Flick: The Flexible IDL Compiler Kit
Version 2.1

Programmer's Manual

Flux Research Group
Department of Computer Science
University of Utah

`http://www.cs.utah.edu/flux/`
`flick@cs.utah.edu`

November 1999

Contents

Part I

Introduction

Preface

This manual describes the implementation details of Flick, the Flexible IDL Compiler Kit. People interested in studying, modifying, or extending Flick should consult this manual as a starting point. Those who simply want to *use* Flick do not need to read this document: use of the Flick compiler tools is described in a separate user's manual.

Chapter 1

Introduction

This guide assumes that the reader is familiar with the basics of distributed application programming: RPCs, RMIs, IDLs, and the various distributed computing standards such as CORBA and ONC RPC. Readers who are unfamiliar with distributed computing should review these terms before proceeding through this manual.

1.1 What is Flick?

Flick is a flexible and optimizing compiler for interface definition languages (IDLS). The name "Flick" is an acronym for "Flexible IDL Compiler Kit." Like a traditional IDL compiler, Flick can read a high-level specification describing an interface to a software component or module, and from that, produce special functions called *stubs* that implement the interface in the C or C++ programming language. The stubs implement remote procedure call (RPC) or remote method invocation (RMI) semantics and allow separate client and server processes to communicate through what appear to be ordinary, local procedure calls or method invocations.

Unlike traditional IDL compilers, however, Flick is implemented as a set of programs, each handling a single phase of compilation. These separate programs allow Flick to support features beyond the capabilities of ordinary IDL compilers:

- *Multiple* IDLs*.* Most RPC and distributed object systems, including CORBA and ONC RPC (Sun RPC), specify *an* IDL and from then on refer to it as *the* IDL. Flick, however, does not introduce a new IDL but instead supports a number of existing ones side-by-side. Any IDL supported by Flick can be used to generate stubs for any environment (with some limitations). Supporting multiple IDLS allows for backward compatibility and migration of existing programs, and also allows developers to choose the best IDL for the task at hand.

- *Flexible language mappings (a.k.a. "presentations").* Flick is designed to allow the appearance and behavior of generated stubs to be altered to suit the needs of the particular client or server in which those stubs are used, without altering the "on-the-wire" protocol between client and server. In other words, Flick provides a framework for creating and modifying language mappings from IDLS to target programming languages (currently, C and C++).

  A language mapping is implemented by what Flick calls a *presentation generator*. The current framework allows Flick users to tweak a language mapping through command line options to a presentation generator, e.g., to change the patterns used to build names, or to tell the presentation generator to create "decomposed" stubs instead of ordinary stubs (as described in Section 11.1.2). Large-scale modifications, however, currently require a Flick programmer to write C++ code to implement the modified language mapping (as a new presentation generator). In the future, we hope to make it possible for "ordinary users" to make more and more complicated changes to language mappings, e.g., through some sort of special-purpose "presentation definition language."

- *Sophisticated code generation and optimization.* Flick was originally targeted toward communication in microkernel-based systems. In that environment, Flick provides the user-level glue between the microkernel's primitive IPC facilities and user-level application code. This target environment implies that Flick-generated stubs must be *very fast*: good microkernels always have extremely fast and tightly optimized primitive IPC paths, so inefficient stubs could easily raise the effective IPC time by an order of magnitude. Today, Flick produces optimized code for *all* of its supported transport facilities including TCP-based transports and lower-level, microkernel-based IPC.

- *Multiple back ends.* Finally, while many RPC and distributed object systems claim to support multiple transport protocols, the protocols they support are generally only slight variations of each other. For example, ONC RPC supports multiple transports in the sense that it can pass messages across either TCP or UDP links. However, the basic data encoding scheme used (XDR) is the same in each case, as are the stubs generated by the IDL compiler. The multiple protocol support is really just a stream-based abstraction layer in the RPC runtime support library. There is little hope, for example, that the ONC RPC system could easily be made to support the CORBA Internet Inter-ORB Protocol (IIOP) even though both are based on TCP.

  Flick, on the other hand, aggressively supports multiple protocols by allowing different stub code generators (back ends) to be used interchangeably. Each back end has complete control over the RPC protocols that it implements: the back end determines not only the underlying transport protocol (e.g., TCP, UDP, Mach 3 messages, direct function call, etc.) but also the data encoding scheme used (e.g., Sun's XDR, DCE's NDR, CORBA's CDR, "dumb" architecture-specific encoding, etc.) and the high-level RPC protocol layered above the low-level stream or message protocol (e.g., ONC RPC, CORBA IIOP, Mach 3's MIG protocol, etc.). Flick contains several back ends supporting extremely diverse protocol combinations.

Flick is being developed as part of the Flux Operating Systems Project at the University of Utah. However, Flick is intended to be an open "kit" that others will extend and adapt for their own projects.

1.2 Organization of this Document

Part I of this manual, which you are reading now, introduces Flick. Chapter 2 contains an overview of Flick's components, a catalog of the directories in the Flick source tree, and a summary of the coding style used in the Flick sources.

Part II of this manual describes the data structures that Flick uses at different stages of IDL compilation. These data structures are the intermediate "languages" used by different parts of Flick and include:

- META, for keeping track of the files that Flick reads and writes (Chapter 3);
- AOI, to represent IDL specifications (Chapter 4);
- MINT, to represent the abstract structure of on-the-wire messages (Chapter 5);
- CAST, to describe C and C++ code (Chapter 6);
- PRES_C, to represent the behavior of generated stubs and skeletons (Chapter 7); and

- SCML, to define templates for other functions that Flick must generate as part of an interface presentation (Chapter 8).

Understanding these intermediate languages — and the associated libraries for using them — is critical to understanding Flick overall.

In Part III, this manual describes the inner workings of Flick's front ends (Chapter 10), presentation generators (Chapter 11), and back ends (Chapter 12). The chapters in this part of the manual describe the individual implementations of each compiler stage, as well as the libraries that are common to each stage. The guide concludes with a summary of the runtime libraries that provide support to the code generated by Flick's various back ends (Chapter 13).

In summary, this document is intended to be a programmer's guide to the inner workings of Flick: the overall architecture, the intermediate data structures, and the individual Flick compiler programs and libraries. This "road map" can only take you so far, however, and it is not a substitute for actually reading and working through the Flick source code. Flick is complex but not obscure — don't be afraid to read the code!

Chapter 2

An Outline of the Source Code

This chapter describes the overall organization of Flick's source code and provides some insights into why things are strucutured as they are. Programmers should read this chapter thoroughly before proceeding to later parts of the manual.

2.1 Flick Programs, Files, and Libraries

Flick works in three phases, each implemented as a separate program: the front end, the presentation generator, and the back end. These steps are illustrated in Figure 2.1. A Flick front end is simply a parser that translates some IDL input into an intermediate representation called an AOI (Abstract Object Interface, '`.aoi`') file. Next, a presentation generator determines how the constructs in the AOI file (the parsed IDL file) are to be mapped onto type and function definitions in the C or C++ programming language. In other words, the presentation generator determines how the generated stubs will appear — e.g., the stubs' function prototypes — and how they will interact with user code — e.g., the stubs' parameter passing conventions. This information about the stubs is written into a PRES_C (Presentation in C/C++, '`.prc`') file. The final phase of compilation, the back end, reads a PRES_C file and one or more SCML (Source Code Markup Language) files, and from those descriptions produces the C or C++ code for the optimized stubs, written to use a particular transport system (e.g., TCP) and message format. A `Makefile` is generally used to run the desired Flick compiler programs in series.[1]

Flick has multiple implementations of each of the three phases described above. There are three separate front ends: one to parse CORBA IDL files, one to parse ONC RPC (Sun) IDL files, and one to parse MIG IDL files. There are five different presentation generators, implementing the CORBA (C and C++), ONC RPC, MIG, and Fluke[2] mappings of IDL constructs onto C or C++. Finally, there are seven separate back ends for producing stubs that use IIOP (C and C++), ONC/TCP, Mach messages, Trapeze,[3] Khazana-style messages,[4] and Fluke IPC.

As shown in Figure 2.1, the separate programs communicate through intermediate AOI and PRES_C files. (An AOI file also contains META data; a PRES_C file also contains META, MINT, and CAST data.[5]) This means

[1] At one time, the `flick-c-pdl` program could act as a driver for the other stages of Flick. That program, however, is very much of of date as of this writing. See the `c/pdl` directory for sources.

[2] Fluke is the operating system being developed by the Flux Project at the University of Utah. Flick produces all of the stub code for Fluke. For more information, refer to `http://www.cs.utah.edu/flux/`.

[3] Trapeze is the fast Myrinet-based networking system from Duke University. For more information, see `http://www.cs.duke.edu/ari/trapeze/`.

[4] Khazana is distributed global memory service in development at the University of Utah. For more information, see `http://www.cs.utah.edu/flux/`.

[5] As of this writing, a PRES_C file also contains AOI, for the benefit of the ONC/TCP back end. This is a bug, not a feature.

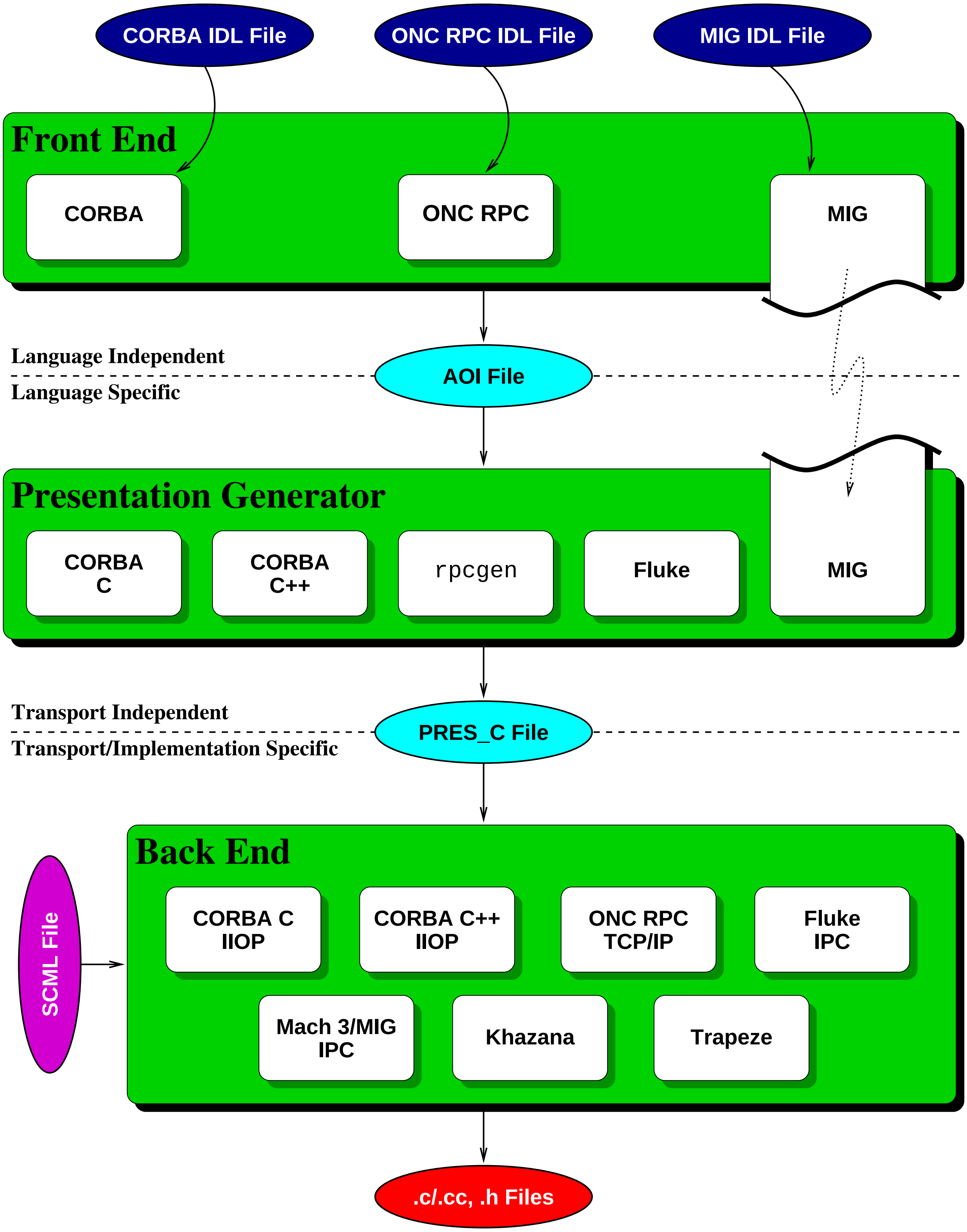

Figure 2.1: Overview of the Flick IDL Compiler.

that Flick must be able to write all of these intermediate languages to files and read them back again. To facilitate these steps, Flick defines these intermediate languages in ONC RPC IDL files (`.x` files, contained in the `mom` directory). When Flick is compiled, these files are processed by `rpcgen` to produce the functions to read and write Flick's intermediate languages to and from files.

Each of Flick's compilation stages is primarily implemented by one or more large libraries of C and C++ code. Different implementations of an IDL compilation stage are created by specializing library C++ classes or overriding library C functions to implement new, tailored behavior. For instance, the ONC RPC presentation generator was created by deriving from the base presentation generator C++ class, and then supplying a small amount of new code to implement the specifics of the ONC RPC language mapping. The libraries that manipulate Flick's intermediate languages are described in Part II of this manual; the libraries for individual compiler stages are described in Part III.

Before moving on to the organization of Flick's source code, here are some retrospective thoughts on the general design and implementation of Flick:

Using `rpcgen` to Generate Code for Flick's Intermediate Representations. The primary reason for using `rpcgen` in defining Flick's intermediate representations was to eliminate the need for programmers to write IR serialization and other I/O code by hand. In retrospect, this early decision has saved the Flick implementors a lot of work. At the same time, however, it has made other parts of the work more difficult. The limitations of the ONC RPC IDL forced compromises in the design of the IRs, e.g., in the representation of scopes and references in AOI. Further, because the `rpcgen`-produced data types are transparent C types, Flick's programmers were tempted to treat them as transparent — not abstract — entities. At many points in the code, `rpcgen`-defined structures are filled out field-by-field, rather than by calling a separate constructor or initialization function. This has made maintenance more difficult, because when the ONC RPC IDL definition of a data type is changed, all the manipulating code must be updated as well. This is not a consistent problem: there are many Flick library functions to help programmers manipulate IRs data structures as abstract objects. It would have been useful, however, to *force* this style of programming throughout Flick's development.[6]

Back Ends. A back end is responsible for generating code that works with a particular transport system, using a particular message format and a particular data encoding format, in conjunction with a certain runtime library. Thus, there are at least four messaging aspects that each back end decides. It would be useful to separate these aspects in Flick's back ends, so that each could be specified separately.[7]

Mix-and-Match. In principle, Flick's design makes it possible and straightforward to mix different front ends, presentation generators, and back ends in novel ways to create a wide variety of stubs. In practice, however, there are several significant barriers to the random mixing of components:

- Combining different front ends and presentation generators is difficult because there is often no obvious mapping for constructs of a "foreign" IDL. For instance, just how should a CORBA IDL `exception` be presented by Flick's ONC RPC presentation generator? In practice, it seems that people don't really want to mix front ends and presentation generators in any case.
- Mixing presentation generators and back ends is principally hindered by the need to provide appropriate runtime support to the generated code. For instance, CORBA object references need to be implemented atop every available transport system, and the Flick developers simply haven't had the time to do the required runtime programming for all "mismatched" cases.

[6]Obviously, we need to implement our own, custom presentation of ONC RPC IDL for developing Flick itself!

[7]It would probably be a bad idea to separate these aspects into separate programs. Who would want an IDL compiler implemented as *six* separate programs?!

- Occasionally, issues arise between mismatched IDL and back end. For instance, how can a non-Mach back end implement a MIG IDL-specified port right? These kinds of conflicts will increase in the future as annotations are attached to IDL specifications, e.g., quality of service constraints.

Therefore, in practice, Flick developers have tended to focus on well-defined pairings between different Flick components.

2.2 Organization of the Source Code

Flick's source code is organized in a directory hierarchy as follows:

- Library header files are generally placed within the `mom` subdirectory. This includes the '`.x`' files that describe Flick's various intermediate data formats as described in Section 2.1. Header files for libraries that are specific to C/C++ code generation are stored in the `mom/c` subdirectory.
- The source code for libraries and programs that are *specific* to C/C++ code generation is stored within the `c` directory.
  - Code for the C/C++-specific libraries is found here, in `c/libcast` and `c/libpres_c`.
  - Code for C/C++ presentation generation is stored within `c/pfe`. ("`pfe`" stands for "presentation front end," which is the old name for what we now call a presentation generator.) The base presentation generator library is in `c/pfe/lib`. Other subdirectories of `c/pfe` contain other presentation generator libraries or specific presentation generators.
  - Code for C/C++ back ends is stored within `c/pbe`. ("`pbe`" is an abbreviation for "presentation back end," which is the old name for what we now call a back end.) The base back end library is contained in `c/pbe/lib`; other subdirectories of `c/pbe` contain other back end libraries or specific back ends.
- The source code for libraries and programs *not specific* to C/C++ code generation is generally stored within immediate subdirectories of the root. For instance:
  - The AOI, META, MINT, and general compiler libraries are contained in the `libaoi`, `libmeta`, `libmint`, and `libcompiler` directories, respectively.
  - Front ends are stored within the `fe` directory. For example, `fe/newcorba` contains the source for Flick's CORBA front end. (Note that the MIG front end is found in `fe/mig`, despite the fact that the MIG front end is specific to the generation of C language stubs.)

In addition, there are top-level directories for other major components of the source code, including the runtime libraries and headers (`runtime`), extra support files for certain platforms (`support`), the Flick testing infrastructure (`test`), and documentation (`doc`).

Below is a detailed listing of the major directories in the Flick source code tree. This catalog is generated automatically by a script that collects the contents of the `WHAT-IS-THIS` files in the Flick source tree. Some of the listed directories — those that contain out-of-date or obsolete code — may not be included as part of general Flick distributions.

.	This is the root of the source tree for Flick, the Flexible, Optimizing IDL Compiler Kit.

`c`	This directory contains all of the Flick libraries and components that are specific to presentation- and code-generation for the C and C++ programming languages.
`c/libcast`	This directory contains general-purpose routines for creating and manipulating C Abstract Syntax Tree (CAST) data structures.
`c/libcparse`	This directory contains lex/yacc code to parse a C source code file into a C Abstract Syntax Tree (CAST). This isn't currently used by Flick, but can be used for testing CAST code.
`c/libpres_c`	This directory contains general-purpose routines for creating and manipulating PRES_C (Presentation in C/C++) data structures.
`c/pbe`	This directory contains the code for all back ends that produce C or C++ code. Aside from certain data structure and general support libraries (libcast, libcompiler, etc.), this directory should contain all code related to the translation of PRES_C (.prc) files to C and C++ source and header files.
`c/pbe/fluke`	This directory contains code for stub generation using the Fluke IPC transport mechanism. This back end produces stubs that are suitable for use within the Fluke kernel server.
`c/pbe/iiop`	This directory contains code for stub generation using the CORBA IIOP transport mechanism.
`c/pbe/iiopxx`	This directory contains code for stub generation using the TAO runtime. TAO is the real-time, open source CORBA ORB from Washington University in St. Louis.
`c/pbe/khazana`	This directory contains code for generating stubs for use within Khazana. Khazana is a distributed service exporting the abstraction of a distributed, secure, persistent, globally shared store that applications can use to store their data. See http://www.cs.utah.edu/flux/ for more information.
`c/pbe/lib`	This directory contains the base implementation of all Flick back ends that produce C or C++ code. The core of the library is an abstract base class; certain member functions must be defined in each derived class to create a complete back end.
`c/pbe/local`	This directory contains (incomplete?) back-end code for generation of a local transport. Intraprocess? Host-local ports? Not really sure. This code is no longer maintained, and it hasn't been substantially changed since 1995, so it is most certainly out of date.
`c/pbe/mach3`	This directory contains back-end code to complete the back end library for the Mach 3 transport. This is different from the Mach3MIG back-end code because it does not attempt to implement the MIG message format. This code is no longer maintained. It has severe bit-rot and almost certainly does not compile or work any longer.
`c/pbe/mach3mig`	This directory contains code to generate Mach 3 transport stubs that emulate the MIG message format.

`c/pbe/mach4` — This directory contains code to produce Mach 4 transport stubs. The code for this back end is incomplete and is no longer maintained.

`c/pbe/sun` — This directory contains code for stub generation using the ONC RPC (Sun RPC) transport mechanism.

`c/pbe/trapeze` — This directory contains code for stub generation using the Trapeze transport mechanism.

`c/pdl` — This directory contains old code for the Presentation Definition Language (PDL) piece of Flick, which also served as a top-level driver program for all of the separate Flick programs (front end, presentation generator, and back end). PDL allowed one to modify a C language presentation (as described by a PRES_C file); however, the PDL program is old, out of date, and is no longer maintained. Some of the code is derived from Sun's 'rpcgen' source code.

`c/pfe` — This directory contains all of the code (except for common data structure manipulation libraries) for translation from AOI to PRES_C. This is called the "presentation generation" phase of IDL compilation.

`c/pfe/corba` — This directory contains the code for Flick's CORBA C language presentation generator. Because the CORBA presentation style is implemented as a library (libcorba), this directory is practically empty.

`c/pfe/corbaxx` — This directory contains the code for Flick's CORBA C++ language presentation generator. Like the CORBA C presentation, this is also implemented as a library (libcorbaxx), so this directory is practically empty.

`c/pfe/fluke` — This directory contains the Fluke/MOM presentation generator for C language stubs. The Fluke language mapping is for the most part like the CORBA C language mapping, but with all the generated stubs and type names changed.

`c/pfe/lib` — This directory contains a "generic" AOI-to-C presentation generator. The generic mapping is largely but not entirely CORBA-like; some of the more complicated mapping procedures are simplified and more 'rpcgen'-like. All C and C++ language presentation generators (except the MIG presentation generator in 'fe/mig') are derived from this generic presentation generator base class, either directly or indirectly.

`c/pfe/libcorba` — This directory contains the CORBA-specific modifications to the "generic" AOI to C language mapping. The CORBA language mapping is organized as a library so that the Fluke presentation generator can be derived from it.

`c/pfe/libcorbaxx` — This directory contains the CORBA C++-specific modifications to the "generic" AOI to C language mapping. It is a library so that other presentation generators can be derived from it.

`c/pfe/sun` — This directory contains the Sun RPC, "rpcgen"-like presentation gen-

	erator. The Sun presentation generator is derived from the "generic" base class for C language presentation generators.
`doc`	This directory contains Flick documentation.
`doc/guts`	This diretcory contains the LaTeX sources for the Flick Programmer's Manual.
`doc/usersguide`	This diretcory contains the LaTeX sources for the Flick User's Manual.
`fe`	This directory contains code for Flick's front ends, which translate from some IDL (CORBA IDL, ONC RPC IDL, MIG IDL, ...) to AOI. Actually, the MIG front end skips over AOI and generates PRES_C directly.
`fe/corba`	This directory contains a "back-end" for the Sun's CORBA IDL compiler front end (which can be obtained from ftp.omg.org). It translates from Sun's CORBA AST to AOI. The code in this directory is out of date and is no longer maintained. Flick's current CORBA front end is in the 'newcorba' directory.
`fe/mig`	This directory contains Flick's MIG front end and presentation generator. MIG contains way too much presentation glop to use the AOI IR, so this MIG front end generates PRES_C directly.
`fe/newcorba`	This directory contains Flick's CORBA IDL front end, which translates CORBA IDL input files into AOI.
`fe/sun`	This directory contains Flick's ONC RPC (a.k.a. Sun RPC) IDL front end, which translates ONC RPC IDL files into AOI files. This front end uses the original 'rpcgen' code to parse the IDL; Flick-specific code then translates the parse tree into AOI format.
`flick-2.1`	This is the root of the source tree for Flick, the Flexible, Optimizing IDL Compiler Kit.
`flick-2.1/c`	This directory contains all of the Flick libraries and components that are specific to presentation- and code-generation for the C and C++ programming languages.
`flick-2.1/c/libcast`	This directory contains general-purpose routines for creating and manipulating C Abstract Syntax Tree (CAST) data structures.
`flick-2.1/c/libpres_c`	This directory contains general-purpose routines for creating and manipulating PRES_C (Presentation in C/C++) data structures.
`flick-2.1/c/pbe`	This directory contains the code for all back ends that produce C or C++ code. Aside from certain data structure and general support libraries (libcast, libcompiler, etc.), this directory should contain all code related to the translation of PRES_C (.prc) files to C and C++ source and header files.
`flick-2.1/c/pbe/fluke`	This directory contains code for stub generation using the Fluke IPC transport mechanism. This back end produces stubs that are suitable for use within the Fluke kernel server.

`flick-2.1/c/pbe/iiop` This directory contains code for stub generation using the CORBA IIOP transport mechanism.

`flick-2.1/c/pbe/iiopxx` This directory contains code for stub generation using the TAO runtime. TAO is the real-time, open source CORBA ORB from Washington University in St. Louis.

`flick-2.1/c/pbe/khazana` This directory contains code for generating stubs for use within Khazana. Khazana is a distributed service exporting the abstraction of a distributed, secure, persistent, globally shared store that applications can use to store their data. See http://www.cs.utah.edu/flux/ for more information.

`flick-2.1/c/pbe/lib` This directory contains the base implementation of all Flick back ends that produce C or C++ code. The core of the library is an abstract base class; certain member functions must be defined in each derived class to create a complete back end.

`flick-2.1/c/pbe/mach3mig` This directory contains code to generate Mach 3 transport stubs that emulate the MIG message format.

`flick-2.1/c/pbe/sun` This directory contains code for stub generation using the ONC RPC (Sun RPC) transport mechanism.

`flick-2.1/c/pbe/trapeze` This directory contains code for stub generation using the Trapeze transport mechanism.

`flick-2.1/c/pfe` This directory contains all of the code (except for common data structure manipulation libraries) for translation from AOI to PRES_C. This is called the "presentation generation" phase of IDL compilation.

`flick-2.1/c/pfe/corba` This directory contains the code for Flick's CORBA C language presentation generator. Because the CORBA presentation style is implemented as a library (libcorba), this directory is practically empty.

`flick-2.1/c/pfe/corbaxx` This directory contains the code for Flick's CORBA C++ language presentation generator. Like the CORBA C presentation, this is also implemented as a library (libcorbaxx), so this directory is practically empty.

`flick-2.1/c/pfe/fluke` This directory contains the Fluke/MOM presentation generator for C language stubs. The Fluke language mapping is for the most part like the CORBA C language mapping, but with all the generated stubs and type names changed.

`flick-2.1/c/pfe/lib` This directory contains a "generic" AOI-to-C presentation generator. The generic mapping is largely but not entirely CORBA-like; some of the more complicated mapping procedures are simplified and more 'rpcgen'-like. All C and C++ language presentation generators (except the MIG presentation generator in 'fe/mig') are derived from this generic presentation generator base class, either directly or indirectly.

`flick-2.1/c/pfe/libcorba` This directory contains the CORBA-specific modifications to the "generic" AOI to C language mapping. The CORBA language mapping is organized as a library so that the Fluke presentation generator can be derived from it.

`flick-2.1/c/pfe/libcorbaxx` This directory contains the CORBA C++-specific modifications to the "generic" AOI to C language mapping. It is a library so that other presentation generators can be derived from it.

`flick-2.1/c/pfe/sun` This directory contains the Sun RPC, "rpcgen"-like presentation generator. The Sun presentation generator is derived from the "generic" base class for C language presentation generators.

`flick-2.1/doc` This directory contains Flick documentation.

`flick-2.1/doc/guts` This diretcory contains the LaTeX sources for the Flick Programmer's Manual.

`flick-2.1/doc/usersguide` This diretcory contains the LaTeX sources for the Flick User's Manual.

`flick-2.1/fe` This directory contains code for Flick's front ends, which translate from some IDL (CORBA IDL, ONC RPC IDL, MIG IDL, ...) to AOI. Actually, the MIG front end skips over AOI and generates PRES_C directly.

`flick-2.1/fe/mig` This directory contains Flick's MIG front end and presentation generator. MIG contains way too much presentation glop to use the AOI IR, so this MIG front end generates PRES_C directly.

`flick-2.1/fe/newcorba` This directory contains Flick's CORBA IDL front end, which translates CORBA IDL input files into AOI.

