their instance memory only. Components that require allocations other than instance memory will require a more sophisticated memory allocator.

Using Interface Pointers in C or Assembly

In C or assembly, an interface pointer is in effect a pointer to a structure that represents an instance of the component. The methods of the interface are invoked through macros that allow the method calling mechanism to be hidden. There is a macro for each interface method whose name is formed by concatenating the namespace, the interface name, and the method name with an underscore character as the separator. The interface pointer that identifies the instance of the invoked component is always passed explicitly as the first parameter to the macro.

The following code example can be used to obtain an `EXAMPLES::IG711` (Listing 2-2 on page 2-9) interface pointer from the `VCSE::IBase` interface pointer returned by the `Create` function in Listing 2-5 on page 2-63.

Listing 2-12. C Interface Pointer

```
  EXAMPLES_IG711_ptr p_IG711;
  mr = VCSE_IBase_GetInterface( p_VCSE_Ibase, EXAMPLES_IG711_IID,
      (VCSE_IBase_ptr*)&p_IG711 );
  if (MR_FAILURE(mr)) {
     ...  /* if the instantiation fails */
}
```

Once an interface pointer to the desired interface is obtained, you can call the methods of the interface to obtain the services it offers. Given the interface pointer is obtained successfully (see Listing 2-12), the following C code example shows how to use the macro that calls the `Compress` method. The `Compress` method converts 128 values to their equivalent µ-law encoding values:

```
  mr = EXAMPLES_IG711_Compress(p_IG711, 128, rawData, muData);
```

Each interface method returns an `MRESULT` value, which indicates the success or failure of the invocation. `MRESULT` values can be tested with the `MSUCCESS` and `MFAILURE` macros. The macro `MSUCCESS` is passed an `MRESULT` value and returns a nonzero value if the call was successful or a zero value if the call failed. Similarly, the macro `MFAILURE` is passed an `MRESULT` value and returns a nonzero value if the method invocation failed or a zero value if the invocation was successful. The value returned by a method should always be tested to ensure the invocation is successful.

Using Interface Pointers in C++

In C++, an interface pointer is in fact a pointer to a C++ class whose member functions are the methods of the interface. It follows that a method can be invoked directly by a call to the C++ member function. The C++ calling mechanism passes the ‘`this`’ pointer for the component instance automatically.

The code example in Listing 2-13 can be used to obtain an `EXAMPLES::IG711` interface pointer from the `VCSE::IBase` interface pointer returned by the `Create` function in Listing 2-5 on page 2-63.

Listing 2-13. C++ Interface Pointer

```
EXAMPLES_IG711_ptr p_IG711;
mr = p_VCSE_IBase->GetInterface (EXAMPLES_IG711_IID,
  (VCSE_IBase_ptr*)&p_IG711);
if (MR_FAILURE(mr)) {
   ...  /* if the instantiation fails */
}
```

Once an interface pointer to the desired interface is obtained, you can invoke the methods of the interface to obtain the services it offers directly in C++. Given the interface pointer is obtained successfully (Listing 2-13), the following code example shows how to invoke the `Compress` method in C++. The `Compress` method converts 128 values to their equivalent µ-law encoding values:

```
mr = p_IG711->Compress( 128,rawData,muData );
```

In C++, the interface pointer type implicitly specifies the invoked interface, and the invoked method implicitly receives the appropriate ‘`this`’ pointer, which identifies the instance of the component.

Each interface method returns an MRESULT value to indicate the success or failure of the method invocation. MRESULT values can be tested with the MSUCCESS and MFAILURE macros. The macro MSUCCESS is passed an MRESULT value and returns a nonzero value if the method was successful or a zero value if the invocation failed. Similarly, the macro MFAILURE is passed an MRESULT value and returns a nonzero value if the method invocation failed or a zero value if the invocation was successful. The value returned by a method should always be tested to ensure that the invocation is successful.

Destroying Components

The Create function for a component creates an instance of the component; each interface pointer obtained from an instance refers to the same instance of this component. When an instance of a component is no longer required, you can destroy it by invoking the Destroy function for the component. The name of the Destroy function is obtained by prefixing _Destroy with the concatenation of the defining namespace, an underscore, and the component name. The Destroy function for the EXAMPLES_CULaw component has the following prototype.

Listing 2-14. Examples_CULaw_Destroy Function

```
VCSE_MRESULT EXAMPLES_CULaw_Destroy(const VCSE_IBase_ptr iptr);
```

The main points to be noticed:

- Any interface pointer obtained from an instance of a component can be passed as the first parameter to the Destroy function to specify the component instance to be destroyed.
- When a component instance is destroyed, all the interface pointers for the instance become obsolete, and no method functions should be invoked via any of these interface pointers.

Implementation of GetInterface Method

All the interfaces supported by a component share a single implementation of the `GetInterface` method, which is used to provide on request interface pointers for all supported interfaces. The `GetInterface` method is always automatically generated by the VIDL compiler. The following example illustrates the implementation of `GetInterface`, automatically generated by the VIDL compiler for the C version of `EXAMPLES_CULaw` component (defined in Listing 2-3 on page 2-17).

Listing 2-15. Implementing GetInterface Methods

```
VCSE_MRESULT EXAMPLES_CULaw_GetInterface(
        VCSE_IBase_ptr    base,
        const VCSE_RefIID iid,
        VCSE_IBase_ptr    *iptr)
{
  /*
   * GetInterface method for supplying the requested interface
   */

  __ASSIGN_THIS_POINTER(__this,EXAMPLES_CULaw);

  if (!iidcmp(iid, VCSE_IBase_IID))
     *iptr = REINTERPRET_CAST(VCSE_IBase_ptr,
                         STATIC_CAST(EXAMPLES_IG711,__this));
  else if (!iidcmp(iid,EXAMPLES_IG711_IID))
     *iptr = (VCSE_IBase_ptr)STATIC_CAST(EXAMPLES_IG711,__this);
  else if (!iidcmp(iid,VCSE_IAlgorithm_IID))
     *iptr = (VCSE_IBase_ptr)STATIC_CAST(EXAMPLES_IG711,__this);
  else
     return (VCSE_MRESULT)MR_NOT_SUPPORTED;
```

```
  return (VCSE_MRESULT)MR_OK;
}
```

Essentially, `GetInterface` checks the identifier of the requested interface against the interfaces that are supported and performs the necessary casting to convert the interface pointer to the appropriate structure.

VCSE Optimizations

Method Call Overhead

Invoking a VCSE method is different from invoking a standard library function. Invoking a VCSE method in general involves two additional actions

- Adjusting the interface pointer by adding a value which depends on the internal structure of the component that implements an interface
- Using an indirect rather than a direct function call

In many cases both these overheads are avoidable. In comparison to cases where multiple inheritance is involved, the value of the adjustment to the interface pointer is normally zero. Under normal circumstances, only one implementation of each interface in a particular application is used. As a result, every invocation of each method of an interface causes the same code to be executed.

In general these two overheads cannot be avoided in code generated directly by the C/C++ compiler or by the assembler compiler, since the only information available is the interface structure itself. The requirement to encapsulate or hide the internal structure of a component specifies that the required information cannot be made available in the interface definition. An interface definition describes a service that can be provided by many different components and can, therefore, not contain any information on an individual component. When code is compiled it is not possible to know which implementation of an interface is invoked or if only there is only one implementation of an interface.

Preventing Code and Data Elimination

When the linker is creating an application from its constituent objects and libraries, it builds a static call tree to enable it to identify functions that cannot be called or static data that is not accessed in the application. The linker uses the call tree to eliminate such unused functions and data and thereby it can significantly reduce the size of the resulting application. Unfortunately, the function pointers in a VCSE method-table force the linker to mark all the corresponding function as potentially callable. As a result, the linker is unable to remove unused methods from applications containing no explicit method calls.

If the linker was able to identify exactly which methods were actually used in a particular application then it would be able to eliminate both the unused methods and any functions that are only invoked from these methods.

Standard Method Functions

VCSE component model defines some standard interfaces that act as base interfaces for different classes of interface. For example all VCSE algorithms are expected to extend the `IAlgorithm` interface. The use of these standard base interfaces provides a common structure for each class of interface. For example the `IAlgorithm` interface defines methods that all algorithms must provide to simplify (or standardize) the use of algorithms. `IAlgorithm` specifies that each algorithm must provide an `Activate` and a `Deactivate` method; these methods are invoked before and after an instance of the algorithm is invoked to process any data. The `Activate` method enables each algorithm for example to prepare for the optimized execution of the instance by possibly moving data to internal memory or execute any other necessary action.

In some cases however there may not in fact be any action that an algorithm wishes to make prior to executing. In this case the component model imposes the overhead of invoking both the `Activate` and `Deactivate` empty methods that will have no useful effect and may simply return a success indication. An application cannot simply avoid invoking such methods since that assumes that the application understands the internal structure of the components being used. Even if the application could avoid invoking these methods for one particular implementation of an interface, the replacement of the algorithm by another implementation may well require the use of the `Active` and `Deactivate` methods for correct execution.

The elimination of such unnecessary method calls must be based on information internal to the component implementation and cannot be associated with the interface definition.

Improving Program Efficiency

None of the above overheads can be eliminated when compiling or assembling either the component or the code that invokes a component because the encapsulation of the internal structure of each component means that the compiler cannot be aware of which component provides the implementation of the interface. However when the linker is building the static executable for the application it becomes possible to associate the invocation of a method on an interface with the component that is providing the implementation of the interface.

To enable the linker to provide these optimizations, it has to have access to various pieces of information such as

- The location of each method table and the identity and number of methods supported by the interfaces associated with the method table.
- The contents of each method table entry (for fine tuning of optimization possibilities.

- The location of every VCSE method call and the interface identity and individual method associated with that call.
- The set of constant VCSE methods, that is, methods that have no effect other than returning a constant value on each invocation of the method.

Given access to all of the above information then the linker is capable of eliminating very many of these overheads.

The information needed by the linker is automatically provided by decorating the assembler, the C or C++ code that the VIDL compiler generates. Neither the developer of a component or the user of a component needs to take any special action to provide the information needed to enable the linker to provide all the optimizations. The one exception to this is that only the component developer can mark methods which have no action but to return a constant value.

The developer can identify such constant return methods by the use of a `#pragma VCSE_RETURNS` in C and C++ and the `.vcse_returns` directive in Assembler.

The example below shoes how you would mark a C or C++ method to notify the system that the methods only action is to return the constant value `MR_OK`.

Listing 2-16. Example of a Constant VCSE Method in C

```
static __VCSEMETHOD VCSE_MRESULT LOCAL_CAgg_AggB(
   VCSE_IBase_ptr base,
   int one,
   int *funcid)
{
  component LOCAL_CAgg *__this = (component LOCAL_CAgg *)base;
  //####SCF Start of interface member function, LOCAL::CAgg::AggB
  {
```

```
#pragma VCSE_RETURNS
   return (VCSE_MRESULT)MR_OK;
  }
  //####ECF End of interface member function, LOCAL::CAgg::AggB
}
```

In the case of C and C++, the compiler will check that the value being returned is a constant expression that can be evaluated at compile time.

If the method had been define in assembler then the code would be similar to that shown in Listing 2-17

Listing 2-17. Example of a Constant VCSE Method in Assembler

```
      align 2;
_LOCAL_CAgg_AggB:
        link  4;

        R0 =   0;
        P0=[FP+ 4];
        unlink;
        JUMP (P0);

._LOCAL_CAgg_AggB.end:
      type _LOCAL_CAgg_AggB,STT_FUNC;
      .vcse_returns _LOCAL_CAgg_AggB,0;
```

The assembler code for the function include carries out all the normal actions including setting the return value. The `.vcse_returns` directive specifies the name of a standard function and the value that the function always returns.

When a function is marked as returning a constant value then the linker can remove the code from the executable and replaces all calls on the method by sampling setting the appropriate return value register.

Because there is normally only one implementation of most interfaces in an application the above optimizations can be deployed frequently. Apart from marking any constant return functions the only action that the developer or user of a component needs to take to trigger the optimizations is to request the linker to effect the VCSE optimizations on the linker property page for a project.

VCSE Algorithms

The VCSE `IAlgorithm` interface has been defined to provide a framework that various algorithms can support. It is much easier for an applications developer if all algorithms provide a common set of methods and style of use. The VCSE `IAlgortithm` interface essentially defines three methods.

- `Activate`, which is invoked when the instance of an algorithm is about to be used.
- `Deactivate`, which is invoked when an instance of an algorithm will not be used for some period of time.
- `Reset`, which can be invoked at any time and which brings the instance of the algorithm to a defined state.

In addition to the above methods, an algorithm would provide one or more methods that are invoked to carry out the actual processing of the algorithm.

The primary use for the `Activate` method is to allow the algorithm to prepare itself for efficient execution of the processing methods. The `Activate` method may therefore be responsible for copying data from external to internal memory to allow the data to be processed most efficiently. The `Deactivate` method similarly provides an opportunity for the instance of an algorithm to preserve any state that will be needed the next time it is called on to process data, and it may do this by copying some data from internal to external memory for example.

In general algorithms should not be directly involved in memory allocation or management since only the application using the algorithm has sufficient knowledge of the whole system memory requirements to make optimal decision over memory use. It is for this reason that an algorithm is expected to allocate the memory it needs by using the `IMemory` interface supplied to it when an instance is created. In this way the algorithm can delegate memory allocation to the controlling application. Similarly the `Activate` and `Deactivate` methods should (ideally) delegate the necessary memory management to the application and this can be done by supplying an interface pointer to the algorithm which is used within the `Activate` and `Deactivate` methods.

The `IAlgorithm2` interface extends the `IAlgorithm` interface and provides a much simpler and more flexible memory management approach while leaving the allocation and management of memory to be handled directly by the application. The `IAlgoritm2` approach allows an algorithm to have quite detailed control over the type of memory it requires to perform very efficiently. In addition the `IAlgoritm2` approach enables an application to have one instance of an algorithm processing some data while the transfer of data between external and internal for another instance can be carried on in parallel using DMA.

An `IAlgortihm2` component specifies the types of memory it requires in its documentation and provides a method `AssignMemoryResources` that the application uses to tell the component the location of the memory areas that have been allocated to the instance of the algorithm. The algorithm has a property that the application sets to inform the algorithm of an interface pointer for a memory swapper interface that would be responsible for actually effecting any data movement needed on activation or deactivation of the algorithm instance. After the property has been set the application would invoke the `ConfigureMemorySwapper` method so that the component can set up the memory swapper as it wishes. After this initialization phase the application would use the following sequence of calls when invoking the algorithm.

- It would invoke the `Swap` method on the memory swapper interface to ensure all memory is ready for processing
- It next invokes the `Activate` method on the algorithm instance to let it prepare for processing
- It next invokes the processing methods of the algorithm as required.
- After the current set of processing is complete it would invoke the `Deactivate` method of the algorithm
- Finally it would invoke the `Swap` method on the memory swapper interface to save any information that is needed on subsequent calls.

Two additional advantages of using the `IAlgorithm2` approach are that

- Scarce internal memory resources can be shared between different instances of the one or multiple algorithms as any such content can be preserved by the swapper
- Since the application is responsible for invoking the swapper and the application, the application can use an asynchronous swapper so that one instance can be processing while a second instance of any algorithm is being swapped.

A VCSE component that provides an implementation of the memory swapper that uses DMA to effect swapping and supports asynchronous operation is available on the ADI web site and can be downloaded and installed using VisualDSP.

The `IAlgorithm` and `IAlgorithm2` interfaces are described in detail "Standard Interfaces" in Chapter 3, Standard Interfaces.

Aggregating Components

Although components are primarily intended for application development, they can also be used in component implementation. For example, the developer of a component that requires the use of μ-law encoding and decoding could simply use the `CULaw` component discussed earlier rather than re-implement the algorithm from scratch. Since component implementations are encapsulated, internal use of the `CULaw` component remains hidden from the application.

There are two techniques for component reuse that are normally referred to as *delegation* and *aggregation*. When delegation is used, the outer component acts as a wrapper around the inner component. Calls to any of the methods of the inner component are made via corresponding methods in the outer component. Normally, the inner component needs to provide significant functionality to make the overhead of the extra method call insignificant by comparison.

Aggregation is a different technique that allows an interface from an existing or inner component to be combined with the interfaces in the outer component. It has the advantage that when the aggregated interface methods are called, the original methods in the existing component are executed directly without any overhead.

Aggregation can be difficult to implement correctly since the combined components must appear to the user of the outer component as a single entity that obeys all the rules of the Component Model seamlessly. The VIDL compiler automatically generates the support necessary, so components can be aggregated automatically without the developer being aware of how aggregation operates in detail.

The usefulness of aggregation can be seen by examining a real world example. Suppose we have an `MP3` component with an `IMp3` interface, which allows MP3 encoded music to be played. Suppose you wish to create a component for use in MP3 players that responds to voice commands through an `IVoice` interface. You can either implement support for both the voice interface and the MP3 support from scratch, or you can decide to use the existing MP3 component and concentrate on the new software needed to support voice commands.

Aggregation allows you to incorporate the existing MP3 component into your new component, so that it offers both the IMp3 and the IVoice interfaces. By doing this, you are giving the user of the component full access to the IMp3 interface with no additional overhead involved in its use. You do not require the source for the MP3 component in order to exploit it within your component by use of aggregation.

Implementation of Aggregation

To explain how aggregation operates, we have two components CRed and CBlue that implement interfaces IRed and IBlue, respectively. We wish to make a new component CRedGreenBlue that provides the three interfaces IBlue, IRed, and IGreen by implementing IGreen directly and aggregating the implementations of IBlue and IRed from CBlue and CRed.

The three components and their interfaces can be represented as shown in Figure 2-4.

A simplified version of the VIDL definition for these components is in Listing 2-18.

Listing 2-18. Aggregation Example

```
namespace EXAMPLES {
   [iid(“24e7d634-d6c8444c-b66c91fa-92fc4cf1”)]
   interface IRed extends IBase {};

   [iid(“5bf3a04f-a541fe42-a9741ad9-43d85370”)]
   interface IBlue extends IBase {};

   [iid("72c53695-c7d4d843-a21f074b-bdba49fb")]
```

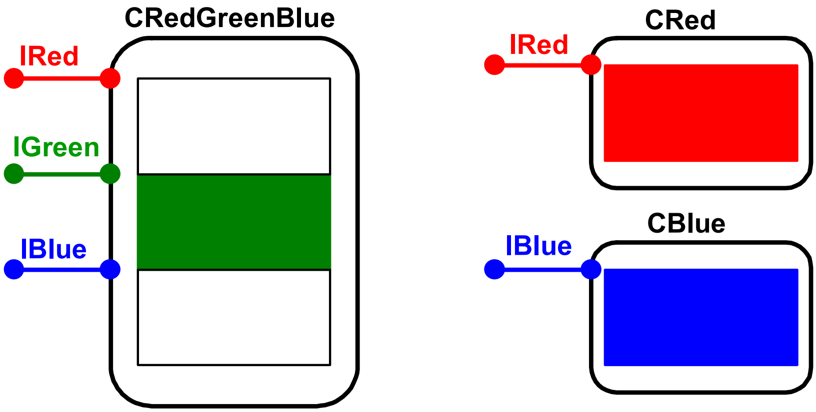


Figure 2-4. Aggregation Example

```
interface IGreen extends IBase {};
[aggregatable,
company("ADI"),
title("RED"),
category("EXAMPLES")]
component CRed implements IRed;

[aggregatable,
company("ADI"),
title("BLUE"),
category("EXAMPLES")]
component CBlue implements IBlue;

company("ADI"),
title("REDGREENBLUE"),
category("EXAMPLES")]
component CRedGreenBlue implements IRed,IGreen,IBlue {
  aggregates IRed from CRed;
  aggregates IBlue from CBlue;
```

```
    };
};
```

When a component provides an interface, the Component Model requires that a call to its `GetInterface` method must be capable of returning every other interface provided by the component. When the `IRed` and `IBlue` are aggregated into `CRedGreenBlue`, their `GetInterface` methods, which are provided by `CRed` and `CBlue` components, must somehow be able to return the `IGreen` interface.

The problem of handling requests for `IBlue` and `IGreen` within `CRed`, and requests for `IRed` and `IGreen` within `CBlue`, is solved by having each aggregated component handle interface requests with an additional variant of `IBase`, called `INonDelegatingBase`, that has a method called `NonDelegatingGetInterface`. All components that support aggregation, such as the `CRed` and `CBlue` components, must implement this interface in addition to `IBase`. The `NonDelegatingGetInterface` method of `INonDelegatingBase` handles all requests for interfaces implemented directly by the aggregated component. Consequently, it is referred to as the non-delegating version of `GetInterface`. By contrast, the `GetInterface` method of `IBase` in the aggregated components `CRed` and `CBlue` handles all interface requests by re-calling the `GetInterface` method in the aggregating component `CRedGreenBlue`. Consequently, it is referred to as the delegating version of `GetInterface`.

The methods `CRed_GetInterface` and `CBlue_GetInterface` both delegate their requests back to `CRedGreenBlue_GetInterface`. In turn, `CRedGreenBlue_GetInterface` handles any request for `IGreen` directly, but hands requests for `IRed` and `IBlue` back to `CRed_NonDelegatingGetInterface` and `CBlue_NonDelegatingGetInterface`, where they can be handled correctly.

To see how the interaction between the `GetInterface` and the `NonDelegatingGetInterface` methods operates, the following examples show simplified versions of the `CRedGreenBlue_GetInterface`, `CRed_GetInterface`, and `CRed_NonDelegatingGetInterface` functions.

In Listing 2-19 on page 2-84, the `CRedGreenBlue_GetInterface` handles requests for `IGreen` directly but forwards requests for `IRed` to `CRed_NonDelegatingGetInterface` and requests for `IBlue` to `CBlue_NonDelegatingGetInterface`. The two `NonDelegatingGetInterface` methods are called via two `INonDelegatingBase` interface pointers, `_this->m_CRed` and `_this->m_CBlue`.

Listing 2-19. GetInterface Method Example

```
VCSE_MRESULT EXAMPLES_CRedGreenBlue_GetInterface(
        VCSE_IBase_ptr     base,
        const VCSE_RefIID  iid,
        VCSE_IBase_ptr     *iptr)
{
  /*
   * GetInterface method for supplying the requested interface
   */
  EXAMPLES_CredGreenBlue *_this = (EXAMPLES_CredGreenBlue *)base;

   if (!iidcmp(iid, VCSE_IBase_IID))
      *iptr = REINTERPRET_CAST(VCSE_IBase_ptr,
                         STATIC_CAST(EXAMPLES_IGreen,_this));
   else if (!iidcmp(iid,EXAMPLES_IGreen_IID))
      *iptr = VCSE_IBase_ptr)STATIC_CAST(EXAMPLES_IGreen,_this);
   else if (!iidcmp(iid,EXAMPLES_IRed_IID))
      VCSE_INonDelegatingBase_NonDelegatingGetInterface
            (_this->m_CRed,iid,iptr);
   else if (!iidcmp(iid,EXAMPLES_IBlue_IID))
      VCSE_INonDelegatingBase_NonDelegatingGetInterface
            (_this->m_CBlue,iid,iptr);
   else
      return (VCSE_MRESULT)MR_NOT_SUPPORTED;
   return (VCSE_MRESULT)MR_OK;
}
```

The non-delegating CRed_NonDelegatingGetInterface shown in Listing 2-20 on page 2-85 handles requests for IBase and IRed directly since these are both implemented by CRed. Note that the request for IBase is satisfied by returning the INonDelegatingBase interface pointer.

Listing 2-20. Non-Delegating GetInterface Example

```
VCSE_MRESULT EXAMPLES_CRed_NonDelegatingGetInterface(
     VCSE_IBase_ptr     base,
     const VCSE_RefIID  iid,
     VCSE_IBase_ptr     *iptr)
{
  /*
   * GetInterface method for supplying the requested interface
   */
   EXAMPLES_CRed *_this = (EXAMPLES_CRed *)base;
   if (!iidcmp(iid, VCSE_IBase_IID))
      *iptr = REINTERPRET_CAST(VCSE_IBase_ptr,
            STATIC_CAST(VCSE_INonDelegatingBase,_this));
   else if (!iidcmp(iid,EXAMPLES_IRed_IID))
      *iptr = (VCSE_IBase_ptr, STATIC_CAST(EXAMPLES_IRed,_this);
   else
      return (VCSE_MRESULT)MR_NOT_SUPPORTED;
   return (VCSE_MRESULT)MR_OK;
}
```

The delegating CRed_GetInterface, shown in Listing 2-21, simply hands all requests back to CRedGreenBlue_GetInterface using the cached interface pointer for the outer component held in this->m_pIBase_outer. If the request is for IRed, it will subsequently get handled by CRed_NonDelegatingGetInterface. Otherwise, it will be handled by CBlue_NonDelegatingGetInterface or directly by CRedGreenBlue_GetInterface.

Listing 2-21. Delegating GetInterface Example

```
VCSE_MRESULT EXAMPLES_CRed_GetInterface(
        VCSE_IBase_ptr    base,
        const VCSE_RefIID iid,
        VCSE_IBase_ptr    *iptr)
{
  /*
   * GetInterface method for supplying the requested interface.
   * Aggregated component delegates the responsibility to
   * the outermost aggregating component
   */

   EXAMPLES_CRed *_this = (EXAMPLES_CRed *)base;

   return
VCSE_IBase_GetInterface(_this->m_pIBase_outer,iid,iptr);
}
```

To enable the three components to call each other's `GetInterface` method, they have to maintain interface pointers to each other. In the examples above, these are represented by `m_CRed` and `m_pIBase_outer`. These pointers are established as the aggregating component creates the aggregated components as part of its own creation process.

The VIDL compiler automatically generates the correct versions of the delegating and non-delegating implementations of `GetInterface` for components that support aggregation, and the entire mechanism outlined above is effected by the automatically generated code. The actual code generated by the VIDL compiler differs in detail from the code in the previous example since it, for example, caches interface pointers to optimize the execution of the component.

Company Namespace Registration

The registration and use of a company namespace or tag is a key element of the approach taken to ensure the names of global entities, such as interfaces and components that are developed by various companies, remain unique. Each organization involved in developing and distributing VCSE components must register a unique namespace and ensure that all their components and interfaces are named within that namespace.

Each global name must be defined within the originating company namespace to ensure that no name clashes can occur. The organization that registers a company namespace is responsible for ensuring that all names defined within the company namespace are unique. An organization is at liberty to define subsidiary namespaces if that simplifies the task of ensuring that all names defined within the company namespace are unique.

An organization that wishes to register the use of a company namespace should send a request to `vcse.register@analog.com`, specifying the desired namespace tag and providing information, such as the full name of the organization and contact information for the person making the request. In general, namespace tags will be registered on a first come first served basis. Analog Devices, Inc. has already registered the `ADI` namespace for its components. The `EXAMPLES` namespace has been reserved for ADI's example interfaces and components used in the VCSE documentation and tutorials.

The `LOCAL` namespace has also been reserved for interfaces and components that will not be distributed outside of creating environment.

3 STANDARD INTERFACES

VisualDSP++ Component Software Engineering (VCSE) defines some standard interfaces offering an essential set of services that any component can exploit as well as a consistent environment for component developers and users.

The standard interfaces are defined within the VCSE namespace. The current set of defined interfaces consists of:

- The `IMemory` interface, which allocates and frees memory as required by a component. An application implements `IMemory` and supplies it to a component. The component exploits this interface to allocate and free memory. For more information, see "IMemory Interface" on page 3-2.

- The `IAlgorithm` interface, which defines a consistent set of services that all VCSE compliant algorithms must provide. All VCSE algorithms are expected to extend the `IAlgorithm` interfaces; therefore, the methods of `IAlgorithm` are available in each VCSE algorithm. For more information, see "IAlgorithm Interface" on page 3-14.

- The `IAlgorithm2` interface, which defines an additional set of services that VCSE compliant algorithms may provide. The `IAlgorithm2` interface extends the `IAlgorithm` interface by defining an efficient memory handling mode for memory-constrained algorithms. For more information, see "IAlgorithm2 Interface" on page 3-18.

- The `IMemorySwapper` interface, which is a standard interface that provides support for swapping data between different memory locations. For more information, see "IMemorySwapper Interface" on page 3-28.
- The `IInstanceFactory` interface, which provides a mechanism that allows a component implementing this interface to supply one or more instances of a requested interface to its users. For more information, see "IInstanceFactory Interface" on page 3-33.
- The `IError` interface, which provides a set of services that enable an application to have centralized error handling across multiple components. The `IError` interface is normally implemented by the application and passed to components, allowing them to report errors in a unified way. For more information, see "IError Interface" on page 3-37.
- The `IName` interface, which can be supported by a component to provide user-friendly names to be obtained for the instances of a component. For more information, see "IName Interface" on page 3-43.

IMemory Interface

The allocation of resources to the various sections of a DSP program often is one of the most difficult aspects of application building. While VCSE supplies a means of formalizing the structure of an application into components performing specific algorithmic or device handling tasks, it does not seek to impose any particular policy regarding resource allocation.

There is one area, however, in which the needs of the application and the needs of the VCSE model are likely to interact — memory allocation. The application will probably need to allocate, either statically or dynamically, working buffers for various purposes, while the VCSE model requires the allocation of memory areas to hold the management and user data associated with each created component instance.

In order to meet these needs, VCSE provides a standard memory allocation interface, `VCSE::IMemory`, to support the VCSE model and to provide application builders considerable freedom in meeting their applications' memory allocation requirements. In addition, `VCSE::IMemory` allows the allocation paradigm to be extended to other resources or to more sophisticated memory allocators.

IMemory and Component Instance Creation

There are two aspects of memory allocation associated with a VCSE component: the storage required to hold the fixed-sized per-instance component data, including VCSE management data; and the dynamic storage requirements of the instance's processing. The component's client can provide an `IMemory` interface to satisfy both needs when a new instance of the component is being created.

A VCSE component is not usually expected to allocate and free memory directly, but instead, to invoke an allocation mechanism provided by its client to carry out such services on its behalf. The `VCSE::IMemory` interface provides such a mechanism allowing a component to request the allocation of specified amounts of various types of memory and their subsequent freeing. The `IMemory` interface is:

- used to obtain memory for the component's instance data as well as the instance data for any aggregated component
- stored in each component's instance as well as in any aggregated component's

- used by the component's methods to obtain and free working memory
- used during instance destruction to free the component instance as well as the instance data for any aggregated component

If desired, an application is free to provide different `IMemory` interfaces, which may implement different allocation strategies, to different components or to different instances of the same component.

To see how a client supplies an `IMemory` instance when creating a component instance, consider the signature of the creation function that VCSE generates for a component `C1` defined within namespace `NS1`:

```
VCSE_MRESULT NS1_C1_Create( const VCSE_IBase_ptr  outer,
                                  VCSE_RefIID          iid,
                                  VCSE_IBase_ptr*      iptr,
                                  VCSE_IBase_ptr       ienvp,
                                  VCSE_HANDLE          token );
```

Parameters `ienvp` and `token` are associated with resource allocation. The `ienvp` argument is an interface pointer obtained from a component that implements `VCSE::IMemory` and possibly other resource allocation interfaces. If an application wishes to control the allocation of memory for a particular component instance, then it should supply a non-`NULL` `ienvp` argument. The allocation component may also implement interfaces that support the allocation of other resources or may implement a more sophisticated memory allocation interface. However, if the client wishes to have control over placement of the component instance's data, then the `ienvp` pointer must provide support for the `VCSE::IMemory` interface.

The second allocation parameter, `token`, provides a means of passing an arbitrary value into the methods defined in `VCSE::IMemory` or other resource allocation interfaces. The `token` value is stored in the newly created component instance's data and is provided to each `Allocate` or `Free` call made by the instance. The component providing the `IMemory` interface may not require specific `token` values, but if it does, then it must describe in its documentation what these values are or how to obtain them. For example, an allocator can use `token` values to implement a strategy of allocating predefined resources to specific component instances.

When an `IMemory` is not supplied at the instance creation time because `ienvp` is `NULL`, memory for the instance's data is obtained and freed entirely under control of the component. If `ienvp` is not `NULL` and calling `GetInterface` on it does not find a `VCSE::IMemory` interface, then the `_Create` function returns an error and the component is not created. The VIDL generated shell uses the macros `__VCSE_malloc` and `__VCSE_free` to allocate and free the instance data, and the component's methods may use these mechanisms for working memory as well.

The default implementations of these macros cause a `NULL` to be returned from `Allocate` and take no action when `Free` is invoked. By default, component creation fails if an `IMemory` interface is not supplied and `__VCSE_malloc` is invoked instead. If component developers want to use the macros as a fallback method, they should be given appropriate definitions in a user-modifiable section of the component header file.

The signature of the first macro is:

```
#define __VCSE_malloc(S)
```

where `S` is the size of the required storage area in the same units as the ones used in C library function `malloc`. The macro returns a valid `VCSE::ADDRESS` value or the `NULL` error indicator.

The second macro's signature is:

```
#define __VCSE_free(ADDR)
```

where `ADDR` is a `VCSE::ADDRESS` value previously obtained from `__VCSE_malloc`. The macro does not return a value.

IMemory Interface Definition

The interface contains only two methods: one for requesting the allocation of a block of memory that meets specified requirements for placement, lifetime, length, and alignment; and one for freeing up a previously obtained block.

The VisualDSP++ Interface Definition Language (VIDL) file that defines `IMemory` also contains the definition of a `struct` type whose members quantify a request for a block of memory in terms of its context, placement, lifetime, length, and alignment. The context member of the structure provides an indication of the use of the requested memory rather than a requirement it must meet. A suitably initialized variable of this type is passed as an argument to the allocation method. Constants denoting valid values for some of the memory request structure's members are specified in enumeration definitions.

The `IMemory` interface definition is shown in Listing 3-1. The interface's methods are described later in this section.

Listing 3-1. IMemory Interface Definition

```
namespace VCSE {

   enum MemType {
      MemAnyType   = 0,
      MemPrimary   = 1,
      MemSecondary = 2,
      MemExternal  = 4,
      MemBank      = 8
   };
   enum MemLifetime {
      MemAnyLifetime = 0,
      MemScratch     = 1,
      MemPersistent  = 2
```

```
    };
    enum MemContext {
       MemInstance = 1,
       MemWorking  = 2
    };

    struct _MemRequest {
       unsigned int   Length;
       unsigned short Alignment;
       unsigned short TypeFlags;
       unsigned short LifetimeFlags;
       unsigned short Context;
       char           BankName[32];
    };

    typedef struct _MemRequest MemRequest;

    interface IMemory extends IBase {
       MRESULT Allocate( [in]  MemRequest Request,
          [in]  HANDLE     Token,
          [out] ADDRESS    Allocation);
       MRESULT Free    ( [in] ADDRESS    Allocation,
          [in] HANDLE      Token);
    };
}
```

Type and Enumeration Descriptions

MemRequest

A client of `IMemory` uses a `MemRequest` structure to describe the attributes of a region of memory that it needs. All the attributes are mandatory: a client must provide valid values for each of them, and a conforming implementation of `IMemory` must satisfy each of them. Some of the attributes can be *multivalued*.[1] For example the value passed for the `TypeFlags` field can be the result of ORing together `MemPrimary` and `MemSecondary`. The combined value being passed requests that the required memory be of either type. An implementation that does not satisfy each attribute does not conform to the interface, but may be useful during application or component development for testing, sizing, or tracing purposes.

The following table lists the members of the `MemRequest` structure and describes their use.

Table 3-1. MemRequest Structure Members

Member	Description
`Length`	The length of the region of memory being requested. The length is measured in addressable units: • on a byte-addressable architecture, a value of 1 means one byte • on a word-addressable architecture, a value of 1 means one word
`Alignment`	The minimum alignment the allocated region must have. Alignment values are measured in addressable units. For example, on a byte-addressable architecture, a value of 4 means the allocated memory must begin at an address that is a whole multiple of 4 bytes. A value of 0 signifies the same alignment that Standard C library function `malloc` supplies—the maximum alignment requirement of the standard C `scalar` types on the target architecture. There are architectures on which certain algorithms are considerably more efficient if their data is aligned more strictly than their basic type requires. A conforming `IMemory` implementation must document the maximum alignment that it can guarantee.
`TypeFlags`	A bit mask specifying the types of memory which can be used to satisfy the allocation request. The meaning of each bit position is defined in the description of the `MemType` enumeration. The bits are examined in the same order that the nonzero members of `MemType` are defined. The first requested type from which memory can be allocated that also satisfies the other request attributes is used. If no bits are set, then an implementation can supply any type of memory.

[1] A multivalued attribute has a single integer value consisting of the combination of several bit values.

Table 3-1. MemRequest Structure Members (Cont'd)

Member	Description
LifetimeFlags	A bit mask specifying the expected duration of the allocation. The meaning of each bit position is defined in the description of the MemLifetime enumeration. The bits are examined in the same order that the nonzero members of MemLifetime are defined. The first requested lifetime from which memory can be allocated that also satisfies the other request attributes is used. If no bits are set, then an implementation can assume any duration is acceptable. In allocation requests that specify multiple values for both type and lifetime, the type takes priority.
Context	One of the two values defined by the MemContext enumeration. The Context is not a requirement, which the allocation must meet, but provides additional information to the memory allocator on the use of the allocated memory.
BankName	A C string of up to 31 characters plus a terminating zero byte specifying a named memory bank from which the allocation must be made. The string must be empty (BankName[0]==0) unless TypeFlags includes the MemoryBank flag. In the latter case, BankName must contain the name of a memory bank from which the requested memory can be allocated.

TypeFlags

The TypeFlags member of the struct is a bit significant enumeration of the types of memory from which a client of IMemory can request an allocation. The MemType enumeration defines the different types of memory that can be requested along with the corresponding bit pattern. The names and general descriptions of the memory types are presented in Table 3-2 on page 3-10. The following supplementary tables give a more precise definition on a per-architecture basis.

A component using IMemory to allocate and free memory must document which memory types it requires. If a component requires an allocation from named banks (MemBank), it must document what steps the user must take during the building or linking of his/her application in order to comply with memory bank requests.

Table 3-2. MemType Enumeration Members

Memory Type	Description
`MemPrimary`	The fastest (non-register) memory, internal to the processor core, suitable for data placement
`MemSecondary`	An alternative internal memory for data placement
`MemExternal`	A memory region, external to the processor core data memory
`MemBank`	A named memory region

Table 3-3. ADSP-BF53x Blackfin Processor Memory Types

Memory Type	ADSP-BF53x Memory
`MemPrimary`	`L1` data memory
`MemSecondary`	`L2 SRAM`
`MemExternal`	External memory
`MemBank`	Named memory bank

Table 3-4. ADSP-21xx DSP Memory Types

Memory Type	ADSP-21xx Memory
`MemPrimary`	`dm` memory
`MemSecondary`	`pm` memory
`MemExternal`	External memory
`MemBank`	Named memory bank

LifetimeFlags

The `LifetimeFlags` member of the structure is a bit-significant enumeration, which lists the expected lifetimes associated with a memory allocation. The `MemLifetime` enumeration defines the different life times of memory that can be requested along with the corresponding bit pattern. An allocator may use the value of this attribute to select between different allocation strategies.

Table 3-5 describes the members of the enumeration.

Table 3-5. MemLifetime Enumeration Members

Memory Lifetime	Description
`MemScratch`	The allocation will have a relatively short lifetime and may, for example, be freed when the `Deactivate` method of an algorithm is invoked.
`MemPersist`	The allocation will have a long lifetime and may, for example, only be freed when the associated component is destroyed.

Context

The `Context` member specifies the context in which the allocated memory will be used. Its value must be one of the members of the `MemContext` enumeration, which defines two constants to describe the context in which a memory allocation request is made, as described in Table 3-6.