`flick-2.1/fe/sun` This directory contains Flick's ONC RPC (a.k.a. Sun RPC) IDL front end, which translates ONC RPC IDL files into AOI files. This front end uses the original 'rpcgen' code to parse the IDL; Flick-specific code then translates the parse tree into AOI format.

`flick-2.1/libaoi` This directory contains data creation and manipulation routines for the AOI data format. The AOI library is used extensively in Flick's front ends and presentation generators.

`flick-2.1/libcompiler` This directory contains utility code used throughout Flick, including error-checked malloc/calloc/realloc routines, uniform warning and error printing routines, and some code printing routines.

`flick-2.1/libmeta` This directory contains routines for creating and manipulating the metadata within Flick's intermediate files.

`flick-2.1/libmint` This directory contains data creation and manipulation routines for the MINT data format. It is used extensively throughout presentation generation and back end code generation.

`flick-2.1/mom` This directory contains the definitions of all of the ONC RPC IDL-specified intermediate data formats formats used internally by Flick

(AOI, MINT, CAST, PRES_C, and META). These formats are specified in ONC RPC IDL because Flick needs to send these intermediate formats between separate programs, and Flick does that by writing data to intermediate files. Because the formats are specified in IDL, we can use 'rpcgen' to automatically create the code to read and write Flick's intermediate data representations.

This directory also contains the header files for the libraries that manipulate Flick's non-C/C++-specific intermediate representations (i.e., libaoi, libmint, and libmeta).

`flick-2.1/mom/c` This directory contains header files that are specific to the C and C++ language-specific phases of Flick (i.e., the presentation generation and back end stages).

`flick-2.1/mom/c/be` This directory contains additional header files specific to Flick's C/C++ back ends.

`flick-2.1/test` This directory is the root of test tree. The structure of the test directory tree is patterned after the stages of the Flick compiler. There are separate directories to hold the results of each front end, presentation generator, and back end. The output from each stage can be verified against known-good copies of the output files. Driver programs are compiled and linked separately. There is a Makefile in the root of the testing tree that simplifies the testing process, and a README file to explain everything.

`flick-2.1/util` This directory contains Flick utility programs.

`flick-2.1/util/aoid` This directory contains the source code for 'aoid', a pretty-printer for AOI files. When the AOI format is changed, this tool should be modified to reflect those changes. Given a 'filename.aoi', the 'aoid' program will create a human-readable 'filename.aod' file.

`flick-2.1/util/presd` This directory contains 'presd', a pretty-printer for PRES_C files. As PRES_C changes, this program should be changed as well. Given a 'filename.prc', the 'presd' program will create a human-readable 'filename.prd' file.

`libaoi` This directory contains data creation and manipulation routines for the AOI data format. The AOI library is used extensively in Flick's front ends and presentation generators.

`libcompiler` This directory contains utility code used throughout Flick, including error-checked malloc/calloc/realloc routines, uniform warning and error printing routines, and some code printing routines.

`libmeta` This directory contains routines for creating and manipulating the metadata within Flick's intermediate files.

`libmint` This directory contains data creation and manipulation routines for the MINT data format. It is used extensively throughout presentation generation and back end code generation.

`mom`	This directory contains the definitions of all of the ONC RPC IDL-specified intermediate data formats formats used internally by Flick (AOI, MINT, CAST, PRES_C, and META). These formats are specified in ONC RPC IDL because Flick needs to send these intermediate formats between separate programs, and Flick does that by writing data to intermediate files. Because the formats are specified in IDL, we can use 'rpcgen' to automatically create the code to read and write Flick's intermediate data representations. This directory also contains the header files for the libraries that manipulate Flick's non-C/C++-specific intermediate representations (i.e., libaoi, libmint, and libmeta).
`mom/c`	This directory contains header files that are specific to the C and C++ language-specific phases of Flick (i.e., the presentation generation and back end stages).
`mom/c/be`	This directory contains additional header files specific to Flick's C/C++ back ends.
`test`	This directory is the root of test tree. The structure of the test directory tree is patterned after the stages of the Flick compiler. There are separate directories to hold the results of each front end, presentation generator, and back end. The output from each stage can be verified against known-good copies of the output files. Driver programs are compiled and linked separately. There is a Makefile in the root of the testing tree that simplifies the testing process, and a README file to explain everything.
`util`	This directory contains Flick utility programs.
`util/aoid`	This directory contains the source code for 'aoid', a pretty-printer for AOI files. When the AOI format is changed, this tool should be modified to reflect those changes. Given a 'filename.aoi', the 'aoid' program will create a human-readable 'filename.aod' file.
`util/presd`	This directory contains 'presd', a pretty-printer for PRES_C files. As PRES_C changes, this program should be changed as well. Given a 'filename.prc', the 'presd' program will create a human-readable 'filename.prd' file.

2.3 Coding Style

Flick is not implemented in an extremely rigid style. Nevertheless, the code has been written using a few general rules and idioms, and these rules should be followed whenever one adds new code or modifies existing code.

2.3.1 Files

Obviously, new files should be placed in accordance with the existing directory hierarchy described in Section 2.2. File names should be chosen to match the file's existing peers; e.g., if the existing file names use underscores to separate words, the new file name should follow this pattern. Flick source files generally use the following file name extensions:

`.cc`	C++ source files.
`.hh`	C++-only header files.
`.c`	C source files.
`.h`	C header files. Note that C header files must be `#include`-able by C++ code. In general, this means that one uses preprocessor magic to ensure that whenever the file is included by C++ source, the header contents are wrapped within an `extern "C"` declaration.
`.idl`	CORBA IDL files.
`.x`	ONC RPC IDL files.
`.defs`	MIG IDL files.

Each `make`-able directory contains a `GNUmakefile.in` file that is processed by Flick's `configure` script to produce a `GNUmakefile` for building the directory contents. (Flick requires GNU Make.) Most "leaf" `GNUmakefiles` do little more than define a few macros. The `GNUmakerules.*` files at the root of the source tree contain the actual rules used to build various Flick libraries and programs. Consult the user's manual for further help in building Flick.

2.3.2 Aesthetics

When writing new code or modifying existing code, please follow these guidelines:

Maintain consistency. When modifying Flick in any way, the primary rule is to *make new code look like the code that's already there*. A corollary to the rule is this: do not reformat existing code. Keep the same declaration style, commenting style, indentation style, and so on as you write and modify code. Most any good, consistent coding style can be readable, but an *inconsistent* style creates Programmer's Hell. Gratuitous reformatting of existing code makes it difficult to tell from a `diff` what was actually changed and what is just cosmetic.

Comment the code. Source code is useless if people cannot understand it. Comment new code *as you write it*. Fix poorly commented code as you find it. Comments come in a variety of forms, but the preferred style is to have the initiators (`/*`) and terminators (`*/`) on separate lines with each line of text starting with an asterisk (`*`). The result looks clean, even though it can be troublesome to edit afterwards. Be careful not to use '`//`'-style comments in C code; not all C compilers accept such comments.

Avoid tricks; write portable code. Obvious code is best. Avoid tricks and non-portable idioms; in particular, be careful to avoid language extensions provided by `gcc` and `g++`. Use the GNU compiler options `-pedantic` and `-ansi` to weed out GNUisms in new code.

Treat compiler warnings as errors. Not only should your code compile on a variety of platforms with a variety of compilers, it should also compile *without any warnings*. A warning is a signal that something is wrong. A constant stream of warnings makes it too easy for new warnings to hide.

Use C for general-purpose libraries. The libraries for Flick's intermediate representations (AOI, MINT, and so on) are implemented in C so that they may be used in both C and C++ programs. The functions in `libcompiler` are implemented in C for the same reason.

Maintain language consistency. Some parts of Flick are implemented in C, and others are implemented in C++. However, it is confusing when a single component is implemented using both languages. When working on an existing component, use the language that is already in use for that component. When starting a new component, use the language that is common for components of that class.

Use the Flux Project coding style. The Flux coding style should be used in all C- and C++-based programs implemented by the Flux Project. The most obvious hallmark of this style is 8-character indentation stops — chosen to make `vi` users happy. A second hallmark is that braces are "hung" after `if`, `else`, `for`, and `switch`. Braces are not hung, however, around the bodies of functions, structure type definitions, and so on. The following code snippet illustrates these points of the Flux coding style:

```
int
make_payments(. . .)
{
        . . .
        for (i = 0; i < pmt_max; ++i) {
                if (i == pmt_num) {
                        skip_payment(pmts, i);
                        . . .
                } else {
                        double_payment(pmts, i);
                        . . .
                }
                print_current_payments(pmts);
        }
}
```

It is more important to maintain consistency with existing code than it is to apply the Flux coding style. In particular, this means that when "foreign" code is assimilated into Flick (e.g., bits of `rpcgen` for the ONC RPC front end, and bits of MIG for the MIG front end), *the formatting of the foreign code should be preserved.*

Use library routines as much as possible. In particular, avoid treating Flick's intermediate representation types (e.g., the various AOI and PRES_C node types) as "exposed" C structures whenever it is feasible and reasonable to do so. As much as possible, use the appropriate library functions to manipulate IR nodes as abstract data types. This makes the code easier to read and maintain. (This advice is somewhat idealistic. While it is generally straightforward to create nodes through a library interface, there is currently very little library support for accessing node data in any kind of abstract way.)

2.3.3 Storage Allocation

Although Flick-generated stubs and skeletons are careful to manage memory in a robust manner, Flick itself handles memory in a way that is more appropriate to a compiler.

Flick is written with the assumption that memory is plentiful, and this assumption is evident in two ways throughout the code. First, Flick makes frequent use of library routines that implement "allocate memory or panic" behavior (see Section 9.2). Thus, if memory runs out, Flick will exit ungracefully. Second, Flick is often not very careful in freeing memory. Memory may "leak" when the last pointer to a bit of previously allocated storage is lost.

In practice, this style of coding is not problematic for Flick: there is plenty of memory on modern computer systems, the compiler's memory demands are not large in any case, and the various Flick compiler passes run very quickly. It is certainly possible to find IDL inputs, however, that cause Flick to consume inordinate amounts of memory or CPU time. For this and other reasons, we expect to eventually "clean up" Flick's memory management code, either through recoding or through the addition of some kind of garbage collector.

2.3.4 Tags

Many functions in Flick take their arguments as "tag lists": lists of name-value associations. Each association consists of an identifier, representing some attribute, followed by the set of (zero or more) values for that attribute. A special tag identifier signals the end of the tag list. For instance, the function `add_function` accepts a list of tags describing the properties of a C++ function:

```
add_function(tl,
             sc_out_name,
             PFA_Protection, current_protection,
             PFA_Constructor,
             PFA_FunctionKind, "T_out(T_var &)",
             PFA_Scope, scope,
             PFA_ReturnType, cast_new_type(CAST_TYPE_NULL),
             PFA_Parameter, type, "r", NULL,
             PFA_TAG_DONE);
```

Each tag (the `PFA_*` identifiers) in the above example is paired with zero or more data values. Most tags have a single associated value; for example, `PFA_Protection` is paired with the value of the variable `current_protection`. Some tags, however, are associated with no values (e.g., `PFA_Constructor` in the example above) or with multiple values (e.g., `PFA_Parameter`). When a tag has no value, it usually signifies a boolean switch: the attribute is true when the tag appears in the tag list. The special `PFA_TAG_DONE` identifier in the above example signals the end of the tag list.

Tags are generally grouped into sets or "families" according to their purpose. For instance, all of the tag identifiers in the above example begin with `PFA` because they are all "presentation function attributes" as defined in `mom/c/libpres_c.h`. The documentation for any tag set should explain the meaning of each tag identifier and the values that each tag must be associated with. (For more information about the `PFA` tags, see Section 7.3.)

Essentially all functions that accept tag lists are `varargs`-style functions, meaning that they may be called with an arbitrary-length list of tags and values. This is why a special tag is required to mark the end of a tag list. Functions that accept tag lists generally do so because they construct objects (e.g., PRES_C nodes) or just because they have a large number of possible arguments. When used to construct objects, the tags generally correspond to slots or flags within the constructed object. Some subset of the possible tags may not be present in the argument list given to a function: for instance, in the above example, the `PFA_GlobalFunction` tag was not specified. The documentation for a function that takes a tag list should describe what tags are required (if any) and what happens when certain tags are not present in a given tag list.

Because tag lists are generally passed to `varargs`-style functions, *programmers must be extremely careful* to ensure that the right types and number of values are associated with each tag: the C compiler cannot typecheck a tag list! *Most notably, programmers must make sure to use `NULL` rather than `0` when passing a literal null pointer to a tags-based function.* On some architectures (e.g., 64-bit machines), a null pointer and an integer 0 have different representations, and may not even be represented by the same number of bits. In a varargs list, the compiler does not know what types of values are required for each tag, and thus the compiler cannot automatically convert zeros to null pointer values.

The notion of a tag lists can be extended to create entire object hierarchies. For example, the following statement uses tag lists to create a nest of four PRES_C nodes: a PRES_C_INLINE_ATOM containing a PRES_C_MAPPING_TEMPORARY, which contains a PRES_C_MAPPING_ARGUMENT, which contains a PRES_C_MAPPING_DIRECT. Tag lists are used to specify the hierarchy of nodes and fill in the nodes' data slots.

```
ac->length = PRES_C_I_ATOM,
               PIA_Index, 0,
               PIA_Mapping, PRES_C_M_TEMPORARY,
                 PMA_Name, "string_len",
                 PMA_CType, lctype,
                 PMA_PreHandler, "stringlen",
                 PMA_TempType, TEMP_TYPE_ENCODED,
                 PMA_Target, PRES_C_M_ARGUMENT,
                   PMA_ArgList, ac->arglist_name,
                   PMA_Name, "length",
                   PMA_Mapping, PRES_C_M_DIRECT, END_PRES_C,
                   END_PRES_C,
                 END_PRES_C,
               END_PRES_C;
```

The node types are specified by special PRES_C_* macros, which expand into calls to functions. Each function takes a tag list: PIA_* tags specify "PRES_C inline attributes" and PMA_* tags specify "PRES_C mapping attributes." The special tag END_PRES_C marks the end of each tag list.

The tag-based code makes it clear how the four nodes relate to each other. Moreover, the above tag-based statement is easier to read and maintain than the alternative: a block of C statements that create and initialize each node individually.

2.4 Summary and Comments

Flick began as a collection of C-only programs and libraries. Originally, individual presentation generators and back ends would "specialize" library code simply by providing new versions of certain functions. At compile time, the linker would include the specialized function in the generated program. This technique made it difficult to create a specialized version of a function that extended the behavior of the library routine; in object-oriented terms, it was impossible to invoke a parent method. Further, because libraries were simply collections of C functions (and types), it was not inherently obvious which sets of functions applied to which "objects" in the library.

For these reasons, most of Flick was converted to C++ in 1995. Some parts, notably the libraries for manipulating Flick's intermediate languages, remained in C. (This made it easier to write C programs based on these libraries.) The conversion was done in the most straightforward manner: certain C structure types were replaced with C++ classes, and functions on those structures were transformed into methods. Most of the existing C idioms (e.g., the use of stdio, malloc, and so on), however, were left in place. This approach was highly pragmatic: the existing C code worked, so there was no need to replace it; the programmers already knew the C idioms; and in 1995, many C++ implementations were notoriously buggy.

The result was that Flick became a combination of C and C++, programmed in a very C-like style. Largely, Flick has remained in this state, although newer parts of Flick take greater advantage of C++-specific idioms.

The overall design of the original C implementation has survived through Flick's conversion to C++ and through the many extensions that have been made since. The remaining chapters of this manual describe Flick's current design and implementation in detail. In places, this design is starting to show its age. Therefore, where appropriate, this manual will point out problems that have become apparent in the current design and implementation of Flick, and will suggest ways in which these problems might be corrected in the future.

Part II

The Intermediate Representations

Chapter 3

Metadata (META) Representation

To keep track of attributes that are associated with Flick's intermediate data structures, Flick stores *metadata* in the intermediate files that it creates (i.e., AOI and PRES_C files). Metadata allows Flick to track the sources and sinks (destinations) of objects, to describe relationships among those sources and sinks, to attach additional semantics or classifications to data, and so on. All of this information is stored in a format called META.

Metadata is used to classify objects in the intermediate files so that Flick can partition its output in various ways. For example, Flick may put certain types of data in one file and other types of data in another. Alternately, Flick might completely suppress the output of certain kinds of data. The META format is currently constrained to representing relatively simple attributes — e.g., the input files associated with various objects — but the META language may be extended in the future to support more complex kinds of annotations.

3.1 META Overview

The META data structures are defined in `mom/meta.x`, and the library for manipulating META data is contained in the directory `libmeta`.

META provides two basic abstractions: `io_files`, representing input files,[1] and `data_channels`, representing separate "streams" of data within a single file. These structures are used to classify data according to their source and purpose. An `io_file` is referenced by data structures that "arise" from the input file represented by the `io_file` object. A `data_channel` is used to classify the results generated by processing the input definitions. For example, multiple channels could be used to separate the client-side and server-side C declarations that result from a single IDL definition.

In addition to the basic `io_file` and `data_channel` structures, META defines structures for describing sets of files and channels. Sets are described by *masks*: patterns that may be used to find files and channels of interest.

3.2 META Data Structures

The primary metadata objects are contained within arrays which are collected under a root object of type `meta`:

```
struct meta {
```

[1] `io_files` are not currently used to represent output files.

```
        io_file         files<>;
        data_channel    channels<>;
};
```

A reference to an `io_file` or `data_channel` is implemented as an integer index into the appropriate array within a `meta` structure.

```
typedef int io_file_index;
typedef int data_channel_index;
```

Other META data structures, such as descriptions of file and channel sets, are not stored in the root `meta` structure but are instead stored where they are used.

3.2.1 Files

Files are tracked with the `io_file` structure:

```
struct io_file {
        string          id<>;           /* Filename */
        unsigned int    flags;          /* Holds the above flags */
        int             references;     /*
                                         * How many other files have included
                                         * this one
                                         */
        io_file_index   includes<>;     /* References to included files */
};
```

The `flags` field contains a set of the following values:

`IO_FILE_INPUT`
: Indicates that the `io_file` represents an input file.

`IO_FILE_OUTPUT`
: Indicates that the `io_file` represents an output file. *This flag is not currently used by Flick.*

`IO_FILE_BUILTIN`
: Indicates that the `io_file` does not correspond to any actual file. Rather, data from this "file" is already built into Flick.

`IO_FILE_ROOT`
: Indicates that this `io_file` represents the root IDL file from which all others are `#included`.

`IO_FILE_SYSTEM`
: Indicates that this `io_file` represents a system file (i.e., a system header file, generally `#included` with `<>`'s).

The main set of `io_file` objects is generated by a Flick front end when it processes a root IDL file. In addition, the front end will create any "builtin" files that it needs to indicate definitions that are known a priori to the front end. For example, the CORBA front end (described in Section 10.2) creates a builtin `io_file` to represent the source of the predefined `CORBA::Object` interface type. The set of `io_files` created by a front end encodes just about everything that Flick cares about from the files, from their names to their place(s) in the inclusion graph.

The `io_file_mask` structure describes a set of `io_file` objects:

```
struct io_file_mask {
        unsigned int    mask_flags;     /* Flags specific to the mask */
        string          id<>;           /* An identifier to match */
        unsigned int    set_flags;      /*
                                         * Match when all of these flags are
                                         * set in the io_file
                                         */
        unsigned int    unset_flags;    /*
                                         * Match when all of these flags aren't
                                         * set in the io_file
                                         */
};
```

This structure is supposed to be somewhat opaque; `libmeta` provides several functions for dealing with them. The construction of `io_file_masks` is handled by a tag list function (see Section 2.3.4) that sets all the fields and flags based on the tags passed in. For example, the following statement constructs an `io_file_mask` that matches only system files:

```
        meta_make_file_mask(FMA_SetFlags, IO_FILE_SYSTEM,
                            FMA_TAG_DONE);
```

The `meta_make_file_mask` function is fully described in Section 3.3, below.

3.2.2 Channels

Channels are described by the `data_channel` structure:

```
struct data_channel {
        io_file_index           input;          /* input for the channel */
        string                  id<>;           /* semantic id */
        unsigned int            flags;          /* Holds the above flags */
        io_file_index           outputs<>;      /*
                                                 * List of output files for
                                                 * the channel (unused)
                                                 */
};
```

Note that the `output` field is currently unused, because `io_files` are currently used only for input files. The possible flags within the `flags` field are these:

`DATA_CHANNEL_SQUELCHED`
: Indicates that the channel is squelched: nothing on this channel should be output.

`DATA_CHANNEL_DECL`
: Indicates that anything on this channel is a declaration.

`DATA_CHANNEL_IMPL`
Indicates that anything on this channel is a definition or implementation.

The purpose of a channel is to describe a data path for some object, thus allowing Flick to classify it and determine its origin and destination. This knowledge is primarily useful for squelching IDL definitions that should not appear in the output. For example, a Flick back end may need to suppress definitions and/or declarations that arise from IDL system header files: the definitions of system objects may be built into the targeted RPC/RMI runtime library. Similarly, a user might ask Flick to suppress definitions that come from `#included` IDL files, but to keep the corresponding declarations. This is useful when the included IDL files are intended to be compiled separately.

Similar to `io_files`, channels can be selected with the help of a mask structure:

```
struct data_channel_mask {
        unsigned int    mask_flags;     /* Flags for the mask */
        io_file_mask    *input;         /* Input file mask */
        string          id<>;           /* ID to match on */
        unsigned int    set_flags;      /*
                                         * Match when all of these flags are
                                         * set in the io_file
                                         */
        unsigned int    unset_flags;    /*
                                         * Match when all of these flags aren't
                                         * set in the io_file
                                         */
};
```

3.3 The META Library

The META library contains a relatively small number of functions of creating and manipulating META data structures. The source code for the library is in the `libmeta` directory, and the library header file is `mom/libmeta.h`. The various META library functions are summarized below.

`void init_meta(meta *m)`
Initialize the given `meta` structure. The structure is set to contain no files or channels.

`void check_meta(meta *m)`
Check that the given `meta` structure is well-formed. This function uses `assert` to verify the contents of the structure.

`void print_meta(meta *m, FILE *file, int indent)`
Pretty-print the given `meta` structure to the given file, using the specified initial indentation value.

The following functions are useful for manipulating files:

`io_file_index meta_add_file(meta *m, const char *id, int flags)`
If the given `meta` contains an `io_file` with the specified `id` and `flags`, return the index of that file. Otherwise, create a new `io_file` within `m`, initialize it, and return the index of the newly created file.

`io_file_index meta_find_file(meta *m, const char *id, int flags, int absolute_path)`

Find a file in the given `meta` structure. If `id` is null, then match only the `flags`; similarly, if `flags` is zero, match only on `id`. The `absolute_path` argument determines whether `id` is an absolute file name or just the file part (i.e., non-directory part) of the name. Return the index of the located `io_file` or `-1` if no such file exists.

`void meta_include_file(meta *m, io_file_index file, io_file_index included_file)`

Record the fact that the file at index `file` includes the file at index `included_file`.

`void meta_check_file(meta *m, io_file_index file)`

Check the integrity of the `io_file` at the given index in `m`. Errors are reported as `assert` failures (i.e., core dumps).

`void meta_print_file(meta *m, FILE *file, int indent, io_file_index idx)`

Pretty-print the `io_file` at the given index. The initial indentation level is specified by `indent`.

`io_file_mask meta_make_file_mask(int tag, ...)`

Create and return an `io_file_mask` structure, initialized according to the tag list given in the function arguments. (Tag list functions are described in Section 2.3.4.) A mask defines a *match set* of `io_file` patterns; a particular `io_file` can be compared to an `io_file_mask` to determine if the `io_file` meets the criteria specified by the mask, and is therefore a member of the mask's match set.

The tags and associated tag values for the `meta_make_file_mask` function are these:

`FMA_TAG_DONE`

(No required tag values.) Marks the end of the tag list.

`FMA_MatchesID`

(Required tag values: `char *`.) Specifies that the match set should *include* `io_files` whose complete file names match the given string. Matching is done with `strcmp` (i.e., exact string matching).

`FMA_ExcludesID`

(Required tag values: `char *`.) Specifies that the match set should *exclude* `io_files` whose complete file names match the given string.

`FMA_MatchesDirID`

(Required tag values: `char *`.) Specifies that the match set should *include* `io_files` containing file names with directory portions that match the given string.

`FMA_ExcludesDirID`

(Required tag values: `char *`.) Specifies that the match set should *exclude* `io_files` containing file names with directory portions that match the given string.

`FMA_MatchesFileID`

(Required tag values: `char *`.) Specifies that the match set should *include* `io_files` containing file names with file (non-directory) portions that match the given string.

`FMA_ExcludesFileID`

(Required tag values: `char *`.) Specifies that the match set should *exclude* `io_files` containing file names with file (non-directory) portions that match the given string.

`FMA_SetFlags`
: (Required tag values: `unsigned int`.) Specifies that the match set should include `io_files` in which the specified flags are set. Additional flags *may* be set in the `io_file`, but only the specified flags are *required* to be set.

`FMA_UnsetFlags`
: (Required tag values: `unsigned int`.) Specifies that the match set should include `io_files` in which the specified flags are unset.

An `io_file_mask` can specify at most one file name matching function: in other words, do not specify more than one of the above "`ID`" tags when creating a mask. In contrast, an `io_file_mask` can specify matching against both set and unset flags.

`int meta_match_file_mask(meta *m, io_file_mask *ifm, io_file_index file)`
: Compare the specified `io_file` (i.e., the file at index `file` in `m`) against the given `io_file_mask`. Return a non-zero value if the file matches the mask — meaning that the file is a member of the mask's match set — and zero if it does not match.

`void meta_squelch_file(meta *m, io_file_index file, data_channel_mask *dcm)`
: Squelch a set of data channels that come from the specified file *and from all files that are included by that file*. The squelched channels are those that match `dcm`. When a channel is "squelched," Flick will not produce output from data structures that refer to that channel.

`void meta_squelch_files(meta *m, io_file_mask *ifm, data_channel_mask *dcm)`
: Apply `meta_squelch_file` to all of the files in `m` that match `ifm`. The squelched channels are those that match `dcm`.

`void meta_print_file_mask(FILE *file, int indent, io_file_mask *ifm)`
: Pretty-print the given `io_file_mask`.

The following functions are useful for manipulating files:

`data_channel_index meta_add_channel(meta *m, io_file_index input, const char *id)`
: Create a new `data_channel` within `m`. The new channel has the given `id` and refers to the indicated `io_file`. Return the index of the newly created channel.

`data_channel_index meta_find_channel(meta *m, io_file_index input, const char *id, int flags)`
: Find a channel in the given `meta` structure with the given attributes, and return the index of the located channel. If no such channel exists, return `-1`.

`void meta_add_channel_output(meta *m, data_channel_index channel, io_file_index output)`
: Add an output file to the specified channel. *Because `io_files` are currently used only for input files, this function is unused.*

`void meta_check_channel(meta *m, data_channel_index channel)`
: Check the integrity of the `data_channel` at the given index in `m`. Errors are reported as `assert` failures (i.e., core dumps).

`void meta_print_channel(meta *m, FILE *file, int indent, data_channel_index channel)`

Pretty-print the `data_channel` at the given index. The initial indentation level is specified by `indent`.

`data_channel_mask meta_make_channel_mask(int tag, ...)`

Create and return a `data_channel` structure, initialized according to the tag list given in the function arguments. A mask defines a *match set* of `data_channel` patterns; a particular `data_channel` can be compared to a `data_channel_mask` to determine if the `data_channel` meets the criteria specified by the mask, and is therefore a member of the mask's match set.

The tags and associated tag values for the `meta_make_channel_mask` function are these:

`CMA_TAG_DONE`

(No required tag values.) Marks the end of the tag list.

`CMA_MatchesInput`

(Required tag values: `io_file_mask`.) Specifies that the match set should *include* `data_channels` that refer to input files matching the specified `io_file_mask`.

`CMA_ExcludesInput`

(Required tag values: `io_file_mask`.) Specifies that the match set should *exclude* `data_channels` that refer to input files matching the specified `io_file_mask`.

`CMA_MatchesID`

(Required tag values: `char *`.) Specifies that the match set should *include* `data_channels` whose `id` field matches the given string. Matching is done with `strcmp` (i.e., exact string matching).

`CMA_ExcludesID`

(Required tag values: `char *`.) Specifies that the match set should *exclude* `data_channels` whose `id` field matches the given string.

`CMA_SetFlags`

(Required tag values: `unsigned int`.) Specifies that the match set should include `data_channels` in which the specified flags are set. Additional flags *may* be set in the `data_channel`, but only the specified flags are *required* to be set.

`CMA_UnsetFlags`

(Required tag values: `unsigned int`.) Specifies that the match set should include `data_channels` in which the specified flags are unset.

`int meta_match_channel_mask(meta *m, data_channel_mask *dcm, data_channel_index channel)`

Compare the specified `data_channel` (i.e., the channel at index `channel` in `m`) against the given `data_channel_mask`. Return a non-zero value if the channel matches the mask — i.e., is in the mask's match set — and zero if it does not.

`void meta_squelch_channel(meta *m, data_channel_index channel)`

Mark the indicated channel as squelched. When a channel is "squelched," Flick will not produce output from data structures that refer to that channel.

`void meta_squelch_channels(meta *m, data_channel_mask mask)`
Squelch all of the channels in `m` that match the given `data_channel_mask`.

`void meta_print_channel_mask(FILE *file, int indent, data_channel_mask dcm)`
Pretty-print the given `data_channel_mask`.

3.4 Summary and Comments

In retrospect, it is apparent that the concept of a `data_channel` could be generalized to encompass the abilities of an `io_file` structure. Currently, the only real difference between the two structures is that a file can have multiple inputs — the included files — but that a channel can have only one. By allowing channels to have multiple inputs, Flick might be able to express the inclusion file graph as well as other dependencies between channels.

Chapter 4

Abstract Object Interface (AOI) Representation

The Abstract Object Interface (AOI, pronounced "ow-ee") is a set of data structures designed to represent parsed IDL files. Most of Flick's front ends — all but the MIG front end — convert IDL into AOI and then save the resultant AOI to disk. This provides a level of abstraction between Flick's front ends, which simply parse IDL, and Flick's presentation generators, which map interface specifications onto specific programming language constructs.

4.1 AOI Overview

The AOI format is intended to satisfy the following goals:

- **Cleanly represent object interfaces in abstract terms.** AOI was designed to represent object interfaces and IDLs in approximately the way humans design and think about them, irrespective of how they may boil down in implementation. For example, AOI supports separate notions of object methods, attributes, and exceptions, since people tend to think of these as separate concepts, even though all of these are likely to become simply "messages" eventually.

- **Support the features available in typical existing IDLs.** AOI should be well matched to many existing IDLs, partly in order to fulfill the above goal (examining a range of existing IDLs helps to identify "the way humans design and think about them") and partly to support the practical goal of being able to write front ends that convert human-readable IDL files into AOI format.

- **Leave out highly language-specific or presentation-related constructs.** Some IDLs are cluttered with features that are of little or no importance to the abstract RPC interface, such as pointer attributes in DCE or translation types in MIG. These "annotations" describe how an interface is to be mapped onto a programming language: in Flick, the goal is to separate the notions of abstract interface and language mapping, in order to provide maximum flexibility. Therefore, AOI is designed to represent interfaces in abstract terms only, excluding any presentation-level "cruft." (To preserve annotations in an IDL file, one must convert IDL directly into a representation that describes presentation: namely, PRES_C. This is how Flick's MIG front end operates; see Section 10.4 for further details.)

- **Create similar representations for equivalent IDL constructs.** By translating equivalent constructs from different IDLs into similar or identical AOI representations, AOI acts as a kind of multi-IDL pseudo-language. This makes writing later pieces of Flick much simpler. For instance, by making a

CORBA IDL `interface` appear similar to an ONC RPC IDL `version`, the code to map these notions to programming languages is also similar. This, in turn, helps to promote mix-and-match combinations of Flick's front ends and presentation generators. (See Section 2.1, however, for comments about mixed IDL/presentation combinations.)

Finally, because AOI acts as the interface between Flick's front ends and presentation generators, AOI was designed to be simple, and easily written to and read from disk.

4.2 AOI Data Structures

Because AOI is written to and read from files, the AOI format is defined using the ONC RPC IDL. The data structures are defined in `mom/aoi.x`; this file is processed by `rpcgen` when Flick is compiled[1] to produce functions that write and read AOI data to and from files. Other functions for manipulating AOI structures are defined in the AOI library, which is contained the `libaoi` directory.

The root of an AOI tree is a structure that holds a META data structure (as described in Chapter 3) and an array of AOI definitions corresponding to definitions given in the input IDL file(s):

```
struct aoi {
        aoi_def         defs<>;
        meta            meta_data;
};
```

The `meta_data` field holds the META information describing the input IDL files: i.e., the file names and the inclusion graph. The `defs` field is an array of associations between names and bindings, described below.

4.2.1 Definitions

An AOI definition — represented by an `aoi_def` structure — consists of a name, a scope identifier, a type that is bound to the name, and a reference (an `io_file_index`) to the source file for the definition. The file reference is an index into the META array of file descriptions as described in Chapter 3.

```
struct aoi_def
{
        string    name<>;
        int       scope;
        io_file_index idl_file;
        aoi_type  binding;
};
```

The `name` is a string, usually taken from the IDL file without any mangling. The `scope` is a number representing the lexical scope of the definition. In AOI, all `aoi_def` structures are contained in a single array: in other words, scopes are *not* represented by nested `aoi_def` structures. Rather, scoping is represented by keeping track of the lexical scoping depth of each definition, as illustrated in Figure 4.1. The `aoi_defs` are arranged so that the order of the IDL source definitions is preserved. A top level IDL definition has a scoping

[1]The '`.x`' files that define AOI and Flick's other intermediate languages are also processed by `rpcgen` when a Flick source distribution is created, so that users do not need to have `rpcgen` installed and working in order to compile Flick.

CORBA IDL	Corresponding `aoi_def` Values
`module MyMod_1 {`	`name = "MyMod_1", scope = 0, ...`
`  interface MyIntf_1 {`	`name = "MyIntf_1", scope = 1, ...`
`    typedef ... T_1;`	`name = "T_1", scope = 2, ...`
`    typedef ... T_2;`	`name = "T_2", scope = 2, ...`
`  };`	
`  interface MyIntf_2 {`	`name = "MyIntf_2", scope = 1, ...`
`  };`	
`};`	
`module MyMod_2 {`	`name = "MyMod_2", scope = 0, ...`
`};`	

Figure 4.1: Representation of IDL Scopes in AOI.

depth of zero. When a definition introduces a new scope, the AOI "current scope" depth is increased by one: thus, the `aoi_def`s within a scope will have `scope` values that are greater than the the `scope` value of the containing `aoi_def`. Obviously, when an IDL scope ends, the AOI "current scope" depth is decreased by one. This representation of scopes makes it easy to search for the "parent" of an `aoi_def`: simply search backward in the array of `aoi_def`s for a definition with a lower `scope` value, handling zero as a special case. Searching for peer and child definitions is also straightforward.[2]

The `binding` in an `aoi_def` refers to the definition's type, translated from the IDL file and represented by an `aoi_type`, to be described shortly.

Before moving on, note that an `aoi_ref` is a reference to an `aoi_def`, implemented as an index into the `aoi_def` array contained in the top-level `aoi` structure shown in Section 4.2. A null reference is represented by the special constant `aoi_ref_null`.

```
typedef int aoi_ref;
const aoi_ref_null = -1;
```

4.2.2 Fields

An `aoi_field` is similar to an `aoi_def` except it is meant for use in compound types: structures, unions, and so on. An `aoi_field` associates a name and an type, but does not need a scope identifier or source file reference.

```
struct aoi_field
{
        string          name<>;
        aoi_type        type;
};
```

4.2.3 Types

An IDL type is represented by an `aoi_type_u`: a discriminated union with branches that represent all IDL types. An `aoi_type` is a pointer to an `aoi_type_u`. All kinds of name bindings are represented as "types,"

[2] CORBA IDL makes searching for peer and child definitions somewhat more complex. For instance, because a `module` can be opened many times, there may be multiple `aoi_def`s corresponding to single module, and therefore one must search for all of those definitions.

AOI Type Kind	Description
AOI_INTEGER	an integer with a specified minimum value and range
AOI_SCALAR	an integer with range determined by # of bits
AOI_FLOAT	a floating-point type with a specified # of bits
AOI_CHAR	a character with flags and a specified # of bits
AOI_VOID	void
AOI_CONST	an IDL-defined constant value
AOI_ENUM	an enumeration (i.e., a set of symbolic constants)
AOI_ARRAY	an array, fixed- or variable-length
AOI_STRUCT	a structure containing named fields
AOI_EXCEPTION	a structure used to signal exceptions
AOI_UNION	a discriminated union
AOI_OPTIONAL	an optional value (i.e., possibly no value)
AOI_ANY	"any" value, *without* a type tag
AOI_TYPE_TAG	an opaque type identifier
AOI_TYPED	a pair: a type identifier and a value of that type
AOI_INTERFACE	an IDL-defined interface or object type
AOI_FWD_INTRFC	a forward interface or object declaration
AOI_NAMESPACE	a namespace for other definitions
AOI_INDIRECT	a reference to a named type (an `aoi_def`)
AOI_ERROR	*a parse error*

Table 4.1: Summary of the Available AOI Type Kinds.

including bindings to constant values and references to other bindings (i.e., usage of named types). The set of representable types is determined by the `aoi_kind` enumeration, summarized in Table 4.1. The `aoi_type_u` union has a branch or "arm" for each possible `aoi_kind`. The details of a particular AOI type definition are specified by further AOI structures, as shown below.

```
enum aoi_kind
{
        AOI_INDIRECT    = 1,
        ...
        AOI_ERROR       = 20
};

typedef struct aoi_type_u *aoi_type;

union aoi_type_u
switch (aoi_kind kind)
{
        case AOI_INDIRECT:      aoi_ref                 indirect_ref;
        case AOI_INTEGER:       aoi_integer             integer_def;
        case AOI_SCALAR:        aoi_scalar              scalar_def;
        case AOI_FLOAT:         aoi_float               float_def;
        case AOI_CHAR:          aoi_char                char_def;
        case AOI_ARRAY:         aoi_array               array_def;
        case AOI_STRUCT:        aoi_struct              struct_def;
        case AOI_UNION:         aoi_union               union_def;
        case AOI_INTERFACE:     aoi_interface           interface_def;
        case AOI_EXCEPTION:     aoi_exception           exception_def;
        case AOI_ENUM:          aoi_enum                enum_def;
        case AOI_VOID:          void;
        case AOI_CONST:         aoi_const_def           const_def;
        case AOI_NAMESPACE:     void;
        case AOI_OPTIONAL:      aoi_optional            optional_def;
        case AOI_FWD_INTRFC:    aoi_ref                 fwd_intrfc_def;
        case AOI_ANY:           void;
        case AOI_TYPE_TAG:      void;
        case AOI_TYPED:         aoi_typed               typed_def;
        case AOI_ERROR:         void;
};
```

The current AOI type kinds are described below, in the order in which they appear in Table 4.1.

Integers

An `AOI_INTEGER` represents an integer type with a specified (constant) minimum value and range. This type is used to represent all integer types whose values can be represented with not more that 32 bits. This includes unsigned and signed 32-, 16-, and 8-bit types, as well as booleans and certain integer "constants." (If the specified `range` is zero, then the `AOI_INTEGER` describes a type with only one possible value. This is useful for describing other types such as fixed-length arrays.)

```
struct aoi_integer
{
        int      min;
        unsigned range;
};
```

Scalars

An AOI_SCALAR describes an integer type by specifying the number of bits in its range. The minimum value is determined by flags that indicate whether the type is signed or unsigned. Scalars are only used to describe integer types that an AOI_INTEGER cannot represent (e.g., 64-bit and 128-bit integers).

```
typedef u_int aoi_scalar_flags;
const AOI_SCALAR_FLAG_NONE      = 0;
const AOI_SCALAR_FLAG_SIGNED    = 1;
const AOI_SCALAR_FLAG_UNSIGNED  = 2;

struct aoi_scalar
{
        int              bits;
        aoi_scalar_flags flags;
};
```

Floats

An AOI_FLOAT describes a floating point type in terms of the number of bits in the encoding of the type.

```
struct aoi_float
{
        int bits;
};
```

Characters

An AOI_CHAR describes a character type. Again, the range of the type is described by the number of bits required to encode values of the type (generally 8). In addition, a set of flags describe additional properties, such as whether the characters are "signed" or "unsigned."

```
typedef u_int aoi_char_flags;
const AOI_CHAR_FLAG_NONE     = 0;
const AOI_CHAR_FLAG_SIGNED   = 1;
const AOI_CHAR_FLAG_UNSIGNED = 2;

struct aoi_char
{
        int            bits;
        aoi_char_flags flags;
};
```

Voids

An `AOI_VOID` represents a void type: a type with no values.

Constants

An `AOI_CONST` corresponds to a literal constant value. (An `aoi_def` can be used to describe a *symbolic* constant.) The type of the constant is described by an `aoi_type` and is also manifest in the `aoi_const` value itself. AOI currently supports integer, character, and floating-point literals, and arrays and structures built from these literals.

```
typedef long aoi_const_int;
typedef char aoi_const_char;
typedef double aoi_const_float;
typedef aoi_const aoi_const_struct<>;
typedef aoi_const aoi_const_array<>;

enum aoi_const_kind
{
        AOI_CONST_INT           = 1,
        AOI_CONST_CHAR          = 2,
        AOI_CONST_FLOAT         = 3,
        AOI_CONST_STRUCT        = 4,
        AOI_CONST_ARRAY         = 5
};

union aoi_const_u
switch (aoi_const_kind kind)
{
        case AOI_CONST_INT:     aoi_const_int           const_int;
        case AOI_CONST_CHAR:    aoi_const_char          const_char;
        case AOI_CONST_FLOAT:   aoi_const_float         const_float;
        case AOI_CONST_STRUCT:  aoi_const_struct        const_struct;
        case AOI_CONST_ARRAY:   aoi_const_array         const_array;
};

struct aoi_const_def
{
        aoi_type        type;
        aoi_const       value;
};
```

Enumerations

An `AOI_ENUM` represents an enumerated data type, i.e., a set of symbolic constants. The set is represented as an array of `aoi_fields`, which are name-value pairs. The associated values, of course, are expected to be constants, not arbitrary AOI types. The `enum_label` contains any IDL-specified label for the enumeration.

```
struct aoi_enum
```

```
{
        string           enum_label<>;
        struct aoi_field defs<>;
};
```

Arrays

An AOI_ARRAY describes an array type: this includes fixed-length arrays, variable-length arrays (what some IDLs call "sequences"), and NUL-terminated strings. The length of an array is specified as an AOI_INTEGER, thus giving the minimum and maximum lengths of instances of the array type. The AOI_ARRAY description also refers to the type of the array elements, as shown in the code.

```
typedef u_int aoi_array_flags;
const AOI_ARRAY_FLAG_NONE                        = 0x00000000;
const AOI_ARRAY_FLAG_NULL_TERMINATED_STRING      = 0x00000001;
const AOI_ARRAY_FLAG_OPAQUE                      = 0x00000002;
const AOI_ARRAY_FLAG_ALL                         = 0x00000003;

struct aoi_array
{
        aoi_type            element_type;
        aoi_type            length_type;

        aoi_array_flags     flgs;
};
```

Additional properties of the type are specified as a set of flags in the flgs field of the aoi_array structure. The AOI_ARRAY_FLAG_NULL_TERMINATED_STRING flag indicates that the array type represents a NUL-terminated string type. At one time, AOI_ARRAY_FLAG_OPAQUE was used to indicate that the array elements should be packed together when encoded. This flag is no longer meaningful, since data encoding decisions are made elsewhere (i.e., by back ends, based on the element types of MINT arrays).

Structures

An AOI_STRUCT represents a compound data type with a fixed number of named fields (also known as slots or data members). The structure type is represented as an array of (zero or more) slot descriptors, which give the name and type of each slot. The order of the slots is preserved from the IDL source definition of the structure type.

```
typedef aoi_field aoi_struct_slot;
struct aoi_struct
{
        /* Type of each structure slot, in order.  */
        aoi_struct_slot slots<>;
};
```

Exceptions

An AOI_EXCEPTION type is similar to a regular structure type, but with an additional semantic: an AOI_EXCEPTION represents an IDL-specified exceptional runtime condition and its associated data (if any). Because IDLs distinguish between normal structure types and exception types, AOI makes this distinction as well.

```
typedef aoi_field aoi_exception_slot;
struct aoi_exception
{
        aoi_exception_slot slots<>;
};
```

Unions

An AOI_UNION represents a *discriminated* union: a compound data type containing a discriminator that selects from a set of possible "arms" (also called cases, branches, or variants). The value of the discriminator determines which one of the possible union arms is valid.

Each arm is represented by an aoi_union_case containing a discriminator value — the value that selects the arm — as well as a name and type for the arm. (The AOI representation of constant values will be described shortly.)

The union as a whole is essentially represented as a kind of structure with a distinguished discriminator member, an array of aoi_union_cases representing the arms of the union, and an optional "default" arm that is selected when none of the ordinary union cases are selected. If there is no default branch, then the dfault slot of the aoi_union will be null. If there is a default case and that default case is void, then dfault must refer to a valid aoi_field of type AOI_VOID. Finally, the union_label slot holds the IDL-specified name (if any) of the union arms as a whole. This may be used when mapping the discriminated union type into an appropriate C/C++ representation (e.g., a struct with a slot that is a union of the AOI-described union branches).

```
struct aoi_union_case
{
        aoi_const       val;
        aoi_field       var;
};

struct aoi_union
{
        aoi_field      discriminator;
        string         union_label<>;
        aoi_union_case cases<>;
        aoi_field     *dfault;
};
```

Optional Values

An AOI_OPTIONAL represents a "maybe value" type: a type that allows for empty values. Intuitively, an AOI_OPTIONAL corresponds to a pointer type that may be null.

In common IDLs, optional data can be described in several ways: as a variable-length array of zero or one elements, as a discriminated union with a void variant, or as a denoted optional element. (The ONC RPC IDL provides "*" for denoting optional data. CORBA IDL does not provide an equivalent construct.) These techniques for "optionality" are logically equivalent, but each IDL idiom generally results in a different presentation — that is, a different set of generated programming language types. Thus, no technique for describing optional data is redundant, and AOI must describe them all. `AOI_OPTIONAL` corresponds to denoted optional data types; i.e., the ONC RPC IDL "*" construct.

```
struct aoi_optional
{
        aoi_type type;
};
```

Anys

An `AOI_ANY` represents "any type," the type of all possible values. Because `AOI_ANY` is all-encompassing, there is no additional data associated with an `AOI_ANY`.

Unlike the `any` types specified by some IDLs, an `AOI_ANY` does *not* contain an implicit type tag. The notion of a type-tagged value is represented by a separate AOI notion called `AOI_TYPED`. AOI distinguishes between tagged and untagged "any" values in order to provide flexibility.

Type Tags

An `AOI_TYPE_TAG` represents an opaque indicator of a type. In other words, the value of a type tag is a description of type. Tags are opaque in the sense that the compiler portions of Flick do not provide, understand, or enforce any semantics for type tags. Other than marshaling and unmarshaling tag instances, Flick does nothing with tags. Semantics must be provided by runtime libraries.

Since type tags are opaque, an `AOI_TYPE_TAG` has no attributes.

Typed Values

Combining the two notions above, an `AOI_TYPED` represents a pair containing a type description and a value of the described type. In other words, an `AOI_TYPED` represents type-tagged values. The `tag` and `type` fields of an `aoi_typed` structure may conceptually refer to any AOI type, but in practice, the former is always an `AOI_TYPE_TAG` and the latter is always an `AOI_ANY`.

```
struct aoi_typed
{
        aoi_type        tag;
        aoi_type        type;
};
```

Interfaces

An `AOI_INTERFACE` represents an IDL-defined interface or object type. AOI's notion of an interface is flexible enough to represent CORBA `interface` declarations and ONC RPC `version` declarations, and should be flexible enough to represent the corresponding notions of other IDLs as well.

An `AOI_INTERFACE` supports separate notions for attributes ("data members") and operations ("function members"). Interface operations are stored in `aoi_operation` structures, which are contained within the `aoi_interface` structure shown below. Each `aoi_operation` contains several bits of information: the operation name, its request and reply message identifiers, the parameters, exceptions that may be raised, and various flags. The AOI data structures for representing these operation data are shown in the code below. Most of these are what one would expect, except perhaps for the request and reply message identifiers. These are intended to be IDL-specified "suggestions" to the presentation generator and back end stages of Flick. These later stages are responsible for deciding the actual identifiers and codes used to implement an interface. A presentation generator and back end can use identifiers that come from the IDL (as passed through AOI), but are not *required* to use them.[3]

Interface attributes are stored separately from general operations and are represented as special facilities for reading and writing and interface's "data members." An `aoi_attribute` specifies four separate codes: two for reading (request and reply) and two for writing. The type of the interface attribute is stored in the `aoi_attribute` structure.

Note that all of an interface's operation and attribute request and reply codes must be unique. Moreover, the request and reply codes for a single operation or attribute cannot be identical. These restrictions ease the translation of AOI into MINT during presentation generation. (See Section 11.2.1 for details.) In addition, because operations and attributes are quite similar, some presentation generators may choose to translate `aoi_attributes` into various `aoi_operations` for accessing the attributes.

```
enum aoi_direction
{
        AOI_DIR_IN      = 1,
        AOI_DIR_OUT     = 2,
        AOI_DIR_INOUT   = 3,
        AOI_DIR_RET     = 4
};

struct aoi_parameter
{
        string          name<>;
        aoi_direction   direction;
        aoi_type        type;
};

typedef u_int aoi_op_flags;
const AOI_OP_FLAG_NONE          = 0x00000000;
const AOI_OP_FLAG_ONEWAY        = 0x00000001;
const AOI_OP_FLAG_IDEMPOTENT    = 0x00000002;
const AOI_OP_FLAG_SETTER        = 0x00000004;
const AOI_OP_FLAG_GETTER        = 0x00000008;

struct aoi_operation
{
        string          name<>;

        /*
```

[3] In practice, the current scheme for tracking IDL-specified identifiers has proven to be too restrictive in some circumstances. This is briefly discussed in Section 4.5.

```
	 * Code uniquely identifying this operation within this interface;
	 * works the same way as the code_type and code in aoi_interface.
	 */
	aoi_const	request_code;
	aoi_const	reply_code;

	aoi_op_flags	flags;

	/* Parameters to/from this operation.  */
	aoi_parameter	params<>;

	/* Return type, if any. */
	aoi_type	return_type;

	/*
	 * Exceptions that can be raised by this method.  These must be
	 * AOI_INDIRECTS so that we can access the exceptions' names.
	 */
	aoi_type	exceps<>;
};

struct aoi_attribute
{
	string	name<>;

	/* Attribute code/identifier.  */
	aoi_const read_request_code;
	aoi_const read_reply_code;
	aoi_const write_request_code;
	aoi_const write_reply_code;

	/* Type of this attribute.  */
	aoi_type  type;

	/* Is the attribute readonly */
	bool	readonly;
};

struct aoi_interface
{
	/* IDL this interface was defined in - "top-level" interface ID.  */
	aoi_idl_id	idl;

	/*
	 * 'code_type' is the type of the identifying interface, e.g. a
	 * char-array (string) for CORBA, a number for MIG, a struct for DCE
	 * uuid's, etc.  'code' is the actual code identifying this interface.
```

```
         */
        aoi_type        code_type;
        aoi_const       code;

        /* Other interfaces this inherits from. */
        aoi_type        parents<>;

        /* Type of the identifying code of each attribute/operation.  */
        aoi_type        op_code_type;

        /* Operations and attributes accessible through this interface. */
        aoi_operation   ops<>;
        aoi_attribute   attribs<>;

        /* Exceptions that can be raised by any method in this interface.  */
        aoi_type        excepts<>;
};
```

Note that the `idl` and `excepts` slots of an `aoi_interface` are not currently used for anything useful. (The `idl` slot is occasionally used to work around certain translation problems in the presentation generator and back end stages of Flick, but all of these cases should be handled in more principled ways.)

Forward Interfaces

An `AOI_FWD_INTRFC` is essentially a placeholder for an interface or object type that has been named but not yet defined. After the input IDL has been parsed, a front end must make a cleanup pass over the AOI in order to attach `AOI_FWD_INTRFC` definitions to the corresponding "real" `AOI_INTERFACE` definitions. In some IDLS (e.g., CORBA IDL), it is an error to forward-declare and interface and then not define it.

Namespaces

An `AOI_NAMESPACE` describes a namespace in the input IDL file(s). These are used, for instance, to represent CORBA `module` constructs. At present, `AOI_NAMESPACE`s are not entirely suitable for describing ONC RPC `programs` because an `AOI_NAMESPACE` provides nowhere to store the program identifier. See Section 10.3 for details.

Indirect References

An `AOI_INDIRECT` represents a reference to another `aoi_def`, i.e., a reference to a name binding. An `AOI_INDIRECT` is used whenever an IDL source construct refers to a named type — e.g., a previously `typedef`'ed type — or to a named constant, or to any other kind of name binding.

The `indirect_ref` value within an `AOI_INDIRECT` in the index of the `aoi_def` that is being referenced. (Remember that all AOI definitions are stored in a single array.)

Errors

Finally, `AOI_ERROR` is a placeholder used by Flick's front ends to handle IDL parse errors in a graceful manner. Instances of `AOI_ERROR` should never be written to an AOI file.

4.3 The AOI Library

Common functions for creating and manipulating AOI data structures are contained in Flick's AOI library. The source code for the library is in the `libaoi` directory, and the library header file is `mom/libaoi.h`. The library functions are summarized below.

`aoi_const aoi_new_const(aoi_const_kind kind)`
: Return an `aoi_const` that points to a newly allocated `aoi_const_u` structure initialized with the given AOI constant `kind`.

  In the structure returned by `aoi_new_const`, any additional fields for describing the constant are uninitialized and must be set by the caller. For this reason, it is often more convenient to call one of the following, more specific constant constructors:

  `aoi_const aoi_new_const_char(char c)`
  : Return an `aoi_const` that points to a new `aoi_const_u` describing the constant character `c`.

  `aoi_const aoi_new_const_int(int val)`
  : Return an `aoi_const` that points to a new `aoi_const_u` describing the constant integer `val`.

  `aoi_const aoi_new_const_float(double val)`
  : Return an `aoi_const` that points to a new `aoi_const_u` describing the constant floating-point `val`.

  `aoi_const aoi_new_const_string(const char *s)`
  : Return an `aoi_const` that points to a new `aoi_const_u` describing the constant string `s`. Note that a string is represented as an array of characters, *without* a terminating NUL.

  `aoi_const aoi_new_const_string_cat(const char *s1, const char *s2)`
  : Return an `aoi_const` that points to a new `aoi_const_u` describing the concatenation of `s1` and `s2`.

`int aoi_const_eq(aoi_const c1, aoi_const c2)`
: Returns `1` if the two constants are equal (i.e., structurally identical, representing the same constant value) and `0` if they are not.

`void aoi_get_int_size(aoi_integer *ai, int *out_bits, int *out_is_signed)`
: Determine the number of bits (1, 8, 16, or 32) required to represent the given `aoi_integer` type. In essence, this function converts a description of an integer range into a bits/signedness representation. The required number of bits is stored in `*out_bits` and the signedness indicator (zero or one) is stored in `*out_is_signed`.

`void aoi_get_array_len(aoi *a, aoi_array *at, unsigned *min, unsigned *max)`
: Determine the minimum and maximum lengths of the given `aoi_array` type. The minimum length is stored in `*min` and the maximum is stored in `*max`. (If the lengths are equal, then the array is fixed-length.) Note that the entire `aoi` structure must be passed to this function, because the length of the array may be an `AOI_INDIRECT` to another AOI type definition.

`aoi_ref aoi_get_parent_scope(aoi *the_aoi, aoi_ref ref)`
: Return the index (`aoi_ref`) of the AOI definition that started the scope in which the definition at `ref`

is contained. More simply put, find the definition of the parent scope of `ref`. If `ref` refers to a top-level definition (i.e., a definition in scope zero), then the return value is the constant `aoi_ref_null`.

`char *aoi_get_scoped_name(aoi_ref ref, const char *separator)`

Compute and return the fully scoped name of the AOI definition referenced by `ref`, according to the AOI representation of scopes (described in Section 4.2.1). The name components are separated by the `separator` string.

`int aoi_def_find_member(aoi *in_aoi, aoi_def *d, const char *name, aoi_def **out_d, int *out_memindex)`

Search through the given `aoi_def d` for any members that have the given `name`. The kinds of definitions that contain named members are `AOI_STRUCT`, `AOI_UNION`, `AOI_EXCEPTION`, `AOI_ENUM`, and `AOI_INTERFACE`. In the case of interfaces, parents are also searched for the given `name`.