Table 3-6. MemContext Enumeration Members

Allocation	Context Description
`MemInstance`	The allocation request is for memory in which to place a component instance record.
`MemWorking`	The allocation request is for other purposes; for example, a workspace buffer for an algorithm or device handler.

Method Descriptions

Supported methods for the IMemory interface include:

- Allocate
- Free

Allocate

The Allocate method is invoked to supply memory as specified in the MemRequest structure passed as its first argument. If a non-NULL IMemory interface is available to a component's Create function, then its Allocate method is used by the VCSE generated factory code to obtain memory to hold the new instance of the component. The IMemory interface is stored in the component instance's data; therefore, the component methods may also invoke Allocate to obtain working memory.

A component's Create function has a value of type VCSE::HANDLE passed to it. This value must be passed as the Token argument to all Allocate and Free calls made by the component instance, so it is stored in the component's instance data as well. The Token argument is a general-purpose mechanism for passing an arbitrary value to the memory allocation methods. Its use is optional. The documentation for a component implementing IMemory must state whether or not it uses the Token value and, if it does, what the valid values are. In the generated C/C++ code, VCSE::HANDLE is represented as void* on ADSP-BF53x and long int on ADSP-21xx processor architectures.

The method's parameters and possible return values are described in Table 3-7 on page 3-13.

The standard VCSE type VCSE::ADDRESS is used to convey the start address of the allocated memory area back to the Allocate's caller. In the generated C/C++ code, VCSE::ADDRESS is represented as void* on

ADSP-BF53x and `long int` on ADSP-21xx processor architectures, so the returned value must be cast to an appropriate pointer type before the allocated memory can be accessed.

Table 3-7. Allocate Method Parameters and Return Values

Parameter	Type	Description
`Request`	`MemoryRequest`	Contains the values of the attributes that the allocated region of memory must satisfy.
`Token`	`HANDLE`	If called from a component, must contain the `HANDLE` value passed to the instance's `Create` function; otherwise, must contain a suitable value as described in the memory allocation component's documentation.
`Allocation`	`ADDRESS`	Returns the start address of the allocation if the allocation has been successful.
Returned value	`MRESULT`	Indicates the success or failure of the request. A value of `MR_OK` indicates the complete success, while the following values denote various failure conditions. • `MR_NO_MEMORY` All the memory requirements are met except the length. • `MR_BAD_ALIGNMENT` The alignment requirement is out of range. • `MR_BAD_MEMTYPE` The requested memory type is not valid or is not supported. • `MR_BAD_MEMLIFE` The requested memory lifetime is not valid or is not supported. • `MR_BAD_CONTEXT` The supplied context is not a valid value. • `MR_BAD_MEMBANK` The requested memory bank name is not valid or is not supported. • `MR_BAD_HANDLE` The value supplied in `Token` is not valid.

Free

All memory obtained by calling `Allocate` must be released by a corresponding call to `Free` when the memory is no longer required. The request to `Free` an allocation obtained by a call to `Allocate` must be made on the same instance of the `IMemory` interface as the allocation was made. The `Token` parameter must have the same value as the corresponding argument to `Allocate` had when requesting the memory.

The result code values that `Free` may return are: `MR_OK` if the action is completed without an error; `MR_NOT_ALLOCATED_MEM` if the implementation can detect that it is asked to free memory that this instance of the `IMemory` implementation has not allocated; and `MR_NOT_COMPLETED` if any other error condition has occurred.

Under no circumstances should the client attempt to access the freed memory again—no matter what result code `Free` returns.

IAlgorithm Interface

The `VCSE::IAlgorithm` interface represents a set of methods, which must be supported by all VCSE based algorithms. Although each algorithm component must have an implementation of each method, the actual implementation can be very simple; for example, it can return `MR_OK` as its only action.

Since an algorithm is not expected to allocate but rather to use memory allocated by its user, there is a standard memory interface defined that it can use to obtain the memory to meet its needs. The user of an algorithmic interface supplies the memory interface to the algorithm at a component's creation time. See the `VCSE::IMemory` description on page 3-2 for details of this interface.

The algorithm interface also enables the user to supply an error handling interface, which the algorithm instance can use to report errors. See the VCSE::IError description on page 3-37 for details of this interface.

IAlgorithm Interface Definition

The IAlgorithm interface defines a common set of basic control methods that all VCSE based algorithms are required to provide. Since algorithms vary considerably in their requirements for the specification of coefficient values, data sources and destinations, and the like, IAlgorithm makes no requirements in this area. Algorithm providers are expected to extend IAlgorithm with methods allowing the user to specify the particulars of an algorithm instance in a natural way. This can be achieved by providing one or more setup methods that accept fixed sets of arguments and corresponding processing methods without parameters, or by providing one or more processing methods that take suitable arguments.

The methods in this interface must return the result code MR_OK if they execute entirely without problems. The general result code, MR_NOT_COMPLETED, is available for other cases, but algorithm developers are encouraged to define and document specific result codes. The structure of MRESULT codes is described in "VCSE Assembler Macros" on page A-1.

The IAlgorithm interface definition is shown in Listing 3-2. The interface's methods are described on page 3-16.

Listing 3-2. IAlgorithm Interface Definition

```
#include <VCSE_IError.idl>

namespace VCSE {
  interface IAlgorithm extends IBase {
    MRESULT Reset();
    MRESULT Activate();
```

```
    MRESULT Deactivate();
    MRESULT SetAlgorithmErrorInterface(
                         [in] IError ErrorReporter,
                         [in] int Level);
  };
};
```

Method Descriptions

Supported methods for the `IAlgorithm` interface include:

- `Reset`
- `Activate`
- `Deactivate`
- `SetAlgorithmErrorInterface`

Reset

An algorithm instance can be set to a default operational state by calling the `Reset` method. The documentation for the algorithm must describe the default state and the effects of executing the algorithm in the default state.

Calls to the `Reset` method can be made at any time after the algorithm interface has been instantiated.

Activate

An algorithm component must be notified when a particular instance of the interface is about to be used by invoking the `Activate` method to allow the algorithm to prepare itself for optimized execution. The `Activate` method allows the algorithm to execute any necessary initialization or setup code prior to repeated use of the instance of the algorithm.

The `Activate` method must be invoked before using any core computation methods supplied by an interface, which directly or indirectly extends `IAlgorithm`. When multiple instances of an algorithm are created, `Deactivate` and `Activate` are expected to be invoked between calls on different instances.

Deactivate

When an algorithm instance will not be invoked for a period, it must be notified of this by a call to its `Deactivate` method. The `Deactivate` method call enables the algorithm to take any actions to reduce resources the algorithm is consuming; for example, to move some data from the internal to external memory.

A `Deactivate` call must be subsequent to an `Activate` call. Conversely, after a `Deactivate` call, a call to `Activate` must be made before invoking an algorithm interface with a call to any method that triggers the algorithm computations. When multiple instances of an algorithm are created, `Deactivate` and `Activate` are expected to be invoked between calls on different instances.

SetAlgorithmErrorInterface

The `SetAlgorithmErrorInterface` method allows the user of an algorithm to supply an error handler interface to be used by the algorithm instance to report any errors the algorithm detects. If no `SetAlgorithmErrorInterface` call is made, or if the passed interface pointer is `NULL`, then the algorithm will not report errors.

The `Level` parameter is a bit mask whose non-zero bits specify which of the various levels of error reports are required by the caller. See the `VCSE::IError` interface description on page 3-37 for the correspondence of bit positions to error levels.

One error handler interface may be passed into multiple instances of the same algorithm component and into instances of different algorithms. However, if a client application holds more than one interface pointer from the same instance of an algorithm component, then calling `SetAlgorithmErrorInterface` affects *all* interface pointers. (After an algorithm component is instantiated by calling its `Create` function, the client can obtain further interface pointers by calling the `GetInterface` method, assuming the algorithm implements more than one interface.)

Calls to the `SetAlgorithmErrorInterface` method can be made at any time after the algorithm interface is instantiated. For instance, it can be called once immediately after the instantiation, requesting only notification of catastrophic errors; and again at some particular point in the user's code to change the level of information being returned.

Valid Sequence of Method Calls

Figure 3-1 shows the valid sequences of the `IAlgorithm` method calls. In general, the methods `Reset` and `SetAlgorithmErrorInterface` can be invoked at any time between an algorithm instance creation and destruction.

IAlgorithm2 Interface

The `VCSE::IAlgorithm2` interface extends `VCSE::IAlgorithm` by defining an efficient memory handling model for memory constrained algorithms. The memory model separates memory areas used for processing and storage. After creating an `IAlgorithm2` component, memory resources must be assigned to enable the instance to perform its functions. The client assigns the memory resources to meet algorithm and application requirements. The `VCSE::IAlgorithm2` interface extends `VCSE::IAlgorithm` to facilitate the assignment and management of the required memory resources.

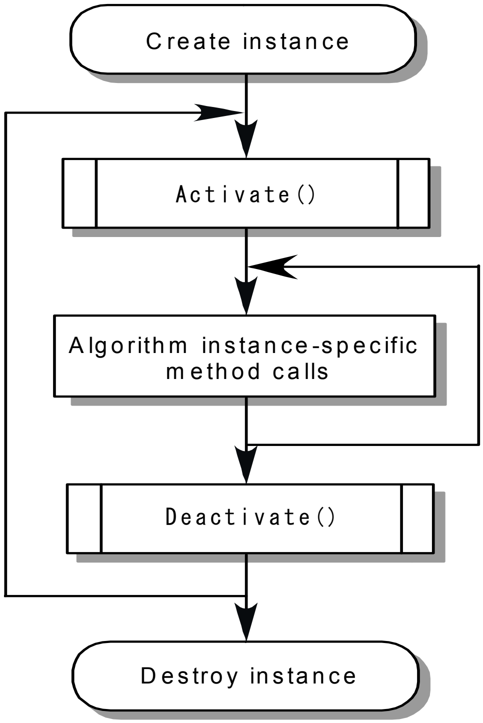


Figure 3-1. Typical Method Calls Sequence

Since an algorithm is not expected to allocate memory but rather to use the memory allocated by its user, a standard memory interface can be used to obtain the required amount memory it needs. The user of an algorithmic interface supplies the memory interface to the algorithm at a component's creation time. See the `VCSE::IMemory` description on page 3-2 for details of this interface.

In general, an `IAlgorithm2` component will use the supplied `VCSE::IMemory` interface only to allocate its instance data and will require additional memory buffers that must be assigned after creation. The additional memory buffers represent time-critical data, typically for algorithmic processing.

The `IAlgorithm2` interface enables an algorithm to operate efficiently in memory constrained situations by supporting the swapping of critical data between fast but limited internal memory and slower but more available external memory. The `IAlgorithm2` interface enables the user to supply a memory swapping interface, which the algorithm instance can use to initialize a component used to swap the algorithm's memory resources. The concepts of memory resources and memory swapping are discussed in "IAlgorithm2 Memory Concept" on page 3-21. See the `VCSE::IMemorySwapper` description on page 3-29 for details on the memory swapping interface.

IAlgorithm2 Memory Concept

An implementation of the `IAlgorithm2` interface may require the use of several categories of memory. A required memory object from one of these categories is called a memory resource. Definitions and short descriptions of these memory resources are provided below to allow a complete description of the `IAlgorithm2` interface.

1. VCSE Instance Memory — Memory needed to support the VCSE component structure. Each instance of a component uses a separate instance memory. A component's instance memory contains pointers to working memory and possibly to storage memory.

2. Working Memory — Memory used by algorithms to perform algorithm-specific work. Working memory should be located within low latency memory, such as internal memory, for optimum performance. Each instance contains its own set of working memory buffers.

   - Persistent Working Memory — Working memory whose content needs to be preserved throughout the lifetime of component.
   - Scratch Working Memory — Working memory whose content only needs to be preserved between the activation and deactivation of a component instance.

3. Storage Memory — If memory swapping is used, the persistent working memory content for a component instance is saved in Storage Memory after deactivation and loaded from that location before activation. This memory should be located in high capacity memory in order to have the lowest strain on memory requirements.

4. Stack Memory — Memory used for local variables during method calls.

5. Algorithm Look Up Tables — Memory shared by all instances of a particular `IAlgorithm2` component. The `IAlgorithm2` component only reads this memory and does not write to it.

An `IAlgorithm2` component must document the requirements for all memory categories, including alignment and memory placement.

Ideally users should place all working memory in internal memory, but internal memory is typically characterized as having a low capacity. When `IAlgorithm2` components are integrated in a system, the developer can adopt any one, or any combination, of the following strategies to deal with limited availability of internal memory.

- The first is the trivial case, when the system integrator determines that there is sufficient internal memory to satisfy all the needs of each component instance during its entire life. The internal memory resource or resources should accommodate both the persistent and scratch memory requirements of the component instance. Each component instance would have exclusive access to at least the persistent part of its internal memory resources. Another trivial case occurs if the application determines that the component instance can meet the real-time constraints even while working in the external memory.
- The second strategy involves using data caching, if it is available. In this case, the application allocates memory in the external memory and caching is enabled. Each component instance has exclusive access to at least the persistent part of its external memory resources.
- The third strategy involves using memory swapping techniques. Memory resources that conform to the memory swapping model have two locations, or images. These two locations are its working and storage locations. The working location is low latency internal memory that is well suited for processing. The storage memory should be large capacity external memory, but may be relatively

slow. When an algorithm processes data, its memory resources should reside in internal memory. While the algorithm is inactive, its memory resources should reside in external memory to allow other objects to make use of the limited internal memory. Conceptually, an instance should have persistent working memory resources swapped into the internal memory and scratch working memory resources assigned in the internal memory when it is activated. These internal memory resources can be assigned to other instances when this instance is deactivated. Before assigning it to other instances, the persistent working memory resources of this instance must be swapped out to the storage memory.

IAlgorithm2 Interface Definition

The `IAlgorithm2` interface extends the definition of an `IAlgorithm` algorithm to enable flexible memory use in a standard way. Since algorithms and application requirements vary considerably, `IAlgorithm2` memory use is configured through its interface to assist meeting a wide range of requirements. The developer must determine the best methods for meeting the algorithm and application requirements.

The methods in this interface must return the result code `MR_OK` if they execute entirely without problems. The general result code `MR_NOT_COMPLETED` is available for other cases, but algorithm developers are encouraged to define and document their own specific result codes. The structure of `MRESULT` codes is described in "VCSE Assembler Macros" on page A-1.

The `IAlgorithm2` interface definition is shown in Listing 3-3. The interface's methods are described later in this section.

Listing 3-3. IAlgorithm2 Interface Definition

```
#include <VCSE_IAlgorithm.idl>
#include <VCSE_IMemorySwapper.idl>
```

```
namespace VCSE {
   interface IAlgorithm2 extends IAlgorithm {

      // methods

MRESULT AssignMemoryResources(
 [in] size_t    NumOfWorkingMemResources,
 [in, size_is(NumOfWorkingMemResources)]
     VCSE::ADDRESS WorkingMemLocations[],
 [in] size_t NumOfStorageMemResources,
 [[in, size_is(NumOfStorageMemResources)]
      VCSE::ADDRESS    StorageMemLocations[]);

MRESULT ConfigureMemorySwapper();

      // properties

      [set, get] ::VCSE::IMemorySwapper MemorySwapper;

   };
};
```

Method Descriptions

Supported methods for the `IAlgorithm2` interface include:

- `AssignMemoryResources`
- `ConfigureMemorySwapper`

AssignMemoryResources

An algorithm instance normally requires that certain memory resources be assigned to it after creation. Use the `AssignMemoryResources` method to assign memory resources. The number of working memory resources is contained in the `NumOfWorkingResources` argument. The number of resources must include all persistent and scratch working buffers. The `WorkingMemLocations` argument is an array of starting addresses of each of the working memory resources. The order of the starting addresses in `WorkingMemLocations` is specified in the documentation for the `IAlgorithm2` component. The number and locations of the storage areas are passed in through the `NumOfStorageResources` and `StorageMemLocations`, respectively. The order of `StorageMemLocations` is also specified in the documentation for the `IAlgorithm2` component.

A one-to-one correspondence should exist between the persistent working memory regions in the `WorkingMemLocations` parameters and the storage memory regions in the `StorageMemLocations` parameter. Each of the corresponding entries marks the starting address of the region specified to hold the data when the component instance is activated and the starting address of the region specified to hold the corresponding data when the component instance is deactivated. The number of entries in the `StorageMemLocations` should be equal to the number of persistent working memory locations. The lengths of each region should also be equal. If the application does not require that a persistent working memory region be swapped, it should make the corresponding entry in the `StorageMemLocations` point to the same memory area as the corresponding entry in `WorkingMemLocations`.

ConfigureMemorySwapper

The `ConfigureMemorySwapper` method is invoked by the client of an `IAlgorithm2`-based component to notify it that it should now supply its configuration parameters to the `IMemorySwapper` interface which was already supplied to the algorithm. The client invokes `ConfigureMemorySwapper` once it has ensured that the associated memory swapper is ready to be used. The `IAlgorithm2` type component is expected to supply these parameters by invoking the `InitMemorySwapper` method of the `VCSE::IMemorySwapper` interface. The client must set the `MemorySwapper` property of this interface before invoking the `ConfigureMemorySwapper` method. The `AssignMemoryResources` method must also be called prior to calling `ConfigureMemorySwapper`.

Properties

MemorySwapper

The `MemorySwapper` property is used to notify the algorithm of the interface pointer for an `IMemorySwapper`-based component. The `IMemorySwapper` interface is used by the application to perform swapping between persistent working memory and associated storage memory if the programmer chooses to use the facilities provided by an `IMemorySwapper` based component. The property allows the associated interface pointer to be both set and read.

Valid Sequence of Method Calls

The `IAlgorithm2` components follow essentially the same sequence of method calls as any other `IAlgorithm` interface. There are a few additional restrictions imposed, due to the extension of the `IAlgorithm` interface.

The typical order of method calls during the creation of the component instance is modeled in Listing 3-4.

Listing 3-4. IAlgorithm2: Creation Phase Call Sequence Pseudo Code

```
IAlgorithm2->Create
IMemorySwapper->Create
IAlgorithm2->setMemorySwapper
IAlgorithm2->AssignMemoryResources
IAlgorithm2->ConfigureMemorySwapper
IMemorySwapper->Swap (Storage to Working Persistent)
IAlgorithm2->Activate
```

```
IAlgorithm2->Reset
IAlgorithm2->Deactivate
IMemorySwapper->Swap (Working Persistent to Storage)
```

As indicated in Listing 3-4, `AssignMemoryResources` must be called before `Reset`. `Reset` will typically work on the assigned memory resources. Also, `setMemorySwapper` and `AssignMemoryResources` must be called prior to calling `ConfigureMemorySwapper`.

An `IAlgorithm2` implementation can be used in two different modes. In synchronous mode each call on the `Swap` method will only return when the swap action is completed. In asynchronous mode the call of a `Swap` method may return before the swap operation is completed, and the application must use the `getSwapStatus` method to determine when the swap operation is complete. The asynchronous mode allows one instance of an algorithm to process data while a separate instance is actually being swapped.

The run-time steps of the algorithm for synchronous swapping are modeled in Listing 3-5.

Listing 3-5. IAlgorithm2: Run-time with Synchronous Swapping Pseudo Code

```
IMemorySwapper->Swap (Storage to Working Persistent)
IAlgorithm2->Activate
IAlgorithm2->Process
IAlgorithm2->Deactivate
IMemorySwapper->Swap (Working Persistent to Storage)
```

Listing 3-5 shows that the working memory buffers are valid at the completion of the `Swap` call. Algorithm specific processing can be called immediately after `Activate`.

In order to support asynchronous swapping, the order of method calls during run-time is modified in Listing 3-6.

Listing 3-6. IAlgorithm2: Run-time with Asynchronous Swapping Pseudo Code

```
IMemorySwapper->Swap (Storage to Working Persistent)
Do something else (or not)
while(IMemorySwapper->getSwapStatus != MR_SWAP_COMPLETE)
IAlgorithm2->Activate
IAlgorithm2->Process
IAlgorithm2->Deactivate
IMemorySwapper->Swap (Working Persistent to Storage)
Do something else (or not)
while(IMemorySwapper->getSwapStatus != MR_SWAP_COMPLETE)
```

Some `IAlgorithm2` components may decide to allow certain methods to be called while deactivated for performance reasons. Low cost method calls do not need to be bracketed by `Activate-Deactivate` method calls and memory swapping may be avoided. In this case, the component would work off the storage memory when deactivated and the working memory when activated. In this scenario it becomes the application's responsibility to determine when it is beneficial to swap and when not to swap memory.

It is the application's responsibility to ensure that it maintains the consistency of the contents of the working and storage memory during swapping.

IMemorySwapper Interface

The `VCSE::IMemorySwapper` interface is a standard interface that provides support for swapping data between different memory locations. The `IMemorySwapper` interface defines a standard mechanism that enables an instance of a component to move data buffers from one location to another, that is to swap memory locations. The definition provides a flexible interface that support synchronous and asynchronous swapping. The IMemorySwapper interface is expected to be closely associated with VCSE::IAlgorithm2 based components.

A component requiring memory swapping services must configure its own instance of an IMemorySwapper component.

IMemorySwapper Interface Definition

The IMemorySwapper interface consists of the two methods: Initialize, Swap and a property: SwapStatus. The Initialize and Swap methods in this interface must return the result code MR_OK if they execute entirely without problems. The general result code MR_NOT_COMPLETED is available for other cases, but algorithm developers are encouraged to define and document their own specific result codes. The structure of MRESULT codes is described in "VCSE Assembler Macros" on page A-1. The SwapStatus property is a read only. Whether a memory swapping is currently in progress or not for a interface can be checked by executing a getSwapStatus inlined function call. The getSwapStatus inlined function call returns MR_SWAP_INPROGRESS if a memory swapping is in progress or MR_SWAP_COMPLETE if it is not.

The IMemorySwapper interface definition is shown in Listing 3-7. The interface's methods are described later in this section.

Listing 3-7. IMemorySwapper Interface Definition

```
namespace VCSE
{
   interface IMemorySwapper extends IBase {
     enum Results_e {
        MR_SWAP_COMPLETE   = MR_ICONSTRUCT(WARN,1),
        MR_SWAP_INPROGRESS = MR_ICONSTRUCT(WARN,2)
   };
   enum  SwapDir_e {
```

```
        SWAP_IN   = 1,
        SWAP_OUT  = 2};
    enum  BlockingPolicy_e {
        NON_BLOCKING = 0,
        BLOCKING = 1};
    [ local ]
    MRESULT Initialize(
        [in]   size_t  NumberOfRegions,
        [in,size_is(NumberOfRegions)] ADDRESS AddrActivated[],
        [in,size_is(NumberOfRegions)] ADDRESS AddrDeactivated[],
        [in,size_is(NumberOfRegions)] size_t  Length[]
    );
    MRESULT Swap(
        [in]  SwapDir_e dir,
        [in]  BlockingPolicy_e block
    );
    [get] MRESULT SwapStatus;
}
```

The component that implements `IMemorySwapper` documents, which of the following two models it supports.

- **Memory Set Sharing.** In this model, both the `IMemorySwapper` and the `IAlgorithm2` implementations use the same table of swap locations. This is possible because the validity of the assignment of an `IMemorySwapper` interface pointer is a subset of the lifetime of the algorithm component. The advantage of using this method is that it saves space when `IMemorySwapper` / `IAlgorithm2` pairs are used. On the other hand, the swapper must verify that `AddrActivated` is equal to `AddrDeactivated` before swapping. This check is required because the table of swap locations is owned and constructed by the

`IAlgorithm2` instance. If `AddrActivated` is equal to `AddrDeactivated`, then swapping is not required. These are the problems characteristically associated with memory sharing.

- **Memory Set Ownership.** In this model, the memory swapper stores its own copy of the set of addresses of working and storage memory in a privately held buffer. The memory swapper controls this buffer. Since swap tables are constructed during initialization and only contain valid swaps pairs — no checking is needed before swapping a memory location. The disadvantage of using this method is the fact that the table between algorithm and its swapper is duplicated. A memory swapper using DMA may have to implement this method in any case.

Method Descriptions

Supported methods for the `IMemorySwapper` interface include:

- `Initialize`
- `Swap`

Initialize

Initialization of a memory swapper is performed via the `Initialize` method. The number of memory regions to be swapped is passed in through the `NumberOfRegions` argument. The `AddrActivated` argument is an array of starting addresses for each memory region used by an `IAlgorithm2` instance when it is activated. The locations of the corresponding regions used by an `IAlgorithm2` instance when it is deactivated are passed in through the `AddrDeactivated` argument. There should be a one to one correspondence between the activated and deactivated memory regions. The length of each region is passed in through the array `Length`.

Swap

The `Swap` method initiates the swapping procedure. The argument `dir` is a member of enumeration `SwapDir_e`. The enumeration `SwapDir_e` has two members `SWAP_IN` and `SWAP_OUT`. The `SWAP_IN` value signals to the memory swapper to move the data from the deactivated regions to the active regions. The `SWAP_OUT` value moves the data in the opposite direction, from active regions to the deactivated regions. The second parameter, `block`, is a member of enumeration `BlockingPolicy_e`. The enumeration `BlockingPolicy_e` describes the mode of transfer, either synchronously or asynchronously.

When the value of block is `BLOCKING` then all memory swapping is effected before the `Swap` method returns.

In general, when the value of block is `NON_BLOCKING`, then not all memory swapping is effected before the `Swap` method returns. The memory swapper processes the transfers as the application continues. In this case the property `SwapStatus` can be used to determine when the swap is completed.

Properties

The only property supported by the `IMemorySwapper` interface is `SwapStatus`.

SwapStatus

The `SwapStatus` property enables the application to determine when swap operation completes. The `SwapStatus` property is a read only property of `IMemorySwapper` interface. Executing the inlined call `getSwapStatus` on `IMemorySwapper` interface returns `MR_SWAP_COMPLETE` when swapping has completed, and `MR_SWAP_INPROGRESS` when swapping is in progress. The `SwapStatus` property enables the user to check for the completion of an asynchronous memory swap.

Intended Use

Normally an instance of a component implementing the IMemorySwapper interface is assigned to an instance of an IAlgorithm2 based component. Either the application or the instance of the IAlgorithm2 based component is expected to initialize the IMemorySwapper instance properly using IMemorySwapper->Initialize() method call. Before the application activates the instance of IAlgorithm2 component (by invoking its Activate method), the application invokes IMemorySwapper->Swap() with the dir parameter set to SWAP_IN. After the application deactivates the instance of IAlgorithm2 component (by invoking its Deactivate method), the application invokes IMemorySwapper->Swap() with the dir parameter set to SWAP_OUT.

If the swapping is to be performed asynchronously, the block parameter must be set to NON_BLOCKING. The application can then use the IMemorySwapper->getSwapStatus() inlined call method to determine if the requested swapping is in progress, or has completed. The application can continue to process other data while the memory swap proceeds.

If the swapping is to be performed synchronously, the block parameter must be set to BLOCKING. The IMemorySwapper->Swap() method returns only after the swapping operation completes.

IInstanceFactory Interface

The IInstanceFactory interface represents a set of creation operations in an abstract and localized manner. It is abstract in the sense that the client of the IInstanceFactory component does not care or know what component may be created to support any returned interfaces, only that it is a component implementing the requested interface. Creation is localized in that all requests for a given interface are performed using the IInstanceFactory interface. The IInstanceFactory interface represents the set of methods that allows a component implementing this interface to supply one or more instances of a requested interface. The use of the term *supply* means that the component implementing this interface internally creates an embedded instance of a component implementing the requested interface and returns a interface pointer to the new instance.

IInstanceFactory Interface Definition

The `IInstanceFactory` interface allows components that implement this interface to manufacture *instances* of another interface and provide them to the client. A single instance of a component can only provide one interface pointer to any of the interfaces that it implements. In order to have more than one interface pointer, users can create multiple instances of the component. Creating multiple instances of a component may not always be appropriate. If developers have a device that supports multiple channels each of which uses some shared control logic, then they need only one instance of the controller and separate interfaces to each of the channels. Also, using the standard component factory methods exposes the component to the application. For software design issues it is recommended that users minimize the number of direct references of specific components. References to the specific components should be localized to reduce the cost of modifications.

The `IInterfaceFactory` interface provides services intended to solve the problems with instantiation, as previously described. Instances can be created through instance factories, when the client requests an object implementing a specific interface. The implementor of the interface supporting `IInstanceFactory` is solely responsible for creating instances of the requested interfaces and can create these instances in whatever manner is suitable. The instances created via `IInstanceFactory` are normally intimately related to the instance that supports `IInstanceFactory`. The internal relationships between these interfaces should not be visible to the user of the interfaces. The provider of the instance factory decides what and how an object is created, as long as the user can expect the returned interface pointer to act as if it is a new instance. Also, creations are localized to the factory implementation.

The methods in this interface must return the result code `MR_OK` if they execute entirely without problems. The general result code `MR_NOT_COMPLETED` is available for other cases, but algorithm developers are encouraged to define and document their own specific result codes. The structure of `MRESULT` codes is described in "VCSE Assembler Macros" on page A-1.

The `IInterfaceFactory` interface definition is shown in Listing 3-5 on page 3-27. The interface's methods are described on page 3-35.

Listing 3-8. IInterfaceFactory Interface Definition

```
namespace VCSE {

   interface IInterfaceFactory extends VCSE::IBase {
      MRESULT RequestInterface(
         [in]   VCSE::RefIID id,
         [in]   int    selector,
         [out] VCSE::IBase   iface);
      MRESULT ReleaseInterface([in] VCSE::IBase iface);
   };
};
```

Method Descriptions

Supported methods for the `IInstanceFactory` interface include:

- `RequestInterface`
- `ReleaseInterface`

RequestInterface

The `RequestInterface` method is the means by which a client obtains an interface pointer for an instance of a desired interface from a component that implements `IInstanceFactory`. The `IInstanceFactory` implementation's documentation must specify the interfaces for which instances are generated. The parameter `id` specifies the unique interface identifier for the desired interface. The corresponding interface pointer is returned via the `iface` parameter, if it can be manufactured successfully.

It is possible that a component implementing this interface has several choices of how to supply a particular interface. The `parameter` selector is meant to identify the choice of how the interface is to be supplied. As an example, the interface for the audio codec could enumerate the different data paths (ADCs and DACs). The audio codec control component would also implement `IInstanceFactory` interface and use it to supply data handling interfaces. Its user can specify a member of the data path enumeration as the selector parameter in requesting a data handling interface. If used, values for selector parameter should be enumerated in the supplied interfaces.

ReleaseInterface

The `ReleaseInterface` interface is the means by which a client releases an interface, returned by `RequestInterfaces`, it no longer needs. The `iface` parameter is the interface that is requested to be released. The `iface` parameter must have been obtained through the same factory object via the `RequestInstance` method.

Example of Use

Consider an audio codec control component for BF-533 EZ-Kit, which includes a single physical audio codec. The audio codec has multiple ADC (analog-to-digital convertors) and multiple DAC (digital-to-analog convertors). Each ADC and DAC can be regarded as a data path. Thus multiple data paths exist in the audio codec and its users might need to access a specific ADC or DAC. The audio codec control component can either choose to implement the data handling interface or supply the data handling interface.

If the component chooses to implement the data handling interface, “a” component instance can only provide “an” interface pointer to the data handling interface. The component can no longer be a singleton, and multiple instances of the components will have to be created. Having multiple instances controlling a single physical codec can lead to design complications and might not be desirable in many cases.

The `IInstanceFactory` interface offers an elegant solution to this problem. It allows the audio codec component to be a *singleton*, meaning that only one instance of the component can exist. This component supplies the data handling interfaces to its users using the `RequestInterface()` method of `IInstanceFactory` interface. The audio codec control component embeds the data handling interface implementation within itself. In this way, the requester is not exposed to the exact type of object that is created but only to the interface that is needed.

The audio codec interface should enumerate its data paths. A member of this enumeration is used as the selector parameter of `RequestInterface()` method call in requesting access to a specific ADC or DAC.

IError Interface

The `VCSE::IError` interface defines a standard mechanism that enables an instance of a component to report errors or to pass other information regarding its operation to the component’s client. A standard interface, whose implementation is provided (directly or indirectly) by the controlling application, allows a standard error handling procedure to be used by the application. An application can use the interface to provide as simple or as complex an error handling process as it requires.

A component requiring error handling services must include a method (in one of the implemented interfaces) that allows the user to pass in an `IError` instance to be used for that purpose.

IError Interface Definition

The single method in this interface, `Error`, reports an error or records other information about the interface operations. The `Error` arguments enable its implementation to discover the severity of the event being reported and to receive arbitrary information about the event.

`IError` also contains a bit-significant enumeration of the various severity levels that can be reported to the method. Although a method call can supply only one specific level, the values are presented as bit-significant. Therefore, components handed an `IError` instance for error reporting may also be handed a bit mask specifying the severities the client is interested to receive. See the `VCSE::IAlgorithm` interface documentation on page 3-15 for an example.

The `IError` interface definition is shown in Listing 3-9 on page 3-38. The interface's only method is described later in this section.

Listing 3-9. IError Interface Definition

```
namespace VCSE {

  enum ErrorLevel {
     ErrorSyslog   = 1,
     ErrorDebug    = 2,
     ErrorWarning  = 4,
     ErrorFatal    = 8
  };

  interface IError extends IBase {
     MRESULT Error([in] IBase RepInterface,
```

```
                    [in]  ErrorLevel    Level,
                    [in]  int           Code,
                    [in]  unsigned int Length,
                    [in, size_is(Length)] unsigned char ErrInfo[]);
  };
};
```

Method Descriptions

The supported method for the `IError` interface is `Error`.

Error

If a non-`NULL IError` interface is supplied to an instance of a component that accepts one, then it must use the `Error` method of the interface to report any detected errors or other events falling into the categories requested by the user of the instance. If there is no mechanism for the user to specify the categories of interest, then the component must report at least fatal errors. The parameters to `Error` are described in Table 3-8 on page 3-40.

ITestReport Interface

The `VCSE::ITestReport` interface is provided as a standard way to communicate diagnostic and documentary messages between the test shell component and the user. Exactly how this is achieved depends entirely on the component implementing the `ITestReport` interface.

For example, generated test shell components make use of this interface in the `__VCSE_SHELL_REPORT` macro defined in the `GLOBALDEF` user-modifiable block of the test shell's method file. For more information, see "Generated Test Shells" on page 4-77.

Table 3-8. Error Method Parameters

Parameter	Type	Description
`RepInterface`	`IBase`	Provides the `IBase` interface of the component instance reporting the error. May be `NULL` if no interface is available, or if the calling code is not a component instance.
`Level`	`ErrorLevel`	Specifies the seriousness of the error being reported. The available levels are: • `ErrorSyslog` Miscellaneous messages the component wishes to record. • `ErrorDebug` Debug information helping to diagnose problems. • `ErrorWarning` Non-fatal error condition that may impact the performance of the component. • `ErrorFatal` A fatal error implying that the component instance may be compromised.
`Code`	`int`	Specifies the error encountered with an integer value. Error codes are specific to each component.
`Length`	`unsigned int`	Specifies the length of data provided with the `ErrInfo` parameter. A value of `0` implies that no additional information is available.
`ErrInfo`	`unsigned char[]`	Supplies additional information associated with the error being reported. One common use of this parameter is to supply a string describing the error.
Returned value	`MRESULT`	Returns `MR_NOT_COMPLETED` if `Error` does not successfully process the request, otherwise returns `MR_OK`.

ITestReport Interface Definition

There are four methods defined for this interface, three for collecting the messages and the other to return the messages in bulk to the user.

The `ITestReport` interface definition is shown in Listing 3-10. The interface's methods are described later in this section.

Listing 3-10. ITestReport Interface Definition

```
namespace VCSE {
  interface ITestReport extends IBase {
     MRESULT AddString( [in, string] char msg[]);
     MRESULT AddStringWithNumber(
              [in,string] char fmt[],
              [in] int Value
     };
     MRESULT AddStringWithNumbers(
            [in,string] char fmt[],
            [in] unsigned short numVals,
            [in,size_is(numVals)] int Values[]
     };
     MRESULT DumpAllMessages();
  };
};
```

Method Descriptions

Supported methods for the `ITestReport` interface include:

- `AddString`
- `AddStringWithNumber`
- `AddStringWithNumbers`
- `DumpAllMessages`

AddString

A simple `char` array is passed to the diagnostics collection component via the `AddString` method. If the string is to contain explanatory text as well as the numerical values of code variables then it will require formatting, for example by the library method, `VCSE_sprintf`, prior to invoking `AddString`:

Listing 3-11. Example C++ Use of AddString Method

```
char *__VCSE_line = VCSE_sprintf(
      "%s : Input Data array size (%d) less than required
(%d)\n",
      "EXAMPLES_CULawc::Compress", N, 256);
p_VCSE_ItestReport->AddString(__VCSE_line);
```

AddStringWithNumber

The `AddStringWithNumber` method provides the means to pass a string and a single integer value to the interface. For example the string could contain a format string and the value could be a result value corresponding to the single %d format-specifier contained in the format string:

Listing 3-12. Example C++ Use of AddStringWithNumber Method

```
p_VCSE_ItestReport->AddStringWithNumber(
            "Test completed in %d cycles\n",
            numCycles);
```

AddStringWithNumbers

For more comprehensive reporting of a set of values, the `AddStringWithNumbers` method provides the means to pass an array of values to the `Report` interface. For example, to record the cycles taken and stack usage the following code could be used:

Listing 3-13. Example C++ Use of AddStringWithNumbers Method

```
integer stats[2];
stats[0] = __builtin_sysreg_read(reg_CYCLES)
            - __vcse_timing_start
            - __this->m_timing_calibration;
__VCSE_STACKUSE(__vcse_stack_end, &stats[1]);
p_VCSE_ItestReport->AddStringWithNumbers(
          "EXAMPLES_CULawc::Compress",
          2,
          stats);
```

DumpAllMessages

To transmit the collection of messages to the user, the `DumpAllMessages` method is provided. This method takes no parameters.

Listing 3-14. Example C++ Use of DumpAllMessages Method

```
p_VCSE_ITestReport->DumpAllMessages();
```

IName Interface

The `VCSE::IName` interface is a standard interface that any component may choose to implement. It provides a means for code holding only an interface pointer to obtain a meaningful name for the component that provides the interface. It also provides the means by which a client can set a meaningful name, so the client can, for instance, distinguish between multiple instances of a component.

An example of code holding an interface that may wish to identify its defining component is an implementation of the `Error` method of the `VCSE::IError` standard interface.

IName Interface Definition

The three methods defined in this interface allow a client to associate a name (or other descriptive text) with a component instance and to retrieve the current size and contents of the name.

The `IName` interface definition is shown in Listing 3-15. The interface's methods are described later in this section.

Listing 3-15. IName Interface Definition

```
namespace VCSE {

   interface IName extends IBase {
      MRESULT SetName( [in, string] char Name[]);
      MRESULT GetName( [in] int Length,
                       [out, string, size_is(Length)] char Name[]);
      MRESULT GetLength([out] int Length);
   };
};
```

Method Descriptions

Supported methods for the `IName` interface include:

- `SetName`
- `GetName`
- `GetLength`

SetName

A component implementing the `IName` interface is required to have a suitable default name associated with it. This default name, set when the factory method is executed, is defined by the component designer and does not have to be distinct for each component instance. The name might be generic, such as the fully qualified component name. If the component implements `IName` by aggregation from another component, then it must call `SetName` on the aggregated component during its own creation in order to set a suitable default name.

A client can also use `SetName` to set the name or other descriptive text to be associated with the component that implements the `IName` interface. For instance, it may do this in order to obtain more meaningful tracing output or to distinguish between multiple instances of the same component.

The name is supplied as a VIDL string whose null-terminated contents `SetName` uses to replace the currently stored name. The `SetName` method must return an error result if it is unable to store the complete name, but it is undefined whether it stores a part of the new name, retains the old name, or follows some other course of action.

The result values that `SetName` returns are:

- `MR_OK` when the complete name is stored successfully.
- `MR_NO_MEMORY` when sufficient memory is not obtained to store the complete name.
- `MR_NOT_COMPLETED` when the complete name is not stored for any other reason, including a fixed-size buffer being too small.

GetName

The `GetName` method copies the current name and terminating null character into the sized string provided by the client. If the string is not long enough, then `GetName` must return an error result and place a null-terminated character sequence in the string, assuming it is not of zero length. The character sequence may be empty but otherwise is undefined.

The result values that `GetName` returns are:

- `MR_NO_ERROR` when the complete name is returned successfully.
- `MR_NOT_COMPLETED` when it fails for any other reason, including the supplied array being too short.

GetLength

The `GetLength` method supplies the length, including the terminating null character, of the current name. The method allows its clients to ensure that a sufficiently large string is supplied to a subsequent `GetName` call. It must return a result of `MR_OK`.

4 VIDL LANGUAGE REFERENCE

The VCSE Interface Definition Language (VIDL) is a descriptive notation for specifying VCSE interfaces and components. The VIDL compiler processes and transforms VIDL specifications into source code fragments. The source code provides skeleton component implementations and interface representations in an appropriate programming language. In practice, a single VIDL specification can be converted by the VIDL compiler into an equivalent representation in C, C++, or a platform assembly language.

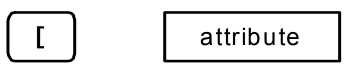
This chapter provides a reference description of the syntax and semantics of VIDL. Syntax is described informally using syntax diagrams rather than grammar rules, and the description of semantics is deliberately as brief and simple as possible. The text includes a number of examples whose purpose is illustrative rather than tutorial. The interpretation of the syntax diagrams is described in "Understanding Syntax Diagrams" on page 4-2. Material relating the principles and practice of VCSE programming is found elsewhere in this manual.

The information about the VIDL syntax and semantics is organized as follows.

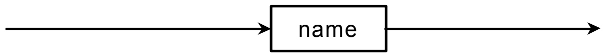
Understanding Syntax Diagrams

In this chapter, the syntax of VIDL statements and elements is illustrated by diagrams, which use notation often referred to as "railroad tracks". The syntax diagrams should be read from left to right and from top to bottom, following the path of the line and the arrows.

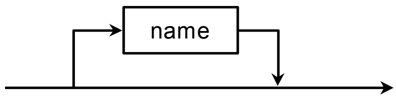
Literal character sequences are shown within rounded rectangles, whereas un-rounded rectangles are used to identify named syntax elements, as shown below:



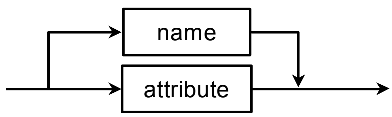
Any required items appear on their own, on the main path:



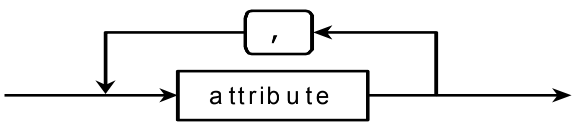
Optional items are shown above or below the main path:



If you can choose from two or more items, they appear vertically, in a stack. If you *must* choose one of the items, one item of the stack appears on the main path:



An arrow returning to the left above or below the main line indicates an item that can be repeated, along with the separator character if that character is necessary:



Lexical Elements

VIDL specifications are constructed from character sequences that identify white space, comments, preprocessing tokens, and language tokens. The VIDL compiler does not see the preprocessing tokens as the C preprocessor removes them prior to compilation.

Character Sequences

A VIDL specification is contained in a text file prepared with a conventional text editor. The file may contain any of the following characters.

- The uppercase and lowercase letters:

  `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

  `a b c d e f g h i j k l m n o p q r s t u v w x y z`

- The decimal digits: `0 1 2 3 4 5 6 7 8 9`
- The special characters:

  `_ : ; , . ' " \ { } [ ] ( ) = | ^ & + - * / ~ % > < #`

- The formatting characters: space, newline, and tab

These characters may be grouped into larger sequences called white space, comments, preprocessing tokens, and language tokens. A token is always formed from the longest possible sequence of characters. For example, the VIDL compiler interprets the character sequence `<<` as a single token denoting a left-shift operator rather than two tokens denoting two less-than operators.

White Space

White space consists of any sequence of formatting characters. White space occurring outside a character literal or string literal may be used to control the layout of a VIDL text file but adds no meaning to the specification it contains. For example, the newline character may be used to split the text within a VIDL file into physical lines. There is no limit either to the length of a line or to the number of lines in the file. The VIDL compiler skips all white space characters when checking the syntax of a VIDL specification.

Comments

Comments may be inserted at any point in a VIDL specification and provide a means to add supplementary documentation. VIDL allows three notations for normal comments, post-comments, and auto-doc comments, as shown in Figure 4-1.

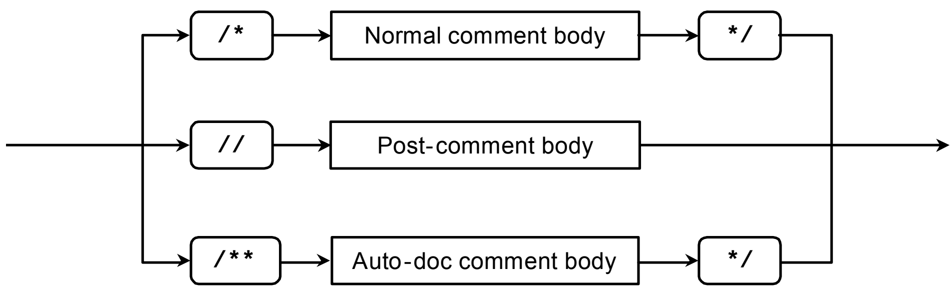


Figure 4-1. `Comment` Syntax Diagram

The body of a normal comment and of an auto-doc comment contains all the characters between the introductory sequence `/*` or `/**` and the termination sequence `*/`. The body of a post-comment contains all the characters up to but not including the newline character that terminates the line. The VIDL compiler discards normal and post-comments, but retains auto-doc comments for further analysis. Auto-doc comments are distinguished by the starting sequence `/**` and are used to provide formatted external documentation. For more information, see "Auto-doc Comments" on page 4-72.

Preprocessing

Every VIDL specification is analyzed by the C/C++ language preprocessor prior to syntax analysis. The preprocessor performs source file substitution, macro expansion, and conditional removal of source text using preprocessing directives that begin with the character `#`. For a description of the C/C++ preprocessor, see the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual* for the appropriate processor platform.

The C++ preprocessor is invoked for all DSP platforms except ADSP-218x DSPs.

The `#include` directive is used to control the inclusion of additional VIDL source text from a secondary input file that is named in the directive. Two available forms of `#include` are shown in Figure 4-2.

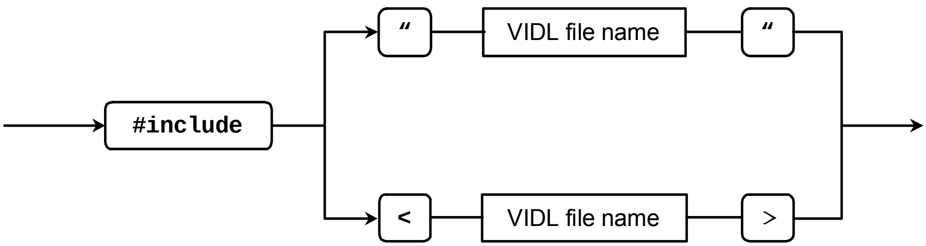


Figure 4-2. `#include` Syntax Diagram

The file, identified by the file name, is located by searching a list of directories. When the name is delimited by quote characters, the search begins in the directory containing the primary input file, then proceeds with the list of directories specified by the `-I` command line switch. When the name is delimited by angle bracket characters, the search proceeds directly with the directories specified by `-I`. If the file is not located within any directory on the search list, the search may be continued in one or more platform-dependent system directories.

VIDL Language Tokens

The characters in a preprocessed VIDL text file are grouped into sequences called language tokens. Language tokens identify the names, keywords, operators, punctuation, numerical and string literals that form the elements of a VIDL specification.

Names

A *name* is a sequence of alphanumeric characters and underscores that contains at least one alphanumeric character, as shown in Figure 4-3. Names are used to identify constants, types, attributes, methods, method parameters, interfaces, components, and namespaces. Namespace names are restricted to names with no underscores. Names are also used to identify tags within the auto-doc comments. Names may be combined with a `::` separator to form a *fully qualified name*.

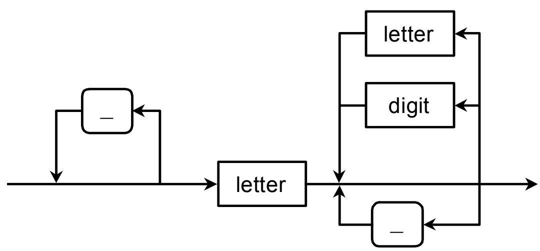


Figure 4-3. Name Syntax Diagram

Keywords

A *keyword* is a name that is reserved by the VIDL language and may not be used as an identifier. The set of keywords is as follows.

Table 4-1. Reserved Keywords

aggregatable	enum	MRESULT	size_is
aggregates	extends	namespace	size_t
algorithm_model	extern	needs	sizeof
alias	first_is	no_algorithm_model	stack_usage
aliasing_check	float	no_aliasing_check	states_used
align	fract	no_array_check	static
array_check	from	no_stack_usage	string
auto	get	no_timing	struct
category	iid	out	struct_pack
char	iid_is	out_assert	struct_pad
clear_state	implements	place	supplies
common	in	pm	testing
company	in_assert	register	timing
complex_double	info	remotable	title
complex_float	init_state	requires	typedef
complex_fract	int	requires_state	typename
complex_long_double	interface	reset_to_state	union
complex_long_fract	is	reuse	unique
component	language	set	unsigned
const	last_i	sets_state	use
distinct	local	signed	version
double	mem_shell	singleton	void
dm	length_is	shared	
document	long	short	

The set of auto-doc tags is as follows. For additional information about auto-docs, see "Auto-doc Comments" on page 4-72.

Table 4-2. Auto-doc Tags

`@author`	`@param`
`@create`	`@postcondition`
`@destroy`	`@precondition`
`@example`	`@return`
`@keyword`	`@sizeof`

Punctuation

The following tokens are used for punctuation.

```
: :: ; , . { } [ ] ( )
```

Operators

The following tokens are used as arithmetic operators.

```
+ - * / % ^ & | ~ << >> == != < <= > >=
```

Numeric Literals

Numbers and strings are represented by integer, real, and string *literals*. They may be combined with appropriate arithmetic operators to form expressions.

Integer Literals

Integer literals are used to denote integer values using sequences of octal, decimal, and hexadecimal digits (see Figure 4-4).

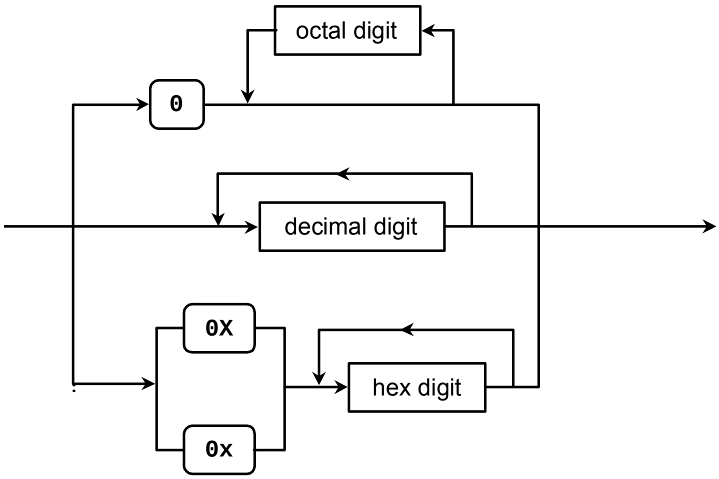


Figure 4-4. Integer Literal Syntax Diagram

- The octal digits: `0 1 2 3 4 5 6 7`
- The decimal digits: `0 1 2 3 4 5 6 7 8 9`
- The hexadecimal digits:

  `0 1 2 3 4 5 6 7 8 9 a A b B c C d D e E f F`

An integer literal defines a value with the VIDL type `int`. Decimal values are distinguished by their first digit, which must not be zero and may be prefixed with the — (minus) unary operator to form negative values. Example of each are: `02274` (octal); `1212`, `34` (decimal); `0x4BC`, `0X4BC` (hexadecimal).

Real Literals

A *real literal* defines a value with the VIDL type `double`. The literal's form is shown in Figure 4-5. Examples are: `2.340`, `2.34e+3`, `2.34E-3`.

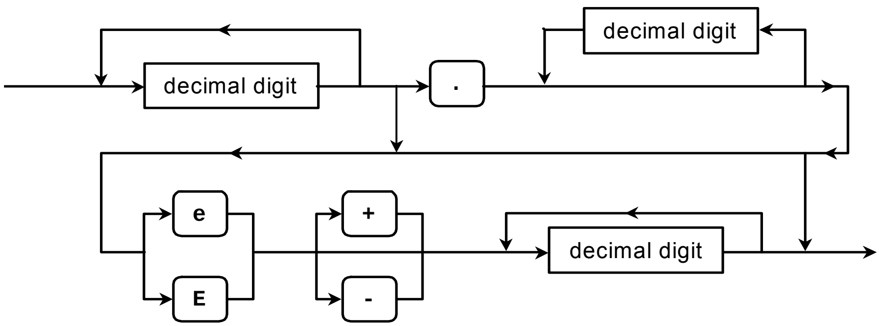


Figure 4-5. Real Literal Syntax Diagram

String Literals

A *character literal* specifies a value of the VIDL type `char`. The character denoted is either a single graphic character or one identified by an escape sequence, as shown in Figure 4-6. Examples are `'0'` and `'A'`.

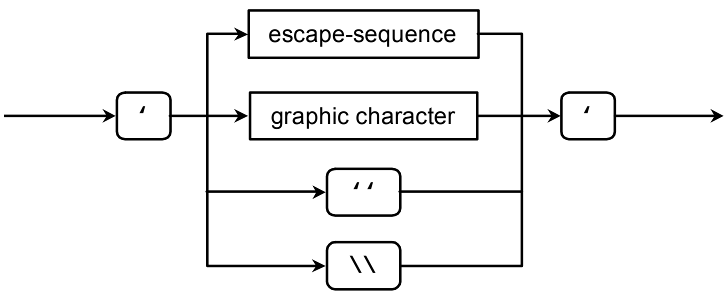


Figure 4-6. Character Literal Syntax Diagram

An escape sequence consists of octal digits, hexadecimal digits, or one of the special escape characters `n t b r f v “ \` (see Figure 4-7). The escape letters represent the non-printing formatting characters for newline, horizontal tab, backspace, carriage return, form feed, and vertical tab. The escape sequences `''` and `\\` respectively denote the single quote and forward slash characters. There can be at most three octal and two hexadecimal digits in any escape sequence. Examples are: `\xA`, `\t`, `\012`.

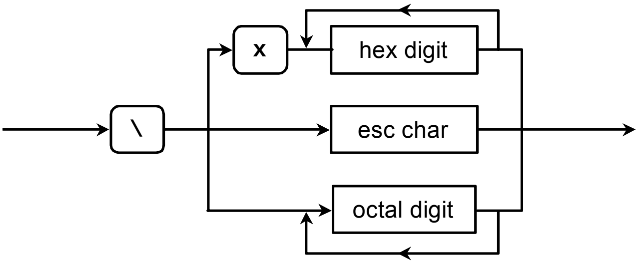


Figure 4-7. Escape Sequence Syntax Diagram

A *string literal* specifies a sequence of zero or more characters, each of which is represented by a graphic or an escape sequence. A double quote character occurring within the string is represented by the escape sequence `\”`. The VIDL compiler maps a string literal to an equivalent representation and type in the implementation language. Examples are: `“www.analog.com\n”`, `“the MRESULT value \”MR_OK\” ...\012”`.

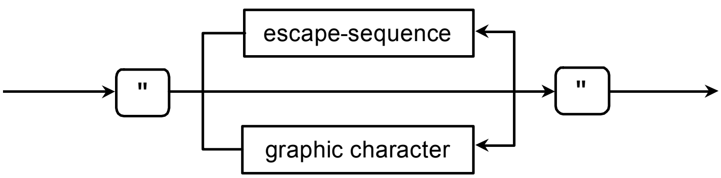


Figure 4-8. String Literal Syntax Diagram

Named Elements

A VIDL specification is composed of *named elements* that describe namespaces, components, interfaces, interface methods, method parameters, types, and constants. Namespaces, components, interfaces, methods and method parameters may be annotated with attributes to provide additional information for the VIDL compiler.

Every named element within a VIDL specification must have a single defining *definition*. The portion of the specification over which a definition applies is called its *scope*. There are three kinds of scope that may occur in a VIDL specification:

- The area of text that is not enclosed by the outermost namespace declaration forms an unnamed scope called the *global scope*. The only named element that can be declared in global scope is a namespace.
- The area of text enclosed by an interface, namespace, component or structure definition forms a named scope whose name is the namespace, interface, or structure name.
- The VIDL compiler maintains a named scope that is associated with a predefined namespace called `VCSE`.

Every use of a name must be preceded by its definition. Because of circular dependencies, it may not be possible to fully define a name prior to its use. In these cases, it is permissible to introduce the name into its scope with a *forward declaration*. A name may have more than one declaration within its scope, but there must be exactly one defining definition. A name with a declaration in an enclosing scope cannot be used and then redefined in the current scope.

When a named element is used in a VIDL specification, it may be referenced using its *qualified* or *unqualified name*. The unqualified form—shown in Figure 4-9—is merely the name introduced by the definition. The qualified form is the unqualified name prefixed with the name of each scope that contains its definition (see Figure 4-10).

Figure 4-9. Unqualified Name Syntax Diagram

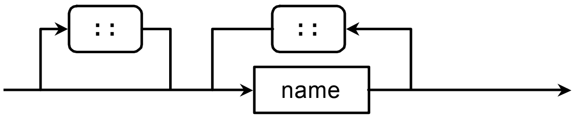


Figure 4-10. Qualified Name Syntax Diagram

A qualified name of the form `ADI::EDSP::IFilter` references an element `IFilter` that is defined in a scope named `EDSP` that is, in turn, defined within an enclosing scope called `ADI`.

In VIDL, scopes are determined by namespace and interface definitions; although in practice, only namespace scopes can be nested. By convention, the global scope is partitioned into distinct company-specific namespaces that allow every named element to be uniquely identified by its fully qualified name.

The order in which scopes are searched for the declaration of an unqualified name may be altered by the `use` *attribute*. For more information, see "use Attribute" on page 4-71.

Therefore, when an element is referenced with an unqualified name `N`, the element is identified by searching the available scopes for its declaration, using the following rules.

1. Search for the declaration of `N` in the current scope. If it is not found, proceed to rule 2.
2. Search the scopes defined by the namespaces in any `use` attribute attached to the current scope. If the declaration is not found, proceed to rule 3.
3. Reapply rules 1 and 2 to all scopes that enclose the current scope. If the declaration is not found, proceed to rule 4.
4. Search the predefined `VCSE` scope.

If the declaration of `N` is not found by rule 4, then the VIDL compiler reports an error. For example, if `N` occurs in scope `B` that is contained by scope `A` that is contained in the global scope, then the VIDL compiler looks for the declaration of `N` by searching scopes `B`, `A`, the global scope, and finally the `VCSE` scope.

When an element is referenced by a qualified name `::S::N` or `S::N`, the element is identified by searching the scopes named by the scope prefixes as follows.

5. If the prefix is `::S`, then `S` must identify a scope `S` declared within the global scope. If the prefix is `S`, then `S` must identify a scope `S` found by application of rules 1 to 4.
6. If `S` is the scope identified by the scope prefix, then the name `N` must be declared in `S`.
7. If the declaration of `N` found by rule 6 identifies another scope `N` and `N` is followed by the token sequence `::M`, then rule 6 is reapplied by substituting `N` for `S` and `M` for `N`.

For example, when the VIDL compiler encounters the qualified name `A::B::N`, rules 1 through 4 must identify the scope prefix `A::` with the scope `A`. Rule 6 must locate the declaration of `B` within scope `A`. Then by rule 7, the declaration of `B` must identify a scope `B`; and by rule 6, `N` must be declared within `B`. If the qualified name has the form `::A::B::N`, the scope prefix `A` must be declared within the global scope.

The VIDL compiler uses the case of each letter to distinguish names that are otherwise identical. Thus, `Region` and `region` are regarded as different names.

Element Attributes

Element attributes supply additional information about namespaces, components, interfaces, methods, method parameters, properties, and structure members to the VIDL compiler. They are specified by attribute lists that precede the definition of the element to which they apply. The attribute form and all of the element attribute forms are shown in Figure 4-11 and Figure 4-12 on page 4-17.

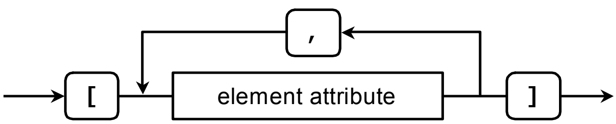


Figure 4-11. Attribute Syntax Diagram

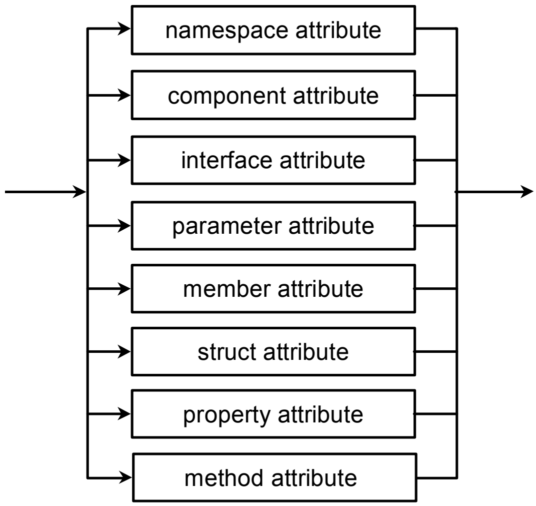


Figure 4-12. Element Attribute Syntax Diagram

The VIDL compiler verifies the attributes supplied are appropriate for the element to which they are applied. In practice, every interface or method parameter definition must be preceded by at least one interface attribute or parameter attribute.

The definitions of the attributes appropriate to each element are covered in the respective sections describing the elements.

Constant Expressions

An expression is composed of binary or unary operators and their operands (see Figure 4-13 through Figure 4-16). An expression whose operands are integer, character literals, or enumeration constants is called a *constant expression*. Only constant expressions are allowed in VIDL. The expression must evaluate to a valid value of integer type.

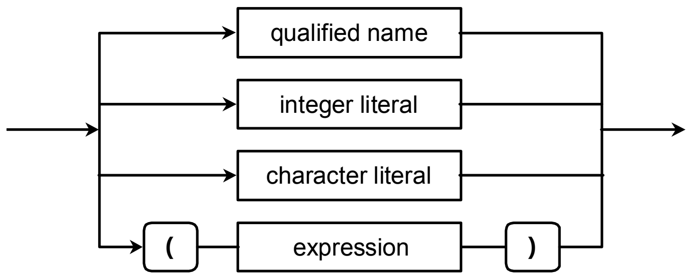


Figure 4-13. Primary Expression Syntax Diagram

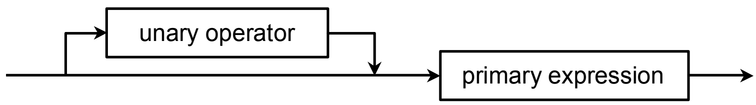


Figure 4-14. Unary Expression Syntax Diagram

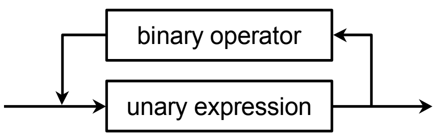


Figure 4-15. Expression Syntax Diagram

Figure 4-16. Constant Expression Syntax Diagram

Table 4-3 and Table 4-4 on page 4-19 list the unary and binary operators in order of decreasing precedence.

Table 4-3. Unary Operators Precedence Chart

Operator	Name	Precedence
+	plus	7
–	minus	7
~	bit negation	7

Table 4-4. Binary Operators Precedence Chart

Operator	Name	Precedence
*	multiplication	6
/	integer division	6
%	remainder	6
+	addition	5
–	subtraction	5
<<	left shift	4
>>	right shift	4
&	bitwise and	3
^	bitwise xor	2
|	bitwise or	1

Unary operators have the highest precedence and are evaluated before any binary operator. Binary operators range from *integer-multiplication* with the highest precedence through to *bitwise-or* with the lowest precedence. Operators are applied to operands according to the following precedence rules.

1. If `o` is any binary operator, `u` is any unary operator, and `X` and `Y` are operands, then the expression `u X o Y` is evaluated as `(u X) o (Y)`.

2. If `o1` and `o2` are binary operators and `X`, `Y` and `Z` are operands, then the expression `X o1 Y o2 Z` is evaluated as `(X o1 Y) o2 (Z)` if the precedence level of `o1` is greater than or equal to the precedence level of `o2`. If the precedence level of `o1` is less than the precedence level of `o2`, then the expression is evaluated as `(X) o1 (Y o2 Z)`.

These rules may be overridden by inserting brackets. For example, in `i*j|k`, evaluation of `|` before `*` can be forced by writing the expression as `i*(j|k)`.

Constant expressions may be used to specify the value of an enumeration constant, an array bound, or an element attribute. For more information about the operands, see “Numeric Literals” on page 4-9 and “Enum Types” on page 4-22. Array bounds are described in “Declarators” on page 4-28 and element attributes in “Element Attributes” on page 4-16. Constant expressions are evaluated by the VIDL compiler and only the resultant numeric value is recreated in the generated files.

Listing 4-1. Example Constant Expressions

```
1000
i - '0'
bits & 0xF0
n*m + 12
(u - v)*(x + y)
(m >> s)&0xF
~(0xF << s)
```

Types

VIDL provides a set of *types* for describing scalar and aggregate values. A type is either an arithmetic base type or a user defined type. Both sets of types are specified by names or constructs that are similar to those found in Analog Devices dialects of the C and C++ programming languages. The set of VIDL types is shown in Figure 4-17 on page 4-21.

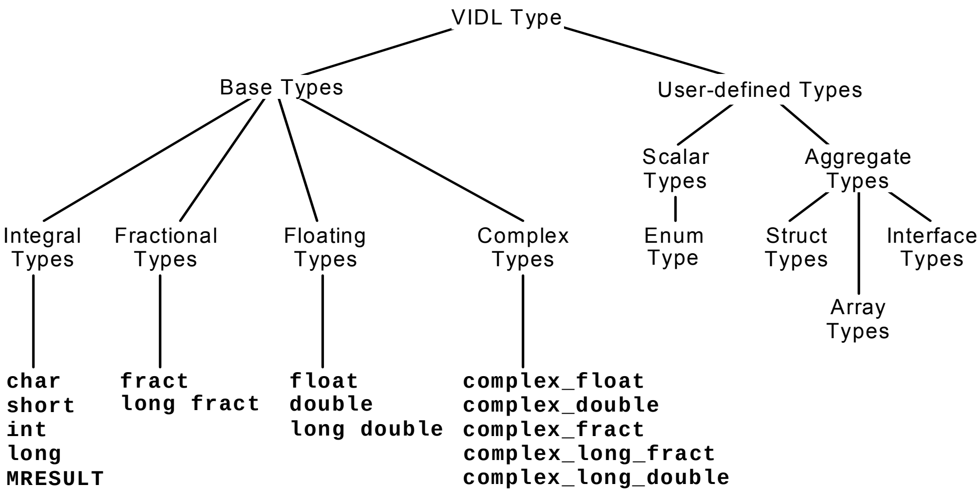


Figure 4-17. VIDL Types

The VIDL compiler maps each VIDL type into an equivalent host type in the implementation language. If there is no equivalent host type, it reports an error.

Base Types

The base types allow integer, fixed-point (fractional), floating-point, and complex arithmetic data to be specified. They are represented by a `type` keyword, which in some cases may be prefixed with a `signed` or `unsigned` qualifier (see Figure 4-18).

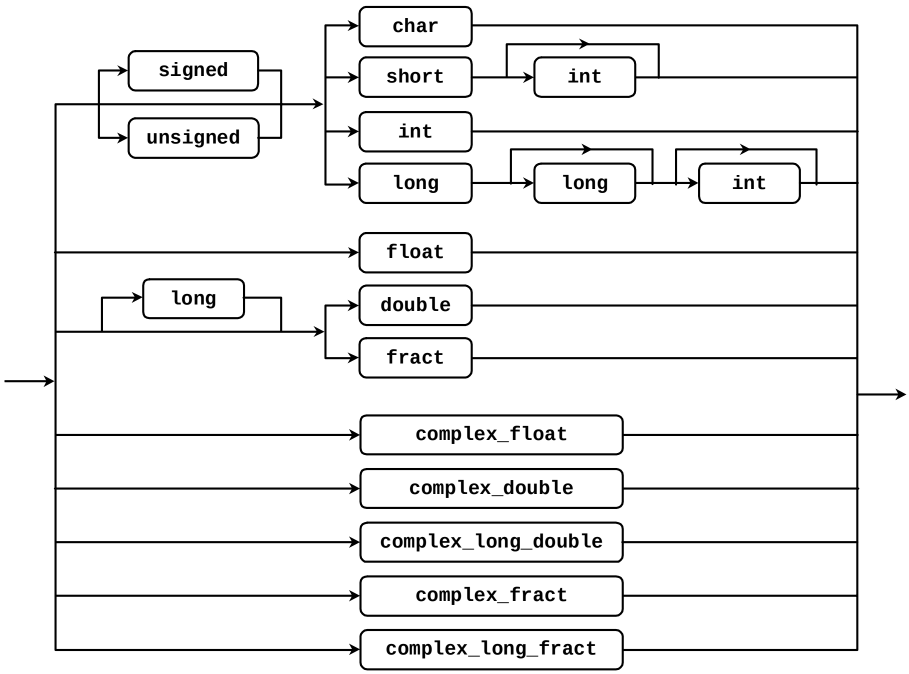


Figure 4-18. Base Type Syntax Diagram

Enum Types

An *enumeration* type specifies the values of one or more enumeration constants. The value of each constant is determined by a constant expression, or by adding one to the value of the preceding constant if no expression is supplied. The value of the first constant is either `zero` or the value of its constant expression. Figure 4-19 and Figure 4-20 on page 4-23 show the enumeration type formats.

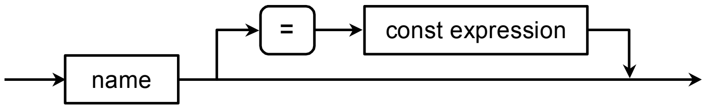


Figure 4-19. Enumerator Syntax Diagram

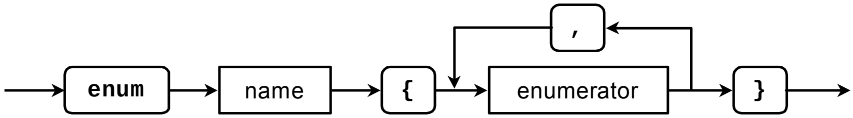


Figure 4-20. `enum` Definition Syntax Diagram

An enumerator specifies a name that denotes its value in the scope in which it is declared. The enumerator may be referenced outside its scope using its qualified name. An enumeration definition specifies a name that denotes the enumeration type and may be used within its scope as a type specifier. The enumeration may be referenced outside its scope by its qualified name. For more information, see "Named Elements" on page 4-13.

Listing 4-2. Enum Example

```
enum Colors { red = 1, green, blue }
enum MemoryType {
      MemoryPrimary     = 1,
      MemorySecondary   = 2,
      MemoryExternal    = 4,
      MemoryBank        = 8,
      MemoryAny         = ( MemoryPrimary   | MemorySecondary |
```

```
                              MemoryExternal | MemoryBank ) }
enum Boundary { top = +10, bottom = -10, left = -20, right = +20 }
```

Structure Types

A *structure* type is an aggregate containing a list of components called *members*. Each member is defined by a declarator that specifies its name and type. A structure defines a scope in which no two members may have the same name. The member declarator form, member list form, and structure definition form are shown in Figure 4-21 through Figure 4-25.

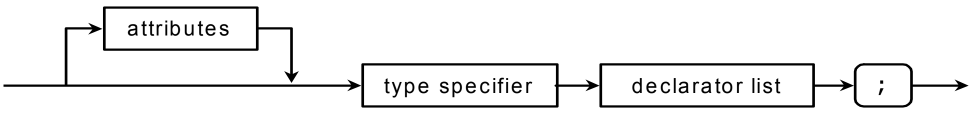


Figure 4-21. Member Declarator Syntax Diagram

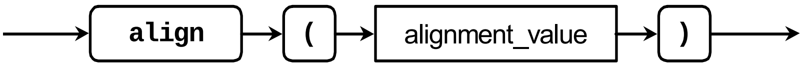


Figure 4-22. Member Attribute Syntax Diagram

In Figure 4-22, the `alignment_value` is an integer with the same constraint as the parameter used in `#pragma align`, which means the value must be zero (default alignment) or a power of two. Refer to the *VisualDSP++ 3.x C/C++ Compiler and Library Manual* for your target processor family or the online Help for more information about pragmas.

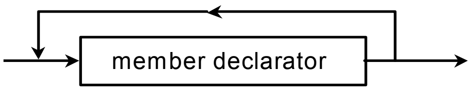


Figure 4-23. Member List Syntax Diagram

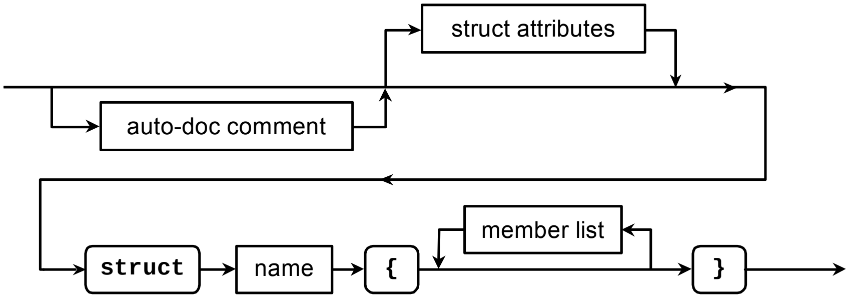


Figure 4-24. `struct` Definition Syntax Diagram

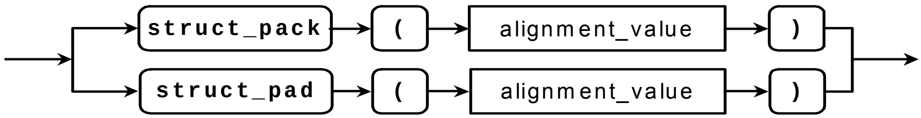


Figure 4-25. `struct` Attributes Syntax Diagram

In Figure 4-25, the `alignment_value` is an integer with the same constraint as the parameter used in `#pragma pack` and `#pragma pad`, which means the value must be zero (default alignment) or a power of two. Refer to the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual* for your target DSP family or the online Help for more information about `#pragmas`.

A structure definition specifies a name that denotes the structure type and may be used within its scope as a type specifier. The structure may be referenced outside its scope by using its qualified name. A structure name cannot be used as a type specifier within its own list of members.

Structure definitions cannot be nested. However, a member may be declared with a type specifier that references a previously defined structure. A structure may be defined with an empty list of members.

Listing 4-3. Struct Example

```
struct Point{ int x; int y; };
[struct_pad(4)] struct Box {
    Point           center;
    [align(2)] int  width, height;
};
[struct_pack(1)] struct MemType {
    int    m_type;
    int    m_life;
    char   m_bank[256];
};
```

Interface Types

An interface defines a name that denotes an *interface type*, which may be used within its scope as a type specifier. In particular, an interface may be used to specify the type of a method parameter. An interface may be referenced outside its scope using its qualified name:

```
MRESULT SetErrorReporter( [in] VCSE::IError ErrorReporter );
```

Interfaces are described in "Standard Interfaces" on page 3-1 and "Interfaces" on page 4-29.

Type Specifiers and Definitions

A type is specified in a parameter or member declaration by a *type specifier*. A type specifier is either the name of the type or a sequence of keywords that identifies a base type, as shown in Figure 4-26. The VIDL base types are described in “Base Types” on page 4-22.

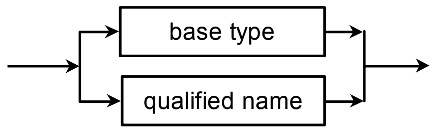


Figure 4-26. Type Specifier Syntax Diagram

A type definition supplies a name for the type, which may be used in its scope as a type specifier. The type may be referenced outside its scope by using its qualified name.

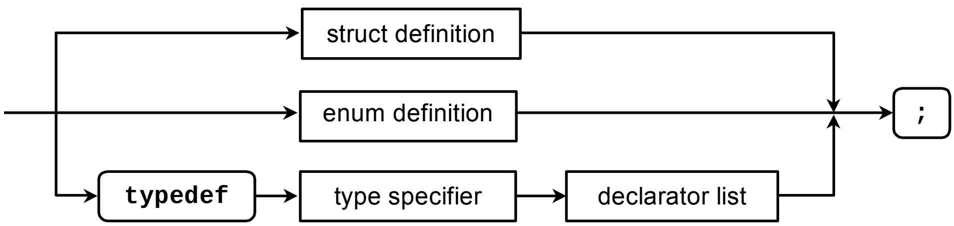


Figure 4-27. `typedef` Syntax Diagram

Listing 4-4. Typedef Example

```
typedef unsigned int u_int;
typedef ::adi::adsp::IFilter adi_ifilter;
enum primary { red, green, blue };
```

Declarators

A *declarator* specifies the name for a method parameter or a structure member. When used in a type definition, a declarator provides a name for the type referenced by the type specifier. It is an error if the name has a previous definition in the scope of the declarator. The declarator and declarator list formats are illustrated in Figure 4-28 and Figure 4-29.

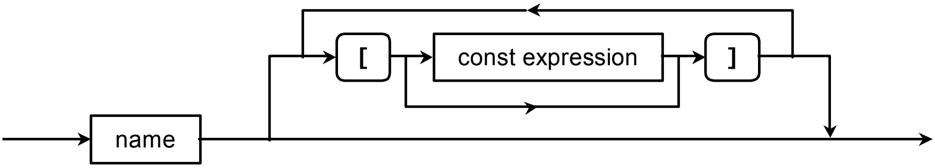


Figure 4-28. Declarator Syntax Diagram

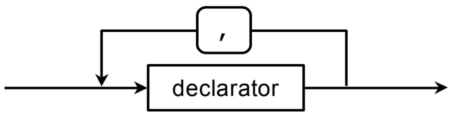


Figure 4-29. Declarator List Syntax Diagram

When the declarator name is followed by one or more pairs of brackets, the name is assigned an *array* type. The element type of the array is provided by the preceding type specifier, and the number of dimensions is specified by the number of bracket pairs.

The number of elements in an array dimension may be specified by a constant expression. If the size of every dimension is specified, the array is called a *fixed array*. If the size of any dimension remains unspecified, the array is called a *conformant array*, and the dimension is said to be *unsized*.

When a declarator is declared with a *conformant* array type, the corresponding member or parameter declarator must be preceded with a `size_is` or `string` parameter attribute that specifies the number of elements in the dimension at run-time. These attributes are defined in "size_is Attribute" on page 4-37 and "string Attribute" on page 4-39.