If a matching member is found, this function returns true (`1`) and sets `**out_d` and `*out_memindex` to refer to the location of the member. `**out_d` is set to point to the `aoi_def` in which the matching member was found; this will always be the same as the original definition except when a member is located in a parent interface. `*out_memindex` is set to the index of the member in the array of named elements in the definition (e.g., the array of slots in an `AOI_STRUCT`, or the array of values inside an `AOI_ENUM`). Because interfaces have two lists of names — operations and attributes — `*out_memindex` may be set specially. If the matching member is an attribute, the `*out_memindex` is offset by the number of operations in the interface. For example, if there were five operations (at array indices zero through four), a `*out_memindex` value of six would correspond to the second attribute.

Of course, if no matching member is found, `aoi_def_find_member` returns false (`0`).

`int aoi_def_has_member(aoi *in_aoi, aoi_def *d, const char *name)`

Returns true if the given `aoi_def` contains a member with the given name. This is a simplified interface to the `aoi_def_find_member` function described above.

`aoi_type aoi_indir(aoi *a, aoi_type t)`

Follow a chain of `AOI_INDIRECT` type definitions, starting at `t`, to determine the underlying "real" type. Return the located `aoi_type` (a pointer to the `aoi_type_u` structure contained in the terminal `aoi_def`). Note that if `t` is not an indirect definition, then `t` is its own underlying type.

`aoi_type aoi_indir_1(aoi *a, aoi_type t)`

Follow a single level of an `AOI_INDIRECT` chain; in other words, "dereference" `t` and return the referenced `aoi_type`. If `t` is not an indirect definition, then this function simply returns `t`.

`aoi_ref aoi_deref_fwd(aoi *a, aoi_ref fwd)`

Locate and return the index of the `AOI_INTERFACE` definition corresponding to the `AOI_FWD_INTRFC` definition at AOI index `fwd`. If the "real" interface definition cannot be found, this function returns `aoi_ref_null`.

(If `fwd` refers to an actual `AOI_INTERFACE`, then `fwd` is returned. If `fwd` refers to a non-interface type, then `aoi_ref_null` is returned.)

This function is primarily useful for front ends only. Other stages of compilation can simply refer to the `aoi_ref` contained within an `AOI_FWD_INTRFC` definition in order to find the actual definition of an interface.

`void aoi_check(aoi *dest)`
: Check the validity of the given `aoi` structure. *At present, this is largely unimplemented.*

`void aoi_readfh(aoi *dest, FILE *fh)`
: Read an `aoi` structure from `fh` into `dest`. (The disk data is stored in XDR format, and the actual I/O is performed by functions that are generated by `rpcgen` from the `mom/aoi.x` file.) If input fails for some reason, this function invokes `panic`, which causes the process to print an error message and exit.

`void aoi_writefh(aoi *dest, FILE *fh)`
: Write the given `aoi` structure to `fh`. (As before, the data in the file is stored in XDR format, and the actual I/O is performed by `rpcgen`-generated functions.) This function invokes `aoi_check` to verify the validity of the AOI data before writing the data to disk. If output fails for some reason, this function calls `panic`.

`void translate_aoi_to_mint()`
: Convert a set of AOI definitions into an equivalent MINT (Message Interface) representation. MINT is the intermediate language for describing, in an abstract way, the messages that will be exchanged between clients and servers. Chapter 5 describes MINT in detail, and Chapter 11 describes Flick's presentation generators, which invoke the AOI-to-MINT translator.

4.4 `aoid`, the AOI Printing Utility

The `aoid` program is a simple pretty-printer that takes AOI input from a file or standard input and outputs a human-readable, ASCII translation. If the AOI input comes from a file, the output is a corresponding '`.aod`' (AOI dump) file. Otherwise, if the AOI input comes from `stdin`, the human-readable output is sent to `stdout`. Note that as the AOI format changes, this program must also be changed to support any new constructs, or modifications to existing ones.

4.5 Summary and Comments

Because AOI has a relatively simple purpose, the AOI format has been quite stable. When features are added, it is generally to support new (or previously unsupported) IDL constructs: only rarely has AOI been changed in order to support the needs of presentation generators. Still, there are several ideas for improving AOI and its library, principally to support Flick's later stages of compilation:

Uniform Handling of Names. Not all name bindings are represented as `aoi_def` structures. Most notably, operation and attribute names are stored within an interface type, not within the top-level `aoi_def` array. The same problem applies to structure, union, enumeration, and exception members. The non-uniform handling of names makes it more difficult to locate and refer to certain entities; AOI would likely be improved if all names were handled in the same way.

Multiple Names. Some IDLs provide multiple names for entities. For example, in ONC RPC IDL, a `program` has both a name and a code. ONC RPC `versions` and operations also have both names and codes. These cases are handled specially in AOI, but in general, AOI should be designed to support multiple identifiers (in multiple identifier spaces) for all nameable objects. This would be useful not only for ONC RPC IDL, but also for CORBA IDL (which has both IDL-given names and "repository identifiers") and MIG IDL (in which a `subsystem` has both a name and a "message base identifier").

Note that if AOI were improved in this way, then MINT would need to be extended in a similar way. This would in fact solve an outstanding problem with MINT: namely, that the top levels of the MINT tree are too specific. Interface and operation identifiers in MINT are too specific, are singular, and are chosen by the presentation generator. This makes it difficult for back ends to construct on-the-wire message identifiers appropriate to their message formats.[4] This problem could largely be solved by allowing AOI to keep multiple names, and extending MINT to preserve the names as "hints" to the presentation generator and back end.

Annotations. Experience has shown that AOI's "purity" is practically inconvenient. It is useful and necessary to distinguish interface specification from presentation-level "cruft" when designing and implementing a flexible IDL compiler such as Flick. However, the decision to exclude all presentation-level details from AOI — and to represent only "pure" interface specifications — means that IDLs with annotations cannot be supported.

An extended AOI format with support for annotations would be useful for a number of interesting projects. Immediately, it would make it possible for one to rearchitect Flick's MIG front end and presentation generator to use AOI as an intermediate format. Later, an annotatable AOI would make it easier for people to experiment with annotated IDL specifications, e.g., specifications with embedded quality-of-service constraints).

`AOI_INTEGER` Versus `AOI_SCALAR`. Obviously, these two types should be unified in some way.

Using `rpcgen`. See Section 2.1 for general comments about using ONC RPC IDL to define Flick's intermediate languages. The AOI library in particular needs more functions that abstract the underlying AOI data structures. (Some of these functions already exist in the individual front ends, and should be relocated into `libaoi`.)

[4] The ONC/TCP back end, for example, goes to painful lengths to generate `program/version`-style message identifiers for operations defined in "foreign" IDLs.

Chapter 5

Message Interface (MINT) Representation

The Message Interface (MINT) intermediate language is designed to represent a collection of RPC/RMI interfaces in terms of the messages — requests and replies — that will be exchanged between clients and servers. Like AOI, MINT descriptions are *abstract*: not concerned with particular message formats, data encoding schemes, or stub styles. These concrete notions are determined by Flick's presentation generators and back ends, *after* the MINT representation of the interfaces has been built.

MINT does not generally stand on its own, because by itself, MINT does not carry enough information to completely represent an interface. Rather, MINT is a part of the more expressive and complete PRES_C language. MINT simply describes collections of abstract messages; other parts of PRES_C attach meanings to those messages, thus creating a complete representation language for interface description.

5.1 MINT Overview

MINT represents a collection of interfaces as a tree, as illustrated in Figure 5.1. (In truth, MINT represents graphs, not trees, but the "top level" of the graph is always a tree.) At the root of the tree is a discriminated union, representing the collection of different IDLs. The separate branches of the union correspond to different IDLs: CORBA, ONC RPC, MIG, or other IDLs to be supported. Although MINT was designed to support interfaces from multiple IDLs at the same time, but in current practice, only one IDL is represented by any one MINT tree.

At the next level of the tree, a node represents a collection of defined interfaces or object type. Here, each branch corresponds to a different interface type. At the third level, a MINT node represents the collection of messages, both requests and replies, that are part of the interface. For each request, there is a fourth-level node (a `MINT_STRUCT`) that describes the request parameter types. For each reply, there is a fourth-level node (a `MINT_UNION`) that describes the types of the reply data. A reply is a union because it represents both normal and exceptional reply messages.

Beyond the fourth level of the tree, MINT describes the types of individual message elements, much as AOI describes the types of IDL operation parameters. Mapping from the AOI representation of an interface and its operations to the corresponding MINT representation is straightforward. Since both AOI and MINT are abstract description languages, the translation can in fact be implemented by a library function shared by all of Flick's presentation generators.[1] The translation from AOI to MINT is "lossy," however: some of the information in the AOI description of an interface or IDL-defined type is lost. This is because MINT only

[1] Currently, Flick provides certain hooks so that presentation generators can "tune" the union discriminator values that appear in the top levels of the MINT tree. This means that the translation from AOI to MINT is not entirely independent of the presentation generator being used. This should be fixed in the future, perhaps through support for flexible naming in AOI and MINT as described in Section 4.5.

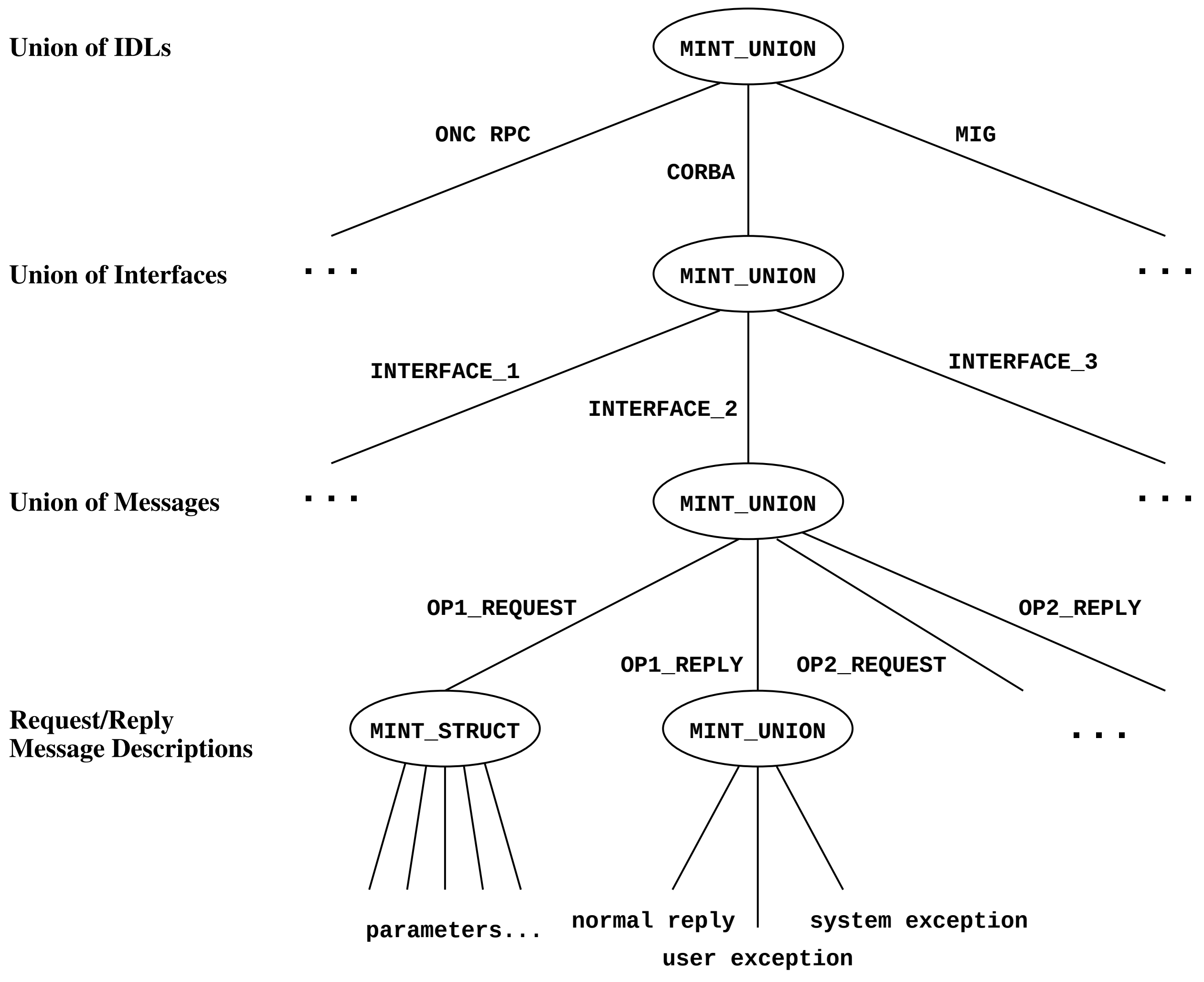

Figure 5.1: The Message Interface (MINT) Tree.

represents the *structure* or *syntax* of messages that will be exchanged between clients and servers. *Semantic* notions — names (type names, structure slot names, ...), inheritance, exceptions, and so on — that give meanings to messages are discarded when AOI is translated into MINT. These semantics are "recovered" when the presentation generator makes a second pass over the AOI while building the PRES_C representation of the types, stubs, and skeletons that will ultimately implement the interface.

5.2 MINT Data Structures

Like AOI, MINT must be written to and read from files, and it therefore defined using the ONC RPC IDL. The data structures are defined in `mom/mint.x`.

The top-level MINT data structure is called `mint_1` and contains two slots. The first slot, `defs`, contains a MINT type graph, represented as a single array of type definitions, with each definition represented by a `mint_def` structure. MINT definitions will be described shortly, in Section 5.2.1. Unlike AOI, there is no semantic significance to the order of the definitions in the array.

The second slot in a `mint_1` is called `standard_refs` and contains a references to certain "well known" and common MINT type definitions. It is convenient to keep these references around because certain basic types are used again and again. A reference to a MINT type definition is a `mint_ref`, which is implemented as an index into the `mint_1` array of `mint_defs`. A null reference is represented by the special `mint_ref_null` value.

```
typedef int mint_ref;
const mint_ref_null = -1;

struct mint_1
{
        mint_def                defs<>;

        /* ‘standard_refs’ are initialized by ‘mint_add_standard_defs’. */
        mint_standard_refs      standard_refs;
};

/*
 * A ‘mint_standard_refs’ contains ‘mint_ref’s for commonly-used MINT types.
 * It is useful to have immediate access to these basic building blocks.
 */
struct mint_standard_refs
{
        mint_ref void_ref;

        mint_ref bool_ref;
        mint_ref signed8_ref;
        mint_ref signed16_ref;
        mint_ref signed32_ref;
        mint_ref signed64_ref;
        mint_ref unsigned8_ref;
        mint_ref unsigned16_ref;
        mint_ref unsigned32_ref;
```

AOI Type Kind	Description
MINT_INTEGER	an integer with a specified minimum value and range
MINT_SCALAR	an integer with range determined by # of bits
MINT_FLOAT	a floating-point type with a specified # of bits
MINT_CHAR	a character with flags and a specified # of bits
MINT_BOOLEAN	true or false
MINT_VOID	void
MINT_ARRAY	an array, fixed- or variable-length
MINT_STRUCT	a collection of unnamed, unordered fields
MINT_UNION	a discriminated union
MINT_ANY	"any" value, *without* a type tag
MINT_TYPE_TAG	an opaque type identifier
MINT_TYPED	a pair: a type identifier and a value of that type
MINT_INTERFACE	a reference to an object instance
MINT_SYSTEM_EXCEPTION	an opaque, system-generated exception

Table 5.1: Summary of the Available MINT Type Kinds.

```
        mint_ref unsigned64_ref;

        mint_ref char8_ref;
        mint_ref float32_ref;
        mint_ref float64_ref;

        mint_ref interface_name_ref;
        mint_ref interface_invoke_ref;
        mint_ref interface_invoke_once_ref;
        mint_ref interface_service_ref;
        mint_ref system_exception_ref;
};
```

5.2.1 Type Definitions

A MINT type definition is implemented by a discriminated union of the possible type kinds, which are summarized in Table 5.1. The details of individual MINT types are specified by further MINT structures as shown in the code below.

It is important to note the differences between an `aoi_def` (described in Section 4.2.1) and a `mint_def`. Unlike an `aoi_def`, a `mint_def` does not include a name, a scope, or any other "semantic" information beyond the essential structure of the type.

```
union mint_def
switch (mint_def_kind kind)
{
        case MINT_VOID:                 void;
        case MINT_BOOLEAN:              void;
        case MINT_INTEGER:              mint_integer_def        integer_def;
```

```
        case MINT_SCALAR:              mint_scalar_def         scalar_def;
        case MINT_FLOAT:               mint_float_def          float_def;
        case MINT_CHAR:                mint_char_def           char_def;
        case MINT_ARRAY:               mint_array_def          array_def;
        case MINT_STRUCT:              mint_struct_def         struct_def;
        case MINT_UNION:               mint_union_def          union_def;

        case MINT_INTERFACE:           mint_interface_def      interface_def;
        case MINT_SYSTEM_EXCEPTION:    void;

        case MINT_ANY:                 void;

        case MINT_TYPE_TAG:            void;
        case MINT_TYPED:               mint_typed_def          typed_def;
};
```

The current MINT type kinds are described below, in the order in which they appear in Table 5.1.

Integers

A `MINT_INTEGER` represents an integer type with a specified (constant) minimum value and range. This type is used to represent all integer types whose values can be represented with not more that 32 bits. This includes unsigned and signed 32-, 16-, and 8-bit types, as well as booleans and certain integer "constants." (If the specified `range` is zero, then the `MINT_INTEGER` describes a type with only one possible value. This is useful for describing other types such as fixed-length arrays.)

Obviously, `MINT_INTEGER` has a direct correspondence with `AOI_INTEGER`.

```
struct mint_integer_def
{
        /* Lowest possible value this integer can take.
           If >= 0, it's an unsigned integer.  */
        int min;

        /* Number of possible values this integer can take in addition to
           'min'.  The maximum legal value is 'min + range'.
           If range is 0, the integer can take only one value -
           i.e. it carries no information; useful for representing fixed-length
           arrays.
           If range is 1, it's a boolean.  */
        unsigned range;
};
```

Scalars

A `MINT_SCALAR` describes an integer type by specifying the number of bits in its range. The minimum value is determined by flags that indicate whether the type is signed or unsigned. Scalars are only used to describe integer types that a `MINT_INTEGER` cannot represent (e.g., 64-bit and 128-bit integers).

Obviously, `MINT_SCALAR` is the MINT equivalent of AOI's `AOI_SCALAR` type.

```
typedef u_int mint_scalar_flags;
const MINT_SCALAR_FLAG_NONE     = 0;
const MINT_SCALAR_FLAG_SIGNED   = 1;
const MINT_SCALAR_FLAG_UNSIGNED = 2;

struct mint_scalar_def
{
        int                     bits;
        mint_scalar_flags       flags;
};
```

Floats

A MINT_FLOAT describes a floating point type in terms of the number of bits in the encoding of the type. Again, there is an obvious and direct correspondence with the equivalent AOI type.

```
struct mint_float_def
{
        /* Currently may be 32 or 64.  */
        int     bits;
};
```

Characters

A MINT_CHAR describes a character type. Again, the range of the type is described by the number of bits required to encode values of the type (generally 8). In addition, a set of flags describe additional properties, such as whether the characters are "signed" or "unsigned." As with most primitive types, the representations of MINT and AOI characters are identical.

```
/* These are flags to modify mint_char's: signed, unsigned, or default. */
typedef u_int mint_char_flags;
const MINT_CHAR_FLAG_NONE     = 0;
const MINT_CHAR_FLAG_SIGNED   = 1;
const MINT_CHAR_FLAG_UNSIGNED = 2;

struct mint_char_def
{
        /* Currently, 'bits' may be 8 (char) or 16 (wchar).  */
        int     bits;
        mint_char_flags flags;
};
```

Booleans

A MINT_BOOLEAN represents the standard boolean type: true or false. Currently, this type is unused: the boolean type is instead represented as a kind of MINT_INTEGER.

Voids

A MINT_VOID represents a void type: a type with no values.

Arrays

A MINT_ARRAY describes an array type: a numbered and ordered collection of values of a single (element) type. The set of possible array lengths is described by a MINT_INTEGER, referenced by the length_type slot of the mint_array_def. The array may be variable-length (if the MINT_INTEGER has a non-zero range) or fixed-length (if the MINT_INTEGER has a zero range). Obviously, the MINT_ARRAY description also contains a reference to the type of the array elements, as shown below.

Note that unlike an AOI_ARRAY, a MINT_ARRAY definition does not have a set of flags describing semantic properties of the array (e.g., NUL-termination).

```
struct mint_array_def
{
        /* Type of each of the elements in this array.  */
        mint_ref element_type;

        /* Type representing the possible lengths of this array.
           Must be an integer type of some kind.
           If the integer's 'range' is zero, it's a fixed-length array.  */
        mint_ref length_type;
};
```

Structures

A MINT_STRUCT describes a collection of *unordered*, *unnamed* values of heterogeneous types. This is different that the usual notion of a "structure" type in most programming languages. Here, a structure is simply a set of (zero or more) independent data elements. Each element has a specific type, but there is no significance to the order of the elements, nor are there any semantic connections between elements. For example, no element describes the "length" or the "type" of another.

In truth, the elements of a MINT_STRUCT are ordered in the sense that order is preserved when translating from an AOI_STRUCT, and the order of elements in an AOI_STRUCT is the order in which the slots were named in the original IDL input. Order is also important when Flick must create constant values that are of specific structure types. But this ordering of slots is only "coincidental" to the meaning of a MINT_STRUCT. The ordering of the slots in a MINT_STRUCT does not, for example, specify or constrain the order in which the structure elements might eventually be written into memory or into a request or reply message. Flick presentation generators and back ends can choose to process the MINT_STRUCT elements in any order they see fit.

```
struct mint_struct_def
{
        mint_ref slots<>;
};
```

Unions

A MINT_UNION represents a *discriminated* union: a compound data type containing a discriminator that selects from a set of possible "arms" or cases. The value of the discriminator determines which one of the possible union cases is valid (i.e., part of the complete value of the union).

Note that the discriminator does not have to be a simple integer type: it can be a complex type such as an array of characters or a structure. This allows MINT to describe unions in which the discriminator values are strings (e.g., interface or operation names) or other complex values (e.g., ONC/TCP program/version pairs, or DCE uuids).

Each arm is represented by a mint_union_case containing a discriminator value — the value that selects the arm — and type for the arm. The MINT representation of constant values is described in Section 5.2.2. The union cases come in no particular order (although order is preserved when translating from an AOI_UNION).

The union as a whole is essentially represented as a kind of structure with a distinguished discriminator member, an array of mint_union_cases representing the arms of the union, and an optional "default" arm that is selected when none of the ordinary union cases are selected. If there is no default branch, then the dfault slot of the mint_union_def will be set to mint_ref_null. In this case, a discriminator value that does not match any of the cases is considered illegal (i.e., decoding will fail on an incoming message containing a union with a bad discriminator). If there is a default case and that default case is void, then dfault must refer to a valid MINT_VOID type node.

```
struct mint_union_case
{
        mint_const      val;
        mint_ref        var;
};

struct mint_union_def
{
        /* Union discriminator variable.
           Type must be based on an int or enum prim_type.  */
        mint_ref discrim;

        /* Variable for each non-default case.
           Any case can be void, meaning no additional data. */
        mint_union_case cases<>;

        /* Variable for the default case, 'mint_ref_null' if no default.
           (If there's a default case but it's void, then 'dfault' should
           point to a MINT_VOID.) */
        mint_ref dfault;
};
```

Anys

A MINT_ANY represents "any type," the type of all possible values. Because MINT_ANY is all-encompassing, there is no additional data associated with a MINT_ANY.

Type Tags

A MINT_TYPE_TAG represents an opaque indicator of a type. In other words, the value of a type tag is a description of type. Like AOI_TYPE_TAGs, MINT_TYPE_TAGs are opaque in the sense that the compiler portions of Flick do not provide, understand, or enforce any semantics for type tags. Other than marshaling and unmarshaling tag instances, Flick does nothing with tags. Semantics must be provided by runtime libraries.

Since type tags are opaque, a MINT_TYPE_TAG has no attributes.

Typed Values

A MINT_TYPED describes pairs containing a type description and a value of the described type. In other words, a MINT_TYPED represents type-tagged values. The tag and ref fields of a mint_typed_def structure may conceptually refer to any MINT type, but in practice, the former is always a MINT_TYPE_TAG and the latter is always a MINT_ANY.

There is an obvious and direct correspondence between the MINT and AOI representations of type-tagged "any" values.

```
/* used for type-tagged data, e.g. MIG's polymorphic type */
struct mint_typed_def
{
        mint_ref        tag;
        mint_ref        ref;
};
```

Interfaces

A MINT_INTERFACE represents a reference to an instance of an IDL-defined object type. Note that this is quite different than the notion described by an AOI_INTERFACE. An AOI_INTERFACE describes the type of object instances; a MINT_INTERFACE describes the type of object *references*. In common IDLs, one passes object references in messages, not the objects themselves.[2]

A MINT_INTERFACE has a single datum describing the high-level RPC characteristics of the reference: e.g., whether the reference is for sending or receiving requests. (The current set of characteristics are derived from those implemented in the Mach operating system.) More specific properties, such as the on-the-wire encoding of object references, are not represented by MINT: by being abstract, MINT allows different back ends to choose different encodings.

```
enum mint_interface_right
{
        MINT_INTERFACE_NAME             = 0,    /* just names a interface   */
        MINT_INTERFACE_INVOKE           = 1,    /* can invoke an interface  */
        MINT_INTERFACE_INVOKE_ONCE      = 2,    /* can invoke only once     */
        MINT_INTERFACE_SERVICE          = 3     /* can service an interface */
};

struct mint_interface_def
```

[2] CORBA 2.3 introduced a notion of "objects by value," which Flick does not yet support. It is likely that the MINT representation of a by-value object type would simply be a MINT_STRUCT.

```
{
        mint_interface_right    right;
};
```

System Exceptions

Finally, a MINT_SYSTEM_EXCEPTION describes the type of exceptions generated by the RPC/RMI runtime layers. These are *implicit* exceptions, not generally defined by in IDL. *Explicit* exceptions, on the other hand, are defined in IDL to be a part of an interface. Explicit exceptions are represented as ordinary MINT types (e.g., as MINT_STRUCT); only implicit, "system-generated" exceptions are represented by MINT_SYSTEM_EXCEPTION.

System exceptions are handled specially because different RPC standards define different properties for system exceptions. For example, CORBA defines a standard set of exceptions that may be raised by the underlying ORB, and also defines the data to be communicated within these system exceptions. ONC RPC, on the other hand, defines a completely different method for communicating RPC errors to client programs.

Thus, in order to be general, MINT must support system exceptions in an abstract way: MINT defines when they may occur, but does not define the actual content or structure of system-generated exceptions. (These things are determined by Flick's individual back ends, long after the translation from AOI to MINT takes place.) In a sense, a MINT_SYSTEM_EXCEPTION is a kind of placeholder that a back end must fill in as part of implementing a complete message protocol.

Because system exceptions are opaque to MINT, a MINT_SYSTEM_EXCEPTION has no attributes.

5.2.2 Constants

In general, MINT is only concerned with the structure of data, not with particular data values. Some data types, however, incorporate constants as part of their descriptions. Discriminated unions are a prime example. (In fact, discriminated unions are the only MINT type that currently makes use of constants.) To support union discriminators and possibly other future MINT type kinds, MINT provides a way to express constant values.

A constant is represented as a mint_const, which is a pointer to a mint_const_u. A mint_const_u is a discriminated union of the possible constant kinds, as shown in the code below. There are three primitive constant types (integers, characters, and floats) and two constructed types (structures and arrays).

A primitive constant can be described as a literal value (a MINT_CONST_LITERAL) or as a symbolic value (a MINT_CONST_SYMBOLIC). Symbolic values are used when the specific value of a constant is unimportant in terms of abstract message format; in general, IDL-specified constants are represented as literals and IDL-unspecified constants (e.g., the discriminator values that select between normal and exceptional reply messages) are symbolic. In MINT, it is not necessary to define the value of a symbolic constant: MINT really only concerns itself with types, not specific values. Of course, a symbolic constant must *eventually* be assigned a value if the constant is to be used in generated stub code. This value is generally assigned by a Flick back end or by one of Flick's runtime library header files. In any case, it happens outside the scope of MINT.

A compound (array or structure) constant is simply represented as an array of other MINT constants. In a MINT_CONST_ARRAY, the number of sub-nodes is the array length, and the nodes themselves are the array elements, in order. In a MINT_CONST_STRUCT, the number of sub-nodes is the number of slots in the structure type. When matching a constant structure against a particular MINT_STRUCT type description (e.g., when interpreting a union discriminator value in terms of the defined union discriminator type), the slots of the constant and type descriptions are paired in order. The number of elements in each must be the same.