Example:

```
/* Declarators: */
xref[10]
cval
coord[10,20]

/* Declarator lists: */
xcord, ycord
ncoef, coef_a[10], coef_b[10]
```

Interfaces

An interface definition specifies the name, the base interface from which it is extended, and the body. The name of the interface may be used as a type specifier, described in "Type Specifiers and Definitions" on page 4-27, or as an interface name within its scope. The interface may be referenced outside its scope using its qualified name. An interface may also be *declared* and its name used as a type specifier, prior to the interface definition. However, a warning occurs if the interface is not defined in the same scope as the declaration.

Figure 4-30 through Figure 4-32 on page 4-30 provide syntax diagrams for interface declarations and interface definitions.

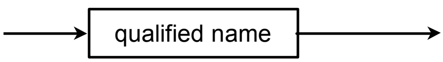


Figure 4-30. Interface Name Syntax Diagram

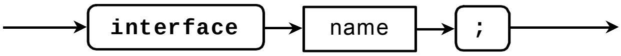


Figure 4-31. `interface` Declaration Syntax Diagram

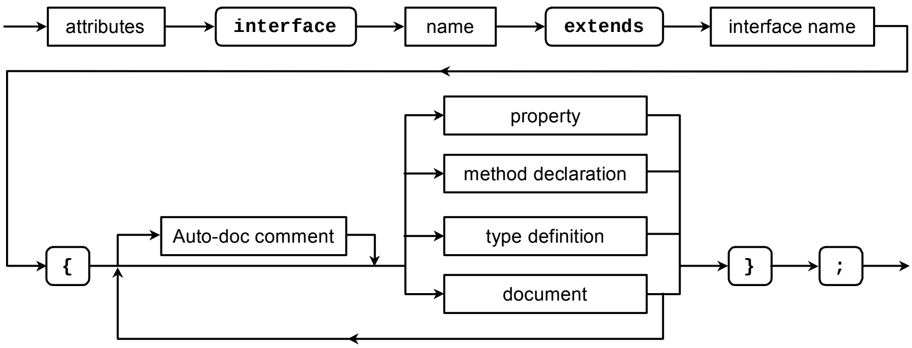


Figure 4-32. Interface Definition Syntax Diagram

An interface definition must be preceded by an attribute list that contains an `iid` attribute (see Figure 4-33). The list may also contain a use Attribute, which is described in "use Attribute" on page 4-71, a needs Attribute, which is described in "needs Attribute" on page 4-58, or a supplies Attribute, which is described in "supplies Attribute" on page 4-61.

Figure 4-33. `iid` Attribute Syntax Diagram

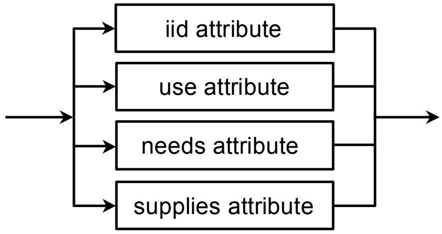


Figure 4-34. Interface Attributes Syntax Diagram

An `iid` attribute supplies an *interface identifier*, which provides a unique binary identification code for the interface. The code is a sequence of 32 hexadecimal digits generated by support utilities within the VisualDSP++ environment.

By convention, an interface name must start with the capital letter `I`. The name `IBase` is reserved for the predefined root interface `VCSE::IBase`.

The base interface specified in an interface definition must either be a previously defined interface or the root interface `VCSE::IBase`. Every interface is a direct or indirect extension of `IBase`.

The methods provided by an interface are specified by the method declarations within its body in addition to the methods provided by its base interface. The root interface `IBase` contains a single method called `GetInterface`, which is provided on all other interfaces. For example, there are interfaces `I1`, `I2`, and `I3`, where `I3` extends `I2`, which extends `I1`, which extends `IBase`. Suppose that the bodies of `I1`, `I2`, and `I3` respectively contain declarations for the methods `M1`, `M2`, and `M3`. Then the methods of `I1` are `{GetInterface, M1}`, the methods of `I2` are `{GetInterface, M1, M2}`, and the methods of `I3` are `{GetInterface, M1, M2, M3}`. If the list of method declarations in an interface body is empty, then the interface provides only the methods in its base interface.

The body of an interface defines a scope in which its methods are declared.

Listing 4-5. Interface Identifier Example

```
namespace Example {

    enum tagRefNotes { A, B, C, D, E, F, G };
    [iid("51c45584-0a17d611-a5580010-4b7cac83")
      use(::ADI::Dolby)]
    interface IInstrument extends IBase {
      MRESULT Select( [in, string] char tune[256] );
      MRESULT Plug( [in] IChannel chOut );
      MRESULT Play( [in] long ticks );
    };
    [iid("d15d56b8-0a17d611-a5580010-4b7cac83")]
    interface ITuner extends IInstrument {
      MRESULT GetRefNote( [in, string] char name[],
                          [out] RefNotes note );
    };
    [iid("108f48d3-0a17d611-a5580010-4b7cac83")]
    interface ITunable extends IInstrument {
      MRESULT Retune( void );
    };
}
```

Methods

A method declaration—shown in Figure 4-35—specifies the name of the method, the return type, and the type of each method parameter. An error occurs if the name has already been assigned to another method in the same interface or in a direct or indirect base interface.

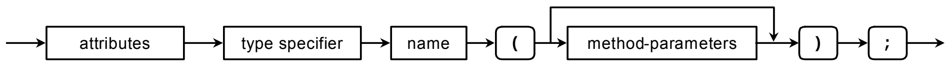


Figure 4-35. Method Declaration Syntax Diagram

The type specifier for the result type must be the predefined type `VCSE::MRESULT`. The implementation of the method provided by a component is expected to return a value of this type.

Method Attributes

The only attribute that can be supplied for a method is the `local` attribute, described below.

local attribute

Figure 4-36. `local` Attributes Syntax Diagram

The `local` attribute specifies that the associated method cannot be invoked remotely but it must *always* be executed on the local processor since at least one of the passed parameters cannot be serialized for transmission to or from another processor.

Method Parameters

Method parameters are specified by a list of parameter declarators, as shown in Figure 4-37 and Figure 4-38 on page 4-34. A method with no parameters is indicated by omitting the parameter list or supplying the keyword `void`. A parameter declarator must include one or more parameter attributes (see Figure 4-39 on page 4-35). The type of the parameter is supplied by the type specifier. If the `const`, `volatile`, or memory type (`pm` or `dm`) qualifiers are supplied, they are included in the C or C++ representation of the method declaration generated by the VIDL compiler.

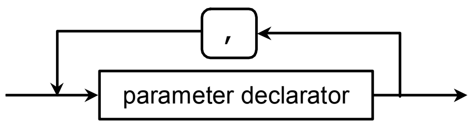


Figure 4-37. Parameter List Syntax Diagram

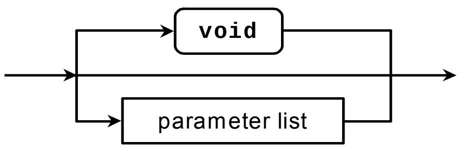


Figure 4-38. Method Parameters Syntax Diagram

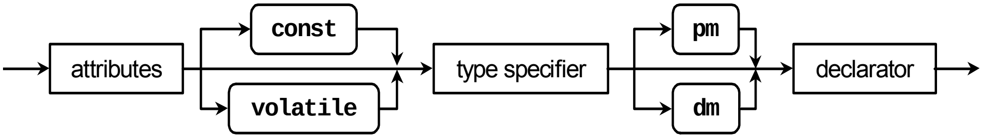


Figure 4-39. Parameter Declarator Syntax Diagram

Parameter Attributes

A parameter must be preceded by a list of parameter attributes. Figure 4-40 lists valid parameter attributes; a syntax diagram for each attribute appears in Figure 4-41 through Figure 4-48. For a description of each attribute, refer to the appropriate sections.

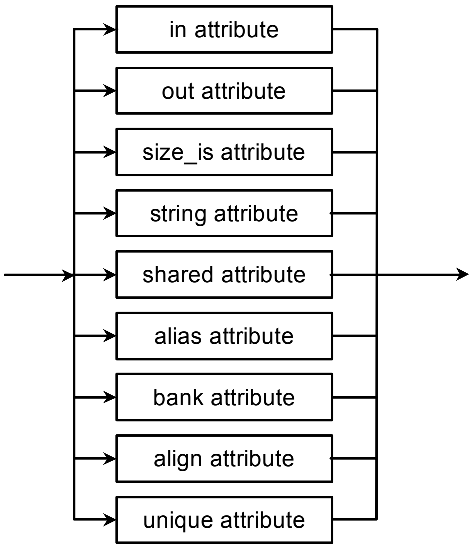


Figure 4-40. Parameter Attribute Syntax Diagram

A parameter's list of attributes must contain at least one of the direction attributes (`[in]` and `[out]`) to indicate how the parameter's value is transmitted between the method and its calling environment. Both attributes `[in]` and `[out]` can be specified in a parameter's list of attributes. The VIDL compiler uses the direction attributes to construct appropriate parameter declarations in C or C++. The other attributes are optional, and their use depends, in part, on the type of the method parameter.

in Attribute

The `in` attribute specifies an *input parameter* value that is transmitted from the calling environment to the method when the method is called.

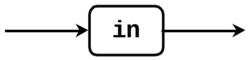


Figure 4-41. `in` Attribute Syntax Diagram

If the parameter type is a base type or an enumerated type, the parameter is passed by value. The `const` qualifier may also be used to indicate that the method should not modify the value. If the parameter type is an array, string, or structure type, the parameter is passed by reference. The VIDL compiler adds the `const` qualifier to ensure the parameter value, which is visible in the calling environment, cannot be changed by the method.

out Attribute

The `out` attribute specifies an *output parameter* value that is transmitted from the method to the calling environment when the method returns. The VIDL compiler arranges for the parameter to be passed by reference to make the final value available in the calling environment. An output parameter should not be prefixed with a `const` qualifier: an error occurs if a parameter qualified with the `[out]` attribute is also prefixed with a `const` qualifier.

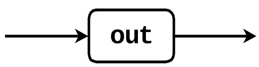


Figure 4-42. `out` Attribute Syntax Diagram

If the attribute list contains both in and out attributes, then the parameter is both an input and output parameter. The parameter value is transmitted from the calling environment to the method when the method is called, and then transmitted back from the method to the calling environment when the method returns. The VIDL compiler arranges for an input-output parameter to be passed by reference.

Any access to the input value of an input-output parameter is performed indirectly because the parameter is passed by reference. If the parameter has a scalar type, then it may be more efficient to supply an input parameter, which can be accessed directly, and a separate output parameter to return the value.

size_is Attribute

The size_is attribute specifies the number of elements in each unsized dimension of a conformant array. The number of expressions supplied in the attribute must match the number of unsized dimensions in the array, and each expression must have an integer type. If the attribute occurs within a parameter declarator, then the operands of the expression may include any of the preceding parameters in the method parameter list.

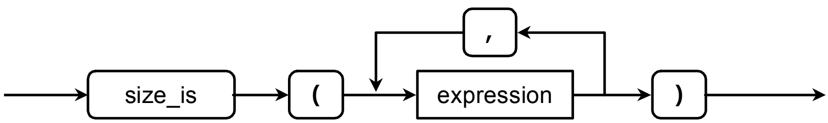


Figure 4-43. size_is Attribute Syntax Diagram

If the attribute occurs within the last member declarator of a structure type, then the expression may include any of the preceding members in the structure. In each case, the run-time value of the expression determines the number of elements in the corresponding unsized dimension.

The current VIDL compilers for ADSP-21xx and Blackfin processors support the `size_is` attribute in parameter declarator only. Support for member declarator will be implemented in future releases.

Example 1:

```
MRESULT M([in] int n, [in] int m, [in, size_is(n, m)] int x[][])
```

When method `M` is called with first and second parameters `10` and `100`, the parameter `x` may be accessed as if it had been declared as `x[10][100]`.

Example 2:

```
MRESULT N([in] int n, [in] int m, [out, size_is(n)] int y[])
```

When method `N` is called with first and second parameters `10` and `100`, the parameter `y` may be accessed as if it had been declared as `int y[10]`.

The information supplied by a `size_is` attribute is only used when the array parameter must be physically copied between memory or address spaces. When a method and its calling environment use the same memory, the run-time overhead is restricted to passing the extra parameters, giving the size of each array dimension. In example 1, if `M` was only called with actual parameter `d[10][100]`, then `M` could be declared as:

```
MRESULT M([in] int x[10][100])
```

and there would be no need to pass the array dimensions as parameters `n` and `m`.

A `size_is` attribute is still required when a method returns an array as an output parameter. In example 2, `n` must supply the size of the actual array to store the values of the formal parameter `y`. If method `N` finds that the array is not large enough, then it has the option of simply discarding the excess values or returning an error code as the case may be.

The parameter supplied as the argument to `size_is` can never be qualified with the direction attribute `[out]`, even when an array is returned as an output parameter. So in example 2, the VIDL compiler reports an error if `n` is previously declared with the attributes `[out]` or `[in,out]`.

string Attribute

The `string` attribute indicates that a method parameter or structure member, which is a character array, is to be treated as a null-terminated string.

Figure 4-44. `string` Attribute Syntax Diagram

The current VIDL compilers for ADSP-21xx and Blackfin processors support the `string` attribute in a parameter declarator only. Support for the attribute in a member declarator will be implemented in future releases.

When the array must be copied between memory or address spaces, all characters up to and including the null are copied.

Example 1:

```
MRESULT M( [in, string] char x[] )
```

All characters including the terminating null character are supplied to the parameter `x`. A parameter declared in this way is called a *conformant string*.

Example 2:

```
MRESULT N( [in] int n, [in, size_is(n), string] char y[] )
```

The number of characters (excluding the terminating null character) in the string `y` transmitted to callee is the minimum of the value of `(n-1)` and the length of the argument string computed by `strlen`. The transmitted string is always terminated with a null character. The total number of characters (including the terminating null character) written to `y` must not exceed the value of `n`.

Example 3:

```
MRESULT O( [in] int n, [out, size_is(n), string] char z[] )
```

The number of characters (excluding the terminating null character) in the string `z` returned to the caller is the minimum of the value of `(n-1)` and the length of the argument string computed by `strlen`. The returned string is always terminated with a null character.

Examples 2 and 3 imply a conformant string parameter is always null-terminated.

A conformant string parameter declared with an `out` attribute must always include a `size_is` attribute.

shared Attribute

The `shared` attribute indicates an array or structure passed as a method parameter is located in a memory region accessible to both the method and its calling environment.

Figure 4-45. `shared` Attribute Syntax Diagram

When the method and its caller run on different processors, the operations that copy the parameter from one processor memory to the other can be avoided. When the method and its calling environment are located on the same processor, or the parameter has a simple arithmetic base type, the `shared` attribute has no effect.

Example:

```
MRESULT M( [in] int n, [in, size_is(n), shared] int x[] )
```

Within `M`, any access to `x` is an access to the memory region occupied by the actual parameter.

alias Attribute

The `alias` attribute indicates an array or structure passed as an input parameter is to be treated as an alias of another input parameter with the same type, size, and shape.

Figure 4-46. `alias` Attribute Syntax Diagram

Example 1:

```
MRESULT M( [in] int x[64], [in, alias] int y[64] )
```

When the method `M` is called with `M(a, b)`, where `a` and `b` are different arrays, a copy of each array is made and the `alias` directive has no effect. When `M` is called with `M(a, a)`, the `alias` directive causes a single copy of `a` to `x` and ensures that all accesses to `y` are accesses to `x`. When the method and its calling environment are located on the same processor, the attribute has no effect.

Example 2:

```
MRESULT N( [in] int x[64], [in, out, alias] int y[64] )
```

When the method `N` is called with `N(a, a)`, a single copy of `a` is made to the parameter `x`, and all accesses to `y` become accesses to `x`. Moreover, when any values of `x` are modified within `N`, these modified values are returned as elements of the `out` parameter `y`.

bank Attribute

The `bank` attribute allows a method parameter to be associated with a named memory bank.

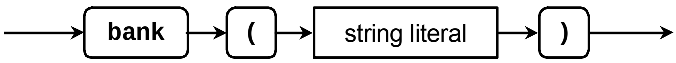


Figure 4-47. `bank` Attribute Syntax Diagram

When two parameters are associated with different banks, their elements may be accessed without possibility of memory conflicts.

```
MRESULT M( [in, bank("B1")] int x[64], [in, bank("B2")] int y[64] )
```

The parameters `x` and `y` of the method `M` are associated with different memory banks called "`B1`" and "`B2`", and the C or C++ compiler will assume no conflicts occur when their elements are accessed. It is the calling environment's responsibility to ensure this is, in fact, the case for the actual arrays supplied to the parameters `x` and `y`.

The `bank` attribute is not supported on ADSP-21xx DSPs.

align Attribute

The `align` attribute allows the actual alignment for an array to be specified in architectural addressing units.

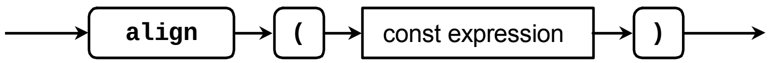


Figure 4-48. `align` Attribute Syntax Diagram

On many processors, an array is word or double word aligned even when the natural alignment associated with the element type of an array is smaller. Use of the `align` attribute allows the true array alignment to be communicated to the C or C++ compiler. This information is often critical in enabling vector loop optimizations. By default, parameters are assumed to have natural alignment unless qualified by the `align` attribute.

The value of the constant expression must be zero (default alignment) or a power of two. A value of zero means the alignment of the corresponding argument is unknown.

```
MRESULT M( [in, align(4)] short x[200] )
```

In the example (which is for a byte addressable architecture, such as the ADSP-BF53x processor), the `align(4)` attribute indicates the array `x` is word aligned, although the `short` data type is half-word aligned.

While the `align` attribute is supported on the ADSP-21xx family of processors, it only has relevance as a means of documenting that an `in` array needs to be declared as aligned in the called program unit for optimal or correct performance within the method.

document Statement

The `document` statement can be used within an interface definition or a component definition block to allow additional documentation be supplied for a method. In the case of an interface the method is expected to be a method of an interface that is being extended. In the case of a component the method can be supplied by any of the interfaces that the component implements. The `document` statement must be preceded by an auto-doc comment, which provides the additional information related to the method.

The syntax for `document` statement is

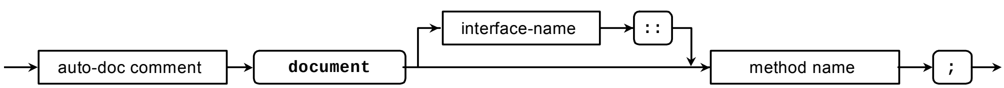


Figure 4-49. `document` Statement Syntax Diagram

In the example below the method Reset is described in the base interface IFace. The Reset method is then also supported by the extended interface IExtFace and the document statement is used to allow the additional information related to the impact of invoking Reset to be defined.

```
interface IFace extends IBase
{
    /**
    * Reset is used to bring the interface to a known state.
    **/
    MRESULT Reset();
};
interface IExtFace extends IFace
{
    /**
    * When reset is invoked all of the extended attributes are
    * automatically set to their default values.
    /**
    document Reset;
};
```

Properties

A property defines a protected field within an interface and the VIDL compiler generates accessor methods depending on the attributes of the property. Accessor methods that are generated give either read or write privileges to the calling environment.

As shown in Figure 4-50, a property declaration specifies the attributes of the properties, the type of the properties, and a list of property names. A property declaration must be declared inside of an interface definition. An error occurs if a property name has already been assigned to another property in the same interface or in a direct or indirect base interface.

Figure 4-50. Property Syntax Diagram

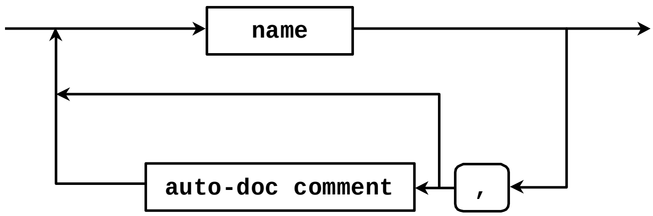


Figure 4-51. Property Name List Syntax Diagram

The type specifier for properties can be any scalar or single-dimensioned array type accepted by the `idl` compiler, excluding multidimensional arrays. Single dimensional arrays are allowed and results in special processing.

Property Attributes

A property list must be preceded by a list of property attributes. Figure 4-50 lists valid property attributes; a syntax diagram for each attribute appears in Figure 4-53 through Figure 4-55.

Valid property values include:

- `get Attribute`, described on page 4-47
- `set Attribute`, described on page 4-48
- `align Attribute`, described on page 4-47

The attributes supplied for a property must contain at least one of the accessibility attributes ([`get`] and [`set`]) to indicate which accessor methods will be defined by the interface. Both attributes, [`get`] and [`set`], can be specified in a property list's attributes list. The VIDL compiler uses the accessibility attributes to define the inlined accessor methods used to access the property.

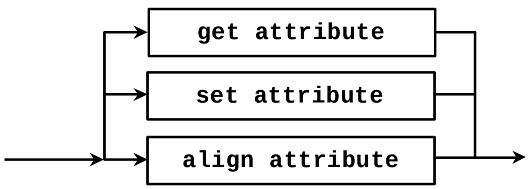


Figure 4-52. Property Attribute Syntax Diagram

get Attribute

The `get` attribute specifies that *read accesses* are allowed from the calling environment to the property.

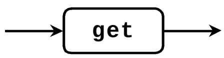


Figure 4-53. `get` Attribute Syntax Diagram

The VIDL compiler generates a method to allow the calling environment to read the value of the property. The name of the VIDL-generated method is the name of the property prefixed with `get`. For non-array type properties this method returns the value of the property's type and does not accept parameters. If the property is an array type, the return value is the type of elements of the array with one parameter that indexes into the array of values for the property.

Example 1:

```
[get] int BitRate, StatusFields[2];
```

Since an interface defines the two properties, `BitRate` and `StatusFields`, the VIDL compiler generates two accessor methods of prototypes:

```
inlined int getBitRate();
inlined int getStatusFields (int _index);
```

set Attribute

The `set` attribute specifies that *write accesses* are allowed from the calling environment to the property.

Figure 4-54. `set` Attribute Syntax Diagram

The VIDL compiler generates a method to allow the calling environment to write a value to the interface property. The name of the VIDL-generated method is the name of the property prefixed with `set`. The return value of this method is `void`. For non-array type properties this method takes in one parameter of the property's type. If the property is an array type, the method includes an additional parameter that indexes into the property.

Example 1:

```
[set] int BitRate, StatusFields[2];
```

Since an interface defines the two properties, `BitRate` and `StatusFields`, the VIDL compiler generates two accessor methods with prototypes:

```
inlined void setBitRate(int _newval);
inlined void setStatusFields(int _newval, int _index);
```

align Attribute

The `align` attribute allows the actual alignment of properties within the interface to be specified in architectural addressing units.

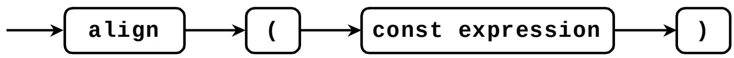


Figure 4-55. `align` Attribute Syntax Diagram

In Figure 4-55, the `alignment_value` is an integer with the same constraint as the parameter used in `#pragma align`, which means the value must be zero (default alignment) or a power of two. Refer to the *VisualDSP++ 3.5 C/C++ Compiler and Library Manual* for your target processor family, or the online Help for more information about pragmas.

Components

A component definition specifies a component in terms of its name, attributes, and the interfaces it provides. A component may provide interfaces by direct implementation, or it may elect to aggregate interfaces provided by other components. The internal details of the implementation are not part of the component's specification, but dependencies on other components are normally recorded by the component's attributes.

Figure 4-56 through Listing 4-59 provide syntax diagrams for a component's declaration and definition.

A component may be declared prior to its full definition. This is a notational convenience that allows a component's name to be introduced prior to its use in an *aggregates* clause or a `requires` attribute, which are defined later in this section. A component definition or declaration introduces a name for the component into the current scope. The component may be referenced outside its scope by using its qualified name.

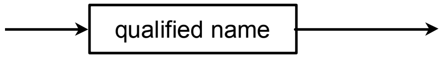


Figure 4-56. Component Name Syntax Diagram

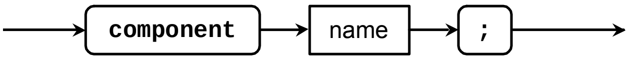


Figure 4-57. Component Declaration Syntax Diagram

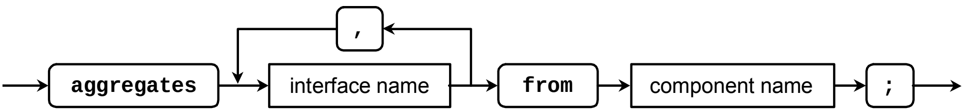


Figure 4-58. Component Aggregation Syntax Diagram

A component definition contains an *implements* clause, which lists the component's external interfaces. The interface list must contain every interface provided by the component—either by direct implementation or aggregation from another component. Each aggregated interface must be identified in a separate `aggregates` clause (see Figure 4-58), which identifies the aggregatable component providing the interface. Where an interface extends another interface, the `implements` and `aggregates` clauses need only contain the name of the derived interface. The extended interfaces are automatically supported by the component.

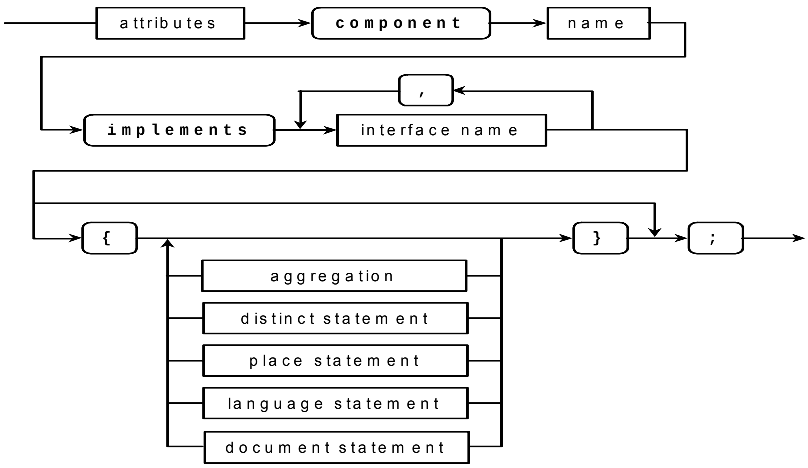


Figure 4-59. Component Definition Syntax Diagram

Example:

```
namespace ADI {
  component CFiddlePlayer;
  component CGuitarPlayer;
  component CKeyBoardPlayer;
  component CBand implements
     IBand, IFiddle, IGuitar, IKeyBoard {
       aggregates IFiddle from CFiddlePlayer;
       aggregates IGuitar from CGuitarPlayer;
       aggregates IKeyBoard from CKeyBoardPlayer;
   };
};
```

In the previous example, the components `CFiddlePlayer`, `CGuitarPlayer`, and `CKeyBoardPlayer` are declared, and the component `CBand` is defined. The `CBand` component provides four interfaces: `IBand`, `IFiddle`, `IGuitar`, and `IKeyBoard`. The first interface is provided directly by `CBand` itself; the remaining three are aggregated from the previously declared components.

The interfaces listed in the `implements` clause and each of their base interfaces may be requested in calls to the component's `Create` factory function and to the `GetInterface` method of the `IBase` root interface.

Component Attributes

A component definition must supply `category`, `component`, and `title` attributes. The set of component attributes is listed in Figure 4-60. Each attribute is briefly described in the following sections.

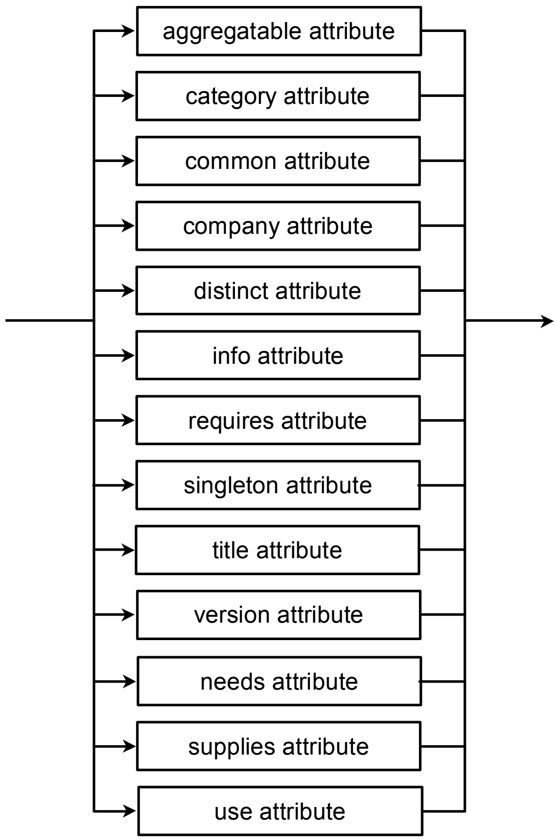


Figure 4-60. Component Attribute Syntax Diagram

aggregatable Attribute

The `aggregatable` attribute identifies a component whose interfaces may be aggregated by another component.

Figure 4-61. `aggregatable` Attribute Syntax Diagram

A component referenced in an `aggregates` clause must be defined to be aggregatable (see Figure 4-61). For more information about the `aggregates` clause, refer to “Components” on page 4-49.

```
namespace ADI {
 [aggregatable,…] component CFiddlePlayer implements IFiddle;
 [aggregatable,…] component CGuitarPlayer implements IGuitar;
 [aggregatable,…] component CKeyBoardPlayer implements IKeyBoard;
};
```

category Attribute

The `category` attribute allows a component to be assigned to one or more component categories.

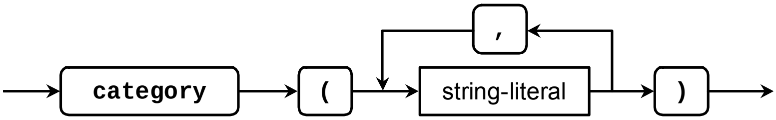


Figure 4-62. `category` Attribute Syntax Diagram

Categories provide hierarchical classification schemes for components based on their functionality. Categories have multipart names that resemble file store path names. The following component categories are predefined.

```
AUDIO
AUDIO\MONO
AUDIO\STEREO
VIDEO
```

The category name is propagated into the component's documentation and packaging information generated by the VIDL compiler. A component definition must provide a `category` attribute.

```
[category("AUDIO"), …] component CDolby implements IDolby;
```

common Attribute

The `common` attribute enables components, which also have the `distinct` attribute, to have a common area for instance storage. The `distinct` interface methods share the same `this`/`__this` pointer for C/C++ implementations.

Figure 4-63. `common` Attribute Syntax Diagram

company Attribute

The `company` attribute identifies the company that developed the component or that acts as the component vendor.

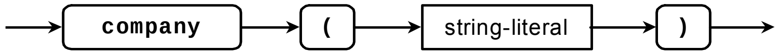


Figure 4-64. `company` Attribute Syntax Diagram

The company name is propagated into the component's documentation and packaging information generated by the VIDL compiler. A component definition must provide a `company` attribute.

```
[category("AUDIO"), company("Analog Devices, Inc"), …]
component CDolby implements IDolby;
```

distinct Attribute

The `distinct` attribute instructs the VIDL compiler to generate separate method shells for interface methods where two or more interfaces, implemented by a component, supply methods that are identical in name and parameter lists.

Figure 4-65. `distinct` Attribute Syntax Diagram

Suppose we have the following (partial) specification.

```
interface I1 extends IBase {
   MRESULT f( [in] int I );
   MRESULT g( [in] int J );
};
interface I2 extends IBase {
```

```
    MRESULT f( [in] int I );
    MRESULT h( [in] int K );
};
component C implements I1, I2;
```

The interfaces `I1` and `I2` each contain a method called `f`, which have identical parameter list signatures (when reproduced in C or C++), and a method called `GetInterface`, which is provided by `IBase`. When a component `C` implements both `I1` and `I2`, the VIDL compiler generates a single shell that contains four methods: `f`, `g`, `h`, and `GetInterface`. Within this shell, the functions `f` and `GetInterface` are shared by both interfaces. When the component `C` is labeled `distinct`, the VIDL compiler generates separate implementation shells for `C` in which *every* method of the interfaces `I1` and `I2`, except `GetInterface`, has a `distinct` method function. In the previous example, if `C` is labeled `distinct`, then the shell for `C`'s implementation of `I1` contains method functions `I1_f` and `I2_g`, and the shell for `C`'s implementation of `I2` contains method functions `I2_f` and `I2_h`. There is a single implementation for `GetInterface` that is shared by each shell. The distinct implementation of such methods is transparent to the user of a component: `I1_f` will be invoked if accessed via an `I1` interface pointer and `I2_f` will be invoked via an `I2` pointer.

If method `f`, in the above example, had a different signature in interface `I1` to that in interface `I2`, then separate methods are generated regardless of whether the `distinct` attribute is used. For a C++ component, this is handled implicitly by the C++ compiler, while for a C component, the VIDL compiler generates separate methods as above.

info Attribute

The `info` attribute allows supplementary information about a component to be supplied as a text string.

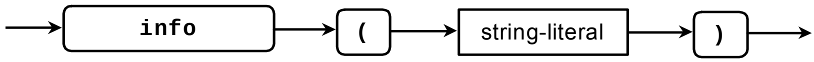


Figure 4-66. `info` Attribute Syntax Diagram

The string may enclose a URL used to link to a webpage provided by the component vendor. The URL is propagated into the component's documentation and packaging information generated by the VIDL compiler.

```
[category("AUDIO"),
   company("Analog Devices, Inc",
   info("http://www.adi.com/dsp/components/audio"), …]
component CDolby implements IDolby;
```

needs Attribute

The `needs` attribute allows a component or an interface to specify interfaces that need to be provided to ensure successful operation. If an interface pointer appears as a parameter within a method of the interface, the VIDL compiler automatically recognizes that the interface or component needs such an interface. Any other interface that the component or interface needs must be explicitly specified using the `needs` attribute. For example the majority of components must be provided with a `VCSE::IMemory` interface; other components require other interfaces.

The syntax of the needs attribute is shown in Figure 4-67

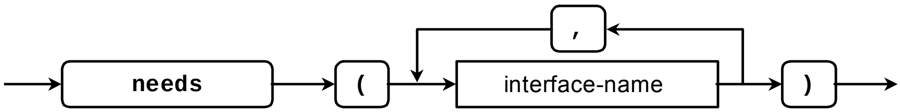


Figure 4-67. needs Attribute Syntax Diagram

The auto-doc generated for the component or interface identifies all needed interfaces, whether they were implicitly or explicitly identified.

requires Attribute

The requires attribute allows a component to specify other components on which it depends. This information is reproduced in the component packaging manifest to ensure that all dependencies on other components are met when installing a component package.

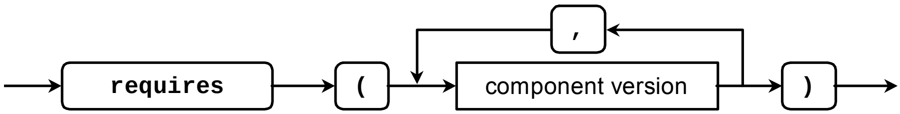


Figure 4-68. requires Attribute Syntax Diagram

Typically, this attribute is used when a component relies on other components for some aspect of its implementation. For example, it may aggregate interfaces from other components or delegate method calls to other components. The required components are specified by name, optionally followed by a version check, which constrains the acceptable versions of the required component. In the case of aggregated components, a requires attribute is only necessary if compatibility with a particular version number is required. Otherwise, the requirement for any version of the aggregated component is automatically included in the component packaging manifest.

```
[requires(ADI::CQuickSort), …]  CSort implements ISort;
[requires(CQuickSort=2.0.0), …] CSort implements ISort;
[requires(::ADI::CQuickSort >=2.0.2), …] CSort implements ISort;
```

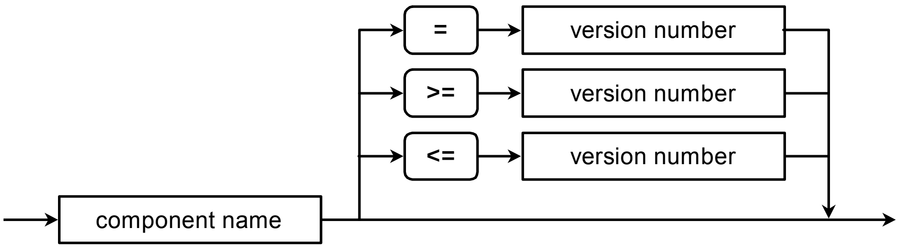


Figure 4-69. Component Version Syntax Diagram

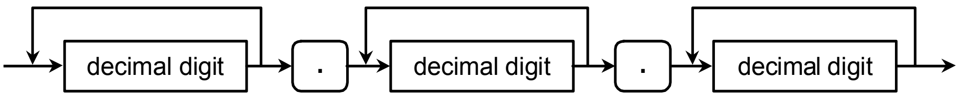


Figure 4-70. Version Number Syntax Diagram

The first decimal digit must be greater or equal to `1`.

singleton Attribute

The `singleton` attribute specifies that only a single instance of the component can exist at any one time and allows the component implementation to be tailored accordingly. The component's `Create` factory function returns an error code if it has been called while an instance already exists.

Figure 4-71. `singleton` Attribute Syntax Diagram

```
[singleton, …] component CMemAlloc implements IMemory;
```

supplies Attribute

The `supplies` attribute specifies that the associated component can supply instances of the nominated interfaces from one or more of its supported interfaces. The auto-doc generated for the component records this information as part of the automatically generated documentation.

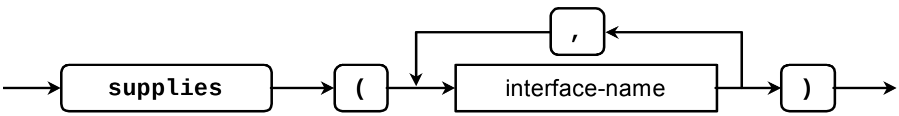


Figure 4-72. `supplies` Attribute Syntax Diagram

title Attribute

The `title` attribute provides a descriptive title for the component being used by the VCSE Component Manager.

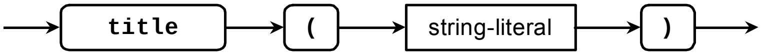


Figure 4-73. `title` Attribute Syntax Diagram

The attribute is propagated into the component's documentation and packaging manifest generated by the VIDL compiler. A component definition must provide a `title` attribute.

```
[title("Dolby 5.1 Decoder"),…] component CDolby implements IDolby;
```

version Attribute

The `version` attribute allows a component version to be specified. The version number is copied into the component's documentation and packaging information generated by the VIDL compiler.

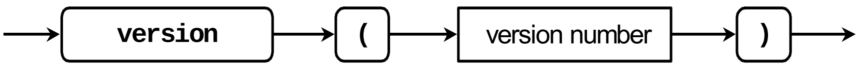


Figure 4-74. `version` Attribute Syntax Diagram

If a component does not have an explicit `version` attribute, then its version number is set to 0.0.0. The component's version number is described on page 4-58.

```
[version(2.0.2)
   category("AUDIO"),
   company("Analog Devices, Inc",
   title("Dolby 5.1 Decoder"),
```

```
    info("http://www.adi.com/dsp/components/audio")]
component CDolby implements IDolby;
```

Additional Statements

In most cases the VIDL compiler's default settings are used. When component developers need more control over how a component is implemented, they can use three additional statements within the VIDL compiler's component definition block:

- The `distinct` statement, described on page 4-64
- The `place` statement, described on page 4-65
- The `language` statement, described on page 4-64

Use the `distinct` statement when only a few methods specify distinct implementations. The remaining methods share implementations. In these cases the use of the `distinct` component attribute is both inappropriate and inefficient.

Where component developers want to locate certain methods in memory segments *other* than the default code segment, they can use the `place` statement to specify which methods on what interfaces are placed and where they are to be placed.

Finally, if component developers use efficient handcrafted Assembler code for their algorithm(s), they must use the `language` statement to generate an Assembler shell for the appropriate method — all other methods can be written in C.

For more detailed descriptions of each of the additional statements as well as their associated syntax diagrams, refer to the sections that follow.

distinct statement

When the `distinct` attribute is specified for a component, any methods with the same name and signature are given separate and distinct implementations. In some situations component developers need to create shared and distinct methods. This level of support, which enables users to specify the names of individual methods provided for distinct implementations, is supplied within the component definition block.

The syntax for the `distinct` statement is

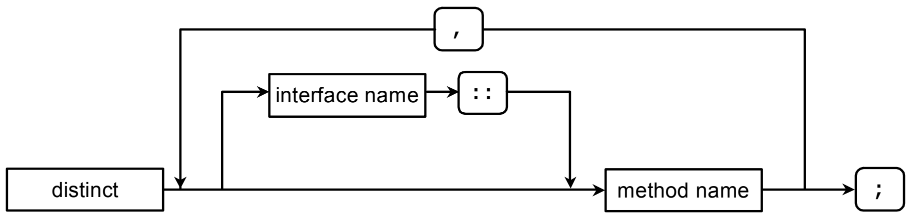


Figure 4-75. `distinct` Method Statement Syntax Diagram

The VIDL compiler ensures that the specified methods (on the specified interfaces) are given an implementation separate from all other methods with the same signature defined on any other interface. Where the qualified interface name is omitted, distinct implementations are generated for *all* implemented interfaces that supply the specified method name.

If for example a method called `DetermineDiff` appears in three interfaces (IG723, IG728 and IGSM) with the same prototype, then by default it would only have a single implementation referenced by each of the three interfaces. If the `distinct` statement below is used

```
distinct IG723::DetermineDiff;
```

then the IG723 interface uses one implementation, while the IG728 and the IGSM interfaces share a separate implementation.

If the `distinct` statement below is used

```
distinct DetermineDiff;
```

then each of the interfaces has a separate implementation.

place statement

Normally each method is placed in the default code section. The `place` statement allows the user to explicitly specify the name of the section in which a method is to be placed. The syntax for the `place` statement is

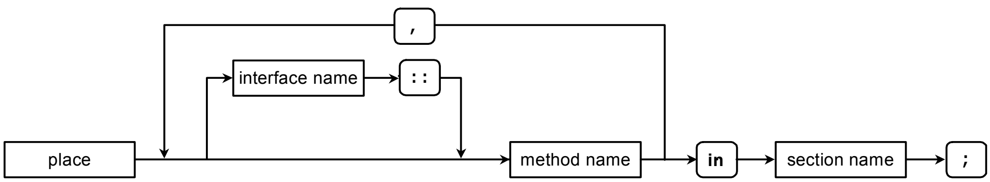


Figure 4-76. Method Placement Statement Syntax Diagram

The compiler ensures that definitions of specified methods (on the specified interfaces) are assigned a section qualifier, which specifies the name of the section where the method is to be placed. When the qualified interface name is omitted all implementations of the specified method name are placed in the specified section.

If for example a method called `DetermineDiff` and a method called `OffsetData` appears in the IG723 interface, then the functions that implement the methods can be placed in a section called `G723_L1code` by using the `place` statement

```
place IG723::DetermineDiff, IG723::OffsetData in G732_L1code;
```

If the `determineDiff` method also appears in the IG728 interface (with a different prototype so that it has a separate implementation for each interface), the function that implements the method for each interface can be placed in the section called `Common_L1code` by using the `place` statement

```
place DetermineDiff in Common_L1code;
```

When no interface name is specified then all methods of that name irrespective of the interface in which they appear are placed in the specified section.

language statement

Normally the VIDL compiler generates shells for every method in the same implementation language. The language used for the shells is specified in the property pages for the VIDL compiler or on the VIDL compiler command line. The `language` statement allows the user to override this default behavior by explicitly specifying the implementation language to be used for a particular method. The `language` statement only allows the implementation language to be either C or assembly. Where the qualified interface name is omitted all implementations of the specified method name are in the specified language.

The syntax for the `language` statement is

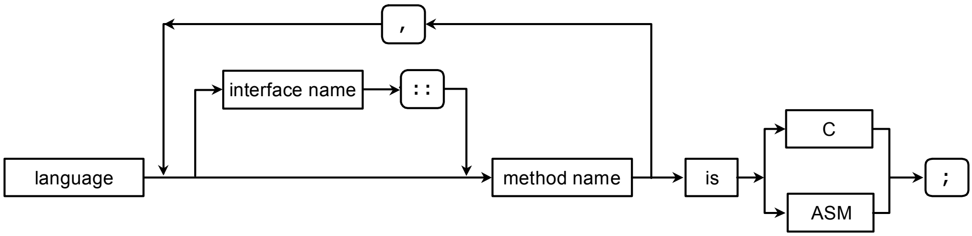


Figure 4-77. Method Language Statement Syntax Diagram

The `language` statement does not currently support components generated in C++.

If for example a method called `DetermineDiff` and a method called `OffsetData` appears in the IG723 interface, the implementation language of the functions that implement the methods can be specified as assembler (`asm`) by using the `language` statement below

```
language IG723::DetermineDiff, IG723::OffsetData is asm;
```

If the `DetermineDiff` method also appears in the IG728 interface (with a different prototype so that it has a separate implementation for each interface), the implementation language of the functions that implement both methods can be specified as "assembler" by using the `language` statement

```
language DetermineDiff is asm;
```

When no interface name is specified then all methods of that name, irrespective of the interface in which they appear, are implemented in the specified language.

Namespaces

A namespace defines a scope containing the definitions of VIDL types, interfaces, components, and nested namespaces. The name of the namespace may be used as a scope prefix in a qualified name (see Figure 4-78) or in a `use` attribute. For information about qualified names, see "Named Elements" on page 4-13.

Namespaces provide a convenient way to partition the global scope in order to avoid name clashes. All named VIDL elements must be enclosed (directly or indirectly) by a namespace.

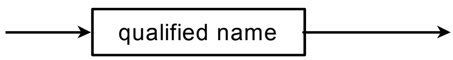


Figure 4-78. Namespace Name Syntax Diagram

Namespaces may have multiple cumulative *declarations*, provided they occur within the same enclosing scope. The namespace declaration and definition forms are shown in Figure 4-79 and Figure 4-80 on page 4-69.

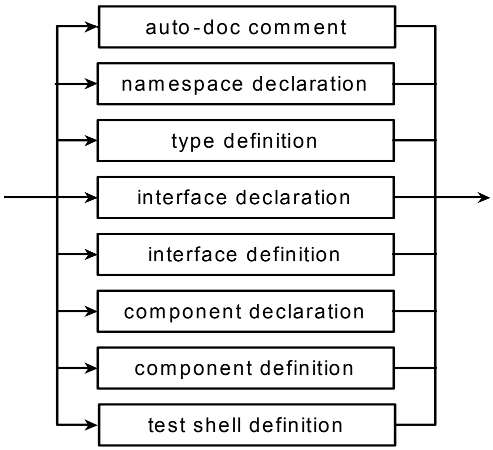


Figure 4-79. Element Definition Syntax Diagram

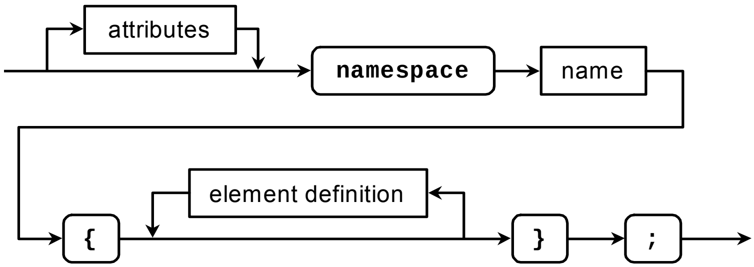


Figure 4-80. Namespace Declaration Syntax Diagram

Listing 4-6. Use of Separate Namespace Blocks

```
/* ACME's types */
namespace ACME {

   typedef unsigned int NType;
   typedef int SType;
};
…
…
 /* ACME's interfaces */
 namespace ACME {
   [iid("10768745-271ad611-a55c0010-4b7cac83")]
   interface ISort extends IBase {
     MRESULT SetData([in] NType N,
                     [in,size_is(N)] SType data[]);
     MRESULT GetData([in] Ntype N,
                     [out,size_is(N)] Stype data[]);
     MRESULT Sort(void);
   };
};
...
 /* ACME's components */
 namespace ACME {
```

```
    [version(1.5.0), company('ACME Software Inc'), …]
    component CQuickSort implements ISort;
    [version(1.5.0), company('ACME Software Inc'), …]
    component CBubbleSort implements ISort;
  };
```

The VIDL shown in the Listing 4-6 may, in practice, have each declaration of the `ACME` namespace located within a separate VIDL file, which may be incorporated into other specifications via the `#include` preprocessor directive. The names defined in the `ACME` namespace can be accessed from any other namespaces using a qualified name. For example, company Analog Devices, Inc. may extend the `ACME::ISort` interface but delegate the implementation of the `ISort` methods to `ACME::CQuickSort`. The dependency is recorded as follows.

```
namespace ADI {
   interface IProcess extends ::ACME::ISort {
      MRESULT ProcessData(void);
   }
   [version (2.0.0), requires(CQuickSort>=1.5.0),
    company("Analog Devices, Inc"), …]
   component CProcess implements IProcess;
};
```

use Attribute

A namespace definition can include a `use` attribute employed to control the order in which namespace scopes are searched when locating the definition of a name. The attribute's form is shown in Figure 4-81.

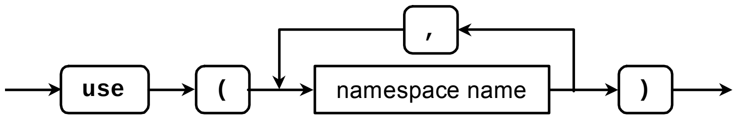


Figure 4-81. `use` Attribute Syntax Diagram

When a name `n` is used in a namespace `X`, the VIDL compiler searches for the definition of `n` in `X`. If the name is not defined in `X`, the VIDL compiler continues the search for `n` in the namespaces listed in any `use` attribute attached to `X`. If the `use` attribute takes the form `[use(Y, Z)]`, `Y` is searched before `Z`. If the name is not found in either `Y` or `Z`, the search continues in the scope that encloses namespace `X`. If there is no enclosing scope, the VIDL compiler searches the predefined namespace `VCSE`.

A `use` attribute can be applied in the previous example to allow `::ACME::CQuickSort` and `::ACME::ISort` to be referred to by their unqualified names:

```
[use(::ACME)] namespace ADI {
   interface IProcess extends ISort {
     MRESULT ProcessData(void);
   }
   [version (2.0.0), requires(::ACME::CQuickSort>=1.5.0),
    company("Analog Devices, Inc")]
   component CProcess implements IProcess;
};
```

When a company tag is used to qualify a name or as a parameter in the `use` attribute, the fully qualified name is preferable to the unqualified one.

The use attributes may also be used to override the normal order in which nested scopes are searched.

```
namespace A {
    typedef unsigned int T;
    namespace B {
       typedef int T;
       namespace C {
          /* Search scopes C, B, A, VCSE  */
          typedef T TC;    /* finds B::T */ };
          [use(A,B)] namespace D {
          /* Search scopes D, A, B, C, VCSE */
          typedef T TD;      /* finds A::T */
       };
    };
 };
```

In the definition of type TC in namespace C, the definition of T is located by searching the scopes C, B, A, and then VCSE. The definition is located in the enclosing namespace B; therefore, TC has type int. In the definition of type TD in namespace D, the definition of T is located by searching the scopes D, A, B, C, and then VCSE. The definition is located in the outer namespace A; hence, TD has type unsigned int.

Auto-doc Comments

Auto-doc comments are stylized VIDL remarks used by the VIDL compiler to generate HTML documentation for components, interfaces, types, properties, and methods. An auto-doc comment is distinguished by its opening /** marker followed by blanks and end of line. There must be a corresponding closing marker */ that occurs on a following line. Each intermediate line must start with an *, optionally preceded with white space.

Auto-doc comments contain an overview description of the component, interface, type, property or method to which they apply, followed by one or more tagged paragraphs. The descriptive text within the comment may contain embedded HTML directives. Auto-doc tags are prefixed with an @ character and allow attributes of the component, interface, or method to be clearly documented and tabulated in HTML. In the following example, the first auto-doc comment provides a summary of the `ISort` interface, and the remaining comments provide documentation for each of the methods.

```
namespace ADI {
/**
 * The ISort Interface provides a generic sorting capability for
 * floating-point data. The data to be sorted must be supplied by
 * calling SetData before attempting to invoke the Sort method.
 * Once Sort has been invoked, the sorted data can be retrieved
 * by the invoking GetData.
 */
 [iid("20aa3d29-4c1ad611-a55c0010-4b7cac83")]
 interface ISort extends IBase {
   /**
    * The SetData method supplies an array of float data values
    * to be sorted.
    * @param N       An input parameter specifying the number of
    *               elements in array parameter data.
    * @param data    An input parameter supplying the data array
    *               be sorted. The corresponding actual array
    *               argument must have at least N elements.
    * @return        MR_OK if the method is successful. An error
    * @postcondition It is valid for the interface to sort data.
    *               See SortData().
    *               code if the method fails.
    */
    MRESULT SetData([in] int N, [in, size_is(N)] float data[]);
```

```
    /**
     * The GetData method retrieves an array of float data
     * values that have been sorted. Must be preceded by a call
     * to Sort.
     * @param N    An input parameter specifying the number of
     *             elements in array parameter data.
     * @param data An output parameter to hold the data array
     *             that has been sorted. The corresponding actual
     *             array argument must have at least N elements.
     * @return     MR_OK if the method is successful. An
     *             error code if the method fails.
@precondition The data has been sorted.  See SortData().
     */
     MRESULT GetData([in] int N, [out, size_is(N)] int data[]);
     *@precondition The data has been sorted.  See SortData().
     /**
      * The Sort method applies a sorting algorithm to the data
      * supplied by a previous call to SetData. The sorting
      * algorithm is provided by the interface implementation.
      * @return      MR_OK if the method successful. An error
      *             code if the method fails.
      * @precondition Interface must be set with data.
      *             See SetData().
      */
      MRESULT Sort(void);
    };
  };
```

The VIDL compiler accepts the following auto-doc tags.

`@param`	Applies to methods and provides a description of a method parameter that includes the name and the nature of the values that are transmitted. incorporated into other specifications via the `#include` preprocessor directive.

`@return`	Applies to methods and provides a description of the values of the type `MRESULT` returned by the method.
`@example`	Applies to interfaces and provides a fragment of example code showing how the methods are called.
`@author`	Applies to components allowing authorship to be attributed to a named individual or organization.
`@keyword`	Supplies a keyword to the index, which is compiled into the HTML based help information. The tag may be included into any auto-doc comment. The keyword is supplied after the tag.
`@create`	Applies to components and allows the components `Create` factory function include some documentation supplied which is specific to this component.
`@sizeof`	Applies to components and allows the components `SizeOf` factory function to include documentation specific to this component.
`@destroy`	Applies to components and allows the components `Destroy` factory function to include documentation specific to this component.
`@precondition`	Applies to methods and allows any conditions that should be met before invoking the method to be specified.
`@postcondition`	Applies to methods and allows any actions that should be taken or any conditions that are valid after invoking the method to be specified.

Specifications

A VIDL specification is a sequence of namespace declarations and auto-doc comments. Each namespace defines a scope that may contain the definitions of nested namespaces, components, interfaces, constants, and types, as well as their related auto-doc comments. The specification format is presented in Figure 4-82.

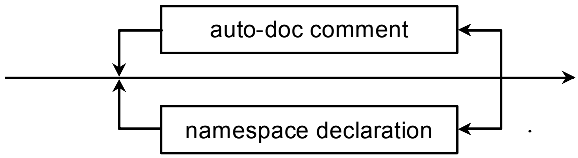


Figure 4-82. VIDL Specification Syntax Diagram

Every component, interface, constant, and type must be declared within a namespace scope.

```
/**
 * ::ADI is the company namespace for Analog Devices, Inc.
 */
namespace ADI {
   /**
    * The CQuickSort component provides an aggregatable
    * implementation of the ADI::ISort interface using a
    * quick-sort algorithm.
    */
   [title("QuickSort"),
     category("SORT"),
     company("Analog Devices, Inc"),
     aggregatable,
     version(1.1.0)]
   component CQuickSort implements ISort;
};
```

Generated Test Shells

The *VIDL test shell syntax* provides a nonintrusive means of testing a component that is either under development, or being used in its targeted application. The VIDL compiler enables users to generate a component identical to the component under test, except that the interface methods of the test shell component contain code prior to and following a call to the actual method of the component under test. The content of this code is determined by the test attributes specified in the VIDL test shell syntax. Placing this code within user-modifiable blocks allows for the amendment and insertion of code for additional testing to be performed.

In most cases, the `Create` factory function of the test shell component contains code used to create an instance of the actual component and stores the associated interface pointers as instance data within the test shell component. When an `VCSE::IMemory`-based component is under test, however, the test shell component functions differently. For that reason, users may choose to reuse the memory management component passed in to the `Create` function via the `ienvp` parameter. For more information, see "Creating Component Instances" on page 2-61.

By using the test shell syntax the developer can express many common test constructs in a simple manner separate from the syntax of the actual component or its implemented interfaces. This unique feature allows both the test shell and its `.idl` file to be added and removed from a project easily.

Overview

In its simplest form the test shell syntax can be expressed as:

```
testing component-name {};
```

This statement generates a simple shell, which allows users to add code in the user-modifiable blocks provided. The statement above also defines the component scope of the test shell syntax for which test attributes can be declared, for example:

```
[array_check] testing component-name {};
```

Within the component scope additional scopes can be nested. The hierarchy of scopes is:

- Component scope (highest)
- Interface scope
- Method scope
- Argument scope

Each test attribute has a target entity, such as a parameter of an interface method. Attributes declared on a scope apply to all appropriate targets within that scope. For example, the test shell attribute `array_check` can be declared on any of the four levels of scope, but it only tests `array-type` parameters of an interface method. In the case of test shell syntax that declares `array_check` for the component scope, array checking is selected for all `array-type` parameters on all methods of all interfaces implemented by the component under test.

At each scope level, additional test attributes can be applied or removed. Only certain attributes can be removed by prefixing the attribute name with `no_`. For example:

```
[array_check] testing component-name {
      foo( [no_array_check] array1, [alias_check]array2 );
};
```

In this example a request is made for array checking for `array-type` parameters of all methods except `foo`, where array checking is deselected for parameter `array1` but remains for parameter `array2`. Alias checking is only selected for `array2` of method `foo`.

With the exception of the component scope, the declaration of all remaining scopes within the VIDL syntax is optional. If the declaration is omitted its use is implied. In the example,

```
[timing] testing component-name {};
```

the `timing` attribute is applied to all methods for all interfaces implemented by the component.

Table 4-5 details the test attributes and their targets and on which scope(s) they can be declared.

Table 4-5. Test Shell Attributes

KeyWord	Scopes				Target of Test
	Component	Interface	Method	Argument	
stack_usage	yes	yes	yes	no	Method
mem_shell	yes	no	no	no	Component
timing	yes	yes	yes	no	Method

Table 4-5. Test Shell Attributes (Cont'd)

KeyWord	Scopes				Target of Test
	Component	Interface	Method	Argument	
algorithm_model	yes	yes	no	no	Component
aliasing_check	yes	yes	yes	yes	Argument
array_check	yes	yes	yes	yes	Argument
states_used	yes	no	no	no	Component
init_state	yes	no	no	no	Component
requires_state	no	no	yes	no	Method
sets_state	no	no	yes	no	Method
reset_to_state	no	no	yes	no	Method
clear_state	no	no	yes	no	Method
in_assert	no	no	no	yes	Method
out_assert	no	no	no	yes	Method
no_stack_usage	no	yes	yes	no	Method
no_timing	no	yes	yes	no	Method
no_aliasing_check	no	yes	yes	yes	Argument
no_array_check	no	yes	yes	yes	Argument

Syntax Structure

The general form of the test shell syntax is:

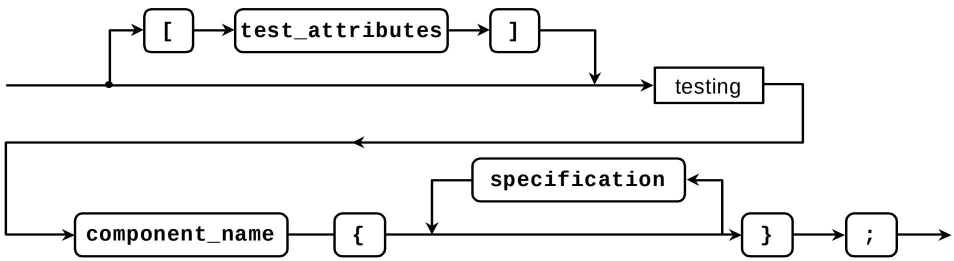


Figure 4-83. Test Shell Syntax Diagram

Where `specification` can be either a test shell interface or a test shell method.

Syntax Rules

There are several general comments that apply to the test shell statements

- Only one definition per test component/interface/method/argument can be included.
- Namespaces are, as usual, allowed to be opened and closed.
- Interfaces and methods can be declared by their fully qualified scope or within a valid `namespace`.
- Test constraints declared at the upper levels are inherited to the lower level, unless turned off by the `no_attribute` attribute.

- Any ambiguous names within a component will be resolved using the standard name resolution rules as described in "Named Elements" on page 4-13.
- Arguments for the test method may be supplied in any order. This means that the order in which they are declared in the test shell does not need to be the same order in which they were declared by the actual interface and not all the arguments need to be referenced.

specification

Each test shell specification can be for either a declaration for a method or an interface as shown in Figure 4-84.

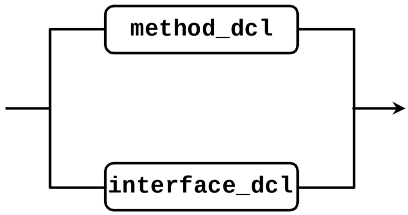


Figure 4-84. `specification` Syntax Diagram

method_dcl

The declaration for a method supplies a set of attributes that applies to each specified method and each attribute applies (by default) to each parameter of each method. Figure 4-85 shows the syntax for a method declaration.

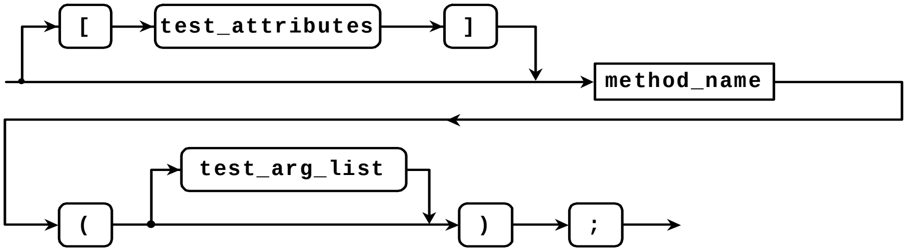


Figure 4-85. `method-dcl` Syntax Diagram

test_arg_list

Figure 4-86 shows the syntax for specifying a set of attributes that apply to a list of argument names. The attributes for the name can either add attributes to the specified argument or be used to override the attribute settings inherited from its method or interface.

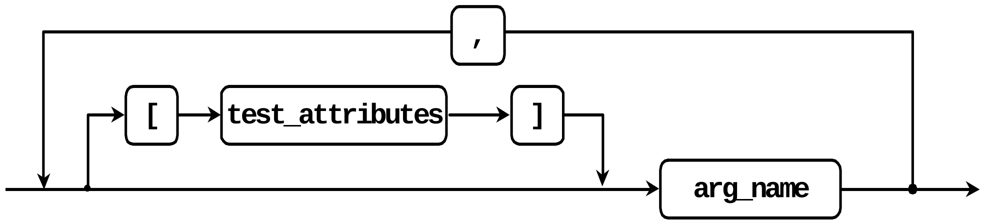


Figure 4-86. `test_arg_list` Syntax Diagram

interface_dcl

The declaration for an interface supplies a set of attributes that applies to the specified interface and each of its supported methods. Each attribute also applies (by default) to each parameter of each method. Figure 4-87 shows the syntax for a method declaration.

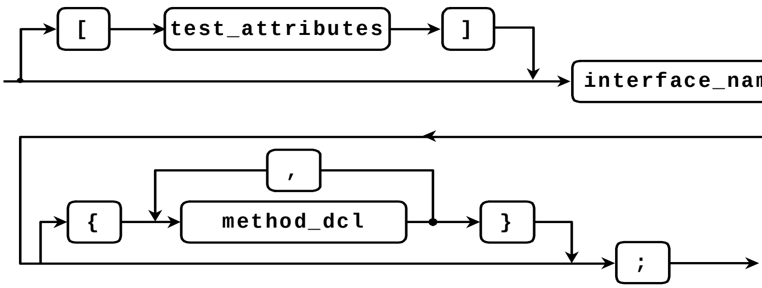


Figure 4-87. `interface_dcl` Syntax Diagram

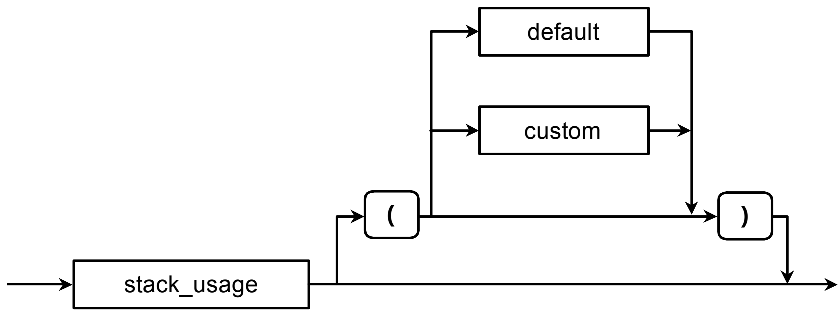
Syntax and Semantics

This section describes the semantics and formats for each of the available test attributes.

- `stack_usage`, described on page 4-86
- `mem_shell`, described on page 4-87
- `timing`, described on page 4-88
- `in_assert`, described on page 4-89t
- `out_assert`, described on page 4-90
- `algorithm_model`, described on page 4-91
- `aliasing_check`, described on page 4-92
- `array_check`, described on page 4-93
- `states_used`, described on page 4-94
- `init_state`, described on page 4-95
- `requires_state`, described on page 4-96
- `sets_state`, described on page 4-97
- `reset_to_state`, described on page 4-98
- `clear_state`, described on page 4-99

stack_usage

Syntax



Description

The `stack_usage` attribute instructs the VIDL compiler to insert code in the test shell component to measure how much the stack grows during the call to each of the required interface methods of the actual component. Stack-use measurements can be performed in one of several ways, as dictated by the optional `default` or `custom` parameters.

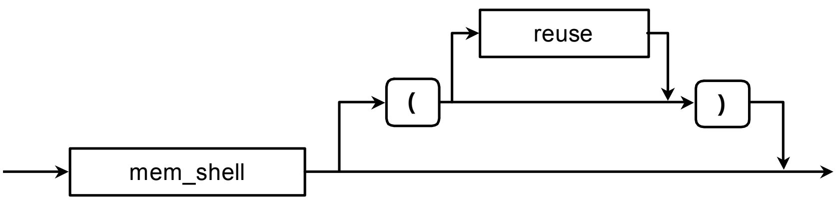
The parameters to `stack_usage` are:

- `default` **input.** Instructs the VIDL compiler to generate code that utilizes the functions provided in the VCSE library to measure the usage of the `MEM_STACK` section of memory, defined in the default linker description file (`.ldf`) for the target processor.
- `custom` **input.** Instructs the VIDL Compile to generate empty macros for the user to add their own implementation of stack usage measurement.

Named Elements	Scope
Component	Valid
Interface	Valid
Method	Valid
Argument	Invalid

mem_shell

Syntax



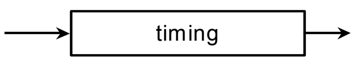
Description

The `mem_shell` attribute instructs the VIDL compiler to generate a test shell for a component that implements `VCSE::IMemory`. Such a test shell component can be used in a client application, in place of an actual memory management implementation, or to report statistics about the memory usage of components used by the application. Additionally, in conjunction with the `-validate-memory` switch passed to the VIDL compiler, code is generated to validate allocated memory before it is freed.

Named Elements	Scope
Component	Valid
Interface	Invalid
Method	Invalid
Argument	Invalid

timing

Syntax



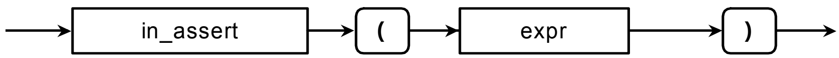
Description

The `timing` attribute instructs the VIDL compiler to insert code in the test shell to collect information about the number of cycles used by a method.

Named Elements	Scope
Component	Valid
Interface	Valid
Method	Valid
Argument	Invalid

in_assert

Syntax



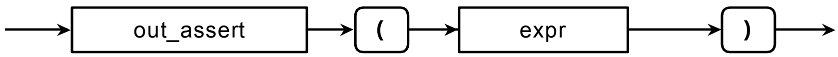
Description

The `in_assert` attribute instructs the VIDL compiler to insert code in the test shell, prior to a forwarding call to an interface method of the component under test, to validate the entry conditions of the method in question according to the given `expr`.

Named Elements	Scope
Component	Invalid
Interface	Invalid
Method	Valid
Argument	Invalid

out_assert

Syntax



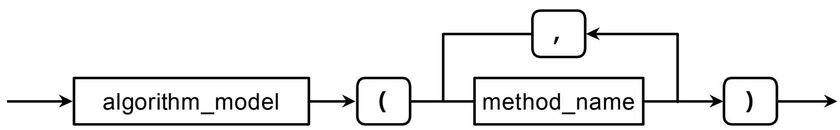
Description

The `out_assert` attribute instructs the VIDL compiler to insert code in the test shell, subsequent to a forwarding call to an interface method of the component under test, to validate the exit conditions of the method in question according to the given `expr`.

Named Elements	**Scope**
Component	Invalid
Interface	Invalid
Method	Valid
Argument	Invalid

algorithm_model

Syntax



Description

The algorithm_model attribute instructs the VIDL Compiler to generate code in the test shell to validate the conformance of an IAlgorithm-based component to the rules described by the documentation of IAlgorithm. Basic rules are as follows:

1. Ensure that Reset is invoked at least once before the first call to a processing method.
2. The Deactivate method is never invoked when it is already deactivated.
3. The Activate method is never invoked when it is already activated.
4. Any methods marked as process methods are only invoked when activated.

Processing methods are marked by passing in their names to algorithm_model or by passing in '*', which marks all methods, excluding the methods of the IAlgorithm interface.

Named Elements	Scope
Component	Valid
Interface	Valid
Method	Invalid
Argument	Invalid

aliasing_check

Syntax

Description

The `aliasing_check` attribute instructs the VIDL Compiler to generate code in the test shell to check that the marked parameter is not aliased to other parameters marked with `aliasing_check` in the same

method. Aliasing occurs when two or more names refer to the same object; that is, when they share the same memory address.

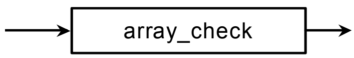
Alias checks are only valid for reference type parameters.

Because a parameter marked with an `aliasing_check` is validated against other parameters marked with `aliasing_check`, more than one parameter must be marked with an `aliasing_check` in order for it to be valid.

Named Elements	Scope
Component	Valid
Interface	Valid
Method	Valid
Argument	Valid

array_check

Syntax



Description

The `array_check` attribute instructs the VIDL Compiler to insert code ion the test shell to perform basic array checks on an array-type parameter.

Basic array checks include:

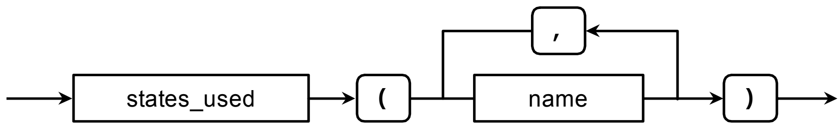
1. If an array is an `out` parameter, then the test shell checks to see if memory areas just beyond the array's boundaries are written to.
2. If an array is an `in`-only parameter, then the program checks to see that the data within the array has not been changed by the invoked method.

Parameters marked with `array_check` must be an array-type.

Named Elements	Scope
Component	Valid
Interface	Valid
Method	Valid
Argument	Valid

states_used

Syntax



Description

The `states_used` keyword declares the states of a test shell component. These defined states can then be used in test attributes:

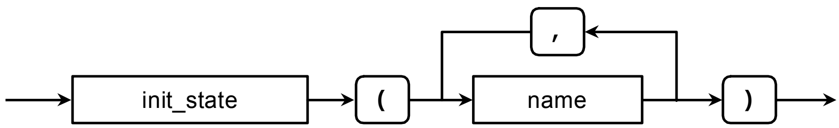
- `init_state`
- `sets_state`
- `clears_state`
- `resets_to state`

State names must be unique and are case sensitive. They are used in conjunction with the attributes (listed above) to check that the order of invocation of the components methods conform to the desired behavior as detailed in the component's documentation. At one level the use of the state test attributes provides a user-definable approach to checking algorithmic components.

Named Elements	Scope
Component	Valid
Interface	Invalid
Method	Invalid
Argument	Invalid

init_state

Syntax



Description

The `init_state` keyword declares which states are initially set to ON after the creation of the test shell.

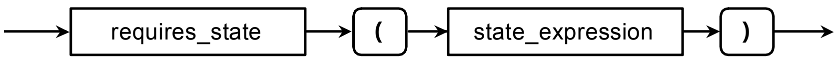
State names must be defined previously in the `states_used` test attribute.

Named Elements	Scope
Component	Valid
Interface	Invalid

Named Elements	Scope
Method	Invalid
Argument	Invalid

requires_state

Syntax



Description

The `requires_state` keyword specifies that the system must be in the declared state prior to the invocation of the marked method. That is, the component should be in the specified state when the method is invoked.

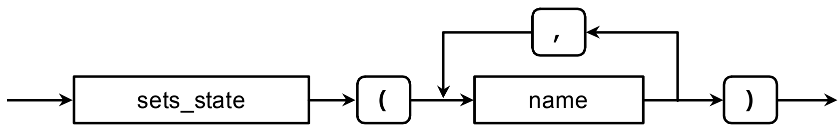
The parameter to `requires_state` is a state expression. State expressions are expression composed of state names, state operations and parenthesis. Valid state operations are negation (~, logical OR (`|`) and logical AND (`&`).

State names must be defined previously in the `states_used` test attribute.

Named Elements	Scope
Component	Invalid
Interface	Invalid
Method	Valid
Argument	Invalid

sets_state

Syntax



Description

The `sets_state` keyword declares a list of states to be turned on as result of the marked method being executed.

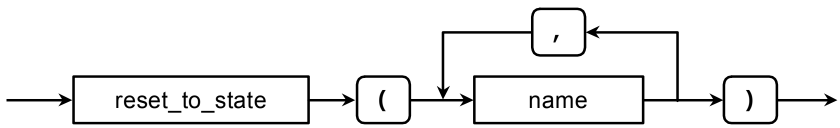
State names used in the input must be defined previously using the `states_used` test attribute.

State names used in the input must not be used in a `clears_state` attribute of the same method.

Named Elements	Scope
Component	Invalid
Interface	Invalid
Method	Valid
Argument	Invalid

reset_to_state

Syntax



Description

When the marked method is invoked, all states defined in `states_used` are turned OFF, except for the states listed as input to the `reset_to_state` attribute.

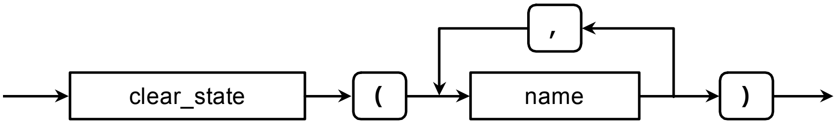
State names used in the input must be defined previously using the `states_used` test attribute.

If a method uses `reset_to_state`, then the method cannot have the test attributes `sets_state` or `clears_state`.

Named Elements	Scope
Component	Invalid
Interface	Invalid
Method	Valid
Argument	Invalid

clear_state

Syntax



Description

The clear_state keyword declares a list of states to be turned off when the marked method is executed.

State names used in the input must be defined previously using the states_used test attribute.

State names used in the input must not be used in a clear_state attribute of the same method.

Named Elements	**Scope**
Component	Invalid
Interface	Invalid
Method	Valid
Argument	Invalid

5 VIDL COMPILER COMMAND LINE INTERFACE

This chapter describes how the VIDL compiler is invoked from the command line, the various types of files processed and generated by the compiler, and the option (switch) set used to tailor its operation.

The chapter contains:

- "Running VIDL Compiler" on page 5-1
- "Processing VIDL Files" on page 5-22
- "Generated Source Files" on page 5-28

The VIDL compiler processes the supplied VIDL source file and generates header files for each specified interface and an implementation shell for each specified component. Each generated header file can be processed by the assembler and C or C++ compiler.

The VIDL compiler lets you specify the language in which the implementation shells are generated. The default implementation language is C; shells in C++ or assembly can also be generated for the platforms that support these languages.

Note that ADSP-218x DSP compilers do not support C++.

Running VIDL Compiler

Use the following syntax for the VIDL compiler command line.

vidl_family [*-switch* [*-switch* …]] *sourcefile*] where:

- `vidl_family` is the name of the VIDL compiler (`.DXE`). Select the name that corresponds to your target processor family:

Table 5-1. Target Processor Families

VIDL Compiler	Processor Family
`vidlblkfn`	ADSP-BF53x Blackfin
`vidl218x`	ADSP-218x
`vidl219x`	ADSP-219x

- `source_file` is the name of the VIDL file to be preprocessed and compiled.

  The file name can include the drive, directory, file name, and file name extension. The compiler supports both Win32- and POSIX-style paths, using either forward or back slashes as the directory delimiter. The compiler also supports UNC path names starting with two slashes and a network name.

  If the file name contains spaces, enclose it in double quotes: "`long file name.idl`". The VIDL compiler expects the file extension to be `.IDL`, ignoring any files that do not have this extension. The compiler only processes the first `.IDL` file it encounters, ignoring all subsequent files with the same extension.

- `-switch` is the name of the switch to be processed. The compiler has many switches that control the generated code and the operation of the compiler. Command line switches are case sensitive, meaning that `-v` is not the same as `-V`.

  Double quotes can be used to embed spaces in switches, and a `\` (backslash) may be used to pass a double quote to the compiler as part of a switch.

Each of the following command lines,

```
vidlblkfn -c++ -trace source.idl
vidl219x -c++ -trace source.idl
vidl218x -c++ -trace source.idl
vidlts -c++ -trace source.idl
vidl21xxx -c++ -trace source.idl
```

runs the VIDL compiler for the appropriate DSP family with:

`-c++`	Elects the generation of C++ component shell files and any associated header file(s). The `vidl218x` option treats the `-c++` option as an error because there is no C++ compiler for ADSP-218x DSPs.
`-trace`	Selects the inclusion of debug code in the component's source files.
`source.idl`	Names the file containing the VIDL specification to process.