Other kinds of complex constants (e.g., unions) are not currently supported, but may be supported in the future if need arises.

```
typedef struct mint_const_u *mint_const;

enum mint_const_kind
{
        MINT_CONST_INT          = 1,
        MINT_CONST_CHAR         = 2,
        MINT_CONST_FLOAT        = 3,
        MINT_CONST_STRUCT       = 4,
        MINT_CONST_ARRAY        = 5
};

enum mint_const_category
{
        MINT_CONST_LITERAL      = 1,
        MINT_CONST_SYMBOLIC     = 2
};

union mint_const_int_u
switch (mint_const_category kind)
{
        case MINT_CONST_LITERAL:        long            value;
        case MINT_CONST_SYMBOLIC:       string          name<>;
};

union mint_const_char_u
switch (mint_const_category kind)
{
        case MINT_CONST_LITERAL:        char            value;
        case MINT_CONST_SYMBOLIC:       string          name<>;
};

union mint_const_float_u
switch (mint_const_category kind)
{
        case MINT_CONST_LITERAL:        double          value;
        case MINT_CONST_SYMBOLIC:       string          name<>;
};

typedef mint_const mint_const_struct<>;
typedef mint_const mint_const_array<>;

union mint_const_u
switch (mint_const_kind kind)
{
        case MINT_CONST_INT:    mint_const_int_u        const_int;
```

```
        case MINT_CONST_CHAR:   mint_const_char_u       const_char;
        case MINT_CONST_FLOAT:  mint_const_float_u      const_float;
        case MINT_CONST_STRUCT: mint_const_struct       const_struct;
        case MINT_CONST_ARRAY:  mint_const_array        const_array;
};
```

5.2.3 Differences Between AOI and MINT Types

Comparing AOI and MINT, one notices that certain AOI types are missing from MINT:

`AOI_ENUM`
: AOI enumeration types describe associations between names and values. In MINT, names are unimportant. Therefore, AOI enumeration types can simply be "reduced" to `MINT_INTEGER` types, and references to specific enumeration values can be reduced to (literal) `MINT_CONST_INTEGER`s.

`AOI_CONST`
: AOI constants definitions are similarly "compiled away." When needed, an AOI constant can be translated into a literal MINT constant.

`AOI_EXCEPTION`
: An `AOI_EXCEPTION` represents an IDL-described exception, i.e., what CORBA calls a "user exception." Except for the semantic that these types are associated with runtime errors, an `AOI_EXCEPTION` simply describes a structure, and so they are translated into `MINT_STRUCT` types.

`AOI_OPTIONAL`
: An optional type is represented as a `MINT_ARRAY` containing zero or one elements.

`AOI_NAMESPACE`
: Namespaces are lexical notions, which have no importance to MINT. Namespace definitions are simply discarded in the translation from AOI to MINT.

`AOI_FWD_INTRFC`
: Similarly, forward interface declarations are lexical notions and are unimportant to MINT.

`AOI_INDIRECT`
: An `AOI_INDIRECT` is a reference to a named type. Because names are unimportant to MINT, indirect references can simply be compiled away by "dereferencing" the `AOI_INDIRECT` type.

`AOI_ERROR`
: An `AOI_ERROR` represents an IDL parse error — obviously, not something MINT needs to describe!

In addition, `AOI_INTERFACE` and `MINT_INTERFACE` represent different notions as described previously in Section 5.2.1. The different parts of an AOI interface type declaration become different MINT constructs:

- An interface or object type declaration is "translated" as a `MINT_UNION` representing the set of messages that may be sent to and from instances of the interface. These are the third-tier nodes in the MINT tree shown in Figure 5.1.
- The `code` slot of an `aoi_interface` structure becomes the MINT constant that selects the represented interface from the second-tier `MINT_UNION` representing the collection of all defined interface types for a single IDL.

- The operations and attributes within an AOI interface become branches from the (third-tier) MINT_UNION that represents the interface. The request and reply codes become the union discriminator values. An AOI-defined operation therefore becomes two branches from the union (request and reply),[3] and an attribute becomes two or four (depending on whether or not the attribute is read-only).
- The data types for a request message (i.e., the `in` and `inout` parameters) are gathered together into a fourth-tier `MINT_STRUCT`.
- The data types for a reply message (i.e., the `inout`, `out`, and `return` values) are gathered under the "normal" reply branch of the (fourth-tier) `MINT_UNION` that represents the structure of a reply message. The `MINT_UNION` itself has branches for normal replies, IDL-defined ("user") exceptional replies, and system exceptions.
- An object reference — i.e., an operation parameter that is a reference to an object instance — becomes a `MINT_INTERFACE` within the description of the request or reply message that contains the reference.

To summarize, the primary distinction between AOI and MINT is that AOI describes interfaces in IDL-like terms, and MINT describes interfaces in terms of the messages that will implement an interface.

5.3 The MINT Library

The `libmint` library is provided to help deal with constructing and manipulating the various MINT data structures. Its header file is `mom/libmint.h` and has the following functions:

`void mint_add_standard_defs(mint_1 *mint)`

Create the standard set of MINT type definitions as described by the `mint_standard_refs` structure type, described in Section 5.2. The MINT type definitions are added to the array of `mint_defs` in `mint`, and the `mint_refs` within `mint->standard_refs` are set to point to the proper definitions.

`mint_ref mint_add_def(mint_1 *mint)`

Return a `mint_ref` that refers to a newly allocated MINT type definition (`mint_def`). The `mint_def` is completely uninitialized and must be filled in by the caller. For this reason, it is often more convenient to call one of the following, more specific constant constructors:

`mint_ref mint_add_union_def(mint_1 *mint, mint_ref discrim_type, int len)`

Return a `mint_ref` to a newly allocated `MINT_UNION` with the given discriminator type and number of cases. The cases are uninitialized; the default case is set to `mint_ref_null`.

`mint_ref mint_add_integer_def(mint_1 *mint, int min, unsigned range)`

Return a `mint_ref` to a newly allocated `MINT_INTEGER` with the given minimum value and range.

`mint_ref mint_add_array_def(mint_1 *mint, mint_ref element_type, unsigned min_len, unsigned max_len)`

Return a `mint_ref` to a `MINT_ARRAY` with the given element type, minimum length, and maximum length. If a MINT array definition with the given attributes already exists in `mint`, return a reference to it. Otherwise, allocate and initialize a new `MINT_ARRAY` definition.

[3] At present, MINT is produced even for the nonexistent "replies" of `oneway` functions.

`mint_ref mint_add_def_tags(mint_def_kind kind, mint_1 *mint, int tag, ...)`
Return a `mint_ref` to a MINT definition of the given `kind`, with the definition slots initialized according to the list of tags. The tags and associated tag values are listed below. *Note that most tags are appropriate for only certain* MINT *type kinds.* `mint_def_add_tags` will call `panic` if given a tag that is inappropriate for the specified MINT type `kind`.

`MINT_TAG_DONE`
(No required tag values.) Marks the end of the tag list.

`MDA_Min`
(Required tag values: `int`.) The minimum value of a `MINT_INTEGER` type.

`MDA_Range`
(Required tag values: `int`.) The range of a `MINT_INTEGER` type.

`MDA_Bits`
(Required tag values: `int`.) The number of bits for a `MINT_SCALAR`, `MINT_FLOAT`, or `MINT_CHAR`.

`MDA_Flags`
(Required tag values: `mint_scalar_flags` or `mint_char_flags`.) The flags for a scalar or character type.

`MDA_ElementType`
(Required tag values: `mint_ref`.) The element type for a `MINT_ARRAY`.

`MDA_LengthType`
(Required tag values: `mint_ref`.) The length type for a `MINT_ARRAY`.

`MDA_Slot`
(Required tag values: `mint_ref`.) A slot within a `MINT_STRUCT`. This tag may be repeated in order to specify multiple structure slots. The slots are added to the structure in the order in which they are specified in the tag list.

`MDA_Discrim`
(Required tag values: `mint_ref`.) The discriminator type for a `MINT_UNION`.

`MDA_Case`
(Required tag values: `mint_const` and `mint_ref`.) A case within a `MINT_UNION` type. This tag may be repeated in order to specify multiple union cases.

`MDA_Default`
(Required tag values: `mint_ref`.) The default case in a `MINT_UNION`.

`MDA_Tag`
(Required tag values: `mint_ref`.) The type tag for a `MINT_TYPED`.

`MDA_Ref`
(Required tag values: `mint_ref`.) The typed reference (i.e., `ref` slot) within a `MINT_TYPED`.

`MDA_Right`
(Required tag values: `mint_interface_right`.) The `right` within a `MINT_INTERFACE` type.

In addition, `libmint.h` provides a set of simple `MINT_*_REF` macros to be used in combination with the above tags to aid in the construction of MINT type definitions. See the header file for details. *Flick implementors must use these macros with caution, however, so as not to introduce portability problems. With these macros, it is easy to create an argument list that contains multiple calls to* `mint_add_def_tags`. *Each call is a separate parameter, and each call has side effects, so the order in which the calls take place is significant. However, C doesn't guarantee an order of evaluation for function parameters. This means that the calls might be evaluated in different orders, depending on the compiler, thus producing different (although logically equivalent)* `mint_def` *arrays. This in turn would cause Flick to produce different* PRES_C *files on different platforms, thus making testing much more difficult!*

`void mint_get_int_size(mint_1 *mint, mint_ref itype, int *out_bits, int *out_signed)`

Determine the number of bits (8, 16, or 32) required to represent the MINT type referenced by `itype`. In essence, this function converts a description of an integer range into a bits/signedness representation. The required number of bits is stored in `*out_bits` and the signedness indicator (zero or one) is stored in `*out_signed`.

`void mint_get_array_len(mint_1 *mint, mint_ref itype, unsigned *min, unsigned *max)`

Determine the minimum and maximum lengths of the MINT array referenced by `itype`. The minimum length is stored in `*min` and the maximum is stored in `*max`. (If the lengths are equal, then the array is fixed-length.)

`int mint_add_struct_slot(mint_1 *mint, mint_ref struct_itype)`

Add a new slot to the `MINT_STRUCT` referenced by `struct_itype`, and return the index of the new slot. The slot contents are uninitialized.

`int mint_add_union_case(mint_1 *mint, mint_ref union_itype)`

Add a new case to the `MINT_UNION` referenced by `union_itype`, and return the index of the new case. The case contents are uninitialized.

`mint_ref mint_find_union_case(mint_1 *mint, mint_ref union_r, mint_const const_r)`

Return a reference to the MINT type node that is selected by `const_r` in the `MINT_UNION` referenced by `union_r`. (`union_r` must point to a union type.) If the constant does not correspond to any explicit case of the union, then this function returns the `dfault` branch of the union, which may be `mint_ref_null`.

`mint_const mint_new_const(mint_const_kind kind)`

Return a `mint_const` that refers to a newly allocated `mint_const_u` initialized with the given MINT constant `kind`. Other fields of the returned constant description are uninitialized and must be filled in by the caller. Therefore, it is often more convenient to call one of the following, more specific constant constructors:

`mint_const mint_new_const_int(int val)`

Return a `mint_const` that points to a newly allocated `mint_const_u` describing the constant integer `val`.

`mint_const mint_new_const_string(const char *val)`
Return a `mint_const` that points to a newly allocated `mint_const_u` describing the constant string `val`. The string is represented as an array of characters, *including* a terminating NUL. (Note that this is different than the behavior of the `aoi_new_const_string` function described in Section 4.3.)

`mint_const mint_new_const_from_aoi_const(aoi_const aoiconst)`
Return a `mint_const` that points to a newly allocated `mint_const_u` that is the MINT equivalent of the given AOI constant.

`mint_const mint_new_symbolic_const(mint_const_kind kind, const char *name)`
Return a `mint_const` that points to a newly allocated `mint_const_u` describing a symbolic constant with the given `type` and `name`.

`int mint_const_cmp(mint_const c1, mint_const c2)`
Compare two MINT constants. Return `1` if the constants are unequal and `0` otherwise. Two constants of different types are never equal. Two primitive literal constants are equal only if they contain the same literal values; two primitive symbolic constants are equal only if they have the same name (and are of the same type, as just mentioned). Two compound constants are equal if they are structurally identical and have equal corresponding elements.

Note that a literal constant and a symbolic constant are never equal.

`void mint_const_check(mint_const mc)`
Check the validity of the given `mint_const`.

`void mint_1_check(mint_1 *mint)`
Check the validity of the given `mint_1` structure. If an error is found, this function calls `panic`, which causes the process to print an error message and exit.

`void mint_1_readfh(mint_1 *dest, FILE *fh)`
Read a `mint_1` structure from `fh` into `dest`. The disk data is stored in XDR format, and the actual I/O is performed by functions that are generated by `rpcgen` from the `mom/mint.x` file. If input fails for some reason, this function invokes `panic`.

`void mint_1_writefh(mint_1 *src, FILE *fh)`
Write the given `mint_1` structure to `fh`. (As before, the data in the file is stored in XDR format, and the actual I/O is performed by `rpcgen`-generated functions.) This function invokes `mint_1_check` to verify the validity of the MINT data before writing the data to disk. If output fails for some reason, this function calls `panic`.

5.4 Summary and Comments

MINT allows Flick to represent messages in an abstract manner, without describing the exact message formats or data encodings that will eventually be used. There are ways, however, in which the current implementation of MINT is problematic: either too abstract, or not abstract enough.

Primitive MINT Types. Some of the basic MINT types, such as floats and characters, are perhaps too abstract. For instance, it is necessary to know more about the range of a floating-point type that just the total number of bits. Different IDLs may have different implicit semantics for floats, e.g., different expectations about the sizes of the exponent and mantissa. These expectations should be represented in MINT (and properly in AOI as well). In a similar way, character types should describe additional properties such as the character set from which the character is chosen. Representing *all* of the encoding details of floats and characters is likely not to be worthwhile. A compromise would be to refer to some encoding details by the names of standards, e.g., `"ASCII"`, `"Unicode UTF-16"`, or `"IEEE ..."`.

Interface and Operation Names. The identifiers for interface, operations, and other "unpresented" parts of a MINT tree are not sufficiently abstract.

The union discriminators used in the upper levels of the MINT tree are chosen by the presentation generator from the names and codes described in AOI. In practice, these MINT union discriminators then become *the* identifiers used by back ends when implementing their message formats. This is a problem: the presentation generators should not be choosing the identifiers that should be used in "unpresented" parts of a message. To fix this problem, MINT must be extended to represent abstract interface and operation identifiers. These abstract identifiers should allow for multiple names, carried forward from AOI. (See the discussion in Section 4.5.) It would be the job of a back end to translate a MINT abstract identifier into a concrete value, much in the way that back ends already handle MINT symbolic constants.

Chapter 6

C Abstract Syntax Tree (CAST) Representation

The C Abstract Syntax Tree (CAST) intermediate language is a collection of data structures built to provide an internal representation of C/C++ source code. This representation allows Flick to generate code as it goes along and then make modifications to the code after some analysis has been done. The definitions of the CAST data structures are contained in mom/cast.x and the library code can be found in c/libcast and its header file is in mom/c/libcast.h.

6.1 CAST Overview

The CAST data structures are designed to represent the various aspects of C/C++ source code. At the root there is the cast_scope which corresponds to a lexical scope, so it contains all declarations and definitions. These declarations and definitions are then represented by cast_defs which contains any information needed to produce the correct source code. Then, beneath the cast_defs there can be types, expressions, and other things representing their counterparts in C/C++ source code. These CAST trees can then be printed out using the cast_w_* functions in the CAST library (described in Section 6.3), although it is easiest to just call cast_w_scope on the root scope to get everything printed out.

6.2 CAST Data Structures

The following descriptions of the data structures show the C type definitions followed by a simple representation of what output they generate. This representation is pretty simple: just think of the stuff between the angle brackets as code to be executed, or as variable names to be referenced.

6.2.1 Scoped Names

CAST uses a cast_scoped_name to hold any information about a fully scoped name in C++. The structure is made up of an array of cast_name_s structures which hold a name and an array of template arguments for that name. Each template argument can either be another name, a type, or an expression.

```
typedef struct cast_name_s cast_scoped_name<>;

enum cast_template_arg_kind
```

```
{
  CAST_TEMP_ARG_NAME    = 1,
  CAST_TEMP_ARG_TYPE    = 2,
  CAST_TEMP_ARG_EXPR    = 3
};
union cast_template_arg_u
switch(cast_template_arg_kind kind)
{
  case CAST_TEMP_ARG_NAME:  cast_scoped_name  name;
  case CAST_TEMP_ARG_TYPE:  cast_type         type;
  case CAST_TEMP_ARG_EXPR:  cast_expr         expr;
};
struct cast_name_s
{
  string                  name<>;
  cast_template_arg_array  args;    /* C++ Only */
};
```

The CAST library functions for scoped names are built to pass the `cast_scoped_name` structure by value if it is not going to be changed or it is newly created, otherwise they are passed by pointer. For example, the `cast_new_scoped_name` constructs a new `cast_scoped_name` from a null-terminated `varargs` list of strings and then returns the new scoped name by value. However, the `cast_add_scope_name` function takes a pointer to a scoped name since it will be changing the structure. Here is a full list of the pertinent `libcast` functions:

`void cast_add_scope_name(cast_scoped_name *scname, char *name, cast_template_arg_array args)`
: Adds another name and set of template arguments to a scoped name.

`void cast_del_scope_name(cast_scoped_name *name)`
: Deletes the last name in the given scoped name.

`void cast_prepend_scope_name(cast_scoped_name *scname, char *name, cast_template_arg_array args)`
: Adds the given scope name to the front of a scoped name.

`int cast_cmp_scoped_names(const cast_scoped_name *scn1, const cast_scoped_name *scn2)`
: Compares two scoped names, just like a regular `strcmp`. Note: this does not yet compare the template arguments so it is possible to get a false positive.

`cast_scoped_name cast_new_scoped_name(const char *name, ...)`
: Constructs a new `cast_scoped_name` from a null terminated list of strings. For example, `cast_new_scoped_name("CORBA", "Boolean", NULL)` would result in the name `"CORBA::Boolean"`.

`cast_scoped_name cast_set_scoped_name(cast_scoped_name *scname, char *name, cast_template_arg_array args, ...)`
: This is a more flexible version of `cast_new_scoped_name` which allows one to construct a name with template arguments. It works by setting the values of `scname` or constructing a new one if it is null. The varargs are tuples of strings for the names, and the template arguments for that name, terminated by a null.

`int cast_scoped_name_is_empty(cast_scoped_name *scname)`
: Returns TRUE if `scname` is empty.

`cast_scoped_name cast_copy_scoped_name(cast_scoped_name *name)`
: Returns a copy of `name`.

The CAST library also defines some handy global variables:

`extern cast_scoped_name null_scope_name`
: This is a `cast_scoped_name` initialized with zero length and value. This variable is usually used to initialize other `cast_scoped_names` created on the stack.

`extern cast_scoped_name empty_scope_name`
: This is a `cast_scoped_name` initialized with an empty string as its only name.

6.2.2 Types

Any C/C++ types in CAST are represented in the `cast_type_u` union and usually referenced by `cast_type` which is a pointer to a `cast_type_u`.

Primitive Types

The `cast_primitive_type` structure is able to represent any primitive types like booleans, characters, integers, and floats, as well as any modifiers for those types. The `name` field is a bit of a hack so that it can be treated internally as a primitive, but when output, the string in `name` will be printed rather than the primitive name.

```
enum cast_primitive_kind
{
  CAST_PRIM_CHAR    = 1,
  CAST_PRIM_INT     = 2,
  CAST_PRIM_FLOAT   = 3,
  CAST_PRIM_DOUBLE  = 4,
  CAST_PRIM_BOOL    = 5  /* C++ Only */
};
typedef unsigned cast_primitive_modifier;
const CAST_MOD_SIGNED      = 0x0001;
const CAST_MOD_UNSIGNED    = 0x0002;
const CAST_MOD_SHORT       = 0x0010;
const CAST_MOD_LONG        = 0x0020;
const CAST_MOD_LONG_LONG   = 0x0040;
struct cast_primitive_type
{
  cast_primitive_kind  kind;
  cast_primitive_modifier  mod;
  cast_scoped_name  name;
};

<if name is null>
  <mod> <kind>
```

```
<else>
  <name>
```

Enumerations

The `cast_enum_type` represents an enumeration with a list of names and their value(s), as well as the name of the enumeration itself.

```
struct cast_enum_field
{
  /* The name of the slot */
  string name<>;

  /* The value (should probably be a literal primitive */
  cast_expr val;
};
struct cast_enum_type
{
  /* Enum tag. */
  cast_scoped_name name;

  /* value and name of each slot */
  cast_enum_field  slots<>;
};

enum <name> {
  <slots[0].name> = <slots[0].val>,
  ...,
  <slots[n].name> = <slots[n].val>
}
```

Arrays

The `cast_array_type` can represent an array of a certain length and type.

```
struct cast_array_type
{
  /* Constant expression that determines the array's length.
     Can be omitted if outside of anything or at the end of a structure;
     means array is variable-length.  */
  cast_expr    length;

  /* Type of each element.  */
  cast_type    element_type;
};

<element_type> [<length>]
```

Pointers

The `cast_pointer_type` adds a level of pointer indirection and points to the real type.

```
struct cast_pointer_type
{
  /* What the pointer points to.  */
  cast_type    target;
};

<target> *
```

References

The `cast_reference_type` adds a level of C++ variable reference.

```
struct cast_reference_type
{
  /* What the reference refers to. */
  cast_type    target;
};

<target> &
```

Functions

The `cast_func_type` represents a function and any related attributes, such as parameters, exceptions, initializers in constructors, and various flags. The parameters are an array of `cast_param`s which contain the type, name, and default value of the parameters.

Also, there is a flag field to specify options for the parameter. Currently, there is only the `CAST_PARAM_IMPLICIT` flag which means that the parameter is implicit and should not be part of the function prototype. An example usage of this flag is the `this` variable in a member function of a C++ class, or a global variable, like `errno`, which a function might be using.

```
typedef unsigned cast_param_spec;
const CAST_PARAM_IMPLICIT  = 0x0001;
struct cast_param
{
  cast_param_spec  spec;
  string           name<>;
  cast_type        type;
  cast_init        default_value;  /* C++ Only */
};

typedef cast_param cast_func_param;
typedef unsigned cast_func_spec;
const CAST_FUNC_VIRTUAL    = 0x0001;  /* Prepends 'virtual' to the prototype */
const CAST_FUNC_PURE       = 0x0002;  /* Appends '= 0' to make it pure */
const CAST_FUNC_INLINE     = 0x0004;  /* Prepends 'inline' */
```

```
const CAST_FUNC_EXPLICIT   = 0x0008;  /* Prepends 'explicit' */
const CAST_FUNC_CONST      = 0x0010;  /* Appends 'const' */
const CAST_FUNC_OPERATOR   = 0x0020;  /*
                                       * Prepends 'operator' to the cast_def
                                       * name unless it's empty, in case it
                                       * is printed before the return type,
                                       * so you can do type cast operators.
                                       */
struct cast_func_type
{
  cast_func_param    params<>;
  cast_type          return_type;
  cast_func_spec     spec;              /* C++ Only */
  cast_type_array    exception_types;  /* C++ Only */
  cast_expr_array    initializers;      /* C++ Only */
};

<spec & CAST_FUNC_VIRTUAL|CAST_FUNC_INLINE|CAST_FUNC_EXPLICIT> <return_type>
(<if !implicit><params[0].type and params[0].name> = <params[0].default_value>,
 ...
 <if !implicit><params[n].type and params[n].name> = <params[n].default_value>
)
<spec & CAST_FUNC_CONST>
throw(<exception_types[0], ..., <exception_types[n]>)
: <initializers[0]>, ..., <initializers[1]>
<spec & CAST_FUNC_PURE>
```

Aggregate Types

The `cast_aggregate_type` is used for compound types, things like unions, structures, and classes. Members are then added to the `cast_scope` just like adding a global variable to the root scope. Of course, other declarations and definitions can be added to the scope if it is C++, as well as parents in the `parents`.

```
typedef unsigned cast_parent_flags;
const CAST_PARENT_PUBLIC     = 0x0001;
const CAST_PARENT_PROTECTED  = 0x0002;
const CAST_PARENT_PRIVATE    = 0x0004;
const CAST_PARENT_VIRTUAL    = 0x0008;
struct cast_parent_spec
{
  cast_parent_flags  flags;
  cast_scoped_name   name;
};

enum cast_aggregate_kind
{
  CAST_AGGREGATE_STRUCT = 1,
  CAST_AGGREGATE_UNION  = 2,
```

```
  CAST_AGGREGATE_CLASS  = 3
};
struct cast_aggregate_type
{
  cast_aggregate_kind  kind;
  cast_scoped_name     name;
  cast_parent_spec     parents<>;  /* C++ Only */
  cast_scope           scope;      /* Adding things besides vars is C++ Only */
};

<kind> <name> : <parents[0].flags> <parents[0].name>,
                ...,
                <parents[n].flags> <parents[n].name>
{
  <scope>
};
```

Qualifiers

The `cast_qualified_type` is used to attach any qualifiers, const and volatile, to a type.

```
typedef int cast_type_qualifier;
const CAST_TQ_CONST       = 0x01;
const CAST_TQ_VOLATILE    = 0x02;
struct cast_qualified_type
{
  cast_type_qualifier qual;
  cast_type           actual;
};

<qual> <actual>
```

Templates

The `cast_template_type` is used to declare a template type with a certain set of parameters.

```
enum cast_template_param_kind
{
  CAST_TEMP_PARAM_TYPE      = 1,
  CAST_TEMP_PARAM_CLASS     = 2,
  CAST_TEMP_PARAM_TYPENAME  = 3,
  CAST_TEMP_PARAM_TEMPLATE  = 4
};
union cast_template_param_u
switch(cast_template_param_kind kind)
{
  case CAST_TEMP_PARAM_TYPE:      cast_type              type_param;
  case CAST_TEMP_PARAM_CLASS:     void;
```

```
  case CAST_TEMP_PARAM_TYPENAME:  void;
  case CAST_TEMP_PARAM_TEMPLATE:  cast_template_param_t  params<>;
};
struct cast_template_param
{
  string                  name<>;
  cast_template_param_u  u;
  cast_template_arg      default_value;
};

typedef unsigned cast_template_flags;
const CAST_TEMP_EXPORT    = 0x0001;
struct cast_template_type
{
  cast_template_flags  flags;
  cast_template_param  params<>;
  cast_type            def;
};

<flags>
template < <params[0].u and params[0].name> = <params[0].default_value>,
           ...,
           <params[n].u and params[n].name> = <params[n].default_value> >
  <def>
```

Named Types

Referencing types through their name is done by using a set of different kinds which get output in slightly different ways:

`CAST_TYPE_NAME`
: A plain name, usually from a `typedef`.

`CAST_TYPE_STRUCT_NAME`
: A name for a structure, it will print `struct` before the scoped name.

`CAST_TYPE_UNION_NAME`
: A name for a union, it will print `union` before the scoped name.

`CAST_TYPE_ENUM_NAME`
: A name for an enumeration, it will just print the scoped name.

`CAST_TYPE_CLASS_NAME`
: A forward class declaration, it will print `class` before the scoped name, so this cannot be used like a regular type.

Voids

Void types are handled simply by setting the kind of the `cast_type` to `CAST_TYPE_VOID`.

Nulls

Null types are used for the return type of C++ constructors and destructors. They are handled by setting the kind of the `cast_type` to `CAST_TYPE_NULL`.

Union of All Types

All of the specific types are brought together under the umbrella of the cast_type_u union which holds them all and a few simple ones.

```
enum cast_type_kind
{
  CAST_TYPE_NAME        = 1,
  ...
  CAST_TYPE_TEMPLATE    = 17   /* C++ Only */
};
union cast_type_u
switch (cast_type_kind kind)
{
  case CAST_TYPE_NAME:         cast_scoped_name      name;
  case CAST_TYPE_PRIMITIVE:    cast_primitive_type   primitive_type;
  case CAST_TYPE_POINTER:      cast_pointer_type     pointer_type;
  case CAST_TYPE_ARRAY:        cast_array_type       array_type;
  case CAST_TYPE_FUNCTION:     cast_func_type        func_type;
  case CAST_TYPE_ENUM:         cast_enum_type        enum_type;
  case CAST_TYPE_STRUCT_NAME:  cast_scoped_name      struct_name;
  case CAST_TYPE_UNION_NAME:   cast_scoped_name      union_name;
  case CAST_TYPE_ENUM_NAME:    cast_scoped_name      enum_name;
  case CAST_TYPE_VOID:         void;
  case CAST_TYPE_NULL:         void;
  case CAST_TYPE_QUALIFIED:    cast_qualified_type   qualified;
  case CAST_TYPE_AGGREGATE:    cast_aggregate_type   agg_type;
  case CAST_TYPE_CLASS_NAME:   cast_scoped_name      class_name;
  case CAST_TYPE_REFERENCE:    cast_reference_type   reference_type;
  case CAST_TYPE_TYPENAME:     cast_scoped_name      typename_name;
  case CAST_TYPE_TEMPLATE:     cast_template_type    template_type;
};
```

There are a number of functions in the CAST library for constructing and initializing these structures. There are also some functions and variables dealing with `cast_type_array`s which can be used whenever an array of CAST types is needed.