Each of the following command lines,

```
vidlblkfn -hdr -Ic:\interfaces interface.idl
vidl219x -hdr -Ic:\interfaces interface.idl
vidl218x -hdr -Ic:\interfaces interface.idl
```

runs the VIDL compiler for the appropriate DSP family with:

`-hdr`	Selects the generation of only the header files for any interfaces specified in the VIDL file.
`-Ic:\interfaces`	Specifies the directory `c:\interfaces` is to be searched when the pre-processor is including files.
`interface.idl`	Names the file containing the VIDL specification to process.

When providing an input or output file name as an optional parameter, use the following guidelines.

- Use a file name, including the extension, with either an unambiguous relative path or an absolute path. A file name with an absolute path includes the drive, directory, file name, and extension. Enclose long file names within double quotes: "`long file name.idl`".
- Verify the compiler is using the correct file. If you do not provide the complete file path as part of the parameter or add additional search directories, the VIDL compiler looks for input in the current project directory.

The VIDL compiler defines the preprocessor macros listed in Table 5-2 to have the value `1`.

Table 5-2. Preprocessor Macros

Compiler	Preprocess Macros
`vidlblkfn`	`__ADSPBLACKFIN__`
`vidl218x`	`__ADSP218X__`
`vidl219x`	`__ADSP219x__`

VIDL Compiler Switches

This section describes the command-line switches used when compiling VIDL source files. A summary of the switch set, organized by type, is in Table 5-3 and Table 5-3 on page 5-5. A more in-depth description of each switch, listed in alphabetical order, follows the tables.

Table 5-3. VIDL Compiler Common Switches

Switch	Reference	Description
`-@ filename`	on page 5-8	Reads command line input from the specified file.
`-accept-any-include-file`	on page 5-8	Accepts `#include` statements that specify any file type and not just `.idl`.
`-all-idl`	on page 5-8	Generates headers and implementation shells for interfaces and components in all nested included files.
`-asm`	on page 5-9	Generates assembly based implementation shells, overrides the default (C based shells).
`-c++`	on page 5-9	Generates C++ based implementation shells, overrides the default (C based shells).
`-copyright filename`	on page 5-9	Specifies copyright text to be inserted in generated source files.
`-cppflags flags`	on page 5-9	Passes additional information to the C preprocessor.
`-Dmacro[=def]`	on page 5-10	Defines the named macro(s).
`-dryrun`	on page 5-10	Displays, but does not perform, the main driver actions.
`-embedded`	on page 5-10	Suppresses generation of factory methods and headers to facilitate use of components embedded within components.
`-generic`	on page 5-10	Generates code suitable for compilation with C/C++ compilers other than those supplied with VisualDSP++.
`-harness`	on page 5-11	Generates a test program for the component.
`-hdr`	on page 5-11	Generates interface headers; does not generate component shells.
`-h[elp]`	on page 5-11	Outputs a list of command line switches with brief descriptions.
`-Idirectory`	on page 5-12	Appends the specified directory to the standard search path.

Table 5-3. VIDL Compiler Common Switches (Cont'd)

Switch	Reference	Description
`-lghtwt`	on page 5-13	Directs the VIDL compiler to generate light-weight components — components with no instance data and minimal overhead.
`-mcd`	on page 5-13	Generates implementation shells for multiple components.
`-M`	on page 5-12	Generates make rules only; does not compile.
`-MM`	on page 5-12	Generates make rules and compiles.
`-no-adoc`	on page 5-13	Does not generate HTML documentation files.
`-no-shell`	on page 5-13	Suppresses generation of the test shell component defined in the specified VIDL file.
`-no-vla`	on page 5-14	Does not generate variable-length arrays in C implementation shells.
`-no-xml`	on page 5-14	Does not generate the XML component manifest.
`-overwrite`	on page 5-14	Allows already existing test harness program to be overwritten.
`-path-def` *path*	on page 5-14	Specifies an alternative driver configuration file.
`-path-html` *directory*	on page 5-15	Specifies the location of HTML documentation template files.
`-path-install` *directory*	on page 5-15	Directs the VIDL compiler to use the specified directory as the base directory for all VIDL tools, include directories, and configuration files.
`-path-output` *directory*	on page 5-15	Specifies the location of non-temporary files.
`-path-temp` *directory*	on page 5-15	Specifies the location of temporary files generated by the driver.
`-path-`*tool path*	on page 5-14	Specifies the location of the named compilation tool.

Table 5-3. VIDL Compiler Common Switches (Cont'd)

Switch	Reference	Description
-proc *processorID*	on page 5-15	Generates code for the specified Blackfin processors. Only one -proc is permitted on a command line.
-save-temps	on page 5-18	Saves intermediate compilation files.
-shell-only	on page 5-19	Directs the VIDL compiler to generate only the test shell component and suppresses generation of the actual component defined in the specified VIDL file.
-si-revision <revision>	on page 5-19	Directs the VIDL compiler to produce code (where applicable) that conforms to the given revision number. Format of <revision> is <major>.<minor> with minor <=255.
-states-verbose-errors	on page 5-19	Directs the VIDL compiler to insert more verbose messages into the component test shell when reporting non-conformance of the component to the states strategy defined in the specified VIDL file.
-trace	on page 5-19	Generates debug code.
-U*macro*	on page 5-20	Undefines the named macro(s).
-umb-verbose	on page 5-20	Directs the VIDL compiler to generate User Modifiable Blocks, which are more prominent in the generated code and containing explanatory comments.
-validate-memory	on page 5-20	Directs the VIDL compiler to generate code within an IMemory-based test shell component to validate the allocated memory before attempting to free it.
-v[ersion]	on page 5-20	Displays version information of the driver.
-verbose	on page 5-21	Displays command line information for all invoked compilation tools.

-@ *filename*

The `g-@ filename` switch specifies that the contents of the named file, which holds driver options, are to be read and placed directly after the `-@` switch on the command line.

The specified *`filename`* argument normally contains only valid options but may also contain source file names. Spaces, tabs, or newline characters can separate the driver options. Any line containing a `#` indicates the remainder of the line is a comment.

When the argument to this switch is a directory, any VIDL source files within the given directory are to be placed on the command line.

-accept-any-include-file

The `-accept-any-include-file` switch overrides the default behavior of the VIDL compiler by including (`#include`) other file types, such as `.H`, in addition to `.IDL` files.

By default, the VIDL compiler only `#include .IDL` files. The `-accept-any-include-file` requests the VIDL compiler to relax this restriction and include other file types, such as `.H` files.

-all-idl

The `-all-idl` (generate sources for all VIDLs) switch directs the compiler to generate interface header files and component shells for interfaces and components defined in any included files, as well as the main VIDL source file. By default, the VIDL compiler generates only interface header files and component shells for interfaces and components defined directly in the main VIDL source file.

-asm

The `-asm` (generate assembly shells) switch specifies assembly language shells are to be generated for any component defined directly in the VIDL file. Interface header files are also to be generated for each interface defined directly in the main VIDL file.

When neither `-asm` or `-c++` is specified, the compiler generates C language shells.

The `-asm` switch cannot be used in conjunction with `-c++` or `-hdr`.

-c++

The `-c++` (generate C++ shells) switch specifies C++ language shells are to be generated for any component defined directly in the VIDL file. Interface header files are also to be generated for each interface defined directly in the main VIDL file.

When neither `-asm` or `-c++` is specified, the compiler generates C language shells.

The `-c++` switch cannot be used in conjunction with `-asm` or `-hdr`.

The `-c++` switch is not supported for the ADSP-218x DSPs.

-copyright *filename*

The `-copyright` (specify copyright file) switch specifies the name of a file, which contains a copyright statement that is to be copied to the start of each generated source file.

-cppflags *flags*

The `-cppflags` (pass to C preprocessor) switch directs the VIDL compiler to pass *`flags`*, an option or a list of options, to the C preprocessor invoked via the VIDL front-end.

-D*macro*[=*definition*]

The `-D` (define macro) switch directs the compiler to define a macro. When the optional definition string is not included, the compiler defines the macro as the string `'1'`. If a definition is required to be a character string constant, then it must be surrounded by escaped double quotes. Note that the compiler processes all `-D` switches before any `-U` (undefine macro) switches on the command line.

Only simple macros can be defined this way—macros accepting arguments must be defined in the source files. A warning is generated when a predefined macro is redefined.

> This switch can be invoked with the **Global definitions** field located in the VisualDSP++ IDDE's **Project Options** dialog box, **VIDL** page selection.

-dryrun

The `-dryrun` switch direct the compiler to display the command lines of each of the processes the driver invokes without processing them.

-embedded

The `-embedded` switch suppresses the generation of factory methods (`<component>_Create`, `<component>_SizeOf`, `<component>_Destroy`) and the factory header file, `<component>_factory.h`, to streamline the use of components embedded within other components.

-generic

The `-generic` switch directs the compiler to generate C/C++ code that can be compiled using alternative compilers to those supplied with VisualDSP++.

Applications that only use one component will compile with Microsoft Visual C++ 6.0 or `gcc 3.2` with no warnings by adding `__GENERIC__` to the list of preprocessor definitions. For multi-component applications, the additional `/FORCE:MULTIPLE` switch is required to be passed to the Microsoft Visual C++ linker to demote `LNK2005` errors to `LNK4006` warnings, informing you that an interface IID has already been defined and that the second definition will be ignored. This is normal. For `gcc 3.2`, no additional linker options are required and no warnings are generated.

-harness

The `-harness` switch directs the compiler to generate a test program for the components defined directly in the main VIDL source file.

By default, the VIDL compiler does not overwrite an already existing test harness source file. If you wish the compiler to overwrite an existing test harness source file, you must also supply the `-overwrite` option.

-hdr

The `-hdr` switch specifies that only the interface header files are to be generated for each defined interface. Any component definitions are validated, but the component shells are not generated.

The `-hdr` switch cannot be used in conjunction with `-c++` or `-asm`.

-h[elp]

The `-h` or `-help` switch directs the compiler to display a list of switches, including a brief description of each switch, that the driver recognizes. This is the default if no other switches are given.

-I*directory* [{,|;} *directory*...]

The `-I` (include directory) switch directs the compiler to add the specified directories to the `#include` file search path. Multiple include directories can be given as a semicolon- or comma-separated list of directories searched in the order specified.

When multiple occurrences of this switch appear on the command line, they are searched in the order specified on the command line.

All directories specified with this switch are searched before the standard include directory is searched.

-M

The `-M` (generate make rules only) switch directs the compiler not to compile the source file but to send to standard output the rules suitable for the make utility, describing the dependencies of the generated files. The format of the make rules output by the compiler is:

```
object_file:include_file …
```

The `-M` switch cannot be used in conjunction with `-MM`.

-MM

The `-MM` (generate make rules and compile) switch is similar to `-M`. The difference is that the VIDL compiler does not halt compilation after preprocessing and proceeds to generate the interface header and component shell files.

The `-MM` switch cannot be used in conjunction with `-M`.

-lghtwt

The `-lghtwt` switch directs the VIDL compiler to generate components that have no instance data and minimal overhead. Memory for such components is allocated statically at compile time.

-mcd

The `-mcd` (generate multiple component shells) switch directs the compiler to accept more than one component definition in the main VIDL source file and to generate shells for each such component. The VIDL compiler only accepts one component definition in the main `.IDL` source file by default.

-no-adoc

The `-no-adoc` (no documentation) switch instructs the compiler not to generate the HTML documentation files from the auto-doc description blocks in the VIDL files. The auto-doc comments are still processed and validated.

-no-shell

The `-no-shell` switch directs the VIDL compiler to ignore the IDL defining the test shell components in the specified VIDL file.

The switch is only applicable when the VIDL contains statements defining a test shell component. By default, the VIDL Compiler generates code for both the actual and test shell components defined directly in the main VIDL source file.

-no-vla

The `-no-vla` (no variable length arrays) switch instructs the compiler not to generate the variable length arrays for conformant array parameters when a C component shell is being generated.

Variable length arrays are never generated for a C++ component shell.

-no-xml

The `-no-xml` (no XML output) switch instructs the compiler not to generate the `.XML` component manifest when a component shell is being generated.

-overwrite

The `-overwrite` switch directs the VIDL compiler to overwrite any existing test harness source file when the `-harness` option is specified. When this option is omitted, the compiler issues a warning that the existing test harness file will not be overwritten.

-path-[cpp|fe|pr|be] *path*

The `-path-`*`tool path`* (tool location) switch directs the compiler to use *`path`* as the location for the specified compilation tool. Respectively, the tools are the preprocessor, front-end, presentation, and back-end. Use this switch to override the default version of the tool, or that implied by the `-path-install` switch.

-path-def *path*

The `-path-def` switch directs the VIDL compiler to use the specified path instead of the default `vidl_driver.def` file, or that implied by the `-path-install` switch.

-path-html *directory*

The `-path-html` *`directory`* (`.HTML` files location) switch directs the compiler to use the specified directory as the location of the `.HTML` template files. The compiler uses the specified templates when generating the HTML documentation instead of those found in the default directory. This is useful when working with multiple versions of the tool set.

-path-install *directory*

The `-path-install` switch directs the compiler to use the specified directory as the base directory for all VIDL tools, include directories, and configuration files. For example, if `-path-install c:\myVIDL` is specified, then `vidlblkfn` (for example) looks for all VIDL compiler tools in the `C:\myVIDL\Blackfin\etc` directory.

-path-output *directory*

The `-path-output` *`directory`* (output location) switch directs the compiler to place all the generated files in the specified directory. This is useful when the directory containing source files is read-only, or there is insufficient space available to copy the generated files.

-path-temp *directory*

The `-path-temp` switch directs the VIDL compiler to use the specified directory instead of the default location for temporary files.

-proc *processorID*

The `-proc` *`processorID`* (compile for a specific processor) switch directs the VIDL compiler to generate component shells for the specified processor.

Blackfin Processor Switches

On Blackfin processors, the compiler accepted values for *processorID* are: AD6532, ADSP-BF531, ADSP-BF532, ADSP-BF533, ADSP-BF535, and ADSP-BF561. Compiling with any of these switches has the effect of modifying the VIDL to be compiled by passing the switch to the preprocessor. The generated C/C++/Assembler source files are not affected, and only include macro guards to test that macro __ADSPBLACKFIN__ is defined as 1. The processor identity is, however, passed on to the XML component manifest.

-proc AD6532	Directs the compiler to generate code suitable for the AD6532 processor.
-proc ADSP-BF531	Directs the compiler to generate code suitable for the ADSP-BF531 processor.
-proc ADSP-BF532	Directs the compiler to generate code suitable for the ADSP-BF532 (formerly ADSP-21532) processor.
-proc ADSP-BF533	Directs the compiler to generate code suitable for the ADSP-BF533 processor.
-proc ADSP-BF535	Directs the compiler to generate code suitable for the ADSP-BF535 (formerly ADSP-21535) processor.
-proc ADSP-BF561	Directs the compiler to generate code suitable for the ADSP-BF561 processor.

By default, `vidlblkfn` assumes that the ADSP-BF532 Blackfin processor is the target processor.

21xx Processor Switches

On 218x processors, the compiler accepted values of processorID are: ADSP-2181, ADSP-2183, ADSP-2184, ADSP-2185, ADSP-2186, ADSP-2187, ADSP-2188, and ADSP-2189. Compiling with any of these switches has the effect of modifying the VIDL to be compiled by passing the switch to the preprocessor. The generated C/C++/Assembler source files are not affected, and only include macro guards to test that macro `__ADSP218X__` is defined. The processor identity is, however, passed on to the XML component manifest.

`-proc ADSP-2181`	Directs the VIDL compiler to generate code suitable for the ADSP-2181 processor.
`-proc ADSP-2183`	Directs the VIDL compiler to generate code suitable for the ADSP-2183 processor.
`-proc ADSP-2184`	Directs the VIDL compiler to generate code suitable for the ADSP-2184 processor.
`-proc ADSP-2185`	Directs the VIDL compiler to generate code suitable for the ADSP-2185 processor.
`-proc ADSP-2186`	Directs the VIDL compiler to generate code suitable for the ADSP-2186 processor.
`-proc ADSP-2187`	Directs the VIDL compiler to generate code suitable for the ADSP-2187 processor.
`-proc ADSP-2188`	Directs the VIDL compiler to generate code suitable for the ADSP-2188 processor.
`-proc ADSP-2189`	Directs the VIDL compiler to generate code suitable for the ADSP-2189 processor.

On 219x processors, the compiler accepted values of processorID are: `ADSP-2191`, `ADSP-2192-12`, `ADSP-2195`, `ADSP-2196`, `ADSP-21990`, `ADSP-21991`, `ADSP-21992`, and `AD90747`. Compiling with any of these switches has the effect of modifying the VIDL to be compiled by passing the switch to the preprocessor. The generated C/C++/Assembler source files are not affected, and only include macro guards to test that macro `__ADSP219X__` is defined. The processor identity is, however, passed on to the XML component manifest.

`-proc ADSP-2191`	Directs the VIDL compiler to generate code suitable for the ADSP-2191 processor.
`-proc ADSP-2192`	Directs the VIDL compiler to generate code suitable for the ADSP-2192 processor.
`-proc ADSP-2192-12`	Directs the VIDL compiler to generate code suitable for the ADSP-2192-12 processor.
`-proc ADSP-2195`	Directs the VIDL compiler to generate code suitable for the ADSP-2195 processor.
`-proc ADSP-2196`	Directs the VIDL compiler to generate code suitable for the ADSP-2196 processor.
`-proc ADSP-21990`	Directs the VIDL compiler to generate code suitable for the ADSP-21990 processor.
`-proc ADSP-21991`	Directs the VIDL compiler to generate code suitable for the ADSP-21991 processor.
`-proc ADSP-21992`	Directs the VIDL compiler to generate code suitable for the ADSP-21992 processor.
`-proc AD90747`	Directs the VIDL compiler to generate code suitable for the AD90747 processor.

-save-temps

The `-save-temps` (save intermediate files) switch prevents any temporary files created by the driver or compiler from being deleted. When used in conjunction with `-M` or `-MM`, the dependency lists are redirected to the file *basename*(<*idl-file*>|<*infile*>)`.dep`.

-shell-only

The `-shell-only` switch directs the VIDL compiler to generate only the test shell component and suppresses generation of the actual component defined in the specified VIDL file.

The switch is only applicable when the VIDL contains statements defining a test shell component. By default, the VIDL Compiler generates code for both the actual and test shell components defined directly in the main VIDL source file.

-si-revision <revision>

The `-si-revision <revision>` switch directs the VIDL compiler to produce code (where applicable) that conforms to the given revision number. Format of `<revision>` is `<major>.<minor>` with minor <=255.

This switch is passed to the preprocessor; it has no further effect on the generated code.

-states-verbose-errors

The `-states-verbose-errors` switch directs the VIDL compiler to insert code to generate more verbose messages into the test shell component when reporting non-conformance of the component to the states strategy defined in the specified VIDL file.

By default, the VIDL Compiler inserts code to generate a minimal non-conformance message.

-trace

The `-trace` switch directs the compiler to generate debug code in component source files to record the entry and exit of each method.

-U*macro*

The `-U` (undefine macro) switch undefines the specified macro. The compiler processes all `-D` (define macro) switches on the command line before any `-U` switches.

The `-U`*`macro_name`* switch on a command line is equivalent to `#undef` *`macro_name`* in a source file.

A warning is generated when a predefined macro is undefined.

-umb-verbose

The `-umb-verbose` switch directs the VIDL compiler to generate User Modifiable Blocks which are more prominent in the generated code and containing explanatory comments.

By default, the VIDL Compiler generates minimal User Modifiable Blocks with no explanatory comments.

-validate-memory

The `-validate-memory` switch directs the VIDL compiler to generate code, within an `VCSE::IMemory`-based test shell component, to validate the allocated memory before attempting to free it.

By default, such generated test shell component code does not validate memory before attempting to free it.

-v[ersion]

The `-v` or `-version` directs the compiler to display the version number of the compiler driver.

-verbose

The `-verbose` switch directs the compiler to display the command lines of each of the compilation processes that the driver invokes.

Processing VIDL Files

Three categories of source files are associated with VCSE:

- Interface Definition Language files (.IDL) name and describe VCSE-conformant components and interfaces, as well as specify which of the available interfaces are supported by the components. The VIDL language is described in "VIDL Language Reference" on page 4-1.

- Standard header files (.H) give access to the VCSE system features, such as response values returned from component functions and interface methods, or macros facilitating interface member calls from C and assembly code.

  Use these headers in component implementations or in component-based applications. The VCSE standard headers are described in "Standard Files" on page 5-24.

- C++, C, and assembly source and header files (.CPP, .C, .ASM, .H) are generated by the VIDL compiler in response to a VIDL specification supplied as its input.

  These files contain specific details of the interfaces and components described in the VIDL file. These generated files also contain standard sections of code that either assist the component developer to create and debug a component or to enable the component to interoperate with applications and other components.

  The VIDL generated files are described in "Generated Source Files" on page 5-28.

File Organization

The general organization of the standard header files and the generated files follows the same principles.

File Names

Apart from the basic VCSE support headers, `vcse.h`, each standard and generated file is named according to the namespaces and the interface or component name with which it is associated.

- From left to right, the file name prefix consists of the name of each namespace (underscore separated, from outer to inner) in which the interface or component definition is located.
- The prefix is followed by the interface or component name, all separated from the prefix by a single underscore.
- For some files, such as method definitions, the name is followed by a suffix of a single underscore and a single word, which indicates its content.
- The file name preserves the letter case of the VIDL base file name. The file name extensions are: `.CPP` for C++ source files, `.C` for C source files, `.ASM` for assembly source files, `.H` for header files, and `.HTML`, `.HHC`, `.HHK` for HTML files.

Start-of-File Comments

Each standard file begins with a corporate Analog Devices copyright comment statement, the name of the file, and an indication of the processor family for which it is intended.

Each generated file begins with a comment providing the file name, a brief purpose description, the date and time of creation, and the version number of the VIDL compiler used to produce the file. Files containing no user-alterable sections have a short warning comment to this effect.

End-of-File Comments

All generated non-header files have a terminating comment that includes the file name.

Header Files Guards

Standard and generated header files have a conditional compilation construct to prevent multiple inclusion of the file contents.

The name of the preprocessor symbol acting as the guard is constructed as follows: two leading underscores; the complete file name (not converted to uppercase) up to the `.H` extension; and a tail of an underscore, capital `H`, followed by two underscores.

Language Identifications

Standard and generated header files can be included into C, C++, and assembler compilations. This means sections of the files must be excluded from preprocessing when their contents are not appropriate for the language being used. The preprocessor symbols used to control section inclusion and exclusion are `__cplusplus`, `_LANGUAGE_C`, and `_LANGUAGE_ASM`. The first two symbols are defined automatically by the C/C++ compiler driver when the user chooses C++ or C mode, and the other is defined by the assembler driver.

Standard Files

Three files give access to the fixed features of the VCSE system: `vcse.h`, `vcse_asm.h`, and `VCSE_IBase.h`. The first and second files are intended for inclusion into components and component clients written in C++, C, and assembly, either directly or at the end of an inclusion chain starting with a generated interface or component header. The second file also defines a set of assembler macros used by components implemented in assembly language. The third file, `VCSE_IBase.h`, is the interface header for the root interface `VCSE::IBase`.

The `vcse_asm.h` file is always target processor-specific. In general, `vcse.h` and `VCSE_IBase.h` may be target-specific in terms of the C/C++ basic types they use since these can be mapped to different hardware entities for the various targets. This convention is catered for by VisualDSP++'s organization of `include` directories, which does not expect header files to be shared across architectures.

Contents of vcse.h

The `vcse.h` file is the main standard header file used by VCSE interface header files and the generated component source files. The content of the standard VCSE header file is outlined as follows.

1. As described in "Language Identifications" on page 5-24, three preprocessor variables are used to distinguish between the different possible implementation languages. One and only one should be defined in each VCSE compilation. The `vcse.h` header verifies the inclusion of the appropriate preprocessor variable.

2. Some of the code generated by VCSE may use functions or macros from the ANSI C run-time library when it is generated in trace mode. When compiling C and C++ files, `vcse.h` ensures the appropriate standard header files are included.

3. Interfaces are represented as method tables. The `vcse.h` file defines a type and macros to enable these method tables to be defined. The type `VCSE_DELTA` assists with method table creation, and the macros `__INVOKE_VARARGS`, `__INVOKE_NOARGS`, and `__UPCAST` assist with invocation of the methods defined in an interface.

   The `__INVOKE_*` macros are not intended for direct use by a client or component developers. The header files generated for each interface definition include a macro for each method, specifically for use with interface pointers. Each such macro calls the appropriate `__INVOKE_*` support macro.

Even for assembly written components, the method tables are constructed in C, so there are no corresponding assembler structure declarations; although, equivalent assembler macros for accessing the method tables and method calls are provided.

Each method table entry consists of an instance of a `VCSE_DELTA` struct followed by a function pointer. The delta member is the offset to be added to an interface pointer to point to the component implementing the interface. Its value is either zero (for the first or only interface implemented by a component) or a small negative multiple of the size of a pointer. For example, on a byte-addressable architecture, the offset for the third interface implemented by a component is `-12`.

4. All interface functions are defined to return a value of a particular type, `MRESULT`. The `vcse.h` file defines this type for C and C++. For the assembler, the result type is assumed to be a short integer to fit in the standard function result register, as defined by the platform's run-time model.

   For C++ applications, `MRESULT` is defined inside the standard VCSE namespace. For C applications, it is given a prefix `VCSE_`, `VCSE_MRESULT`.

   The `vcse.h` header also defines the set of standard VCSE `MRESULT` codes. The codes are defined in "MRESULT Codes" on page B-2.

5. VCSE generates two main types of data structures from the VIDL specifications supplied by the user—interfaces and components. To reinforce the difference between the two data structures, `vcse.h` defines `interface` and `component` as synonyms for `struct`, allowing the generated code to use the synonyms appropriately.

6. Finally, `vcse.h` includes the header file `vcse_asm.h` to provide access to the macros and definitions used by the assembly language programmer.

Contents of vcse_asm.h

The `vcse_asm.h` file is the standard assembly language header file. It defines macros used by the generated assembly component source files. Where possible, the VIDL compiler generates the same assembly text for all the target processors. Processor-specific content is wrapped in macros defined in `vcse_asm.h`. The contents of the standard `vcse_asm.h` file are outlined as follows.

1. The header file first defines a set of helper macros used when constructing names for items, such as for the interface `iid`. The helper macros are not meant to be used directly by the assembler programmer but by other macros.

2. The header file then includes macro definitions for the code section in the generated code, function start and end, and function entry and exit.

3. To support the method-calling macros defined in the generated interface headers, the macros `__GET_METHOD` and `__INVOKE` are defined. Users are not expected to call `__INVOKE` directly; instead, to call it via the method invocation macros generated in the interface header files. In addition, the macro `__CHECK_VCSE_RESPONSE` is likely to be used by a client to verify the results returned by the methods.

4. In addition to the common macros, the header defines some platform-specific convenience macros. The VIDL compiler generated code does not use these macros, but the various method implementations provided by the component might. These macros are provided to facilitate the tasks of setting up stack frames and making function calls conforming to the C run-time model.

Contents of VCSE_IBase.h

The `VCSE_IBase.h` file is the interface header file for the VCSE base interface `VCSE::IBase`. It is suitable for inclusion into C++, C, or assembly files. `VCSE_IBase.h` contains:

1. The external declaration of the interface identifier variable that holds `IBase`'s unique identifier.

2. A `typedef` for a pointer to the `struct` type that implements the `IBase` interface as well as the definition of the `struct` type.

3. The definition of a macro `VCSE_IBase_GetInterface`, which is used for calling the `IBase` interface's sole member function from C and assembly source files. C++ clients use a normal method call to invoke the interface functions.

4. A `typedef` for `VCSE_IBase_methods`, which is the C equivalent of the C++ method table associated with the `VCSE::IBase` class.

No structure definitions for the interface appear in the assembly portion of the file since the assembly implementation of interfaces relies on the Analog Devices assemblers 'importing' the `typedef` names and `struct` layouts from the C portion of interface header files.

Generated Source Files

The VIDL compiler produces several header, source, and HTML and source files in response to the interface and component definitions found in the VIDL input presented by the user.

Table 5-4 through Table 5-6 on page 5-32 summarize and describe the compiler generated files. In addition to the notations in "File Names" on page 5-23, the following applies to all of the generated file names.

<NS>	Represents the namespace components.	<I>	Represents the interface name.
<C>	Represents the component name.		All other characters are literals.

Interface Definitions

The VIDL compiler generates the interface files for all interfaces that the specified interface directly or indirectly extends. The main file generated for an interface specification is the interface header file. Both the creator of a component implementing the interface and the client using the interface require this header file.

In addition to the interface header, the VIDL compiler normally produces a set of `.HTML` files, which combines information from the auto-doc comments and the VIDL specification, to document the interface and its use. The generated `.HTML` files are held in the `html` subdirectory. All the generated interface files, along with the corresponding `.IDL` file, are normally distributed to all users of the interface. A summary of files generated for each interface definition is found in Table 5-4.

Table 5-4. Interface Source Files

File Name	Description
`<NS>_<I>.h`	Contains definitions of: types that represent the interface as well as a pointer to the interface; macros that facilitate calling the methods of the interface; and types that represent the method table layout. Also contains the definition of the unique interface identifier. Any C++, C, or assembly client module calling methods of the interface includes this file. Any C++, C, or assembly component module implementing the interface or constructing a method table for the interface includes this file.
`html\<NS>_<I>.html`	Main `.HTML` file; displays the generated documentation for the interface. Also creates a frame to display a table of contents or an index.
`html\<NS>_<I>_BASE.html`	Provides comprehensive information on the interface. Displays on the right-hand side of the frame created by `html\<NS>_<I>.html`.

Table 5-4. Interface Source Files (Cont'd)

File Name	Description
`html\<NS>_<I>_TOC.html`	Triggers the creation of the table of contents, which is displayed on the left-hand side of the frame created by `html\<NS>_<I>.html`.
`html\<NS>_<I>_INDEX.html`	Triggers the creation of the automatically generated index, which is displayed on the left-hand side of the frame created by the file `html\<NS>_<I>.html`.
`html\<NS>_<I>_hhc.html`	Defines the table of contents. Displays on the left-hand side of the frame created by `html\<NS>_<I>.html`.
`html\<NS>_<I>_hhk.html`	Defines the automatically generated index. Displays on the left-hand side of the frame created by the file `html\<NS>_<I>.html`.

In addition to the specific files generated for each interface, a common set of files is also generated (shown in Table 5-5).

Table 5-5. Common Generated Documentation Files

File Name	Description
`html\vcsehtml.css`	Defines a Common Style Sheet, which is referenced by all of the generated `.HTML` files. Allows the appearance of the HTML text to be controlled.
`html\vcsehtml.js`	Specifies a common Java Script element. The file is referenced in each of the generated `.HTML` file.

Component Definitions

The VIDL compiler produces a set of C, C++, or assembly source files for each processed component specification. The generated file set provides a framework for the component implementation and is referred to as an implementation shell. It provides all the necessary generic code needed to conform to the VCSE component object model. The generated files provide the definitions of each method, leaving the developer to complete the generated shell by providing the implementation of each method. Normally, the component distribution includes all the generated interface files.

In addition to the implementation shell source files, the VIDL compiler produces a set of `.HTML` files, which combines information from the auto-doc comments and the VIDL input, to document the component and each interface supported by the component. The generated `.HTML` files are held in the `html` subdirectory. For more information about these files, see "Component Documentation Files" on page 5-35.

Only the factory header file, `<NS>_<C>_factory.h`, should be distributed with the compiled component; the other source files are implementation-only files and, normally, are not distributed.

C Based Components

The set of files that the VIDL compiler generates for each C-based component definition is summarized in Table 5-6. This table represents the full set of files that can be generated. Certain command line switches will reduce the number of files generated.

Table 5-6. C Component Source Files

File Name	Description
`<NS>_<C>_factory.h`	Contains the external declarations of the non-interface method functions of the component. Any C++, C, or assembly module that creates, destroys, or queries the size of an instance of the component includes this file.
`<NS>_<C>.h`	Contains the layout (in C++ and C) of the struct that implements the component, including sections for the implementor to add private members and other component-related declarations. Used by all C++ and C component modules or imported into all assembly component modules that require to access the members of the component instance struct.
`<NS>_<C>.c`	Contains C definitions of the component's factory functions and the `GetInterface` and `NonDelegatingGetInterface` methods. The component developer can add custom code to the `Create` and `Destroy` function definitions to control the allocation and initialization of the component's instance data.
`<NS>_<C>_test.c`	Contains a C-based test harness program, which creates the component and calls each of the defined interfaces and then destroys the component. This file is only generated when the `-harness` switch is supplied.
`<NS>_<C>_methods.c`	Contains C function definitions for the remaining interface methods implemented by this component. Also contains the definition and static initialization of the component's method tables. The component developer adds custom code to the function definitions in order to implement the required functionality.

C++ Based Components

The set of files that the VIDL compiler generates for each C++ based component definition is summarized in Table 5-7. This table represents the full set of files that can be generated. Certain command line switches will reduce the number of files generated.

Table 5-7. C++ Component Source Files

File Name	Description
`<NS>_<C>_factory.h`	Contains the external declarations of the non-interface method functions of the component. Any C++, C, or assembly module that creates, destroys, or queries the size of an instance of the component includes this file.
`<NS>_<C>.h`	Contains the layout in C++ and C of the struct that implements the component, including sections for the implementor to add private members and other component-related declarations. Used by all C++ and C component modules or imported into all assembly component modules that require to access the members of the component instance struct
`<NS>_<C>.cpp`	Contains C++ definitions of the class management functions associated with the component, such as a class constructor, the factory functions, operators `new` and `xdelete`, the `GetInterface` and `NonDelegatingGetInterface` methods. Component developer adds custom code to the `Create` and `Destroy` function definitions to control the allocation and initialization of the component's instance data.
`<NS>_<C>_test.cpp`	Contains a C++ based test harness program, which creates the component and calls each of the defined interfaces and then destroys the component. This file is only generated when the `-harness` switch is supplied.

Assembly Based Components

The set of files that the VIDL compiler generates for each assembly based component definition is shown in Table 5-8. This table represents the full set of files that can be generated. Certain command line switches will reduce the number of files generated.

Table 5-8. Assembly Component Source Files

File Name	Description
`<NS>_<C>_factory.h`	Contains the external declarations of the non-interface method functions of the component. Any C++, C, or assembly module that creates, destroys, or queries the size of an instance of the component includes this file.
`<NS>_<C>.h`	Contains the layout in C++ and C of the struct that implements the component, including sections for the implementor to add private members and other component-related declarations. Used by all assembly component modules or included into all C++ and C component modules that require to access the members of the component instance struct.
`<NS>_<C>.c`	Contains C definitions of the component's factory functions and the `GetInterface` and `NonDelegatingGetInterface` methods. Also contains the definition and static initialization of the component's method tables. For an assembly component, the factory functions and the `GetInterface` method are generated in C. Hence, this file must be compiled with C and included in the set of component object files.
`<NS>_<C>_methods_asm.asm`	Contains assembly function definitions for the remaining interface methods implemented by this component. The component developer adds custom code to the function definitions in order to implement the required functionality. A standard set of assembly macros is provided for accessing parameters and elements of the component's instance data.
`<NS>_<C>_test.c`	Contains a C based test harness program, which creates the component and calls each of the defined interfaces and then destroys the component. This file is only generated when the `-harness` switch is supplied.

Component Documentation Files

For each component processed in the VIDL input file, the VIDL compiler produces a set of .HTML files to combine information from the auto-doc comments and the VIDL input. The generated .HTML files are held in the html subdirectory. In addition to the documentation files for the component, all the documentation files for each supported interface are integrated with the component's documentation.

The .HTML files generated for each component are shown Table 5-9.

Table 5-9. Component Specific Documentation Files

File Name	Description
html\<NS>_<C>.html	The main .HTML file; displays the generated documentation for the component. In addition, the file creates a frame to display a table of contents or an index.
html\<NS>_<C>_BASE.html	Describes the component in detail. Displays on the right-hand side of the frame created by the file html\<NS>_<C>.html.
html\<NS>_<C>_TOC.html	Triggers the creation of the table of contents, which is displayed on the left-hand side of the frame created by the file html\<NS>_<C>.html.
html\<NS>_<C>_INDEX.html	The .HTML file, which triggers the creation of the automatically generated index, which is displayed on the left-hand side of the frame created by the file html\<NS>_<C>.html.
html\<NS>_<C>_hhc.html	Defines the table of contents. Displays on the left-hand side of the frame created by the file html\<NS>_<C>.html.
html\<NS>_<C>_hhk.html	Defines the automatically generated index. Displays on the left-hand side of the frame created by the file html\<NS>_<C>.html.

In addition to the specific files generated for the component, a common set of files is also generated, as shown in Table 5-5 on page 5-30.

Component Manifest File

For each component specified in the VIDL input file, the VIDL compiler produces an XML based manifest file. Use this file to control the packaging wizard when the component is being packaged for distribution. The name of the packaging file is `<NS>_<C>.xml`. The packaging wizard combines the contents of the `.XML` file with information derived from the wizard steps to complete a component package.

Test Shell Files

The files generated by the VIDL compiler when the test shell syntax is present are very similar to those generated for the actual component, except that no Assembler files are generated. The names of the files are listed in Table 5-10.

Table 5-10. Test Shell Files

Type	Actual Component	Test Shell Component
Component header	ADI_C1.h	ADI_C1_VCSETEST.h
Component Management		
C	ADI_C1.c	ADI_C1_VCSETEST.c
C++	ADI_C1.cpp	ADI_C1_VCSETEST.cpp
Component Methods		
C	ADI_C1_methods.c	ADI_C1_VCSETEST_methods.c
C++	ADI_C1_methods.cpp	ADI_C1_VCSETEST_methods.cpp
ASM	ADI_C1_methods_asm.asm	ADI_C1_VCSETEST_methods.c
Component Factory Header	ADI_C1_factory.h	ADI_C1_VCSETEST_factory.h

6 VCSE RULES AND GUIDELINES

Read this chapter if you develop, deploy, or use VCSE components. The chapter documents the rules and best programming practices associated with the software components' successful development and successful inclusion into DSP applications.

VCSE provides a model or framework to aid the development and use of software components in DSP applications running on Analog Devices DSP processors. Two major aims of VCSE are the promotion of software interoperability and reuse, in a language-neutral way. Although the VCSE model ensures these aims *can* be met, it cannot guarantee that they always *are* met for any particular component or application, especially since assembly is one of the supported languages. For this reason, the model and tools support must be supplemented with rules and guidelines to obtain the maximum benefit when using components.

The rules and guidelines cover two broad areas, although the two areas sometimes overlap:

- Programming, see "Programming" on page 6-6
- Packaging, see "Packaging" on page 6-14

Issues concerning the correct operation of a component, considered in isolation, come under *programming*; while issues concerning a component's inclusion in an application that may use other components come under *packaging*.

Paragraphs labeled '**Rule**' describe actions or practices that are mandatory; applications may fail to build or run properly if they, or some components they include, fail to obey a rule.

Paragraphs labeled '**Guideline**' describe actions or practices that we strongly recommend you to follow. Applications may not fail to build or run if guidelines are not heeded, but they may be harder to debug or to deploy.

Components described as *algorithms* are those that implement the standard interface `VCSE::IAlgorithm` or an interface derived from it. Some rules and guidelines differ according to whether or not a component they apply to is an algorithm.

The rules and guidelines are described as being specific to algorithm or non-algorithm components. Where a component implements multiple interfaces that define a mixture of algorithms and non-algorithms, the rules and guidelines apply to the parts of the component that implement the algorithm or non-algorithm interfaces, respectively. Thus, a client using only the algorithm interfaces offered by the component can consider the component (as a whole) to be an algorithm component even though it contains (unused) code that may break some of the 'algorithm' rules.

Summary

The following tables summarize the presented rules and guidelines. Those that are common to the development and use of all components are listed first, followed by those rules and guidelines that apply only to algorithms, and finally those that apply only to non-algorithms.

- Table 6-1, "Common Component Rules" on page 6-3
- Table 6-2, "Common Component Guidelines" on page 6-4
- Table 6-3, "Algorithm Component Rules" on page 6-5

- Table 6-4, "Algorithm Component Guidelines" on page 6-5
- Table 6-5, "Non-algorithm Component Rules" on page 6-5

There are no guidelines for non-algorithm components.

Following the tables are sections providing a detailed description of each rule or guideline.

Table 6-1. Common Component Rules

Rule	Description
Programming	For a component, use the interface pointer supplied to its `Create` factory function (parameter `ienvp`) to obtain an interface pointer to a memory allocator; use this interface for all memory allocations. For more information, see "Resource Allocation" on page 6-6.
Programming	For a client, supply an interface pointer obtained from a component implementing an appropriate memory allocator to the `Create` function when instantiating a component. For more information, see "Resource Allocation" on page 6-6.
Programming	Client-component interactions must follow C run-time model specifications for the target processor. For more information, see "Registers and Stack" on page 6-9.
Programming	The documentation for every component that requires a memory allocation interface other than `VCSE::IMemory` must include or refer to a detailed description of the interface. For more information, see "Resource Allocation" on page 6-6.
Programming	Document self-modifying components as only sequentially reusable. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Programming	Document components saving data in fixed memory locations as only sequentially reusable. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Packaging	Use your company tag when naming files, globally visible labels, and LDF's sections and variables to avoid name clashes. For more information, see "Name Clashes" on page 6-14.
Packaging	Document your component's memory characteristics. For more information, see "Memory" on page 6-16.

Table 6-1. Common Component Rules (Cont'd)

Rule	Description
Packaging	Document your component's processing cycle characteristics. For more information, see "Processing" on page 6-17.
Packaging	Document all the required non-memory resources for your component. For more information, see "Non-memory Resource Requirements" on page 6-17.

Table 6-2. Common Component Guidelines

Guideline	Description
Programming	For a component, define an interface (VIDL) for a non-memory resource allocation. For a client, implement that interface in conjunction with the appropriate memory allocator. For more information, see "Non-memory Resource Requirements" on page 6-17.
Programming	Clients and components should follow C run-time model specifications for the target processor. For more information, see "Registers and Stack" on page 6-9.
Programming	Components should use the standard memory allocation interface `VCSE::IMemory` where possible. For more information, see "Resource Allocation" on page 6-6.
Programming	For assembly written components, use the `#include VCSE.h` macros to set up stack frames and refer to outgoing function call arguments. For more information, see "Registers and Stack" on page 6-9.
Programming	Avoid self-modifying code in component specifications. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Programming	Avoid fixed location data variables in component code. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Packaging	Use the linker's data elimination features for applications that employ assembly written components. For more information, see "Code and Data Elimination" on page 6-18.
Packaging	Ensure that your component objects are usable in various addressing models. For more information, see "Addressing Models" on page 6-18.

Table 6-3. Algorithm Component Rules

Rule	Description
Programming	Do not modify the interrupt controls and structures. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Programming	Document any algorithm components whose methods rely on specific configurations or performance characteristics of the interrupt system. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Programming	Document your algorithm component's re-entrancy capabilities. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Programming	Do not switch processor modes. For more information, see "Processor Modes" on page 6-13.
Programming	Document the algorithm component's requirement to be in a specific processor mode. For more information, see "Processor Modes" on page 6-13.
Programming	Do not access any core peripherals. For more information, see "Core Peripherals" on page 6-14.
Programming	Do not access code or data at absolute memory addresses. For more information, see "Address Clashes" on page 6-15.

Table 6-4. Algorithm Component Guidelines

Guideline	Description
Programming	Design your algorithm components to provide the most re-entrancy capabilities. For more information, see "Interrupt System and Re-entrancy" on page 6-10.

Table 6-5. Non-algorithm Component Rules

Rule	Description
Programming	Document all the processor's interrupt system operations and alterations. For more information, see "Interrupt System and Re-entrancy" on page 6-10.
Programming	Restore the processor's original mode once a method's execution is completed. For more information, see "Processor Modes" on page 6-13.

Table 6-5. Non-algorithm Component Rules (Cont'd)

Rule	Description
Programming	Document how the core peripherals are accessed by your component. For more information, see "Core Peripherals" on page 6-14.
Programming	Do not access code or data at absolute addresses, except memory-mapped registers. For more information, see "Address Clashes" on page 6-15.

Programming

The rules and guidelines defined in this section cover two major aspects of embedded applications design—resource usage, including memory allocation; and processor usage, including the interrupt system. The objective is to describe how a component and its client should conduct themselves in order for the component to obtain the resources and environment it needs to function, and the client to obtain the results and services envisaged when the application is planned.

Resource Allocation

All VCSE components require the allocation of at least one resource—an area of memory to hold the data associated with each instance of the component created by a client. Each instance may also require additional memory (working storage) and access to other resources, such as an I/O peripheral or a hardware timer.

The `Create` function called by a client to create a new component instance has two parameters associated with resource allocation: an interface pointer and a token. The interface pointer is obtained from some other component implementing the memory allocation interface appropriate to the component being instantiated. This may be the `VCSE::IMemory` standard interface, or it may be some other interface, as described in the component's documentation. The code initially generated by the VIDL compiler assumes that it is the `VCSE::IMemory` interface, but the component developer may alter that code to use a different allocator. An arbitrary token value will be passed as an argument to the `Allocate` and `Free` methods of the `IMemory` instance in code generated by the VIDL compiler. User specified allocators may utilize or ignore the token as desired.

See "Standard Interfaces" on page 3-1 for more information about the `VCSE::IMemory` interface and component instantiation. It is possible to pass `NULL` for the interface pointer and use a different mechanism for allocating memory when instantiating a component, but this is intended only as an aid during initial component development and as a method of bootstrapping memory allocator components.

Rule: Every component, with the exception of any that implements a memory allocator interface, must use the interface pointer passed to its `Create` function to obtain the allocator interface and to use that interface to satisfy all its memory requirements.

Rule: Every client must supply an interface pointer obtained from a component implementing a suitable memory allocation interface to the `Create` function of every component that the client instantiates, with the exception of components that themselves implement memory allocators. Details of the allocator interface that a component requires are referenced in the component's documentation.

The component that supplies the interface pointer used as an argument to a `Create` call can implement other interfaces besides its memory allocator. One way to organize resource allocation for a particular application is to develop a composite component. The interface pointer passed to the `Create` functions of all instantiated components acts as a gateway to all of the allocators.

Rule: If a component requires its clients to provide an instance of a memory allocator other than `VCSE::IMemory`, then its documentation must contain or refer to a full description of that alternative memory allocation interface.

Guideline: Component developers should use `VCSE::IMemory` as their memory allocation interface whenever possible since clients are likely to already have a component that implements the memory allocator. If this is impossible, then consider using some other already-published interface or providing a component that implements your custom interface.

Guideline: If your component requires the allocation of a resource other than memory, either use a published interface or, if necessary, publish a new interface definition. The interface is to be used by the component for allocation and freeing of the resource. The implementation of such additional resource interfaces should be accessed via the same component that provides the memory allocator interface to the component.

Applications may centralize the management of resources with a monitor component aggregating the interfaces provided by separate resource allocation components. Clients of the monitor may access its resource interfaces by calling the `GetInterface` method of its `IBase` interface.

See "Packaging" on page 6-14 for rules concerning documentation of a component's resource usage.

Processor Usage

Registers and Stack

A VCSE component can be implemented in C++, C, or assembly and must be usable by a client application written in any of these languages. To achieve this, the points of interaction between a client and component, the `Create` and `Destroy` functions, and the interface methods must adhere to the C run-time model for the targeted processor. The major points covered by a run-time model are:

Register usage	Specifies which registers are available as scratch registers, which must be preserved across a function call, and which have special usage.
Function call	Specifies how arguments and control are passed to a function and how results and control are returned.
Stack maintenance	Specifies the alignment that stack pointer registers must maintain and the details of any areas that must be created for parameter passing and other purposes.
Data size/alignment	Specifies the memory sizes and alignment requirements of the fundamental data types.

The C run-time model is described in the *VisualDSP++ 3.x C/C++ Compiler and Library Manual* supplied with VisualDSP++ for each target architecture family.

Rule: All interactions between a client and a component must obey the target processor's C run-time model.

Processing that is strictly internal to an application or component code does not need to conform to the run-time model. For example, the implementation of an interface method can invoke support functions that accept more register based arguments than the model specifies or that return multiple results in multiple registers. As long as no effects of this are discernible once control returns to the code that invokes the interface method, this is acceptable. However, we recommend that all code follow the platform's run-time model since non-standard code is a common source of hard-to-find bugs and also reduces flexibility (in terms of code replacement or reuse).

Guideline: All client and component code should adhere to the C run-time model of the target platform.

On some platforms, a common problem found in assembly code written to follow the C run-time model is failure to provide the proper on-stack storage area for outgoing arguments. For instance, on Blackfin processors, even if the function being called takes no arguments, the caller must provide a three-word area at the top of the stack. The called function is at liberty to use this area as temporary storage. Failure to provide the area may result in the caller's own temporary storage (perhaps containing a return address or saved registers) being overwritten.

Guideline: In assembly code, use the macros made available via `#include VCSE.h`. The macros can help to set up proper stack frames and correctly refer to outgoing function call arguments.

Assembly programmers need to understand the C run-time model.

Interrupt System and Re-entrancy

An 'ideal' VCSE component has the following characteristics.

- Multiple instances of the component can coexist
- The instance structure can address or contain all the modifiable data of one component instance
- The component's methods require no exclusive access to any system resource, apart from the memory for each instance structure

- The component's methods do not require to run in any particular execution mode (supervisor or system mode)
- The component's methods do not require to be non-interruptible, or fail if interrupt processing overhead exceeds some limit

Such a component is likely to be usable in all situations, from a simple single thread of control in a standalone application through various flavors of cooperative and preemptive multitasking systems. Not many components can achieve such ubiquity; therefore, adhering to programming and documentation rules for the components can help a potential user to judge whether a particular component fits into their system.

The general principle is components that are algorithms must not alter the execution environment to suit their needs, but must document the environment they require. Components implementing peripheral or resource managers may change the execution environment, but must document the changes and the circumstances in which they occur.

Rule: No algorithm component may modify the interrupt controls and structures in any way.

Rule: Any algorithm component with methods relying on specific configurations or performance characteristics of the interrupt system must document their requirements. Examples include methods that must execute with interrupts disabled, or that fail to work to specification if interrupt processing overhead exceeds a certain threshold.

The reentrancy capabilities, from least restrictive to most restrictive, that a component might posses are:

1. Interleaved execution of methods of the same component instance is possible
2. Interleaved execution of methods of different instances of the same component is possible

3. Execution of a component instance is preemptable, but not in favor of another instance of the component, meaning the component as a whole is only serially reusable

4. Execution of a component instance is not preemptable

5. Only one instance of the component is allowed to be created and executed

Rule: The documentation for each algorithm component must state its reentrancy capability, either for the component as a whole or for each method.

Guideline: When developing an algorithm component, try to achieve at least point 2 (found on the previous page in the list of reentrancy capabilities) to allow the most flexibility in the application design.

Non-algorithm components, particularly peripheral handlers, may need to install interrupt handlers and modify interrupt control registers in order to function. The component's documentation must state what changes the component makes to the target processor's interrupt system, when, and under what circumstances.

Rule: Any use or modification of the processor's interrupt system must be fully stated in the component documentation.

In a VCSE component, only a single copy of the code—the `Create` and `Destroy` functions and the methods—exists. In fact, this code works on different sets of data (the instance data), allowing a component to be timesliced or interleaved between different incarnations of itself. Obviously, if a method modifies its code to suit the instance that is executing at that point, it loses the ability to execute other instances of itself at the same time.

Rule: Components that employ self-modifying code must classify themselves in their documentation as only serially reusable.

Guideline: If at all possible, components should not use self-modifying code since it restricts the application designer's options in deploying the component.

In a similar manner, components saving data in fixed memory locations, where 'fixed' means not allocated by a memory allocator, are not generally preemptable and must be documented as such.

Rule: Components saving data in fixed memory locations must classify themselves in their documentation as only serially reusable.

Guideline: If at all possible, components should not use fixed location data variables since it restricts the application designer's options in deploying the component.

Processor Modes

Some of Analog Devices DSPs feature processor modes in which different subsets of the total processor capabilities are available. Usually there is a user mode in which all of the computational and most of the control capabilities are available, and a system or supervisor mode in which the remaining control aspects are operative. The overall decision as to which processor modes should be in effect at each point is left to the application designer. Non-algorithm components may need to switch modes at certain points in their processing but are required to restore the original mode before returning to the application.

Rule: Algorithm components must not switch processor modes.

Rule: Algorithm components with methods that require a specific processor mode must document this requirement.

Rule: Non-algorithm components may alter the processor mode during execution of a method, but must restore the original mode before returning to the caller.

Core Peripherals

Algorithm components are presumed to be structured as 'pure' algorithms—they perform some computation upon data supplied as arguments and return results in specified memory locations, as described in "IAlgorithm Interface" on page 3-14. Thus, algorithms should have no reason to use the processor's I/O peripherals to obtain data or to output results. For producing debugging or tracing output, components should use the `IError` mechanism described in "IError Interface" on page 3-37.

Rule: Algorithm components must not access the core peripherals.

Non-algorithm components may need to use the core peripherals; indeed their function may be to manage access to one or more of the peripherals. The only requirement in this circumstance is that the component documentation must list, on a method-by-method basis, which peripherals are used and summarize how or why they are used.

Rule: The documentation for a component that accesses core peripherals must describe how the peripherals are used.

Packaging

Name Clashes

There is no requirement that the code and data comprising a VCSE component should be contained entirely within the source files generated by the VIDL compiler—you, the component developer, are free to call functions and reference data defined in other files. (Of course, the corresponding object (`.DOJ`) files must be added to the component library (`.DLB`) file that gets distributed for inclusion into client applications.)

To avoid the possibility of name clashes with other developers' components or with clients' code, there is a simple naming rule: all externally visible names created by a component developer must use the developer's company tag. The VIDL compiler takes care of the names of methods, types, enumerations, and structures defined in properly specified .IDL files:

- Developer defined filenames, C function and data variable names, and names in Linker Definition Files (.LDF) must have a prefix consisting of the company tag followed by an underscore.
- Externally visible C++ function, class, and data variable names must be defined within an outer namespace whose identifier is the company tag; further inner namespaces are acceptable.
- Assembler global names must use a prefix consisting of an underscore, the company tag, and another underscore.

It is your responsibility to ensure inclusion of any two components into the same application will not result in name result in name clashes. The possibility of name clashes within the company namespace can be reduced by ensuring that all names incorporate any embedded namespace and the name of the component using the style of names generated by the VIDL compiler.

Rule: You must use your company tag when naming the files, globally visible code and data names, and LDF names (sections and variables) to ensure there are no clashes of global names between their separate components.

Address Clashes

As a developer of a VCSE component, you have no control over the allocation of memory addresses to any of the component's code or data. The designers of the applications into which the component is included, along with the system linker, control the layout of code and data. Accordingly, every VCSE component must be link-time relocatable; that is, apart from references to memory-mapped registers, the code and initialized data must not refer to absolute (literal) addresses, but must refer to relocatable labels.

Rule: No algorithm component may access code or data at absolute memory addresses.

Rule: No non-algorithm component may access code or data at absolute memory addresses, apart from accesses to memory-mapped registers.

Memory and Processing Characteristics

If you are considering using a third party component in your application to obtain some part of the application's functionality, you need to know what effect that component may have on your system's memory and MIPS budgets. Does it fit? Is it fast enough? In the case of multichannel data streams, how many channels is the application able to support using this component? To aid these calculations, a VCSE component must have its memory and processing characteristics documented and available for evaluation.

Memory

The minimum documentation for a component's memory characteristics consists of:

1. The total size of code and initialized data that gets linked into an application using the component. Supply separate totals for architectures that differentiate between program and data memory.

2. Typical and maximum figures for the additional data memory associated with one instance of the component, including the size of the instance structure itself and any other working storage. If the memory requirement is dependent on the values of parameters that the client supplies, use the names that the parameters have in the interface definition (VIDL file).

3. A breakdown of the totals from 1 and 2, in terms of the different memory attributes that the component's allocator interface defines. For example, if the standard allocator `VCSE::IMemory` is used, then a breakdown by `MemoryType` and `MemoryLifetime` is appropriate.

Rule: A component's documentation must include at least the minimum memory usage characteristics.

Processing

The minimum documentation for a component's processing characteristics is a list of typical and maximum cycle counts for the execution of the `Create` and `Destroy` functions and each of the component's methods. The counts must be obtained from hardware or a cycle-accurate simulator, and the source of the counts must be stated.

If a cycle count is dependent on the values of parameters which the client supplies, the documentation must quote the names the parameters have in the interface definition (`.IDL` file). If the cycle counts depend on the type of memory allocated to any of the component's code, static data, or instance data, the documentation must specify which type is required for each critical element in order to achieve the best performance.

Rule: A component's documentation must include at least the minimum processing cycle characteristics.

Non-memory Resource Requirements

If a component uses, or requires, some system resource other than memory, you must document these requirements. If a specific peripheral is required (for example, a particular DMA channel), document this requirement as well.

Rule: A component's documentation must list the non-memory resources it needs.

Code and Data Elimination

The VisualDSP++ linker performs exclusion of functions and data areas from builds when it can detect the code or data in question is unused. This feature is available automatically for C/C++ programmers. Assembler programmers should define a label consisting of a period (`.`), the name of the function or variable, another period, and the letters "`end`" (`.`*`function-name`*`.end`) immediately at the end of each function or data item that may be omitted if never referred to. The macros `__STARTFUNC` and `__ENDFUNC` available via `#include VCSE.h` generate the appropriate labels for the start and end of a function.

Guideline: For assembly written components, use the VisualDSP++ linker's features to enable exclusion of potentially not-needed code and data, such as debugging code, from applications that include the assembly components.

Addressing Models

The compilation systems of some DSP platforms allow a choice of addressing model—applications with limited memory requirements can be built in a way that minimizes code size by assuming all addresses are in some way 'short' or 'near'. Other systems support various types of memory and allow some variability in the allocation of code and data to each memory type.

You should attempt to develop your components, which are delivered as object code (not user-modifiable source code) as universal as possible. If universality is not possible or imposes too great a performance or size overhead for some class of applications, consider providing alternative versions of your components, compiled to use the different addressing models.

Guideline: Ensure your components, as supplied in object file format, are usable in as wide a class of addressing models as possible.

A VCSE ASSEMBLER MACROS

This appendix lists and describes the VCSE assembly macros available to to developers of assembler components and applications by the `#include <vcse.h>` statement.

The information is presented as follows.

- "General Overview of Macro Definitions" on page A-1
- "Implementation of Macros on Blackfin Processors" on page A-9
- "Implementation of Macros on ADSP-21xx DSPs" on page A-16

General Overview of Macro Definitions

This section presents a functional summary of each of the macros available. Processor-specific information is reserved for the following sections, where a more detailed description is given.

In some cases, the processor-specific implementation of a macro may differ from that described in this section. Please refer to the section that is relevant to your target DSP family for further information.

Method Result Macros

Macros provided for constructing method result values and testing the result values returned from method calls are listed as follows.

VCSE_MRESULT

Expands into the appropriate data definition directive when defining a memory location to hold a method result.

MR_ICONSTRUCT(F,I)

Constructs a method result value literal.

`F`	Determines whether result code denotes a failure code (`F=1`) or otherwise (`F=0`).
`I`	Specific failure or warning value.

MR_FAILURE(mr)

Checks the returned method result for failure status.

`mr`	Register containing method result value.

MR_SUCCESS(mr)

Checks the returned method result for success status.

`mr`	Register containing method result value.

__CHECK_VCSE_RESPONSE(handler)

Checks the status of the returned method result against `MR_OK` and calls the `handler` function if different, passing the result code as the first parameter.

`handler`	Symbol of handler function.

Accessing Factory Functions

Every VCSE component has three *factory functions*, which client applications use to create and destroy instances of the component and to obtain an indication of the size of a component's per-instance data structure.

__CREATOR(C)

Forms the symbol name for component `C`'s `Create` factory function.

`C`	Fully qualified component name.

__DESTROYER(C)

Forms the symbol name for component `C`'s `Destroy` factory function.

`C`	Fully qualified component name.

__SIZEOF(C)

Forms the symbol name for component `C`'s `SizeOf` factory function.

`C`	Fully qualified component name.

Invoking Interface Methods

The usual approach for invoking an interface method is to use the macro the VIDL compiler generates for it in the interface header file. For example, method `Filter` in an interface `ADI::FILTERS::IFir` would have a macro called `ADI_FILTERS_IFir_Filter(P)` defined in the interface header file. Alternatively, the following constituent elements of the above macro call can be used separately.

`P`	Register holding pointer to interface.

__INVOKE(P,T,M)

Invokes an interface method M for the interface of type T. Assumes that the method's user arguments are already set up. Uses __GET_METHOD(P,T,M) defined in the next section.

P	Register holding pointer to interface.
T	Fully qualified interface name.
M	Method name.

__GET_METHOD(P,T,M)

Calculates the pointer to the method's code and its first argument.

P	Register holding pointer to interface.
T	Fully qualified interface name.
M	Method name.

Function Writing Macros

The definition of a function in assembly, especially one that follows the C run-time model, requires the use of certain directives and instruction sequences. The directives are concerned with making the function's name, size, and visibility available in the generated object file; the instruction sequences are required for setting up stack frames, saving and restoring preserved registers, and returning function results. The following macros are available to help with these tasks.

__STARTFUNC(Name,Visibility)

Generates the assembler directives to mark the start of an assembly written function.

`Name`	Symbol name of the function. Remember to include a leading underscore if the function is called from C or C++ code.
`Visibility`	Determines whether the function has global (`Visibility=__GLOBAL`) or local scope (`Visibility=__LOCAL`).

__ENDFUNC(Name)

Generates the assembler directives to mark the end of an assembly written function.

`Name`	Symbol name of the function.

__LINK(N)

Generates a new stack frame by pushing the relevant registers on to the stack and reserving enough space for local variables. This macro is required if the function is a non-leaf function. It should be used in conjunction with `__EXIT` or `__RETURN(Value)`.

`N`	Stack space (words) required for local variables.

__PUSH(Reg)

Pushes the named register on to the run-time stack.

`Reg`	Valid register name.

__POP(Reg)

Pops the run-time stack, placing the top value in to the named register.

`Reg`	Valid register name.

__ALLOCSTACK(N)

Allocates space on the run-time stack.

N	Stack space required.

__FREESTACK(N)

Frees space on the run-time stack.

N	Stack space required.

__arg0 to __arg9

Where the DSP architecture and instruction set allow, the stack locations for outgoing arguments can be directly referenced using these macros.

__STORE_ARG(n,Reg)

Where the DSP architecture and instruction set disallow the implementation of the `__argN` macros, an alternative macro is provided. Note that use of `__STORE_ARG(n,Reg)` may be less efficient than direct methods.

N	Argument index.
Reg	Valid register name containing value to be stored.

__EXIT

Generates code required to exit from a non-leaf function. The macro restores the registers pushed on the stack by `__LINK(N)`. No value is returned.

__LEAF_EXIT

Generates code required to exit from a leaf function.

__RETURN(Value)

Generates code required to exit from a non-leaf function and returns `Value`. The prime use of the macro is to return method result values.

`Value`	Valid value that can be used by the mechanism by which values are returned from a function.

__LEAF_RETURN(Value)

Generates code required to exit from a leaf function and returns `Value`. The prime use of the macro is to return method result values.

`Value`	Valid value that can be used in the mechanism by which values are returned from a function.

Miscellaneous

__LA(R,V)

Loads the register `R` with the address of variable `V`.

`R`	Valid register that can be assigned the address of a variable.
`V`	Variable name.

__VCSE_ASM_TRACE(A1,A2)

Calls to this macro are generated by the VIDL compiler when you request tracing code to be placed at the start and end of method bodies, but it may be of more general use. It concatenates two string literal arguments, `A1` and `A2`, and calls a small function in the C run-time library to write the result to `stdout`.

`A1`	First string literal.
`A2`	Second string literal.

__VCSE_PRINT_VAR(A1,A2,V)

This is another macro used by VCSE generated tracing code. It concatenates two string literal arguments, `A1` and `A2`, appends a carriage control and a line feed, and passes the result and the value `V` into a call of a simplified `printf`-like function.

`A1`	First string literal.
`A2`	Second string literal.
`V`	Value to be output.

Implementation of Macros on Blackfin Processors

C Run-Time Model

The macros provided within `vcse_asm.h` assume that the C run-time model is implemented, which is always the case for the assembly implementation of interface methods. The macros, therefore, make use of certain reserved registers, as described in the *VisualDSP++ 3.x C/C++ Compiler and Library Manual for Blackfin Processors*.

You need to take this into consideration and insert additional code if the macros are used outside of the context of the C run-time model.

Method Result Macros

Macros provided for constructing method result values and testing the result values returned from method calls are listed as follows.

VCSE_MRESULT

This macro expands into the appropriate data definition directive when defining a memory location to hold a method result. On Blackfin processors, the directive is `.BYTE2`.

MR_ICONSTRUCT(F,I)

Use this macro to construct a method result (`MRESULT`) value literal, combining the failure indicator `F` (which should be `1` if the specified result code, `I`, denotes a method failure and `0` otherwise) and a specific failure or warning code value, `I` (which should be a decimal number in the range `0-255`). See "MRESULT Codes" on page B-2 for further details on the construction of `MRESULT` values.

The following code fragment is an example of how the MR_ICONSTRUCT macro can be used.

```
#define warn 0
#define fail 1
#define NOT_FOUND     MR_ICONSTRUCT(fail,3)
#define CREATED_NEW   MR_ICONSTRUCT(warn,4)
.
.
CC = ...
R0 = NOT_FOUND;
IF CC R0 = CREATED_NEW;
RETS;
```

MR_FAILURE(mr) and MR_SUCCESS(mr)

These macros can be used to determine whether or not a returned method result value represents a failure or otherwise. MR_FAILURE sets CC to 1 if mr represents a failure code; otherwise, the macro sets CC to zero. MR_SUCCESS does the opposite.

The following is an example of how to use the macros immediately after every method call.

```
MR_FAILURE(R0)
IF CC JUMP .my_error_label;
```

__CHECK_VCSE_RESPONSE(handler)

This macro provides an alternative way to check whether a method call is successful. Assuming the result code is still in R0, the macro compares the result with the predefined value MR_OK. If the values are not equal (the method reported either a failure or a warning), then the user supplied function handler is called with the result code in R0.

Accessing Factory Functions

Every VCSE component has three factory functions, which client applications use to create and destroy instances of the component and to obtain an indication of the size of a component's per-instance data structure. Each of the macros `__CREATOR(C)`, `__DESTROYER(C)`, and `__SIZEOF(C)` takes the fully qualified name of a VCSE component and expands it into the name of the component's `Create`, `Destroy`, and `Sizeof` functions, respectively.

Taking `__CREATOR` as an example:

```
#define FIR ADI_FILTERS_CFir /* fully qualified component name */
  .
  .
/* load up Create's arguments */
...
call __CREATOR(FIR)
MR_FAILURE(R0)
IF CC JUMP .no_fir;
```

If at a later time a different FIR component is to be used in the application, all that needs to be changed is the `#define` of `FIR`.

Invoking Interface Methods

The usual approach for invoking an interface method is to use the macro the VIDL compiler generates for it in the interface header file. For example, method `Filter` in an interface `ADI::FILTERS::IFir` would have a macro called `ADI_FILTERS_IFir_Filter(P)` defined in the interface header file `<ADI_FILTERS_IFilter.h>`. To invoke the method, use the following code.

```
/* load up Filter's arguments into R1, R2 and stack slots */
/* ... */
/* and then invoke Filter */
```

```
ADI_FILTERS_IFir_Filter(P)
MR_FAILURE(R0)
IF CC JUMP .error_3;
```

In the macro call, `P` is either the name of the register containing the interface pointer or an addressing expression, such as [`FP-24`] or [`P3+4`]), for the location where it is stored.

Each of the generated method call macros ultimately uses a macro called `__GET_METHOD(P,T,M)` to obtain a pointer to the method's code and to calculate its first argument. In situations where the same method of the same interface pointer is being called repeatedly, it may be appropriate for users to call `__GET_METHOD` directly, save the code pointer and argument value, and use these values to call the method subsequently.

The `P` parameter to `__GET_METHOD` is either the name of the register holding the interface pointer whose method is required, or an addressing expression from which it can be loaded. The `T` parameter is the name of the interface, and `M` is the name of the required method. The macro puts the method's code pointer into register `P0` and its required first argument into `R0`. The macro also overwrites `R3` and `P1`.

Instead of using the `ADI_FILTERS_IFir_Filter` macro to call the `Filter` method, as shown in the previous example, an application could use `__GET_METHOD`:

```
/* outside main loop  */
__GET_METHOD(P,ADI_FILTERS_IFir,Filter)
P3 = P0;  /* save method code address  */
R7 = R0;  /* save method's first argument */
.
.
/* ... inside main loop */
/* load up Filter's arguments into R1, R2, and stack slots
/* ...
/* then load up saved first argument */
```

```
R0 = R7;
/* and call the method */
call (P3);
MR_FAILURE(R0)
IF CC JUMP .error_8;
```

Function Writing Macros

The definition of a function in assembly, especially one that follows the C run-time model, requires the use of certain directives and instruction sequences. The directives are concerned with making the function's name, size, and visibility available in the generated object file. The instruction sequences are required for setting up stack frames, saving and restoring preserved registers, and returning function results. The following macros are available to help with these tasks.

__STARTFUNC(Name,Visibility) and __ENDFUNC(Name)

These two macros generate the assembler directives, which mark the start and the end of an assembly written function. Because the `Name` argument is used 'as is', it is important to include a leading underscore if the function is to be called from C or C++ code.

The `Visibility` argument to `__STARTFUNC` should be one of the symbols `__GLOBAL` or `__LOCAL`, depending on whether you want the function name to be visible from outside this file.

__LINK(N)

This macro is an alternative name for the Blackfin processor link instruction, which creates a new stack frame by pushing the return address and old `FP` on the stack and decrementing `SP` by the requested number of bytes to allocate space for the function's on-stack variables.

__PUSH(Reg) and __POP(Reg)

The __PUSH macro generates an instruction to push Reg onto the run-time stack. The actual argument supplied for Reg can be anything that is valid for a Blackfin processor PUSH or PUSH_MULTIPLE instruction, such as a register name, a register range in parentheses (__PUSH((R7:5))), or a comma separated pair of ranges (__PUSH((R7:5,P5:4))).

The __POP macro accepts a similar argument to __PUSH and generates the appropriate Blackfin processor pop or pop_multiple instruction.

__ALLOCSTACK(N) and __FREESTACK(N)

The first macro generates an instruction to adjust SP downwards by N bytes to create new space on the run-time stack. N must be a multiple of four with a maximum value of 60. Use __ALLOCSTACK to create the stack slots needed for holding the outgoing arguments of calls made from a function. The __FREESTACK macro adjusts SP in the opposite direction in order to free up temporarily allocated stack space.

__arg0 to __arg9

The C run-time model includes rules defining where a function must place the arguments for a function it calls. Often, these passing places are split between registers and slots on the stack; on Blackfin processors, for instance, the first three arguments are passed in R0-R2 and the remainder on the stack.

The __arg*N* macros expand to addressing expressions, which give the correct location for the first ten arguments. The __arg0, __arg1, and __arg2 macros give R0, R1, and R2 (respectively), while __arg3 gives [SP+12], __arg4 gives [SP+16], and so on.

__EXIT and __LEAF_EXIT

These macros generate the appropriate instructions for exiting non-leaf and leaf functions, respectively. A leaf function is one that calls no other functions and does not issue a link instruction in its prolog. Both macros require the effects of any `__PUSH` and `__ALLOCSTACK` calls to be undone first by calling corresponding `__POP` and `__FREESTACK` macros.

__RETURN(Value) and __LEAF_RETURN(Value)

These macros generate instructions to assign `Value` to the result register `R0` and exit the function (using `__EXIT` or `__LEAF_EXIT` as appropriate). The actual argument used for `Value` can be anything that can be directly assigned to `R0`, such as another register, an immediate value, or the contents of a location (for example, `[P1 + 4]` or `B[P3 + 5](X)`).

Miscellaneous

__LA(R,V)

This macro is a shorthand for the two instructions, `R.H = V; R.L = V;`. Its main use is to load the address of a variable into a register.

__VCSE_ASM_TRACE(A1,A2)

The VIDL compiler calls this macro when you request tracing code to be inserted at the start and end of method bodies, but it may be of more general use. It concatenates two string literal arguments, `A1` and `A2`, and calls a small function in the C run-time library, `_Write`, to write the result to `stdout`. The macro preserves `RETS`, `R7-R0`, and `P5-P0`.

__VCSE_PRINT_VAR(A1,A2,V)

This is another macro used by the VCSE generated tracing code. It concatenates two string literal arguments, `A1` and `A2`, appends a carriage control and a line feed, and passes the result and the value `V` into a call of a simplified `printf`-like function.

The `V` argument must be assignable to register `R1` (another register, an integer literal, or an addressing expression, such as `W[P3 + 12](Z)`), while the concatenation of `A1` and `A2` makes up a format specification for printing `V`. The macro preserves the same registers as `__VCSE_ASM_TRACE` does.

Implementation of Macros on ADSP-21xx DSPs

C Run-Time Model

The macros provided within `vcse_asm.h` assume that the C run-time model is implemented, which is always the case for the assembly implementation of interface methods. The macros, therefore, make use of certain reserved registers, as given in Table A-1.

Table A-1. Reserved Registers for ADSP-21xx DSP C Run-time Model

ADSP-218x DSPs		ADSP-219x DSPs	
Register	Use	Register	Use
`I4`	Stack pointer (`SP`)	`I4`	Stack pointer (`SP`)
`M4`	Frame pointer (`FP`)	`I5`	Frame pointer (`FP`)
`M7`	`-1`	`M5`	`-1`
`M1`	+1		

Table A-1. Reserved Registers for ADSP-21xx DSP C Run-time Model

ADSP-218x DSPs		ADSP-219x DSPs	
M2	0		
M6	0		

You need to take this into consideration and insert additional code if the macros are used outside of the context of the C run-time model.

Method Result Macros

Macros provided for constructing method result values and testing the result values returned from method calls are listed as follows.

VCSE_MRESULT

This macro expands into the appropriate data definition directive when defining a memory location to hold a method result. On ADSP-21xx DSPs, this is simply `.VAR`.

MR_ICONSTRUCT(F,I)

Use this macro to construct a method result value literal (`MRESULT`), combining the failure indicator `F` (which should be `1` if the specified result code, `I`, denotes a method failure and `0` otherwise) and a specific failure or warning code value, `I` (which should be a decimal number in the range `0-255`). See "MRESULT Codes" on page B-2 for further details on the construction of `MRESULT` values.

The following code fragment shows one way to use the `MR_ICONSTRUCT` macro.

```
#define warn 0
#define fail 1
#define NOT_FOUND     MR_ICONSTRUCT(fail,3)
```

```
#define CREATED_NEW   MR_ICONSTRUCT(warn,4)
.
.
AR = ...
AX1 = NOT_FOUND;
IF NE JUMP .end_func;
AX1 = CREATED_NEW;
.END_FUNC:
RTS;
```

MR_FAILURE(mr) and MR_SUCCESS(mr)

These macros can be used to determine whether or not a returned method result value represents a failure or otherwise. `MR_FAILURE` sets the uppermost bit (`15`) of `AF` to `1` if `mr` represents a failure code, otherwise it sets it to zero. Similarly, `MR_SUCCESS` sets the uppermost bit of `AF` to `1` if `mr` represents a success code, and zero otherwise. The `MR_SUCCESS(mr)` macro also modifies `AR` and `SR0`.

The following is an example of how to use the macros immediately after every method call.

```
MR_FAILURE(AX1)
IF NE JUMP .my_error_label;
```

__CHECK_VCSE_RESPONSE(handler)

This macro provides an alternative way to check whether a method call is successful. Assuming the result code is still in `AX1`, the macro compares the result with the predefined value `MR_OK`. If the values are not equal (the method reports either a failure or a warning), then the user supplied function handler is called with the result code in `AX1` pushed on to the outgoing argument stack. The macro modifies `AR`.

Accessing Factory Functions

Every VCSE component has three factory functions, which client applications use to create and destroy instances of the component and to obtain an indication of the size of a component's per-instance data structure. Each of the macros `__CREATOR(C)`, `__DESTROYER(C)`, and `__SIZEOF(C)` takes the fully qualified name of a VCSE component and expands it into the name of the component's `Create`, `Destroy`, and `Sizeof` functions, respectively.

Taking `__CREATOR` as an example:

```
#define FIR ADI_FILTERS_CFir  /*fully qualified component name */
  .
  .
  /* load up Create's arguments */
  ...
  call __CREATOR(FIR)
  MR_FAILURE(AX1)
  IF NE JUMP .no_fir;
```

If at a later time a different `FIR` component is to be used in the application, all that needs to be changed is the `#define` of `FIR`.

Invoking Interface Methods

The usual approach for invoking an interface method is to use the macro the VIDL compiler generates for it in the interface header file. For example, method `Filter` in an interface `ADI::FILTERS::IFir` has a macro `ADI_FILTERS_IFir_Filter(P)` defined in the interface header file <ADI_FILTERS_IFilter.h>. To invoke the method, use the following code.

```
/* load up Filter's arguments into the stack slots */
/* ... */
/* and then invoke Filter */
ADI_FILTERS_IFir_Filter(P)
```

```
MR_FAILURE(AX1)
IF NE JUMP .error_3;
```

In the macro call, `P` is the name of the register containing the interface pointer.

Each of the generated method call macros ultimately uses a macro called `__GET_METHOD(P,T,M)` to obtain a pointer to the method's code and to calculate its first argument. In situations where the same method of the same interface pointer is being called repeatedly, it may be appropriate for users to call `__GET_METHOD` directly, save the code pointer and argument value, and use these values to call the method subsequently.

The `P` parameter to `__GET_METHOD` is the name of the register holding the interface pointer whose method is required. The `T` parameter is the name of the interface, and `M` is the name of the required method.

On ADSP-218x DSPs, the macro puts the method's code pointer into register `I6` and its required first argument into `AR`. The macro also overwrites `AX0`, `AY1`, and `I0`.

On ADSP-219x DSPs, the address bus is 24 bits wide, so two registers are required to hold the method's code pointer. The macro puts the lower 16 bits of the pointer into register `I1`, the upper 8 bits of the pointer into register `IJPG`, and the method's required first argument into `AR`. The macro also overwrites `AX0`, `AY1`, and `I0`.

Instead of using the `ADI_FILTERS_IFir_Filter` macro to call the `Filter` method, as shown in the code example, an application can use the `__GET_METHOD` macro (see Table A-2 on page A-21).