`cast_type_array cast_set_type_array(cast_type_array *array, cast_type type, ...)`
: Sets the values of `array` (or construct a new one if it is NULL) with the array of values passed in the null-terminated argument list.

`int cast_add_type_array_value(cast_type_array *array, cast_type type)`
: Adds another element to `array`, initializing it with `type`, and then return the index of the new element.

`extern cast_type_array null_type_array`

This is a global that can be used for initializing other `cast_type_arrays` to be empty.

6.2.3 Expressions

Literal Primitives

A `cast_lit_prim` represents a primitive literal value of a primitive type.

```
union cast_lit_prim_u
switch (cast_primitive_kind kind)
{
  case CAST_PRIM_CHAR:   char    c;
  case CAST_PRIM_INT:    long    i; /* XXX long long */
  case CAST_PRIM_FLOAT:  float   f;
  case CAST_PRIM_DOUBLE: double  d; /* XXX long double */
  case CAST_PRIM_BOOL:   char    b; /* C++ Only */
};
struct cast_lit_prim
{
  cast_primitive_modifier  mod;
  cast_lit_prim_u          u;
};
```

```
<mod> <u>
```

Function Calls

A `cast_expr_call` represents a function call with a set of arguments.

```
struct cast_expr_call
{
  cast_expr        func;
  cast_expr_array  params;
};
```

```
<func>(<params[0]>, ..., <params[n]>)
```

Member Selection

A `cast_expr_sel` represents the selection of a member of an aggregate type.

```
struct cast_expr_sel
{
  cast_expr         var;
  cast_scoped_name  member;
};
```

```
<var>.<member>
```

Unary Operations

A `cast_unary_expr` represents a unary operation on another expression.

```
enum cast_unary_op
{
  CAST_UNARY_DEREF     = 1,  /* * (unary) */
  CAST_UNARY_ADDR      = 2,  /* & (unary) */
  CAST_UNARY_NEG       = 3,  /* - (unary) */
  CAST_UNARY_LNOT      = 4,  /* ! (unary) */
  CAST_UNARY_BNOT      = 5,  /* ~ (unary) */
  CAST_UNARY_PRE_INC   = 6,  /* ++foo */
  CAST_UNARY_PRE_DEC   = 7,  /* --foo */
  CAST_UNARY_POST_INC  = 8,  /* foo++ */
  CAST_UNARY_POST_DEC  = 9   /* foo-- */
};
struct cast_unary_expr
{
  cast_unary_op  op;
  cast_expr      expr;
};

<op><expr>
```

Binary Operations

A `cast_binary_expr` represents a binary operation over two expressions.

```
enum cast_binary_op
{
  CAST_BINARY_MUL    = 1,   /* * (binary) */
  CAST_BINARY_DIV    = 2,   /* / */
  CAST_BINARY_MOD    = 3,   /* % */
  CAST_BINARY_ADD    = 4,   /* + (binary) */
  CAST_BINARY_SUB    = 5,   /* - (binary) */
  CAST_BINARY_SHL    = 6,   /* << */
  CAST_BINARY_SHR    = 7,   /* >> */

  CAST_BINARY_LT     = 8,   /* < */
  CAST_BINARY_GT     = 9,   /* > */
  CAST_BINARY_LE     = 10,  /* <= */
  CAST_BINARY_GE     = 11,  /* >= */
  CAST_BINARY_EQ     = 12,  /* == */
  CAST_BINARY_NE     = 13,  /* != */

  CAST_BINARY_BAND   = 14,  /* & */
  CAST_BINARY_BXOR   = 15,  /* ^ */
  CAST_BINARY_BOR    = 16,  /* | */
```

```
  CAST_BINARY_LAND   = 17,  /* && */
  CAST_BINARY_LOR    = 18,  /* || */

  CAST_BINARY_ASSIGN = 19,  /* = */

  CAST_BINARY_COMMA  = 20   /* , */
};
struct cast_binary_expr
{
  cast_binary_op  op;
  cast_expr       expr[2];
};

<expr[0]> <op> <expr[1]>
```

Type Casting

A `cast_expr_cast` represents a type cast on another expression.

```
struct cast_expr_cast
{
  /* Expression to cast.  */
  cast_expr  expr;

  /* Type to cast it to.  */
  cast_type  type;
};

(<type>)<expr>
```

Conditional Expressions

A `cast_cond_expr` represents a conditional expression with a test expression and its positive and negative results.

```
struct cast_cond_expr
{
  cast_expr  test;
  cast_expr  true_expr;
  cast_expr  false_expr;
};

<test> ? <true_expr> : <false_expr>
```

The new Operator

A `cast_op_new_expr` represents a usage of the `new` operator.

```
struct cast_op_new_expr
{
  cast_expr  placement;
  cast_type  type;
  cast_init  init;
};
```

```
new (<placement>) <type><init>
```

The delete Operator

A `cast_op_delete_expr` represents a usage of the `delete` operator.

```
struct cast_op_delete_expr
{
  long       array;
  cast_expr  expr;
};
```

```
delete <if array>[]</if> <expr>
```

Variable Names

Variable names and other name expressions are a `cast_expr` set to `CAST_EXPR_NAME` with a scoped name.

String Literals

A string literal is simply a `cast_expr` set to `CAST_EXPR_LIT_STRING` and a C string set in the `cast_expr` structure.

The sizeof Operator

A `sizeof` is done by setting the kind of the `cast_expr` to `CAST_EXPR_SIZEOF_EXPR` or `CAST_EXPR_SIZEOF_TYPE` and then setting the value of what is to be evaluated.

The typeid Operator

A `typeid` is done by setting the kind of the `cast_expr` to `CAST_EXPR_TYPEID_EXPR` or `CAST_EXPR_TYPEID_TYPE` and then setting the value of what is to be evaluated.

Types as Expressions

Sometimes it is necessary to use a type as an expression, such as in a macro call. This can be done by using the kind `CAST_EXPR_TYPE`.

Union of All Expressions

```
enum cast_expr_kind
{
  CAST_EXPR_NAME              = 1,
  ...
  CAST_EXPR_TYPE              = 22
};
union cast_expr_u
switch (cast_expr_kind kind)
{
  case CAST_EXPR_NAME:              cast_scoped_name     name;
  case CAST_EXPR_LIT_PRIM:          cast_lit_prim        lit_prim;
  case CAST_EXPR_LIT_STRING:        string               lit_string<>;
  case CAST_EXPR_CALL:              cast_expr_call       call;
  case CAST_EXPR_SEL:               cast_expr_sel        sel;

  case CAST_EXPR_UNARY:             cast_unary_expr      unary;
  case CAST_EXPR_CAST:              cast_expr_cast       cast;
  case CAST_EXPR_SIZEOF_EXPR:       cast_expr            sizeof_expr;
  case CAST_EXPR_SIZEOF_TYPE:       cast_type            sizeof_type;

  case CAST_EXPR_BINARY:            cast_binary_expr     binary;
  case CAST_EXPR_OP_ASSIGN:         cast_binary_expr     op_assign;
  case CAST_EXPR_COND:              cast_cond_expr       cond;
  case CAST_EXPR_CONST_NAME:        cast_scoped_name     const_name;
  case CAST_EXPR_CONST_CAST:        cast_expr_cast       c_cast;
  case CAST_EXPR_DYNAMIC_CAST:      cast_expr_cast       d_cast;
  case CAST_EXPR_REINTERPRET_CAST:  cast_expr_cast       r_cast;
  case CAST_EXPR_STATIC_CAST:       cast_expr_cast       s_cast;
  case CAST_EXPR_OP_NEW:            cast_op_new_expr     op_new;
  case CAST_EXPR_OP_DELETE:         cast_op_delete_expr  op_delete;
  case CAST_EXPR_TYPEID_EXPR:       cast_expr            typeid_expr;
  case CAST_EXPR_TYPEID_TYPE:       cast_type            typeid_type;
  case CAST_EXPR_TYPE:              cast_type            type_expr;
};
```

6.2.4 Statements

Statement Blocks

A `cast_block` represents a block of executable statements.

```
typedef unsigned cast_block_flags;
/* The statements in the block are in reverse order */
const CAST_BLOCK_REVERSE   = 0x00000001;
/* The statements in the block should be treated as if they were
   inlined with the rest of the statements in the parent block.
   (e.g. there is no '{' when printed) */
```

```
const CAST_BLOCK_INLINE    = 0x00000002;

struct cast_block
{
  /* Variable declarations and such at beginning of block.  */
  cast_scope        scope;

  /* Statements following decls, but preceeding "regular" code.
     (Mainly useful for initializing temporary vars, etc.) */
  cast_stmt         initials<>;

  /* Statements following them ("regular" code).  */
  cast_stmt         stmts<>;
  cast_block_flags  flags;
};

{
  <scope>

  <initials[0]>
  ...
  <initials[n]>

<if !reverse>
  <stmts[0]>
  ...
  <stmts[n]>
  <else>
  <stmts[n]>
  ...
  <stmts[0]>
}
```

If Statements

A `cast_if` represents an if statement and its true and false branches.

```
struct cast_if
{
  cast_expr    test;
  cast_stmt    true_stmt;
  cast_stmt    false_stmt;  /* optional */
};

if( <test> )
  <true_stmt>
else
  <false_stmt>
```

While and Do/While Loops

A `cast_while` can be used to represent both `while` and `do/while` loops, depending on the `cast_stmt_kind`.

```
struct cast_while
{
  cast_expr    test;
  cast_stmt    stmt;
};

while( <test> )
  <stmt>

do
  <stmt>
while( <test> )
```

For Loops

A `cast_for` represents a `for` loop.

```
struct cast_for
{
  cast_expr    init;  /* optional */
  cast_expr    test;  /* optional */
  cast_expr    iter;  /* optional */
  cast_stmt    stmt;
};

for( <init>; <test>; <iter> )
  <stmt>
```

Switch Statements

A `cast_switch` represents a `switch` statement. The stmt slot is used as the body so it should be a `cast_block` with a number of `cast_cases` for each branch.

```
struct cast_switch
{
  cast_expr    test;
  cast_stmt    stmt;
};

switch( <test> )
  <stmt>
```

Statement Labels

A `cast_label` attaches a label to a statement. The `users` slot is used internally to track the number of times this label has been used, if the count is zero then this label will not be printed, but the statement will still be output.

```
struct cast_label
{
  string      label<>;
  cast_stmt   stmt;
  int         users;
};
```

```
<if users><label>:</if>
  <stmt>
```

Case Statements

A `cast_case` represents a case branch in a switch statement.

```
struct cast_case
{
  cast_expr    label;
  cast_stmt    stmt;
};
```

```
case <label>:
  <stmt>
```

Catch Blocks

A `cast_catch` represents a `catch` block in C++.

```
/* C++ Only */
struct cast_catch
{
  cast_type    type;
  string       name<>;
  cast_stmt    block;
};
```

```
catch( <type and name> )
  <block>
```

Try Blocks

A `cast_try` represents a `try` block in C++.

```
/* C++ Only */
struct cast_try
{
  cast_stmt     block;
  cast_catch    handlers<>;
};

try
  <block>
<handlers[0]>
...
<handlers[n]>
```

Break Statements

Setting the `cast_stmt` kind to `CAST_STMT_BREAK` will make a break statement. Since there is nothing else associated with a break statement, it has no structure

Continue Statements

Setting the `cast_stmt` kind to `CAST_STMT_CONTINUE` will make a continue statement. Since there is nothing else associated with a continue statement, it has no structure

Null Statements

Setting the `cast_stmt` kind to `CAST_STMT_NULL` will make a null statement which just prints out a ';'

Empty Statements

Setting the `cast_stmt` kind to `CAST_STMT_EMPTY` will make a completely empty statement, nothing will be printed. This is useful for deleting statements in the middle of blocks without having to move all of the following statements back one.

Union of All Statements

```
enum cast_stmt_kind
{
  CAST_STMT_EXPR      = 1,
  ...
  CAST_STMT_DECL      = 20   /* C++ Only */
};
union cast_stmt_u
switch (cast_stmt_kind kind)
{
  case CAST_STMT_EXPR:      cast_expr     expr;
  case CAST_STMT_BLOCK:     cast_block    block;
  case CAST_STMT_IF:        cast_if       s_if;
  case CAST_STMT_WHILE:     cast_while    s_while;
  case CAST_STMT_DO_WHILE:  cast_while    s_do_while;
```

```
  case CAST_STMT_FOR:       cast_for      s_for;
  case CAST_STMT_SWITCH:    cast_switch   s_switch;
  case CAST_STMT_BREAK:     void;
  case CAST_STMT_CONTINUE:  void;
  case CAST_STMT_GOTO:      string        goto_label<>;
  case CAST_STMT_LABEL:     cast_label    s_label;
  case CAST_STMT_CASE:      cast_case     s_case;
  case CAST_STMT_DEFAULT:   cast_stmt     default_stmt;
                                          /* optional: */
  case CAST_STMT_RETURN:    cast_expr     return_expr;
  case CAST_STMT_TEXT:      string        text<>;
  case CAST_STMT_NULL:      void;
  case CAST_STMT_EMPTY:     void;
  case CAST_STMT_TRY:       cast_try      try_block;
  case CAST_STMT_THROW:     cast_expr     throw_expr;
  case CAST_STMT_DECL:      cast_scope    decl;
};
```

6.2.5 Definitions and Declarations

Function Definitions

A `cast_func_def` represents a function definition. The `type` slot should be a function type that is usable in a definition, and the `block` slot is the function body.

```
struct cast_func_def
{
  cast_func_type  type;
  cast_block      block;
};

<type>
<block>
```

Variable Definitions

A `cast_var_def` represents a variable definition for a function.

```
enum cast_init_kind
{
  CAST_INIT_EXPR       = 1,
  CAST_INIT_AGGREGATE  = 2,
  CAST_INIT_CONSTRUCT  = 3  /* C++ Only */
};
union cast_init_u
switch (cast_init_kind kind)
{
  case CAST_INIT_EXPR:       cast_expr        expr;
  case CAST_INIT_AGGREGATE:  cast_init_array  subs;
```

```
  case CAST_INIT_CONSTRUCT:  cast_expr_array  exprs;
};
struct cast_var_def
{
  cast_type  type;

  /* Optional - if present, specifies the variable's initializer.  */
  cast_init  init;
};
```

Include Directives

A `cast_include` represents an include directive for the C preprocessor.

```
struct cast_include
{
  string    filename<>;
  bool      system_only;
};

<if system_only>
  #include <<filename>>
<else>
  #include "<filename>"
```

Using Directives

A `cast_using_kind` indicates the type of scope that is to be ‘used’.

```
/* C++ Only */
enum cast_using_kind
{
  CAST_USING_NAME       = 1,
  CAST_USING_TYPENAME   = 2,
  CAST_USING_NAMESPACE  = 3
};

using <cast_using_kind>
```

Direct Code

A `cast_direct` is used to inject any string into the current scope. For example, one can insert comments or preprocessor directives using this structure.

```
struct cast_direct
{
  string    code_string<>;
};
```

Union of All Definitions

The `cast_def` structure brings everything together.

```
enum cast_storage_class
{
  CAST_SC_NONE      = 0,
  CAST_SC_AUTO      = 1,
  CAST_SC_STATIC    = 2,
  CAST_SC_EXTERN    = 3,
  CAST_SC_REGISTER  = 4,
  CAST_SC_MUTABLE   = 5  /* C++ Only */
};
enum cast_def_kind
{
  CAST_TYPEDEF     = 0x00000001,
  ...
  CAST_FRIEND      = 0x00001000   /* C++ Only */
};
union cast_def_u
switch (cast_def_kind kind)
{
  case CAST_TYPEDEF:      cast_type         typedef_type;
  case CAST_TYPE:         cast_type         type;
  case CAST_FUNC_DECL:    cast_func_type    func_type;
  case CAST_FUNC_DEF:     cast_func_def     func_def;
  case CAST_VAR_DECL:     cast_type         var_type;
  case CAST_VAR_DEF:      cast_var_def      var_def;
  case CAST_DEFINE:       cast_expr         define_as;
  case CAST_INCLUDE:      cast_include      include;
  case CAST_DIRECT_CODE:  cast_direct       direct;
  case CAST_NAMESPACE:    cast_scope       *new_namespace;
  case CAST_USING:        cast_using_kind   using_scope;
  case CAST_LINKAGE:      cast_scope       *linkage;
  case CAST_FRIEND:       cast_type         friend_decl;
};

/* C++ Only */
enum cast_def_protection {
     CAST_PROT_NONE      = 0,
     CAST_PROT_PUBLIC    = 1,
     CAST_PROT_PROTECTED = 2,
     CAST_PROT_PRIVATE   = 3
};

struct cast_def
{
  /* C identifier of type (or whatever) to be declared/defined.
```

```
     Length 0 if none.  */
  cast_scoped_name    name;

  /* Storage class for the declaration/definition.  */
  cast_storage_class  sc;

  /* Description of definition.  */
  cast_def_u          u;

  /* The data_channel this definition is a part of */
  data_channel_index  channel;

  cast_def_protection  protection;  /* C++ Only */
};
```

6.3 The CAST Library

`cast_new_*`
: Many constructor functions are available so one does not have to build the structures manually.

`int cast_find_def(cast_scope **scope, const cast_scoped_name name, int kind)`
: Locates a definition with `name` in `scope`. The return value is either -1, if it was not found, or the index of the definition in `scope`. If the name is scoped then it will try to descend down into child scopes, and then set `scope` to the child scope where the definition was located. The `kind` parameter is a bitmask that causes the function to only match on definition kinds that are set in the mask.

`int cast_find_def_pos(cast_scope **scope, int scope_pos, const cast_scoped_name name, int kind)`
: Similar to `cast_find_def` except it allows one to start searching at a specific position. This is useful in case a definition match, but it was not the correct one. For example, a forward class declaration would match, but if one were searching for the actual class definition, one would have to do another `cast_find_def`.

`cast_type cast_find_typedef_type(cast_scope *scope, cast_type name_ctype)`
: Returns the real definition of a `typedef`.

`cast_enum_type *cast_find_enum_type(cast_scope *scope, cast_type ctype)`
: Returns the `cast_enum_type` that was specified with the passed in enumeration named type.

`cast_aggregate_type *cast_find_struct_type(cast_scope *scope, cast_type ctype)`
: Returns the `cast_aggregate_type` that was specified with the passed in struct named type.

`cast_aggregate_type *cast_find_union_type(cast_scope *scope, cast_type ctype)`
: Returns the `cast_aggregate_type` that was specified with the passed in union named type.

`cast_aggregate_type *cast_find_class_type(cast_scope *scope, cast_type ctype)`
: Returns the `cast_aggregate_type` that was specified with the passed in class named type.

`void cast_block_absorb_stmt(cast_block *b, cast_stmt st)`
Adds the statement to the `cast_block`. If the statement is also a `cast_block` then it will transfer any statements and declarations to the parent block.

`int cast_find_label(cast_block *block, const char *label)`
Returns the index of the `cast_label` in the block that matches given label

`extern cast_expr_array null_expr_array`
A global expression array available for initializing to an empty expression array.

`cast_expr_array cast_set_expr_array(cast_expr_array *array, cast_expr expr, ...)`
Set, or create if `array` is null, the values in an expression array.

`extern cast_init_array null_init_array`
Similar to null_expr_array, except for `cast_init`s.

`cast_init_array cast_set_init_array(cast_init_array *array, cast_init, ...)`
Similar to the `cast_expr` version.

`extern cast_template_arg_array null_template_arg_array`
Similar to null_expr_array, except for template arguments. This is handy for passing to a `cast_add_scope_name` since it requires a template argument array.

`cast_template_arg_array cast_set_template_arg_array(cast_template_arg_array *array, cast_template_arg template_arg, ...)`
Similar to the `cast_expr` version.

`cast_check(cast_scope *scope)`
Check the data structures to ensure they have valid data. This is done before CAST is written to a file or after it has been read from a file, usually as part of `pres_c_check`.

`int cast_expr_const(cast_expr expr)`
Returns true if `expr` is a constant expression, very helpful when trying to do optimizations.

`cast_expr aoi_const_to_cast_expr(aoi_const c)`
Converts an `aoi_const` to a `cast_expr`.

`int cast_cmp_type(cast_type a, cast_type b)`
Returns zero if the two types are the same, less than zero if type `a` is "less than" `b`, or greater than zero if type `a` is "greater than" `b`. The exact notion of lesser and greater are not exactly clear in all cases (e.g., if the types are completely different, the scalar difference between the two `kind` enumerations is returned). For similar (read "same") types, a more intelligent "difference" can be taken (e.g., two `CAST_TYPE_TYPENAME`s return the "string difference", similar to `strcmp`, of the two typenames).

`int cast_cmp_expr(cast_expr a, cast_expr b)`
Returns zero if the two expressions are the same, or nonzero based on a similar lesser/greater relationship as `cast_cmp_type`, above.

`int cast_cmp_init(cast_init a, cast_init b)`
Returns zero if the two initializers are the same, or nonzero based on a similar lesser/greater relationship as `cast_cmp_type`, above.

`cast_w_*()`
: Uses the `w_printf` functions to output the various CAST structures.

6.4 Summary and Comments

CAST was originally restricted to exactly the syntax of C, with minor accommodations for comments and preprocessor directives. Over time, CAST was extended to describe C++, and was also enhanced with certain pseudo-syntactic features such as implicit function arguments (`CAST_PARAM_IMPLICIT`), types used as expressions (`CAST_EXPR_TYPE`, so that types can be passed as arguments to macros), and label reference counts (so that unused labels can be silently dropped during output). These extra-syntactic features make it significantly easier to create PRES_C descriptions of stubs.

CAST allows Flick to create and modify C/C++ code at a fine-grain level, before it is output. This is an important feature that allows Flick to create code in ways that would be difficult to accomplish through other means such as template instantiation. Unfortunately, CAST's fine-grained nature also makes CAST an inconvenient data structure for some tasks, such as instantiating big blocks of mostly predetermined code. For code-building tasks that are not suited to CAST, Flick uses a separate intermediate representation called SCML, described in Chapter 8.

A second problem inherent to CAST it that — by design — it is tied to C and C++. CAST originally supported only C and was later extended for C++, but it would likely be infeasible to extend CAST to languages that are very unlike C. If Flick were extended to such languages, it would be useful to replace CAST with a more abstract code description language, designed around the logical organization of code (e.g., flow control) rather than around any specific language syntax. Flick would then map this abstract language onto concrete programming languages. This mapping could even be user-customizable.

Chapter 7

Presentation in C/C++ (PRES_C) Representation

A PRES_C file contains information describing the user-visible behavior of a set of stubs to be generated and anything else pertinent to the presentation. These files are used by the presentation generators to communicate their intent to the back ends. This makes it possible to separate the presentation, the C declarations and definitions, from the implementation.

The first few sections in this chapter describe the overall concepts embodied in the PRES_C format; the last section, Section 7.2, describes the actual XDR data structures representing PRES_C files.

7.1 PRES_C Overview

7.1.1 Stubs

A PRES_C file contains the blueprints for a set of *stubs*. Each of these blueprints, or *stub definitions*, completely defines one stub which will be produced when the PRES_C file is run through a C back end. A stub is a C function or data object which, when compiled and linked into a C program, can be invoked or otherwise utilized by that program as defined in the PRES_C file. There are several different types of stubs that can be defined in a PRES_C file (and hence produced by a back end).

Marshaling and Unmarshaling Stubs

These stubs are able to marshal and unmarshal the various data types defined by the presentation. The PRES_C file actually contains only stub "templates" for such functions, rather than the complete stub definition. This is the one exception to our stub definition above. The back end is responsible for "specializing" the stub for a particular parameter direction (and perhaps other aspects), based on how the stub is used. If a stub is used in multiple ways, multiple versions of the stub will appear in the generated code. For stubs that are not used, no code is generated. Because of Flick's aggressive inlining of the marshaling/unmarshaling code, most of these stubs are rarely produced.

Client Stubs

A client stub is a C function that can be called from anywhere in the program it is linked with. A client stub function packages up a request message, sends it to a server, and optionally waits for, receives, and unpacks a reply message from that server. The C function's parameters and return value may be used to pass data to and from the server, as well as to specify the target of the RPC (the object to be invoked) and to return error status. Exactly how these C function parameters are used is entirely defined by the client stub definition; there are no fixed rules that must apply to all stubs, such as the target object being specified in a particular parameter, or the return value being used to return an error status code. The name of the C client stub function is also defined in the PRES_C file; although it is usually based on the name of an RPC operation defined in an IDL file, it doesn't have to be.

A client stub, when invoked, generally performs the following basic steps, not necessarily in this exact order:

1. Determine the target of the RPC: e.g., extract the target object reference from the appropriate C function parameter.
2. Marshal the request message using the appropriate C function parameters.
3. Send the request message to the server.
4. Await and receive the server's reply message.
5. Allocate any storage needed for unmarshaled reply data and not already allocated by the caller of the client stub.
6. Unmarshal the reply message.
7. Deallocate any storage allocated by the caller to pass request data, which the caller expects the client stub to deallocate on its behalf.
8. Return the appropriate return value, typically a status code or a piece of unmarshaled reply message data.

Of course, for one-way client stubs, steps 4–6 are not performed. Steps 5 and 6 are usually combined in actual stub implementations produced by typical PBEs (see Chapter 12 for more information).

Server Skeletons

A server skeleton is the server-side analog to the client stub: it causes request messages received from clients to be converted into calls to C work functions in the server program. Unlike a client stub, a server skeleton is not generally a C function: instead, it is an opaque transport-specific C object (e.g., a global variable) which the server program uses as a "handle" when registering itself as a server. Once the server has registered one or more object with the ORB, it can begin accepting client messages. These messages are then dispatched to the appropriate object and skeleton to be processed.

Each skeleton acts like a dispatcher by decoding the message enough to figure out which server work function needs to be called and then unmarshaling the rest of the message. Server work functions tend to correspond to operations in interfaces, but some may be implicitly defined by other definitions from the IDL. Each work function that can be called from a particular server skeleton can have a different C function signature, and can use its parameters and return value in a different way: a server skeleton definition in a PRES_C file contains a separate function parameter/return value mapping description for each supported server work function.

When a request message is received from a client, the server skeleton generally causes the following operations to be performed, not necessarily in this exact order:

1. Determine which server work function is to be called, based on the contents of the request message.

2. Allocate any storage needed to hold unmarshaled request message data to be passed to the server work function.

3. Unmarshal the request message.

4. Determine which specific server object was invoked by the client, and indicate this to the server work function.

5. Allocate any storage for reply data from the server work function.

6. Call the server work function.

7. Marshal the reply message. The marshaled data may include data structures referred to by pointer parameters passed to the work function, which the work function filled in or modified during its execution, and/or the data returned by the work function in its return value.

8. Send the marshaled reply message back to the client.

9. Deallocate any storage used to hold reply message data before being marshaled, which the server work function expects the skeleton to deallocate on return. This storage could have been allocated in step 5 above, or by the server work function itself.

10. Deallocate any storage allocated above to hold unmarshaled request message data, which the server work function expects the skeleton to deallocate automatically on return (i.e., storage that the server work function doesn't "take control of" and deallocate itself when finished with it).

11. Return to the ORB so that it can process subsequent messages.

Of course, for one-way RPC requests, steps 7–9 are not performed. Steps 2 and 3 are usually combined in actual server skeletons produced by typical PBEs. Steps 7 and 9 may also be combined in some cases. See Chapter 12 for more information about PBE implementation.

Client Skeletons

Used in clients that use an event-based programming model. Not all PBEs or transport protocols may support client skeletons.

A client skeleton is the client-side equivalent of the server skeleton. Rather than dispatching requests to appropriate server work functions, a client skeleton dispatches reply messages to appropriate handlers, or client work functions. It is responsible for decoding the reply message enough to figure out with which client request it is associated, and calling the reply handler with any information the client saved when the request was made.