Table A-2. __Get_Method Macros

ADSP-218x DSPs	ADSP-219x DSPs

/* outside main loop */
__GET_METHOD(
      P,ADI_FILTERS_IFir, Filter)
SE = I6; /* save method code address */
SI = AR; /* save method's first argument */
/* ... inside main loop */
/* load up Filter's arguments into stack slots */
/* then load up saved first argument */
__PUSH(AR)
/* load up the method's address */
I6 = SE;
/* and call the method */
call (I6);
MR_FAILURE(AX1)
IF NE JUMP .error_8;
	/* outside main loop */
	__GET_METHOD(
	      P,ADI_FILTERS_IFir, Filter)
	SE = I1; /* save method code address */
	MX0=IJPG;
	SI = AR; /* save method's first argument */
	/* ... inside main loop */
	/* load up Filter's arguments into stack slots */
	/* load up saved first argument */
	__PUSH(AR)
	/* load up the method's address and the saved first argument, call the method */
	I1 = SE;
	IJPG=MX0;
	call (I1) (DB);
	__PUSH(AR)
	NOP;
	MR_FAILURE(AX1)
	IF NE JUMP .error_8;

Function Writing Macros

The definition of a function in assembly, especially one that follows the C run-time model, requires the use of certain directives and instruction sequences. The directives are concerned with making the function's name, size, and visibility available in the generated object file. The instruction sequences are required for setting up stack frames, saving and restoring preserved registers, and returning function results. The following macros are available to help with these tasks.

__STARTFUNC(Name,Visibility) and __ENDFUNC(Name)

These two macros generate the assembler directives, which mark the start and the end of an assembly written function. The `Name` argument is used 'as is'; it is important to include a leading underscore if the function is to be called from C or C++ code.

The `Visibility` argument to `__STARTFUNC` should be `__GLOBAL` or `__LOCAL`, depending on whether you want the function name to be visible from outside the file.

__LINK(N)

This macro creates a new stack frame by pushing the old frame pointer `FP` and the return address on the stack and decrementing the stack pointer `SP` by the requested number of bytes to allocate space for the function's on-stack variables. `SI` and `M5` are modified by the `__LINK(N)` macro (on ADSP-218x DSPs only).

__PUSH(Reg) and __POP(Reg)

The `__PUSH` macro generates an instruction to push `Reg` onto the run-time stack. The actual argument supplied for `Reg` must be a `Dreg`.

The __POP macro accepts a similar argument to __PUSH and generates the appropriate code to pop the run-time stack. I1 is modified by the __POP macro (on ADSP-218x DSPs only).

__ALLOCSTACK(N) and __FREESTACK(N)

The first macro generates an instruction to adjust SP downwards by N words to create new space on the run-time stack. Use __ALLOCSTACK to create the stack slots needed for holding the outgoing arguments of calls made from a function. __FREESTACK adjusts SP in the opposite direction in order to free up temporarily allocated stack space. On ADSP-218x DSPs, M5 is modified by both macros (on ADSP-218x DSPs only).

__arg0 to __arg9 (ADSP-219x DSPs only)

The C run-time model includes rules defining where a function must place the arguments for a function it calls. For ADSP-219x DSPs, these passing places are slots on the run-time stack.

The __arg*N* macros expand to addressing expressions, which give the correct location for the first ten arguments. The __arg0 macro gives DM(SP+1), __arg1 gives DM(SP+2), and so on.

__STORE_ARG(n,Reg) (ADSP-218x only)

The pre-modify-offset mode of DAG addressing, used in the above __arg*N* macros on ADSP-219x DSP architectures, is not available on the ADSP-218x DSPs. It is, therefore, not possible to construct these macros using ADSP-218x DSP assembler. The alternative macro, __STORE_ARG(n,Reg), which results in a complete DAG move instruction, is thus provided for consistency. However, use of __STORE_ARG(n,Reg) for multiple arguments is not recommended. A better way is either to use the __PUSH macro for each argument in reverse order or, if the arguments must be added in ascending order, apply the following code example.

```
__ALLOCSTACK(2)
I0 = I4;
MODIFY(I0,M1);    /* I4 now points to the first argument slot */
AX0 = ...;
DM(I0+=M1) = AX0;  /* Store first argument */
AX0 = ...;
DM(I0+=M1) = AX0;  /* Store second argument */
call _my_func;
__FREESTACK(2)
```

If, in the above example, I0 is not available, then the run-time stack can be used to store its value, but it must be pushed prior to __ALLOCSTACK(2) and popped after __FREESTACK(2).

The __STORE_ARG(n,Reg) macro modifies I1 and M3.

__EXIT and __LEAF_EXIT

These macros generate the appropriate instructions for exiting non-leaf and leaf functions, respectively. A leaf function is one that calls no other functions and does not store the linkage information in its prolog. Both macros require the effects of any __PUSH and __ALLOCSTACK calls to be undone first by calling the corresponding __POP and __FREESTACK macros.

Additionally, if __LINK(N) with N>0 has been used in the prolog, then __FREESTACK(N) must be used prior to the use of __EXIT or __RETURN(Value).

The __EXIT macro modifies I6 and SI (on ADSP-218x DSPs only).

__RETURN(Value) and __LEAF_RETURN(Value)

These macros generate instructions to assign `Value` to the result register `AX1` and exit the function (using `__EXIT` or `__LEAF_EXIT` as appropriate). The actual argument used for `Value` can be anything that can be directly assigned to `AX1`, such as another register, an immediate value, or the contents of a location (for example, `DM(I0,M0)`).

Miscellaneous Macros

__LA(R,V)

This macro is provided for consistency with the macros provided for other DSP architectures. For ADSP-21xx DSPs, it simply translates to `R=V;`. Use the macro to load the address of a variable into a register (for example, `__LA(AX0,_my_var)`).

__VCSE_ASM_TRACE(A1,A2)

Calls to this macro are generated by the VIDL compiler when you request tracing code to be placed at the start and end of method bodies, but it may be of more general use. It concatenates its two string literal arguments, `A1` and `A2`, and calls a simplified `printf`-like function to write the result to `stdout`. Please note that `AX1` is used in the macro, and other registers may be clobbered within the `printf`-like function.

__VCSE_PRINT_VAR(A1,A2,V)

This is another macro used by the VCSE generated tracing code. It concatenates two string literal arguments, `A1` and `A2`, appends a carriage control and a line feed, and passes the result and the value `V` to a simplified `printf`-like function.

The V argument must be assignable to register AX1, while the concatenation of A1 and A2 must make up the format specification by which V is output, for example:

```
__VCSE_PRINT_VAR('ADI::FILTERS::Ifir::Filter',' method result is
%x',AR)
```

Please refer to __VCSE_ASM_TRACE(A1,A2) for comments concerning registers usage.

B VCSE MRESULT CODES

This appendix lists and describes the defined MSRESULT codes.

The information is presented as follows.

- "MRESULT Structure" on page B-1
- "MRESULT Codes" on page B-2

MRESULT Structure

An MRESULT is defined as a signed short integer on each DSP platform and, therefore, is a 16-bit signed quantity.

Bit	15	14-8	7-0
Value	F	vcode	icode

The high order bit (bit 15) of an MRESULT indicates whether the return value represents success or failure. If set to zero, the value indicates success. If set to one, it indicates failure. The macros MR_SUCCESS and MR_FAILURE can be also used to test for success or failure.

The next seven bits (bit 14 to bit 8) are reserved for VCSE defined result codes. The low order eight bits (bit 7 to bit 0) are used for interface specific result codes.

The interface specific field icode is set to zero for all VCSE defined result codes. Similarly, the VCSE defined result field vcode should be zero for all interface specific results.

The macro MR_VCODE can be used to access the VCSE defined field.

The macro MR_ICODE can be used to access the interface specific field.

MRESULT Codes

Table B-1 lists and briefly describes the MRESULT codes.

Table B-1. VCSE MRESULT Codes

Code	Description
MR_OK (0x0000)	Indicates the VCSE function or method executed without failure.
MR_FAILED (0x8000)	Indicates the VCSE function or method detects a failure, which does not have a specific result code value.
MR_NOT_SUPPORTED (0x8100)	Indicates the underlying component does not implement the requested interface. The code is returned by a GetInterface method.
MR_NO_MEMORY (0x8200)	Indicates the Allocate method of the IMemory interface does not have sufficient available memory to satisfy a memory allocation request.
MR_NO_AGGREGATION (0x8300)	Indicates an attempt is made to aggregate a component that does not support aggregation.
MR_BAD_AGGREGATION (0x8400)	Indicates the requested interface is not VCSE::IBase. The code is returned by a component's Create function when it is called to create an instance for aggregation into another component.

Table B-1. VCSE MRESULT Codes (Cont'd)

Code	Description
MR_BAD_ALIGNMENT (0x8500)	Indicates the MemRequest struct passed to the Allocate method of the IMemory interface has a bad value for the Alignment member.
MR_BAD_MEMTYPE (0x8600)	Indicates the MemRequest struct passed to the Allocate method of the IMemory interface has a bad value for the TypeFlags member.
MR_BAD_MEMLIFE (0x8700)	Indicates the MemRequest struct passed to the Allocate method of the IMemory interface has a bad value for the LifetimeFlags member.
MR_BAD_CONTEXT (0x8800)	Indicates the MemRequest struct passed to the Allocate method of the IMemory interface has a bad value for the Context member.
MR_BAD_MEMBANK (0x8900)	Indicates the MemRequest struct passed to the Allocate method of the IMemory interface has a bad value for the BankName member.
MR_BAD_HANDLE (0x8A00)	Indicates an invalid Token value is passed to the Allocate and Free methods of a component that implements IMemory.
MR_NOT_COMPLETED (0x8B00)	Indicates the called function or method did not complete its processing. It may have reported a specific error by other means, such as an IError interface. This is a general result code.
MR_NOT_ALLOCATED_MEM (0x8C00)	Indicates the Free method of an IMemory instance is asked to free memory it did not allocate.
MR_INV_PARAM (0x8D00)	Indicates an invalid value for a method argument is not covered by some interface specific result code. This is a general result code.
MR_BAD_IFCE_PTR (0x8E00)	Indicates a NULL interface pointer is passed to a component's Destroy function.
MR_SINGLETON_EXISTS (0x8F00)	Indicates an attempt made to create more than one instance of a [singleton] component. The code is returned by the Create function.

Table B-1. VCSE MRESULT Codes (Cont'd)

Code	Description
MR_BAD_STACK_PTR (0x9000)	Indicates that the stack pointer was invalid when a method was invoked.
MR_DSR_REQUIRED (0x9100)	Indicates that the AnalyseInterrupt method requires that the ProcessInterrupt method be invoked subsequently to process an interrupt completely.
MR_NO_RESOURCES (0x9200)	Indicates that a method or a component failed to allocate sufficient resources (such as memory) to enable it to function correctly.

C VCSE UTILITIES

This appendix describes the VCSE utilities in detail.

vcse_enforce

The vcse_enforce utility can be used to enforce the VCSE naming rules for both globally defined variables and section names within a component library. The vcse_enforce utility scans all the object modules in the library to determine all the globally defined names in the library and to collect all the sections names that are defined in the modules. The utility checks to see if each of the names is prefixed with the component name. If any name is not properly prefixed, each occurrence of the name in each of the object modules is consistently modified with the correct prefix to ensure that the name is unique within the library. For example if the name of the component is ADI::TOOLS::CMemory, each globally defined name is checked to ensure that it is prefixed with _ADI_TOOLS_CMemory_ and each nonstandard section name is prefixed with ADI_TOOLS_CMemory_. If necessary the corresponding prefix is added to each name.

If any changes to an object within the library are required, the library is updated "in place".

The general format for invoking the vcse_enforce utility is as shown below.

```
vcse_enforce [-M] [-MM] [-verbose] [-no_update] [-report] [-help]
[-obfuscate] [-names name_control_file] [-cname component_name]
[-add library_dlb]  component_lib.dlb
```

The significance of each of the options is described in the sections that follow.

-add library.dlb

The contents of a library file (`library_dlb`) specified by the `-add` options will be merged the contents of the specified component library. The utility processes the contents of the library as if it were included in the supplied component library. The `-add` option can appear more than once if more than one library file's contents are to be merged in. If the name of an object file in one of the additional libraries matches the name of an object file in the component library, the object file in the component is overwritten with the contents from the additional library.

No changes are made to any of the library files specified by the `-add` option. Only the object files in the merged component library have name changes applied.

Attempts to rerun the `vcse_enforce` utility on an already-processed component library, when `-add` has been used to merge the contents of one or more libraries, may be unsuccessful. Regenerate the contents of the base component library *before* processing a second time when using `vcse_enforce` and `-add` together.

-cname component_name

By default the `vcse_enforce` utility determines the name of the component from the name of the supplied component library. The `-cname` option can be used to specify the actual name for the component. The elements of the component name must be separated by `::`.

-help

The `-help` option produces a brief usage summary of the options supported by the utility.

-M

The `-M` option generates a dependency list showing the input files upon which the updated component library is dependent. It does *not* check or update the component library.

-MM

The `-MM` option generates a dependency list showing the input files upon which the updated component library is dependent. It then checks and updates the component library appropriately.

-names name_control_file

The name control file can be used to modify the processing of each of the names in the library encountered by the utility. The format and use of the name control files is described in "Name Control File" on page C-5.

-no_update

The `-no_update` option instructs the utility to scan the library to determine and report on the changes it would make. Using this option suppresses the actual update of the library function.

-obfuscate

The `-obfuscate` option should obfuscate any symbol names to which it adds a prefix. It also prefixes any symbol names. When this option is not used, the original names of all global symbols are simply prefixed but not hidden.

-report

The -report option instructs the utility to produce a summary listing showing the names of each object module that needs to be updated. The -report option causes the utility to generate a summary showing the symbol and section names changes that will be made within the library.

-verbose

The -verbose option causes the utility to generate a report detailing each name change that is made to each file.

Operation of the Utility

The vcse_enforce utility scans each of the object files in the library looking for any globally-defined symbol names that are not prefixed with the prefix corresponding to the component name. If any such names are found, then the utility updates the object files in the library. As a result, the definition and any external references to the name are changed to the appropriately prefixed name.

No changes are made if the global definition of the name is marked as being WEAK, since all GUID definitions are weak symbols that are globally defined.

Any section names that are marked as to be loaded with the program are also checked to ensure that any nonstandard section names are properly prefixed. The standard section names that the utility recognizes are shown in Table C-1.

By default the utility determines the name of the component from the name of the supplied library file. The utility examines the filename and replaces each _ with :: to form the full component name. For example, if the name of the supplied library is ADI_TOOLS_CSample.dlb then the default name for the component is ADI::TOOLS::CSample.

Table C-1. Standard Section Names

Architecture	Section Names
Blackfin	program data1 constdata voldata cplb ctor
218x	program data1 data2
219x	program data1 data2 ctor ctor_end

Name Control File

The name control file enables developers to control how names are to be processed. The control file can specify that a name not be changed, or that a name should be changed to a specified name, or that a normally protected name be prefixed with the component name prefix.

Each line of the name control file contains a single name and a definition of how the name is to be processed. The names in the control file can be specified as either symbol or section names. A line containing [symbol] indicates that the following lines contain the names of global symbols. A line containing [section] indicates that the following lines contain the names of sections. A line containing [symbol] or [section] must be present before any names are encountered.

A line containing a name can be in one of three formats.

name

A name by itself implies that the specified name should not be modified even if the name is not properly prefixed.

name=newname

In this case any occurrence of the specified name should be changed to the supplied `newname` whenever it occurs.

name=

In this case, the supplied name is a name that is normally protected from being prefixed (a standard section name for example). This line specifies that each occurrence of the name must be prefixed with the appropriate component prefix.

Blanks before or after any of the names are ignored but embedded blanks are preserved. The occurrence of a `;` or a `/` on a line indicates that the separator and any following text are comments and can be ignored.

The names used by in the objects are ELF symbol names. Any C generated names need to be prefixed with an _ character when used within the name control file.

In the example of a name control file below, the comments indicate how each of the names is to be dealt with.

```
[symbol]

sendto                          ; name is not to be changed
_closesocket = _SocketClose      ; the name _closesocket is always to be
                                 ; changed to _SocketClose

[section]

data1=                          / the standard section name data1 is to
                                / be prefixed with the component prefix
```

vcse_sizer

The `vcse_sizer` utility can be used to determine the total size of each of the sections that are defined in the modules in a library. When this utility is used, it can also determine the size of the instance data for a component. The utility accumulates the size of each section from all the object files in the supplied library. By default the size information is generated as an HTML table to simplify including it in the component documentation. A description file can also be supplied to allow a description for each section to be automatically included in the generated HTML. The size for each section is reported in bytes.

```
vcse_sizer [-text] [-help][-desc desc_filename
   [-cname component_name] component_lib.dlb]
```

The significance of each of the options is described in the sections that follow.

-cname component_name

By default the utility determines the name of the component from the name of the supplied component library. The `-cname` option can be used to specify the actual name for the component. The elements of the component name must be separated by `::`.

-desc desc_filename

The description file can be used to supply a description for each section. The format and use of the description file is described in "Description File" on page C-8.

-help

The -help option produces a brief usage summary of the options supported by the utility.

-text

The -text option requests that the generated size information is output as plain text rather than as HTML text. The information in this case is generated as comma-separated variable information with a separate record for each section.

By default the utility determines the name of the component from the name of the supplied library file. The utility examines the filename and replaces each _ with :: to form the full component name. For example, if the name of the supplied library is ADI_TOOLS_CSample.dlb then the default name for the component would be ADI::TOOLS::CSample.

Description File

The description file consists of a series of entries each of which starts with a section name enclosed in square brackets immediately followed by the description for the section. The description can be HTML text and can be written over multiple lines, if necessary.

Any text encountered before the first section name is ignored.

A sample description file is shown below

[[**program**]]	This section holds all the code generated by the compiler.

[[**data1**]]	The data section, **data1**, is where the compiler puts global and static data in memory.
[[[**constdata**]]]	The const data section holds all the data that was marked as “const”.

vcse_packager

The `vcse_packager` utility can be used to automatically package a component using the specified component manifest file. Optionally, the packaged component can be installed on the system. The packager can also be used to uninstall a component.

```
vcse_packager  [-install]
     [-package package_filename] manifest.xml
```

OR

```
vcse_packager  [-uninstall] component_uuid
```

When the `vcse_package` utility is used, any existing package file is automatically overwritten. The default name of the package file is derived from the name of the manifest file with its file extension replaced by the `.vcp` file extension. When a component is packaged its unique user identifier (UUID) is displayed.

The significance of each of the options are described in the sections that follow.

-install

The `-install` option requests that the generated package should be automatically installed on the current system. Any previous installation will be overwritten with the contents of the newly generated package.

-package

If present, the `-package` option requests that the generated package file should be given `package_filename` as its name.

-uninstall

The `-uninstall` option requests that the component identified by the supplied UUID be uninstalled from the current system.

D PCC — AN EXAMPLE OF VCSE INTERFACE DESIGN

VCSE can be used to various degrees to aid the design and implementation of application programs, from single interface/single component arrangements covering very specific DSP algorithms to large interface hierarchies controlling much of an application's processing. This appendix aims in between these extremes, since it describes some aspects of the design of a set of interfaces intended to ease the use of the system peripheral set on one of Analog Devices processor families.

The description briefly introduces the interface set, describes the motivation for its creation, and defines its scope. Additional detail about the set as a whole and some of the individual interfaces is then supplied, followed by a description of how components that implement the interfaces are built and used. The material is contained in the following sections.

- "Introduction" on page D-2
- "The Interfaces—Detailed Descriptions" on page D-4
- "Implementing a PCC Component" on page D-13
- "Using PCC Components" on page D-15

VIDL files containing the VCSE interfaces mentioned here are available from the VCSE web site, and can be downloaded using the VCSE Component Manager, which is integrated with VisualDSP. The interface files are extensively documented using VCSE's auto-doc feature and contain more detailed information than is included in this appendix. Also available are several example VCSE components that implement some of the interfaces for various members of the Blackfin processor family.

Introduction

This appendix describes some design aspects for a set of VCSE interfaces, referred to as the Peripheral Control Components (PCC) interfaces. The PCC components provide application designers and programmers with control code for Blackfin peripherals. The components are simple to use (and to re-use) in demonstration programs and algorithm test beds. The interfaces are designed to be extensible so that control components for external peripherals can be developed quickly and compliment those interfaces that control the system peripherals. No specific application organization (stand-alone, VDK tasks, Linux processes, and so on) is implied or required.

Motivation

For marketing or educational purposes developers often need to produce programs that are not fully developed products, but which nevertheless perform some useful function. Such a program can, for example, be designed to accept a stream of encoded audio, decode it in real time, and play the results through a codec chip. The middle step, decoding the stream correctly in a timely manner, should be the focus of the project and take most of the development effort. Often, however, considerable time must be spent on developing code to control the on-chip and off-chip peripherals involved in data acquisition and disposal, or in modifying such code designed for a different, and differently-organized, program.

Similarly, design engineers may want to develop simple test bed programs into which they can drop their core algorithms. Afterwards, they need to run these algorithms using representative data in order to characterize their performance and memory requirements. The focus at this point is on how the algorithms perform and whether the signal processing features of the processor are being used to maximum effect – data throughput considerations come later, but in the meantime a simple-to-use way of getting the data into and out of the processor reasonably efficiently and without much development effort is all that is required.

This appendix describes some aspects of a design for a set of VCSE interfaces that encapsulates into a small number of methods and structures the common modes of use of most of the system peripherals found in Analog Devices Blackfin processors, such as SPORT, SPI, UART, and so on. These interfaces, coupled with a set of components that implement them, offer a common framework for building a variety of programs.

Scope

Because of the nature of the topic this appendix deals exclusively with one family of processors – Blackfin – and occasionally with individual members of that family. PCC interfaces and components developed for the other processor families on which VCSE is implemented are for the most part the same as those described here, but would differ both in features used and details. The processing model remains the same across all families.

The interfaces that are described here capture common modes of operating the Blackfin system peripherals and also imply a certain data processing model. The operating modes and processing model are expected to be sufficient for the class of program being targeted but may not fulfill the needs of a complete product-class application.

The Interfaces—Detailed Descriptions

The PCC interfaces fall into several categories:

1. **Setup and control**

   A hierarchy of interfaces concerned with the setup and overall control of the Blackfin's on-chip peripherals and devices attached to them is established. The base interface defines the most basic enumeration values and methods common to all peripherals. Other interfaces extend the base with general information and methods suitable for on-chip ("system") and external peripherals, respectively. These in turn are extended to describe specific peripherals. The control methods in these interfaces deal with functions such as enabling, disabling and resetting a peripheral rather than its data handling capabilities.

2. **Data transfer**

   Two main freestanding interfaces describe a common model of transferring data between an application and a peripheral, each in a distinct manner. Once a peripheral has been initialized and enabled, the methods in these interfaces are used by the application repeatedly to obtain raw data and dispose of the results.

3. **DMA and interrupts**

   Two additional freestanding interfaces are designed to capture fundamental hardware features common to most of the peripherals – DMA and interrupt handling.

4. **Services and callbacks**

   Several other interfaces define common services or actions that the application may need to provide for PCC components. For example, one of these interfaces defines a small set of methods covering synchronization and timing.

A Description for each major interface in the PCC set, along with examples of specific device interfaces are described in Table D-1. Table D-1 lists the interface's name, immediate ancestor, a summary of its contents, and a description of its purpose. Subsequent sections provide a more detailed description of selected interfaces.

Table D-1. Summary of PCC and Device Interfaces

Interface	Extends	Contents and Purpose
IPeripheral	IBase	Defines the root of entire peripheral hierarchy. Enumerates all system peripherals (SPORT0/1, UART0/1, SPI, and so on) and all interrupt sources. This interface also defines methods for basic control (enable, disable, and so on) and for obtaining data transfer interfaces.
ISysPeripheral	IPeripheral	Defines the root of system (on-chip) peripheral hierarchy. Adds methods for obtaining interrupt handler interface (for application's ISR to call) and registering 'logical' interrupt interfaces.
ISPI	ISysPeripheral	Provides the specific interface for control of serial peripheral interface (SPI) system peripherals. Adds SPI-specific enums, structs and methods to allow static setup of SPI peripherals and dynamic switching between master and slave modes.
ISPORT	ISysPeripheral	Provides the specific interface for control of serial port (SPORT) system peripherals. Adds structs and a configuration method to allow static setup of SPORT clocks and transfer modes on RX and TX channels.
IPciHost	ISysPeripheral	Provides the specific interface for control of PCI system peripheral acting in host mode (ADSP-BF535 only). Adds enums, structs and methods for setting up the PCI controller in host mode. This interface also includes methods for discovering what devices are installed on the PCI bus and for obtaining `IPciAgent` interfaces to access them.

Table D-1. Summary of PCC and Device Interfaces (Cont'd)

Interface	Extends	Contents and Purpose
IProgrammable-Flag	ISysPeripheral	Provides the specific interface for control of the programmable flags. Adds enums and functions to allow interrupt attributes to be set and flag state to be queried and set.
IExtPeripheral	IPeripheral	Provides the root of off-chip (external) device hierarchy. Adds method to allow registering of peripherals (probably `IProgrammableFlag`) that in some way control or select the external device.
IEBIUPeripheral	IExtPeripheral	Enables additional functionality for devices connected to the external bus interface unit by adding a means to specify the address range(s) to which the device responds.
IAD1836	IExtPeripheral	Provides an example of an interface for a specific external device, in this case an Analog Devices 1836 audio codec.
ILan91C111	IEBIUPeripheral	Provides an example of an interface for a device connected via the external bus interface unit. In this case it is an SMsC LAN91C111 Ethernet controller.
IDataPort	IBase	Defines a general scheme of data transfer, based on the exchange of lists of receive and/or transmit buffers. Manufactured on demand by a PCC component for client's use.
ICircDataPort	IBase	Defines a general scheme of data transfer, based on the circular reuse of one transmit and/or one receive buffer. Manufactured on demand by a PCC component for client's use.
ICallback	IBase	Defines a general scheme for allowing a VCSE component to invoke a client-supplied method at some point in its processing. The data port interfaces support the `ICallback` interface's use.
IPciAgent	IBase	Defines methods for accessing a PCI device's configuration and data spaces. Manufactured on demand by `IPciHost` for client's use.
IDma	IBase	Defines enums and methods by which a PCC component for a data-transfer peripheral may be set up to use one of the forms of direct memory access available on the platform.

Table D-1. Summary of PCC and Device Interfaces (Cont'd)

Interface	Extends	Contents and Purpose
IInter-ruptHandler	IBase	Defines methods supporting the two-level interrupt handling protocol used by PCC. `IInterruptHandler` interfaces are offered by components implementing system peripherals and may be used in the implementation of components for external devices.
IOE	IBase	Abstracts PCC components' timer and synchronization requirements into a form that each application can implement in keeping with its operating environment.
IThreadedOE	IOE	Extends `IOE` with services that PCC components operating in a threaded environment might require.

IDataPort

The `IDataPort` interface provides a general interface for the exchange of data between a component and its client. The interface is manufactured by the component but the data buffers and associated structures are allocated and owned by the client. The client "lends" the component lists of buffers that are to be filled with data or whose contents are to be consumed. Once a buffer has been filled or emptied a *callback method* may be invoked by the component, which prompts the client to "reclaim" the buffer for processing and/or re-use. As an alternative to supplying a callback method, a client may poll the component to reclaim filled receive buffers and emptied transmit buffers.

All of the data transfer peripheral control components manufacture one or more `IDataPort` (and/or `ICircDataPort`) interfaces as appropriate and use those interfaces' methods as the means of exchanging data with their clients. For example, a component implementing the `ISPI` interface on ADSP-BF535 may be prepared to manufacture up to two `IDataPort` interfaces, one each for SPI0 and SPI1. Once a client has requested one of these interfaces, and after it sets up the SPI's operating characteristics using the `ISPI::ConfigureSPI()` method and enabling it for use by invoking `ISPI::Enable()`, it invokes the `UseReceiveBuffers() /`

`ReturnTransmitBuffers()` and `UseTransmitBuffers() / ReturnTransmitBuffers()` method pairs of the `IDataPort` interface repeatedly to receive and send data via that SPI.

Components that implement interfaces for non-data-handling peripherals such as timers and real time clocks do not manufacture any data exchange interfaces.

ICircDataPort

The `ICircDataPort` interface defines the second of the two data processing models supported by PCC. Similar to `IDataPort`, components implementing interfaces for the data transfer peripherals may manufacture `ICircDataPort` interfaces on request. Again the client allocates and controls buffers for the transmission and reception of data, but in this case only one of each can be supplied to each `ICircDataPort` interface. The transfer model is that once enabled, the component uses a pair of volatile read and write pointers associated with each buffer to determine (a) where to place new data received from the peripheral (for receive buffers), and (b) when new data is available for transfer to the peripheral (for transmit buffers). The component invokes a client-supplied callback when specific points are reached such as a buffer's midpoint or end. However, it will continue using the buffer, wrapping around to the start once the end is reached. The client must consume or produce the data fast enough otherwise the component reports an overflow or underflow event via the callback.

IDma

The data exchange models presented by the two data port interfaces are relatively general purpose and can be used to transfer data between any two software entities. However, they imply or at least do not rule out an item-by-item approach to their implementation that might result in unacceptably low performance if used in PCC components. The `IDma` interface is an interface that a PCC component may optionally implement along with an appropriate system peripheral interface such as `ISPORT`, `ISPI`, and so on. The `IDma` interface provides the component's client with the means to request the use of direct memory access for transfers to and from the peripheral, thus relieving the processing burden of item-by-item transfers.

The Blackfin family supports two main types of DMA, descriptor-based and autobuffering. Normally a direct correlation exists between the type of data port that a client is offered and the type of DMA the component performs. If a client obtains an `IDataPort` interface and uses the `IDma::EnableDMA()` method, then the PCC component normally performs descriptor-based DMA. If an `ICircDataPort` interface is obtained, then the component normally uses autobuffering DMA.

The client uses methods in `IDma` to provide static aspects of the DMA setup to be used for a particular peripheral. For example, when a memory area must be made available for DMA descriptors, or when DMA must be enabled and disabled. All the dynamic information that a component needs for performing each DMA transfer is contained in the client-supplied structure associated with each receive and transmit buffer. This information includes the address and size of the whole buffer, the word size of the data items and any data striding or skipping to be performed.

ICallback

The `ICallback` interface is a simple, but general means of allowing a peripheral control component to signal the occurrence of specific events asynchronously. Its client provides an `ICallback` interface during initialization or during normal processing. The component calls its single method, `ICallback::SignalEvent()`, to signal the occurrence of events specified in the appropriate interface. For example, both `IDataPort` and `ICircDataPort` define a method `UseCallback()` and an enumeration of a set of buffer processing conditions. The client may invoke `UseCallback()` passing in an `ICallback` interface (usually implemented within the client as a lightweight component). After that point, the PCC component calls that interface's `SignalEvent()` whenever one or more of the buffer conditions is true.

The `SignalEvent()` method's parameters include a bitmap of the buffer conditions that caused it to be called, a general `IBase` interface which by convention receives the instance of the interface to which this `ICallback` interface was given, and a VIDL `HANDLE` type that allows arbitrary further information to be passed back to the client.

IInterruptHandler

Hardware interrupts and their service routines are at the heart of all embedded programs, and applications written using the PCC components are no different.

While the application code is responsible for setting up the interrupt system by assigning each interrupt source an appropriate priority, populating the interrupt vector table, setting the interrupt masks, and so on, the knowledge of how to handle the interrupts resides in the PCC components.

The `IInterruptHandler` interface forms the bridge between these two arenas, defining two methods which, with the help of a small amount of VCSE library code, make up a two-stage interrupt handling protocol.

In the same way that a PCC component will manufacture a data port interface for each input and output channel it controls it will also manufacture an `IInterruptHandler` interface for each interrupt source to which it responds. While initializing, an application requests one or more interrupt interfaces from each PCC component it is using. It also installs an interrupt service routine (ISR) for each appropriate hardware interrupt. The format of each ISR is identical: it accesses the appropriate `IInterruptHandler` interface and invokes its `AnalyzeInterrupt()` method. If the method returns the `MR_DSR_REQUIRED` result code then the ISR calls a small VCSE library function to put the `IInterruptHandler` interface on a queue for further action. Once all current interrupts are analyzed the VCSE system takes each queued interface and invokes its `ProcessInterrupt()` method as a deferred service routine (DSR).

Generally, `AnalyzeInterrupt()` saves information about the interrupt cause in its internal state and clears the interrupt, while `ProcessInterrupt()` performs updating buffer list pointers, invoking data port callbacks, and so on. If there is very little processing to do, or if a device is in a critical state, then `AnalyzeInterrupt()` performs all the processing instead and does not request that its DSR be run.

When a system peripheral such as a SPORT is being used to transfer data to or from an external device (such as an audio codec), then it is possible that two separate components are involved. This is also the case when one component is responsible for the Blackfin PCI controller in host mode and other components control specific devices attached to the PCI bus. In these scenarios, once the interrupt handler for the system peripheral verifies that an interrupt relates to normal data transfer and not some controller operation, it passes responsibility to a device-specific component. The `IInterruptHandler` interface also acts as the bridge between these two domains.

The `ISysPeripheral` base interface contains a method that lets dependent components register `IInterruptHandler` interfaces of their own. During its interrupt analysis the system peripheral's interrupt handler selects the appropriate registered interfaces, invokes *their* `AnalyzeInterrupt()` method, and queues their `ProcessInterrupt()` method for later execution if requested.

IPeripheral

The `IPeripheral` device interface is the base interface, or template, for all configuration-and-control aspects of the interfaces offered by system peripheral and external device control components on Blackfin. It contains enumerations of a processor's system peripherals and interrupt sources, and definitions of some general configuration and control methods. The configuration methods include one that allows a client to tell a newly-created component instance which specific peripheral it is to control (for example: SPORT0 or SPORT1, UART0 or UART1, and so on), and another that allows the client to pass in one or more interfaces for services that the component may require. The control methods include general reset, enable/disable, and powerup/powerdown operations.

The interface does not directly define methods for data transfer but it does define the method by which clients can obtain data port interfaces for a peripheral or device.

ISysPeripheral

The `ISysPeripheral` interface extends `IPeripheral` with methods related to interrupt handling:

- The `RequestSystemInterruptHandlers()` method allows clients to obtain the `IInterruptHandler` interfaces that their ISRs should call for first-level processing of hardware interrupts.
- The `RegisterLogicalInterruptHandlers()` method allows the writers of components for external devices to register *their* interrupt handlers with the PCC components for the system peripherals that control those devices.

ISPI

The `ISPI` peripheral interface is an example of a PCC interface for a specific system peripheral type. The `ISPI` interface extends `ISysPeripheral` with peripheral-specific enumerations and structures and specifies a configuration method that allows a client to define all common aspects of an SPI port's operation. In addition a method to switch an SPI port between master and slave modes is provided.

IOE

Software controlling hardware devices often needs a means of measuring the passage of time, to implement timeouts or allow 'settling time' after reconfiguring a device for example. Similarly, components often need some means of guaranteeing themselves exclusive access to a specific resource for a short period. To avoid duplication of effort and to ensure that the implementation of such functions does not interfere with their clients' own control code, PCC components do not implement the functions themselves. Instead, a minimal set of service functions has been defined in the `IOE` interface. *Clients* are required to provide a suitable implementation and pass an `IOE` interface to PCC components as needed and defined in documentation.

The mechanism for passing an `IOE` interface to a component (`IPeripheral`'s `UseServices()` method) can be used to pass other 'service' interfaces from client to component if needed.

Implementing a PCC Component

The implementation of a typical PCC component for a Blackfin system peripheral is concerned with two main areas: (a) setup and overall control and (b) data transfer and interrupt handling.

Setup and Overall Control

The *component* definition in the component's VIDL file specifies which of the peripheral interfaces (`ISPORT`, `ISPI`, `IUART`, and so on) it implements and whether it also implements `IDma`. The methods file generated by the VIDL compiler contains empty function definitions for all the methods defined in the selected peripheral interface, in addition those that the interface inherits from `ISysPeripheral` and `IPeripheral`. It also contains empty definitions of `IDma`'s methods if the component implements it. The component developer populates these empty function definitions with setup and control code that is specific to the type of system peripheral specified. Clients create instances of the component in the normal way by using the VIDL-generated `Create()` factory function and obtain the peripheral and DMA interfaces via the `VCSE::GetInterface()` standard method.

Data Transfer and Interrupt Handling

Rather than have a PCC component explicitly implement the data transfer and interrupt handler interfaces as it does for the control and DMA ones, it is required to be able to manufacture multiple instances of these interfaces on demand. The reason for the different approach is that a data port or interrupt source often deals only with one part of a peripheral's functionality (SPORT1 TX channel, for example) whereas setup and control of the peripheral covers all its functionality. The several data port interfaces that interact with one specific peripheral must all be owned by the same instance of the component that is controlling the peripheral.

There are different strategies that can be followed to implement this interface manufacturing requirement. For example, the developer of a single PCC component could decide to define a suitable number of instances of internal 'lightweight' data port and interrupt handler interfaces and hand them out, suitably customized, upon request. Alternatively, the developer of a suite of PCC components could decide to develop private implementations of the data port interfaces, for example, and create instances of them on demand. Since buffer list and circular pointer handling will be common for all the peripherals, this approach avoids duplication of code.

Using PCC Components

An application that makes use of PCC components performs four main tasks in order to set up the proper operating conditions for them:

1. The application calls the `Create()` factory function of each system peripheral component once for each peripheral to be used. That is, if both SPORTs but only one UART are to be used, then the application must obtain two `ISPORT` interfaces by calling the SPORT component's `Create()` function twice, and one `IUART` interface by calling the UART component's `Create()` function once. The program then tells each interface which peripheral (SPORT0 or 1, UART0 or 1) it is controlling by invoking its `UsePeripheral()` method. If the peripheral's DMA capabilities are to be used then the application uses the standard `VCSE::GetInterface()` method to obtain an associated `IDma` interface and configures the peripheral's DMA characteristics using that interface's methods.

2. The application then calls the `Create()` factory function of each external peripheral component to be used. External peripherals are usually connected to one or more of the system peripherals – for example, an audio codec might be connected to a SPORT for data transfer and a SPI for initial setup. The application then customizes each external peripheral interface by passing it the appropriate system peripheral interfaces it obtained and customized in step 1.

3. The application calls the `OpenDataPath()` method of the peripheral component instances to obtain an `IDataPort` or `ICircDataPort` interface for each data transfer stream it uses.

   If the application is driving a system peripheral directly, as it might for a UART, then it obtains the data port interfaces from the system peripheral component and uses them to transmit and receive data in a format acceptable to whatever is connected to 'the other end' of the peripheral. If the application is driving an external peripheral, such a codec, then it obtains the data port interfaces from that device's component and uses them in accordance with the component's documentation. The component for a codec, for example, may offer the application separate left and right channel data streams but operate *its* data port interfaces with the underlying SPORT in an interleaved manner.

4. The application calls the `RequestSystemInterruptHandlers()` method of each system peripheral interface to obtain `IInterruptHandler` interfaces for each interrupt source on which its use of the peripheral relies. The application's ISRs are responsible for invoking the `AnalyzeInterrupt()` method of the appropriate interfaces and for using VCSE library facilities to schedule an invocation of their `ProcessInterrupt()` methods, if requested.

To complete the set up stage and initiate processing the application programs the Blackfin's interrupt priority system, enables interrupts, and invokes the `Reset()` and `Enable()` methods of each peripheral.

If the application has chosen to supply `ICallback` interfaces to its data port providers then its processing is driven by calls of its `SignalEvent()` method(s) made by the data port components as a result of their `ProcessInterrupt()` execution. Alternatively an application can be designed to poll for data using the appropriate methods in `IDataPort` or by checking the volatile read and write pointers associated with each `ICircDataPort` interface.

I INDEX

N

Q

R

T