As on the server side, a "handle" or pseudo-object reference is used to identify the client skeleton expecting a reply message for the original request. The handle is passed to the server, and the reply is much like a one-way request to the pseudo-object client.

Unlike the server skeleton, however, the client skeleton does not bother to unmarshal the reply message (beyond the point that it needs to determine how to dispatch it), nor does it need to worry about marshaling data returned by the reply handler (there is no reply to the reply inherent in the model, although such functionality is certainly possible within the model).

When a reply message is received from a server, the client skeleton generally causes the following operations to be performed, not necessarily in this exact order:

1. Determine which client work function (reply handler) is to be called, based on the contents of the reply message.
2. Determine which specific client pseudo-object was "invoked" by the server, and indicate this to the reply handler.
3. Call the reply handler.
4. Return to the ORB so that it can process subsequent messages.

Send and Receive Stubs

Used in clients and servers that use an event-based programming model. Not all PBEs or transport protocols may support send and receive stubs.

Send stubs are responsible only for sending a message to either the client or server. They return immediately to the caller, regardless of whether a reply is expected (in the case of a request to the server). Note that a send stub is similar to a one-way client stub in that the message is sent and control returns without waiting for a reply. However, they different from each other in three important ways. First, a send stub is not responsible for marshaling any data; it requires a pre-marshaled message as a parameter. Second, a one-way client stub is used when no reply at all is expected for the RPC invocation; a send stub is used even if a reply is expected, but its receipt will be deferred until later in the client program's execution. Third, a send stub is not limited to the client; they exist on the server side to issue replies back to the client. They are more in harmony with the concepts of message passing than with traditional RPC.

Receive stubs are not well defined for Flick. Ideally, they represent a method of either polling or waiting for the reply from a previous send stub, although the implementation of such a stub has not been laid out for any of Flick's runtime layers. Currently, the supported method of receiving a reply is via the client skeleton.

In a sense, a send stub acts as a "thread fork," allowing both the client and the server to work in parallel. Similarly, a receive stub acts as a corresponding "join" operation, synchronizing the client thread with the server thread and returning the server's result.

Note that although send and receive stubs are not explicitly responsible for marshaling or unmarshaling any data, implementationally they may do a small amount. For example, a CORBA/IIOP client send stub must encode the GIOP header and server object reference into the message before it can be sent to the server.

Continuation Stubs

Used in clients and servers that use an event-based programming model. Not all PBEs or transport protocols may support continuation stubs.

If for some reason a work function or reply handler is not able to complete (perhaps required data has not yet arrived to finish the computation), it may request that the work be deferred until a later time. A continuation stub is used to indicate this to the runtime layer, and provide a way to save partially computed results so the work can be continued later (rather than being entirely restarted). It is left up to the runtime how to "awake" one of these continued work functions to continue its processing.

Message Marshaling and Unmarshaling Stubs

Used in clients and servers that use an event-based programming model. Not all PBEs or transport protocols may support message marshal/unmarshal stubs.

Like their marshal and unmarshal stub cousins, they are responsible for encoding and decoding data. However, rather than working on specific types, they focus on the entire message for a request or reply. By invoking a message marshal stub, one receives an opaque message object ready to be sent to a server (or multiple servers). Likewise, by invoking a message unmarshal stub, one can decode a message object into the appropriate parameters.

7.1.2 Presentation Attributes

For some presentations there are peripheral data types or just extra information that needs to be available in order to produce the correct code. For instance, the CORBA C++ mapping requires the IDL compiler to generate smart pointer classes for many of the defined types. Most of this work can be accomplished during presentation generation and simply added to the CAST part of the PRES_C file; however, producing the bodies of any functions is an implementation issue which should be left to the back end. Thus, we must have a generic way of storing information in the PRES_C, whether it is presentation functions or any other things of interest. The solution was to make a data structure in XDR that could store tagged data so we could create semantically meaningful but otherwise arbitrary data values that could be stored in the PRES_C file.

7.1.3 Marshaling and Unmarshaling

Most of the types of stubs listed above perform marshaling and/or unmarshaling of messages in some context: after all, marshaling and unmarshaling is the central mechanism behind RPC. Marshaling can be defined as conversion of data from language-specific presented data formats (C-language data types in this case) to a language-independent in-transit message format useful for passing the data between a client and a server; unmarshaling is the opposite conversion. There are an infinite number of ways message data can be presented in C, many of them reasonable and commonly used in different RPC systems. Providing a convenient representation of a wide variety of these possible presentations is the primary challenge for the PRES_C file format.

To this end, each PRES_C file contains three main parts:

- A message interface (MINT) type graph, representing the complete on-the-wire message type signature used by the stubs. See Chapter 5 for more information on MINT type graphs.
- A C abstract syntax tree (CAST), representing the complete set of C-language definitions and declarations for the C-language presentation the stubs are required to implement. The CAST stored in a PRES_C file, when converted to C source code, directly becomes the header file to be included by the RPC programmer in client or server code that uses the generated stubs. See Chapter 6 for more information on the CAST format.
- A set of stub definitions, as described above, each containing references into the MINT and CAST sections of the file above, and *mapping* information describing how one corresponds to the other.

Thus, each stub definition in a PRES_C file refers to a "common database" of message type definitions (the MINT), and a "common database" of C-language definitions and declarations (the CAST). For example, to define how a client stub is to marshal a request message and unmarshal the reply, each client stub definition contains:

- A reference to a subsection of the MINT type graph representing the message that the client stub will send to the server.
- A reference to another subsection of the MINT type graph representing the reply message the stub expects to get back from the server.

- A reference to the C function prototype declaration in the CAST defining the C-language type signature this particular client stub will have.

- A tree of *mapping* data structures defining how the C-language parameters to the client stub are to be converted into the request message defined above in the MINT.

- Another tree of mapping structures defining how the server's reply message is to be converted back into presented C-language data types, either by filling in pass-by-reference C data objects whose addresses were passed to the client stub through C parameters, or directly through the client stub function's return value.

In general, the C type definitions are usually more complicated than their corresponding MINT types, although in a few cases the MINT actually contains more information. For example, MINT integer types specify an exact range of allowable values, whereas C integer types are merely the well-known `char`, `int`, `long`, etc.

Mapping Between Presentation and Message

The stubs definitions are actually made up of a number of data structures defined in `pres_c.x` and referred to as "nodes". These nodes are connected together to form a tree that describes how to map to and from CAST and MINT. The nodes themselves are split into two classes of structures called "mappings" and "inlines". A mapping node is used to translate between whole types, a CAST int to a MINT int, for example. An inline node works on an exploded type, for example, a mapping could match to a C structure, but an inline is used to access the slots of the struct. Switching between these modes is done when a `PRES_C_INLINE_ATOM` is used to select what to map from an inline, and when a `PRES_C_MAPPING_STRUCT` is used to expand into an inline. For example, a pointer to a C structure would be a mapping that then referred to an inline, since the pointer is a primitive, but once it has been dereferenced, you are left with a structure.

7.1.4 Memory Allocation and Deallocation

Memory allocation and deallocation must be dealt with whenever any level of pointer indirection appears in the C-language type signatures used by a stub. Basically, any pointer must be initialized to point to *something* before use: either to memory on the stack, the heap, or even to a global variable. Similarly, that memory must be appropriately deallocated after it is no longer needed. Pointer indirection is a necessary part of a viable C presentation for any non-trivial IDL, and different IDLs specify different ways of initializing and releasing these pointers.

To handle a wide variety of different memory allocation and deallocation semantics, the PRES_C format associates some standard memory management information with each level of pointer indirection that appears in the C types a stub uses. This memory management information is contained within a `PRES_C_INLINE_ALLOCATION_CONTEXT` node, along with references to `pres_c_inline` nodes describing the pointer and other associated information (e.g., the length of a sequence, or CORBA's sequence ownership/release flag). Within these child inlines, PRES_C mapping nodes (typically a `PRES_C_MAPPING_POINTER` node; see Section 7.2.4 below) refer back to their allocation context by its unique name. This memory management information specifies:

- Whether, and under what circumstances, the stub should allocate memory and initialize the pointer. If the pointer isn't initialized by the stub, then the stub expects it to have been initialized already by the program code that interacts with the stub.

- Whether, and under what circumstances, the stub should deallocate the memory referred to by the pointer. If the memory isn't deallocated by the stub, then the stub expects the program to deallocate it at some other time at its own convenience.

- If the stub is to allocate or deallocate memory, what memory allocation mechanism the stub should use. Many operating systems have several "levels" of memory allocation: for example, Mach has the low-level page-granular `vm_allocate` mechanism in addition to the high-level `malloc` mechanism provided by the C library; Windows provides three different types of memory management functions (global, local, and heap) in addition to `malloc`. If a piece of memory allocated using one mechanism is freed using another, the process is likely to get hopelessly confused. Thus, whenever a stub may "transfer" a memory region to the programmer or vice versa, they must agree on how the memory is to be allocated.

Although the memory allocation and deallocation semantics are specified in the same mapping data structure tree that is used to specify how marshaling and unmarshaling code is generated, this does not mean that allocation and deallocation occurs at the same time as marshaling or unmarshaling. For example, in the case of a server skeleton that must allocate memory to be used to hold data to be marshaled into the reply message, the memory must be allocated *before* the server work function is called, so the work function will have a place to fill in its reply data; on the other hand the actual marshaling of the reply message can only be done *after* the work function is called, when the data has been filled in. In general, the point at which memory is allocated or freed depends on the type of stub; see Section 7.1.1 above for details on when this happens in each case.

Although each pointer mapping node in a stub's mapping tree specifies both allocation and deallocation semantics for that pointer, in some cases one or the other of these is meaningless. For example, a client stub has no opportunity to allocate memory for request data, because the request data must already be allocated and ready to marshal from when the client stub is first entered. Similarly, a client stub has no opportunity to deallocate memory holding reply data, since the only time it could do so would be during the execution of the stub, and the calling program will not be able to examine the reply data until after the stub returns. Therefore, for client stubs, allocation semantics for request data and deallocation semantics for reply data are ignored.

In cases where a pointer is not merely a simple level of pointer indirection, but in fact holds the address of the first element in an *array* of elements, the length of the block of memory allocated or deallocated naturally depends on the length of the array.

Pointer Aliasing

In some cases, two pointer mapping nodes in a PRES_C file may refer to the *same* actual pointer variable at runtime. For example, most IDLs support a notion of an "inout" parameter, where data for that parameter is to be sent in both the request and the reply message, but the parameter appears only once in the C function prototype for the stub or work function. This parameter is a pointer to a data area to be used for both the request and the reply: the request data is expected to be overwritten during the RPC with reply data. In cases like this, care must be taken that the pointer is not allocated and initialized twice, causing memory to be irrevocably lost, and (even more importantly) is not freed twice.

Presentation back ends that take PRES_C files as input cannot generally be expected to recognize when two pointer mapping nodes refer to the same pointer. Instead, the producer of the PRES_C file must take care to ensure that at most one of those nodes specify memory allocation, and at most one specifies deallocation. If allocation for such a pointer is to be done by the stub, then the allocation semantics must be specified only in the pointer mapping node corresponding to the operation that is done *first*; the other mapping node must specify PRES_C_ALLOC_NEVER. Similarly, if deallocation is to be done by the stub, then the deallocation semantics must be specified only in the mapping node corresponding to the operation that is done *last*. For example, in the case of a server skeleton, allocation semantics for an "inout" pointer parameter must be specified in the pointer mapping node for the unmarshaled request data, whereas deallocation semantics for that same pointer parameter must be specified in the pointer mapping node for the marshaled reply data.

7.1.5 Exceptions

Exceptions as described in the IDL must be expressed in the PRES_C so they can be properly marshaled, unmarshaled, and handled in an appropriate way. Since exceptions can abstractly be viewed as just another data type to be passed over the wire, there isn't any provision for a special exception mapping in PRES_C. However, they do make their presence known by slightly changing the structure of reply messages. Since an exception can occur based on data given to the server by a client, an exception message may be returned by the server in place of the normal reply message. The exception may either be a system-defined exception due to a mal-formed request message (e.g., an undefined operation or unknown object reference), or the exception may be user-defined, generated by the server-side work function indicating the request could not be fulfilled for whatever reason. This creates a union, if you will, of possible reply messages – one that contains the normal reply, one that identifies a system exception, and one that identifies a user exception. The PRES_C represents this as would be expected, a union of the three possible reply messages, the environment or status code in the reply being the discriminator as to which message actually exists on the wire. The method of communicating the exception to the user is handled mostly by the back end; only the presented type and name of the parameter is specified in the PRES_C.

7.2 PRES_C Data Structures

The top-level PRES_C structure is defined in mom/pres_c.x. This structure contains the necessary MINT, CAST, and tag information for producing stubs and any associated presentation.

```
struct pres_c_1
{
        mint_1      mint;
        cast_1      cast;
        cast_1      stubs_cast;
        cast_1      pres_cast;
        aoi         a;
        pres_c_stub stubs<>;
        string      pres_context<>;
        cast_expr   error_mappings<>;
        data_channel_mask       unpresented_channels<>;
        tag_list    *pres_attrs;
        meta        meta_data;
        int         cast_language;
};
```

How individual elements in the MINT and CAST structures link together is specified by four kinds of data types. First, stub types describe the functions generated by Flick necessary for the IPC model. Second, inline data structures traverse the MINT data structure, or link together a MINT interface type to one or more C function parameters. Third, mapping data structures map one C function parameter in to the MINT. Finally, allocation types describe the allocation semantics associated with various pieces of the generated code.

Let us look at each aspect of the PRES_C structure in more detail.

7.2.1 Stub Types

Stubs come in two forms. The first form transforms data between a MINT interface type and a complex data structure. These stubs are called marshal and unmarshal stubs. The second and more common form implements the IPC mechanism between client and server. These are known as client stubs and server skeletons (or dispatch routines).

```
union pres_c_stub
switch (pres_c_stub_kind kind)
{
        case PRES_C_MARSHAL_STUB:       pres_c_marshal_stub     mstub;
        case PRES_C_UNMARSHAL_STUB:     pres_c_marshal_stub     ustub;
        case PRES_C_CLIENT_STUB:        pres_c_client_stub      cstub;
        case PRES_C_SERVER_STUB:        pres_c_server_stub      sstub;
        case PRES_C_CLIENT_SKEL:        pres_c_skel             cskel;
        case PRES_C_SERVER_SKEL:        pres_c_skel             sskel;
        case PRES_C_SEND_STUB:          pres_c_msg_stub         send_stub;
        case PRES_C_RECV_STUB:          pres_c_msg_stub         recv_stub;
        case PRES_C_MESSAGE_MARSHAL_STUB:   pres_c_msg_marshal_stub     mmstub;
        case PRES_C_MESSAGE_UNMARSHAL_STUB: pres_c_msg_marshal_stub     mustub;
        case PRES_C_CONTINUE_STUB:      pres_c_continue_stub    continue_stub;
};
```

Let us look at each PRES_C stub type in more detail.

`PRES_C_MARSHAL_STUB`, `PRES_C_UNMARSHAL_STUB`

Specifies a stub function which marshals/unmarshals a data structure. Although they are generated for every data type defined in the IDL (and in some cases the presentation itself), these stubs are only really needed to handle the complex structures which don't have a default mapping onto a MINT type. Structures with pointers often fall into this category.

Implementationally, both marshal and unmarshal stubs are represented by the same data structure. However, they are separate members of the stub union, and should thus be treated separately.

The marshal/unmarshal structure contains references to the stub function definition in the CAST, and to the interface type in the MINT. The `pres_c_inline` structure links them together. The `seethru_map` slot exports the internal workings of the stub so that it can be easily inlined in other code.

```
struct pres_c_marshal_stub /* Both marshal and unmarshal */
{
        cast_ref        c_func;
        mint_ref        itype;
        pres_c_inline   i;
        /* How the connection to marshal to is specified.  */
```

```
        pres_c_inline   target_i;

        /* The PRES_C mapping that allows us to "see through" or "see into"
           this stub.  We need this mapping when we want to inline the body of
           this marshal stub into a client stub, a server function, etc. */
        pres_c_mapping seethru_map;
};
```

PRES_C_CLIENT_STUB

This is a representation of a client stub. References are kept to the function definition in the CAST structure and to the request and reply interface types in the MINT structures. Two `pres_c_inline` structures link the request and reply types to the C function parameters in the CAST.

```
struct pres_c_client_stub
{
        cast_ref                c_func;
        pres_c_stub_op_flags    op_flags;
        mint_ref                request_itype;
        mint_ref                reply_itype;
        pres_c_inline           request_i;
        pres_c_inline           reply_i;
        mint_ref                target_itype;
        pres_c_inline           target_i;
        mint_ref                client_itype;
        pres_c_inline           client_i;
        mint_ref                error_itype;
        pres_c_inline           error_i;
};
```

PRES_C_SERVER_SKEL, PRES_C_CLIENT_SKEL

For each interface there is at least a server-side skeleton in the PRES_C structure. In some cases, a client-side skeleton will also exist if you have selected the decomposed stubs option. In both cases, the skeleton structure holds a reference to the CAST for the function prototype, the root MINT for all incoming and outgoing messages, and the list of supported functions to which the skeleton can dispatch.

Following are the data structures relevant to both client and server skeletons:

```
struct pres_c_skel
{
        cast_ref                c_def;
        mint_ref                request_itype;
        mint_ref                reply_itype;
        pres_c_func             funcs<>;
};

struct pres_c_server_func
{
```

```
        cast_ref                c_func;
        pres_c_stub_op_flags    op_flags;
        mint_ref                simple_msg_itype;
        pres_c_inline           msg_i;
        mint_ref                target_itype;
        pres_c_inline           target_i;
        mint_ref                client_itype;
        pres_c_inline           client_i;
        mint_ref                error_itype;
        pres_c_inline           error_i;
};
```

A `pres_c_func` is either a `pres_c_server_func` or `pres_c_receive_func`, depending on the stub model in place (typical RPC or event-based message passing). Each is defined almost identical to the `pres_c_client_stub` above.

`PRES_C_SEND_STUB`, `PRES_C_RECV_STUB`

These stubs are used in an event based system to send or receive a pre-marshaled message. Note that the target object is still associated with the send or receive stub. This facilitates the reusability of pre-marshaled messages, in that they can be cached and/or sent to multiple targets. These stubs are non-blocking so waiting for or producing a reply must be done explicitly.

```
struct pres_c_msg_stub
{
        cast_ref        c_func;
        mint_ref        msg_itype;
        pres_c_inline   msg_i;
        mint_ref        target_itype;
        pres_c_inline   target_i;
        int request;
};
```

`PRES_C_MESSAGE_MARSHAL_STUB`, `PRES_C_MESSAGE_UNMARSHAL_STUB`

These stubs are used to marshal and unmarshal a complete message corresponding to a specific interface operation. These are used to build or decode the pre-marshaled messages used by the send and receive stubs described above.

```
struct pres_c_msg_marshal_stub
{
        cast_ref                c_func;
        pres_c_stub_op_flags    op_flags;
        mint_ref                itype;
        pres_c_inline           i;
        int                     client;
        int                     request;
        string                  reply_handler_name<>;
};
```

PRES_C_CONTINUE_STUB

A continuation stub is used to notify the runtime that a work function has not fully completed, but would like to defer its computation until a later time.

```
struct pres_c_continue_stub
{
        cast_ref        c_func;
        mint_ref        itype;
        pres_c_inline   i;
        int request;
};
```

7.2.2 Presentation Attribute Types

The `tag_data` structure is used to hold arbitrary information, but with a particular semantic meaning. This provides the hooks necessary for a back end to generate appropriate implementations for certain pieces of the presentation. For example, a smart pointer class for a data type is something defined by the presentation. However, the implementation must be given by the back end. The tags are used to give semantic meaning to the presentation pieces, allowing the back end to generate an appropriate implementation.

In general, tags are stored in a `tag_list`, which provides a hierarchical list structure. Lists contain `tag_items`, which hold a tag name and its corresponding data. The following is the definition of the tag infrastructure in the PRES_C file:

```
/* Tags are used for storing generic data which
   have some properties specified by the tag */
enum tag_data_kind
{
        TAG_NONE                        = 0x0001,
        ...
        TAG_CAST_INIT                   = 0x000f,

        TAG_ANY_ARRAY                   = 0x8002,
        ...
        TAG_CAST_INIT_ARRAY             = 0x800f
};
const TAG_ARRAY                         = 0x8000;

typedef struct tag_list *tag_list_ptr;
typedef string tag_string_array<>;
typedef char   tag_object_array<>;

union tag_data
switch (tag_data_kind kind)
{
        case TAG_NONE:                          void;
        case TAG_ANY:                           void;
        case TAG_TAG_LIST:                      tag_list_ptr            tl;
```

```
        case TAG_BOOL:                         char                   b;
        case TAG_STRING:                       string                 str<>;
        case TAG_INTEGER:                      int                    i;
        case TAG_FLOAT:                        float                  f;
        case TAG_REF:                          string                 ref<>;
        case TAG_OBJECT:                       tag_object_array       obj;
        case TAG_CAST_SCOPED_NAME:             cast_scoped_name       scname;
        case TAG_CAST_DEF:                     cast_def_t             cdef;
        case TAG_CAST_TYPE:                    cast_type              ctype;
        case TAG_CAST_EXPR:                    cast_expr              cexpr;
        case TAG_CAST_STMT:                    cast_stmt              cstmt;
        case TAG_CAST_INIT:                    cast_init              cinit;

        case TAG_TAG_LIST_ARRAY:
                tag_list_ptr            tl_a<>;
        case TAG_BOOL_ARRAY:
                char                    b_a<>;
        case TAG_STRING_ARRAY:
                tag_string_array        str_a<>;
        case TAG_INTEGER_ARRAY:
                int                     i_a<>;
        case TAG_FLOAT_ARRAY:
                float                   f_a<>;
        case TAG_REF_ARRAY:
                tag_string_array        ref_a<>;
        case TAG_OBJECT_ARRAY:
                tag_object_array        obj_a<>;
        case TAG_CAST_SCOPED_NAME_ARRAY:
                cast_scoped_name        scname_a<>;
        case TAG_CAST_DEF_ARRAY:
                cast_def_t              cdef_a<>;
        case TAG_CAST_TYPE_ARRAY:
                cast_type               ctype_a<>;
        case TAG_CAST_EXPR_ARRAY:
                cast_expr               cexpr_a<>;
        case TAG_CAST_STMT_ARRAY:
                cast_stmt               cstmt_a<>;
        case TAG_CAST_INIT_ARRAY:
                cast_init               cinit_a<>;
};

struct tag_item
{
        string          tag<>;
        tag_data        data;
};
```

```
struct tag_list
{
        struct tag_list *parent;
        tag_item        items<>;
};
```

7.2.3 Inline Types

PRES_C inline types are used to map between high-level MINT and CAST types and also to traverse the MINT or CAST structures separately without directly mapping between the two. They control the presentation of interfaces to the given language and control stub generation.

```
union pres_c_inline_u
switch (pres_c_inline_kind kind)
{
        case PRES_C_INLINE_ATOM:
                pres_c_inline_atom                      atom;

        case PRES_C_INLINE_STRUCT:
                pres_c_inline_struct                    struct_i;
        case PRES_C_INLINE_FUNC_PARAMS_STRUCT:
                pres_c_inline_func_params_struct        func_params_i;
        case PRES_C_INLINE_HANDLER_FUNC:
                pres_c_inline_handler_func              handler_i;

        case PRES_C_INLINE_STRUCT_UNION:
                pres_c_inline_struct_union              struct_union;
        case PRES_C_INLINE_EXPANDED_UNION:
                pres_c_inline_expanded_union            expanded_union;
        case PRES_C_INLINE_VOID_UNION:
                pres_c_inline_void_union                void_union;
        case PRES_C_INLINE_COLLAPSED_UNION:
                pres_c_inline_collapsed_union           collapsed_union;
        case PRES_C_INLINE_VIRTUAL_UNION:
                pres_c_inline_virtual_union             virtual_union;

        case PRES_C_INLINE_TYPED:
                pres_c_inline_typed                     typed;

        case PRES_C_INLINE_THROWAWAY:
                void;

        case PRES_C_INLINE_XLATE:
                pres_c_inline_xlate                     xlate;
        case PRES_C_INLINE_ASSIGN:
                pres_c_inline_assign                    assign;
        case PRES_C_INLINE_COND:
                pres_c_inline_cond                      cond;
```

```
        case PRES_C_INLINE_MESSAGE_ATTRIBUTE:
                pres_c_inline_message_attribute         msg_attr;
        case PRES_C_INLINE_ALLOCATION_CONTEXT:
                pres_c_inline_allocation_context        acontext;
        case PRES_C_INLINE_TEMPORARY:
                pres_c_temporary                        temp;

        case PRES_C_INLINE_ILLEGAL:
                void;
};
```

Let us look at each PRES_C inline type in detail.

`PRES_C_INLINE_ATOM` Type

This is the fundamental inline type and is used to map one single, simple MINT type to one single C type. It also marks the transition from inline mode to mapping mode, by selecting CAST member `index` from the current inline state and passing that to the mapping.

```
struct pres_c_inline_atom
{
        pres_c_inline_index     index;
        pres_c_mapping          mapping;
};
```

`PRES_C_INLINE_STRUCT` Type

This inline type contains one `pres_c_inline` structure for each struct member. This usually corresponds to a container of some kind, or something in the C and MINT with multiple attributes.

```
struct pres_c_inline_struct
{
        pres_c_inline           slots<>;
};
```

`PRES_C_INLINE_STRUCT_UNION` Type

This inline type maps a MINT union into a C struct containing a discriminator tag and the union. For example, a `MINT_UNION` containing a `MINT_INTEGER` and `MINT_CHAR` might have a C mapping like this:

```
struct example_union
{
    mint_type_tag kind;
    union
    {
        int i;
        char c;
    }
}
```

```
struct pres_c_inline_struct_union_case
{
        int                     index;
        pres_c_mapping          mapping;
};

struct pres_c_inline_struct_union
{
        pres_c_inline_atom                      discrim;
        pres_c_inline_index                     union_index;
        pres_c_inline_struct_union_case         cases<>;
        pres_c_inline_struct_union_case         *dfault;
};
```

PRES_C_INLINE_VOID_UNION Type

This inline type maps a MINT union into a C struct containing a discriminator tag and a `void *` pointer to the data. The previous example now might have a C mapping like this:

```
struct example_union
{
    mint_type_tag kind;
    void *data;
}

struct pres_c_inline_void_union_case
{
        cast_expr case_value;
        cast_type type;
        pres_c_mapping mapping;
};

struct pres_c_inline_void_union
{
        pres_c_inline_atom                      discrim;
        pres_c_inline_index                     void_index;
        pres_c_inline_void_union_case           cases<>;
        pres_c_inline_void_union_case           *dfault;
};
```

PRES_C_INLINE_EXPANDED_UNION Type

This inline type maps a MINT union into a C struct containing a discriminator tag and slots for each member of the union. This isn't currently used in Flick.

```
struct pres_c_inline_expanded_union
{
        pres_c_inline_atom      discrim;
```

```
        pres_c_inline           cases<>;
        pres_c_inline           dfault;
};
```

PRES_C_INLINE_COLLAPSED_UNION Type

This inline type forces the selection of a given member of union. Only unions with this member selected will be accepted. This type is often used to traverse down the MINT structure (e.g., the union of all interfaces, and the union of all operations) until a certain section of the MINT structure is reached (e.g., the request of a specific operation), holding fixed the various unions in the MINT along this path.

```
struct pres_c_inline_collapsed_union
{
        mint_const              discrim_value;
        pres_c_inline           selected_case;
};
```

PRES_C_INLINE_VIRTUAL_UNION Type

This inline type represents an "arbitrary" union. It's main purpose is to allow the return from an operation to be either the regular result, a system error, or a user-defined exception. Since the implementation is really left up to the back end, the presentation generator leaves this as an arbitrary union of the three possibilities.

```
typedef pres_c_inline pres_c_inline_virtual_union_case;

struct pres_c_inline_virtual_union
{
        string arglist_name<>; /* Name of associated arglist. */
        pres_c_inline_virtual_union_case discrim;
        pres_c_inline_virtual_union_case cases<>;
        pres_c_inline_virtual_union_case dfault;
};
```

PRES_C_INLINE_THROWAWAY Type

Valid for receive-only mappings, it ignores any MINT or CAST that may remain in the current subtree, effectively throwing away their values.

PRES_C_INLINE_ALLOCATION_CONTEXT Type

An allocation context is the main structure used for dealing with all types of allocation. It does not correspond to any MINT or CAST directly, but provides context for (usually) a pointer type, such as length, maximum allocation size, allocation semantics, etc. In some cases, these are known constants, while at other times the values can be known only at runtime and are included as part of the data structure (e.g., a CORBA sequence's length and maximum fields).

```
struct pres_c_inline_allocation_context
{
        /* Allocation context: */
```

```
        string arglist_name<>;          /* Name of the context's arglist. */

        /*
         * XXX - The arguments below marked with 'XXX' are NOT IMPLEMENTED.
         * There is minimal support in place for handling them, but they are
         * not currently being used to support any particular allocation
         * semantics.
         */
/*XXX*/ pres_c_inline     offset;       /* First element to m/u */
        pres_c_inline     length;       /* # of elements to m/u */
        pres_c_inline     min_len;      /* [hard] minimum (usu. IDL const) */
        pres_c_inline     max_len;      /* [hard] maximum (usu. IDL const) */
        pres_c_inline     alloc_len;    /* allocated # of elements */
        pres_c_inline     min_alloc_len;/* minimum length of allocation */
/*XXX*/ pres_c_inline     max_alloc_len;/* maximum length of allocation */
        pres_c_inline     release;      /* release of struct-owned buffer */
        pres_c_inline     terminator;   /* buffer termination */
        pres_c_inline     mustcopy;     /* must copy data to keep it */
/*XXX*/ int               overwrite;    /* 'out' buffer is preallocated */
/*XXX*/ allocation_owner  owner;        /* entity ownership */

        pres_c_allocation alloc;        /* the allocation semantics */

        /*
         * The real inline to be handled:
         * 'ptr' must eventually lead to a PRES_C_MAPPING_INTERNAL_ARRAY node,
         * a PRES_C_MAPPING_POINTER node, or a PRES_C_MAPPING_OPTIONAL_POINTER
         * node, which is responsible for actually instantiating the
         * allocation.
         */
        pres_c_inline     ptr;
};
```

PRES_C_INLINE_XLATE Type

This inline transforms the current CAST type to a known, fixed type. This is useful when performing conversions between the presented type of a parameter and the type with which it is viewed by the stubs. For example, MIG allows the user to name somewhat abstract data structures (e.g., one can specify the size of a structure, but one cannot explicitly specify its contents or layout), but the stubs must have a concrete representation in order to marshal or unmarshal the data (e.g., a structure of a known size can be blindly copied into and out of the data stream). This node facilitates the conversion between the user-defined type (unknown to the IDL compiler) and an equivalent type the stubs can use to handle the data.

```
struct pres_c_inline_xlate
{
        /* the _real_ pres_c_inline... */
        pres_c_inline           sub;
```

```
        pres_c_inline_index     index;
        cast_type               internal_ctype;

        string                  translator<>;
        string                  destructor<>;
};
```

PRES_C_INLINE_ASSIGN Type

This inline allows an arbitrary CAST expression to be assigned to the current entity. This isn't currently used in Flick.

```
struct pres_c_inline_assign
{
        /* the _real_ pres_c_inline... */
        pres_c_inline           sub;

        pres_c_inline_index     index;
        cast_expr               value;
};
```

PRES_C_INLINE_COND Type

This is very similar to a union inline, except that it refers to a boolean "discriminator". The runtime value of the discriminator will select which inline is actually part of the execution path.

```
struct pres_c_inline_cond
{
        /* This index must refer to a boolean C variable
           (or at least an integerish type that can be used as a boolean).
           At runtime, if the value of that C variable(/parameter/slot)
           is true, marshal using true_inl; otherwise use false_inl.  */
        pres_c_inline_index     index;
        pres_c_inline           true_inl;
        pres_c_inline           false_inl;
};
```

PRES_C_INLINE_MESSAGE_ATTRIBUTE Type

This inline allows an "attribute" to be embedded in the tree, describing a semantic entity that is entirely left up to the back end to decide how to implement. There is no associated CAST or MINT for this inline.

```
enum pres_c_message_attribute_kind
{
        /* sets some attribute flags */
        PRES_C_MESSAGE_ATTRIBUTE_FLAGS = 0,

        /* how long to wait for reply */
        PRES_C_MESSAGE_ATTRIBUTE_TIMEOUT = 1,
```

```
        /* index of when received */
        PRES_C_MESSAGE_ATTRIBUTE_SEQUENCE_RECEIVED = 2,

        /* client reference */
        PRES_C_MESSAGE_ATTRIBUTE_CLIENT_REFERENCE = 3,

        /* flag indicating work function should make its own copy of
           the data if it wants to keep it */
        PRES_C_MESSAGE_ATTRIBUTE_SERVERCOPY = 4

        /* maybe in the future, SID will be moved here */
};

struct pres_c_inline_message_attribute
{
        pres_c_message_attribute_kind   kind;
};
```

`PRES_C_INLINE_HANDLER_FUNC` Type

This inline describes a function to be called to handle the PRES_C tree that normally would be rooted in this location. In the decomposed presentation style, this replaces the `pres_c_inline_func_params_struct`. The slots all represent parameters associated with the presentation, such as the object reference, marshaled message, environment, etc.

```
struct pres_c_inline_handler_func
{
        /*
         * One element for each parameter to the handler function.
         * These are all ``special'' presentation-only parameters.
         */
        pres_c_inline_struct_slot slots<>;
};
```

`PRES_C_INLINE_TEMPORARY` Type

This inline introduces a new (temporary) variable into the scope for the given mapping, rather than selecting a member of the inline state (a parameter or data member). This allows one to marshal/unmarshal data that isn't explicitly presented (e.g., a string's length or an exceptional reply when there is no environment parameter). In some cases, this may even be a constant that needs to be "hardcoded" into the PRES_C file (in which case it is optimized to be a constant instead of a temporary variable). This is common in allocation contexts where important pieces of the context are known, such as the length of a fixed array.

```
enum pres_c_temporary_type
{
        TEMP_TYPE_PRESENTED,
        TEMP_TYPE_ENCODED
```

```
};
struct pres_c_temporary
{
        pres_c_mapping          map;    /* The mapping for the data */

        string                  name<>; /* A hint used in naming the temp */
        cast_type               ctype;  /* The type of the temporary */
        cast_expr               init;   /* Initialization expression */
        pres_c_temporary_type   type;   /* How this temporary is used */

        /*
         * 'is_const' is TRUE (non-zero) if 'init' evaluates to some constant
         * expression.  This is used to avoid creating a temporary variable for
         * constant values.  The initialization expr is simply passed down
         * instead of a temporary variable name.  If this flag is set, you are
         * asserting that the resulting expr will NEVER be used as an lvalue.
         */
        int             is_const;

        /*
         * Handlers indicate to the back end what this temporary is used for so
         * it can resolve the mismatch.  For example, it may simply be mapped
         * to a macro that has parameters for the original variable and the
         * temporary.  We define both pre- and post-handlers, so that different
         * processing can be done both before encode and after decode.
         */
        string          prehandler<>;
        string          posthandler<>;
};
```

`PRES_C_INLINE_ILLEGAL` **Type**

This is really a slot-holder that tells the code generator to emit an unconditional error. This is useful when an inline is required to exist, but it can't (or shouldn't) happen in practice. For example, the PRES_C structure is such that a reply *must* consist of a union of the normal reply case, a system exception case, and a user exception case. In the case of Flick's decomposed stubs presentation, there is a special marshaling function to encode an exceptional reply. Since it makes no sense for one to encode an exceptional reply when there is no exception, the normal reply case is given as a `PRES_C_INLINE_ILLEGAL`. Thus, if the generated function is ever called with no exception, an error is returned.

7.2.4 Mapping Types

PRES_C mappings serve to specify exactly how a MINT type is to be translated to the corresponding CAST type. This combined with the inline information completely specifies stub generation.

While inline structures can map MINT types to more than one parameter in a function, mappings control one and only parameter. So, inline types which present more than one parameter will have more than one mapping type.

```
union pres_c_mapping_u
switch (pres_c_mapping_kind kind)
{
        case PRES_C_MAPPING_DIRECT:
                void;
        case PRES_C_MAPPING_STUB:
                pres_c_mapping_stub                   mapping_stub;
        case PRES_C_MAPPING_POINTER:
                pres_c_mapping_pointer                pointer;
        case PRES_C_MAPPING_INTERNAL_ARRAY:
                pres_c_mapping_internal_array         internal_array;
        case PRES_C_MAPPING_STRUCT:
                pres_c_mapping_struct                 struct_i;
        case PRES_C_MAPPING_FLAT_UNION:
                pres_c_mapping_flat_union             flat_union;
        case PRES_C_MAPPING_SPECIAL:
                pres_c_mapping_special                special;
        case PRES_C_MAPPING_XLATE:
                pres_c_mapping_xlate                  xlate;
        case PRES_C_MAPPING_REFERENCE:
                pres_c_mapping_reference              ref;
        case PRES_C_MAPPING_TYPE_TAG:
                void;
        case PRES_C_MAPPING_TYPED:
                void;
        case PRES_C_MAPPING_OPTIONAL_POINTER:
                pres_c_mapping_optional_pointer       optional_pointer;
        case PRES_C_MAPPING_IGNORE:
                void;
        case PRES_C_MAPPING_SYSTEM_EXCEPTION:
                void;
        case PRES_C_MAPPING_DIRECTION:
                pres_c_mapping_direction              direction;
        case PRES_C_MAPPING_SID:
                pres_c_mapping_sid                    sid;
        case PRES_C_MAPPING_ARGUMENT:
                pres_c_mapping_argument               argument;
        case PRES_C_MAPPING_MESSAGE_ATTRIBUTE:
                pres_c_mapping_message_attribute      message_attribute;
        case PRES_C_MAPPING_INITIALIZE:
                pres_c_mapping_initialize             initialize;
        case PRES_C_MAPPING_ILLEGAL:
                void;
        case PRES_C_MAPPING_VAR_REFERENCE:
                pres_c_mapping_var_reference          var_ref;
        case PRES_C_MAPPING_PARAM_ROOT:
                pres_c_mapping_param_root             param_root;
```

```
        case PRES_C_MAPPING_ELSEWHERE:
                void;
        case PRES_C_MAPPING_SELECTOR:
                pres_c_mapping_selector                 selector;
        case PRES_C_MAPPING_TEMPORARY:
                pres_c_temporary                        temp;
        case PRES_C_MAPPING_SINGLETON:
                pres_c_mapping_singleton                singleton;
};
```

Let us look at each mapping type in more detail.

`PRES_C_MAPPING_DIRECT` Type

This mapping directly maps simple types together. A MINT integer to a C integer is a direct mapping. No additional information is needed.

`PRES_C_MAPPING_STUB` Type

The C parameter must be a named type. The appropriate stub will then be called to marshal/unmarshal this parameter. No additional information is needed. Note that Flick aggressively inlines the stub's code into the generated code, so only in rare cases will a call to a marshal/unmarshal stub actually be emitted (e.g., a recursive data structure).

```
struct pres_c_mapping_stub
{
        int     mapping_stub_index;
};
```

`PRES_C_MAPPING_DIRECTION` Type

This mapping provides context to mappings that hang under this node. This mapping represents the direction of the parameter we are currently handling.

```
enum pres_c_direction
{
        PRES_C_DIRECTION_UNKNOWN = 0,
        PRES_C_DIRECTION_IN = 1,
        PRES_C_DIRECTION_OUT = 2,
        PRES_C_DIRECTION_INOUT = 3,
        PRES_C_DIRECTION_RETURN = 4
};

struct pres_c_mapping_direction
{
        pres_c_direction dir;
        pres_c_mapping   mapping;
};
```

PRES_C_MAPPING_POINTER Type

This mapping inserts a level of pointer indirection on top of the target mapping. The arglist_name identifies the allocation context associated with the pointer.

```
struct pres_c_mapping_pointer
{
        string                  arglist_name<>;
        pres_c_mapping          target;
};
```

PRES_C_MAPPING_VAR_REFERENCE Type

This mapping allows the conversion from a reference type to an actual type.

```
struct pres_c_mapping_var_reference
{
        pres_c_mapping          target;
};
```

PRES_C_MAPPING_OPTIONAL_POINTER Type

This is similar to the standard pointer mapping in that it adds a level of pointer indirection onto the target mapping. However, the value of the pointer has a semantic meaning in that it determines the presence or absence of data.

```
struct pres_c_mapping_pointer
{
        string                  arglist_name<>;
        pres_c_mapping          target;
};
```

PRES_C_MAPPING_STRUCT Type

A struct mapping maps elements of the interface type into a compatible C type. The mapping of each element is specified by the struct_i inline type parameter. Note that this returns the PRES_C walker to inline mode; we map a single entity (a structure) into multiple parts (its fields).

```
typedef pres_c_inline pres_c_mapping_struct;
```

PRES_C_MAPPING_INTERNAL_ARRAY Type

This node is associated with the data buffer of an array. When processing this mapping, other pieces of the array must also be known, such as the length, maximum length, etc. These are provided by the allocation context identified by arglist_name.

```
struct pres_c_mapping_internal_array
{
        string          arglist_name<>;
        pres_c_mapping  element_mapping;
};
```

PRES_C_MAPPING_FLAT_UNION Type

This node maps a MINT union onto a "flat" C union. The first case is an integerish discriminator, with each subsequent case a structure starting with the discriminator in the first slot.

```
struct pres_c_mapping_flat_union
{
        pres_c_mapping  discrim;
        pres_c_mapping  cases<>;
        pres_c_mapping  dfault;
};
```

PRES_C_MAPPING_SPECIAL Type

This mapping maps an arbitrary MINT type onto an arbitrary C type by specifying a marshaling function to be invoked by the stub to marshal/unmarshal this MINT type. This mechanism allows us to handle specialized presentations.

```
struct pres_c_mapping_special
{
        string marshaler_name<>;
};
```

PRES_C_MAPPING_XLATE Type

This is almost identical to its inline counterpart. It transforms the current CAST type into a known, fixed type.

```
struct pres_c_mapping_xlate
{
        cast_type               internal_ctype;
        pres_c_mapping          internal_mapping;

        string                  translator<>;
        string                  destructor<>;
};
```

PRES_C_MAPPING_REFERENCE Type

This mapping is for object references. Each back end must define this to allow the conversion between on-the-wire and in-memory representations for object references.

```
enum pres_c_reference_kind
{
        PRES_C_REFERENCE_COPY             = 0,  /* send a copy      */
        PRES_C_REFERENCE_MOVE             = 1,  /* send my instance */
        PRES_C_REFERENCE_COPY_AND_CONVERT = 2   /* send a converted copy.
                                                 * This corresponds to
                                                 * MIG's MAKE, since the OS
```

```
                                                             * converts a RECV right
                                                             * to a SEND right
                                                             */
};

struct pres_c_mapping_reference
{
        /*
         * 'kind' refers to the semantic: either copy, move, or MIG's
         * copy_and_convert (make).  'ref_count' is how many references we
         * are working with (currently it's always 1).
         */
        pres_c_reference_kind kind;
        int ref_count;
};
```

`PRES_C_MAPPING_SID` Type

This mapping identifies the current parameter as a security identifier.

```
enum pres_c_sid_kind
{
        PRES_C_SID_CLIENT       = 0,    /* effective SID of the client */
        PRES_C_SID_SERVER       = 1     /* required SID of the server  */
};

struct pres_c_mapping_sid
{
        pres_c_sid_kind kind;
};
```

`PRES_C_MAPPING_MESSAGE_ATTRIBUTE` Type

Similar to its inline counterpart, an "attribute" is embedded in the tree describing a semantic entity. The difference is that there is MINT and CAST for this mapping, allowing a run-time parameter to convey special instructions to the stub or runtime.

```
struct pres_c_mapping_message_attribute
{
        pres_c_message_attribute_kind kind;
};
```

`PRES_C_MAPPING_ARGUMENT` Type

Arguments are the method by which we can allow other branches of the PRES_C tree to see the parameter or data of the current branch. The best example is an array. The length is one piece of data on the wire, while the contents of the array elements is another. It is obvious, however, that they are related, and the array's length is necessary to process the array's data. The mechanism Flick uses for this is to save the length as an argument to some higher node (the allocation context), and when the array is processed, the length can be retrieved from the list of arguments. The CAST type and expression are saved in the argument list when this node is reached.

```
struct pres_c_mapping_argument
{
        string          arglist_name<>; /* Name of the arglist itself. */
        string          arg_name<>;     /* Name of argument in the list. */
        pres_c_mapping  map;
};
```

`PRES_C_MAPPING_INITIALIZE` Type

This node allows the current parameter or data item to be initialized to some specified CAST expression.

```
struct pres_c_mapping_initialize
{
        cast_expr       value;
};
```

`PRES_C_MAPPING_SELECTOR` Type

This is a special node for handling "presented structures", structures that exist in the presentation, but not in the MINT. It selects a member of the current CAST scope, and then maps only that entity. This is particularly useful for directly handling the CORBA environment discriminator.

```
struct pres_c_mapping_selector
{
        int             index;
        pres_c_mapping  mapping;
};
```

`PRES_C_MAPPING_PARAM_ROOT` Type

This node indicates that the current CAST expression and type describe a formal parameter to a client stub or server work function. On both the client and server, `ctype` may describe the internal (actual) type of the parameter, if different from the formal type. On the server side, the type may be further transformed, such as changing array types into pointer-to-element types (to facilitate proper allocation and ease unmarshaling).

```
struct pres_c_mapping_param_root
{
        cast_type               ctype;
        cast_init               init;
        pres_c_mapping          map;
};
```

`PRES_C_MAPPING_SINGLETON` **Type**

This node has the sole purpose of returning to inline mode, so we can treat the current entity as more than a single entity. This is primarily used for interjecting an allocation context when we only have a mapping available. The required pieces of the allocation context can be temporaries, with the pointer finally being mapped with the original CAST and MINT from this singleton node.

```
struct pres_c_mapping_singleton
{
        pres_c_inline inl;
};
```

7.2.5 Allocation Types

Allocation types specify what function provides space for a buffer to be marshaled/unmarshaled and when that function is to be invoked. The events for allocation are specified by flags. The function which does the allocation is specified by name; the function must have a certain prototype (specified by the back end).

The allocation structure is rather complex because of the many different allocation semantics that can and need to be expressed. At the highest level, there is a list of allocation cases, one for each possible parameter direction (since each direction may need to be handled differently). Each of these cases can either allow or disallow allocation for that particular direction. If allowed, the allocation semantics are expressed by a set of flags and an allocator. The allocator can be either a specific name, a special type of allocator (such as static or out-of-line), or an arbitrary allocator (essentially one that doesn't matter, allowing the code generator to choose an optimal one, such as stack).

The complete set of allocation flags and structures follow.

```
const PRES_C_ALLOC_NEVER                = 0x00000000;
const PRES_C_ALLOC_IF_NULL              = 0x00000001;
const PRES_C_ALLOC_IF_TOO_SMALL         = 0x00000002;
const PRES_C_ALLOC_IF_TOO_LARGE         = 0x00000004;
const PRES_C_ALLOC_ALWAYS               = 0x00000008;
const PRES_C_ALLOC_EVER                 = 0x0000000f;
const PRES_C_ALLOC_CLEAR                = 0x00000010;
const PRES_C_REALLOC_IF_TOO_SMALL       = 0x00000200;
const PRES_C_REALLOC_IF_TOO_LARGE       = 0x00000400;
const PRES_C_REALLOC_ALWAYS             = 0x00000800;
const PRES_C_REALLOC_EVER               = 0x00000f00;
const PRES_C_REALLOC_CLEAR              = 0x00001000;
const PRES_C_FAIL_IF_TOO_SMALL          = 0x00002000;
const PRES_C_DEALLOC_NEVER              = 0x00000000;
const PRES_C_DEALLOC_ALWAYS             = 0x00080000;
const PRES_C_DEALLOC_EVER               = 0x000f0000;
const PRES_C_DEALLOC_NULLIFY            = 0x00100000;
const PRES_C_DEALLOC_ON_FAIL            = 0x00200000;
const PRES_C_RUN_CTOR                   = 0x01000000;
const PRES_C_RUN_DTOR                   = 0x02000000;

enum pres_c_allocator_kind
```

```
{
        PRES_C_ALLOCATOR_DONTCARE,
        PRES_C_ALLOCATOR_STATIC,
        PRES_C_ALLOCATOR_OUTOFLINE,
        PRES_C_ALLOCATOR_NAME
};

union pres_c_allocator
switch (pres_c_allocator_kind kind)
{
        case PRES_C_ALLOCATOR_DONTCARE:         void;
        case PRES_C_ALLOCATOR_STATIC:           void;
        case PRES_C_ALLOCATOR_OUTOFLINE:        string ool_name<>;
        case PRES_C_ALLOCATOR_NAME:             string name<>;
};

struct pres_c_allocation_case
{
        pres_c_alloc_flags      flags;
        pres_c_allocator        allocator;
        cast_init               alloc_init;
};

enum pres_c_allocation_allow
{
        PRES_C_ALLOCATION_INVALID = 0,
        PRES_C_ALLOCATION_ALLOW = 1
};

union pres_c_allocation_u
switch (pres_c_allocation_allow allow)
{
        case PRES_C_ALLOCATION_ALLOW:   pres_c_allocation_case  val;
        case PRES_C_ALLOCATION_INVALID: void;
};

struct pres_c_allocation
{
        pres_c_allocation_u cases[PRES_C_DIRECTIONS];
};
```

7.3 The PRES_C Library

`void pres_c_1_init(pres_c_1 *pres)`

Initializes the base structure of `pres`.

`void pres_c_1_check(pres_c_1 *pres)`

Checks the validity of the structure and data in `pres`.

`void pres_c_1_readfh(pres_c_1 *dest, FILE *fh)`
Uses XDR to read the contents of the file into `dest` from the file `fh`.

`void pres_c_1_writefh(pres_c_1 *src, FILE *fh)`
Uses XDR to write the contents of `src` to the file `fh`.

`pres_c_inline pres_c_new_inline(pres_c_inline_kind kind)`
Creates a new, empty PRES_C inline structure of the given type `kind`.

There are also the following related functions to create specific PRES_C inlines, allowing the initialization of important data fields. They should all be self-explanatory, based on the descriptions given above.

`pres_c_inline pres_c_new_inline_struct(int slots)`

`pres_c_inline pres_c_new_inline_func_params_struct(int slots)`

`pres_c_inline pres_c_new_inline_struct_union(int mappings)`

`pres_c_inline pres_c_new_inline_virtual_union(int mappings)`

`pres_c_inline pres_c_new_inline_expanded_union(int mappings)`

`pres_c_inline pres_c_new_inline_void_union(int mappings)`

`pres_c_mapping pres_c_new_mapping(pres_c_mapping_kind kind)`
Creates a new, empty PRES_C mapping structure of the given type `kind`.

`pres_c_inline pres_c_set_inline_tags(pres_c_inline_kind kind, pres_c_inline_inl, int tag, ...)`
This is a tag based constructor for many of the PRES_C inline types.

`PIA_Index`
(Required tag values: `pres_c_inline_index`.) The index for a `pres_c_inline_atom`.

`PIA_Mapping`
(Required tag values: `pres_c_mapping`.) Specifies a `pres_c_mapping`.

`PIA_Slot`
(Required tag values: `int` and `pres_c_inline`.) Adds a slot to an inline struct or function parameters struct.

`PIA_Flags`
(Required tag values: `pres_c_*_flags`.) Specifies any PRES_C flags.

`PIA_Return`
(Required tag values: `int` and `pres_c_inline`.) The return inline for a `pres_c_func_params_struct`.

`PIA_Discrim`
(Required tag values: *depends on the type of union*.) The discriminator for a union. The attribute values depend on the type of union you are specifying, so just match them up with the union structure.

`PIA_Default`
(Required tag values: *depends on the type of union*.) The default case for a union. The attribute values depend on the type of union you are specifying, so just match them up with the union structure.

`PIA_Case`
(Required tag values: *depends on the type of union*.) A (non-default) case for a union. The attribute values depend on the type of union you are specifying, so just match them up with the union structure.

`PIA_DiscrimVal`
(Required tag values: `mint_const`.) The discriminator value for a discriminated union.

`PIA_SelectedCase`
(Required tag values: `pres_c_inline`.) The selected case for a collapsed discriminated union.

`PIA_Tag`
(Required tag values: `pres_c_inline`.) The tag for a `pres_c_inline_typed`.

`PIA_Inl`
(Required tag values: `pres_c_inline`.) The inline for a `pres_c_inline_typed`.

`PIA_Offset`
(Required tag values: `pres_c_inline`.) The offset inline of an allocation context describing the numerical offset of the first data item (e.g., an array that starts at 5 instead of 0). This is currently not used or fully supported.

`PIA_Len`
(Required tag values: `pres_c_inline`.) The (actual) length inline of an allocation context describing the length of the data.

`PIA_MinLen`
(Required tag values: `pres_c_inline`.) The minimum length inline of an allocation context describing the minimum allowable length for the data (e.g., a fixed array cannot be smaller than its specified length).

`PIA_MaxLen`
(Required tag values: `pres_c_inline`.) The maximum length inline of an allocation context describing the maximum allowable length for the data (e.g., a fixed or bounded array).

`PIA_AllocLen`
(Required tag values: `pres_c_inline`.) The (actual) allocated length inline of an allocation context describing the allocated buffer length in which the data is or should be placed.

`PIA_MinAllocLen`
(Required tag values: `pres_c_inline`.) The minimum allocated length inline of an allocation context describing the minimum allowable buffer length that can be allocated for the data (e.g., CORBA requires that a bounded sequence be allocated to the maximum bound, regardless of the actual amount of data the sequence holds).

`PIA_MaxAllocLen`
(Required tag values: `pres_c_inline`.) The maximum allocated length inline of an allocation context describing the maximum allowable buffer length that can be allocated for

the data. This is not currently used or fully supported. A specific problem arises when the maximum allocated length is smaller than the actual or maximum length of the data.

`PIA_Release`
: (Required tag values: `pres_c_inline`.) The release inline of an allocation context describing whether the buffer is wholly owned (not reference by anything other than the array or sequence (e.g., a CORBA sequence's release flag).

`PIA_Terminator`
: (Required tag values: `pres_c_inline`.) The terminator inline of an allocation context, representing the terminating element of a terminated array. Typically, this is only used for strings, and the terminator is thus the nil character.

`PIA_MustCopy`
: (Required tag values: `pres_c_inline`.) The inline of an allocation context indicating whether the data must be copied to keep a reference to it. Currently, this is under experimental use in the Mach3/MIG back end to help implement the "servercopy" feature of MIG.

`PIA_Alloc`
: (Required tag values: `pres_c_allocation`.) The allocation semantics of an allocation context.

`PIA_Overwrite`
: (Required tag values: `int`.) A flag in an allocation context indicating that the buffer can be overwritten rather than freshly allocated. This is not currently used or fully supported.

`PIA_Owner`
: (Required tag values: `allocation_owner`.) The owning entity of a the data or buffer of an allocation context. This is not currently used or fully supported.

`PIA_Ptr`
: (Required tag values: `pres_c_inline_atom`.) The main (pointer) inline of an allocation context.

`PIA_ArgList`
: (Required tag values: `char *`.) The name of the associated argument list.

`PIA_Name`
: (Required tag values: `char *`.) The name of a temporary variable.

`PIA_CType`
: (Required tag values: `cast_type`.) The CAST type of a temporary variable.

`PIA_Value`
: (Required tag values: `cast_expr`.) The initializer (or initial value) for a temporary variable. It is not necessarily that this be a constant expression.

`PIA_PreHandler`
: (Required tag values: `char *`.) The name of a run-time handler function that must be executed prior to marshaling or unmarshaling the temporary variable.

`PIA_PostHandler`
: (Required tag values: `char *`.) The name of a run-time handler function that must be executed after marshaling or unmarshaling the temporary variable.

`PIA_IsConst`
: (Required tag values: `int`.) The flag indicating whether the temporary's initialization is constant.

`PIA_TempType`
: (Required tag values: `pres_c_temporary_type`.) The type of temporary (associated with the presentation or the encoding).

`pres_c_mapping pres_c_set_mapping_tags(pres_c_mapping_kind kind, pres_c_mapping map, int tag, ...)`

Set the values of `map` according to the given list of tags and values. If `map` is null, a new `pres_c_mapping` node of kind `kind` is created and filled in according to the tags. The modified mapping is returned. The list of available tags is:

`PMA_Index`
: (Required tag values: `int`.) The index for a mapping stub.

`PMA_Mapping`
: (Required tag values: `pres_c_mapping`.) A `pres_c_mapping`.

`PMA_Dir`
: (Required tag values: `pres_c_direction`.) The parameter direction for a mapping direction.

`PMA_Target`
: (Required tag values: `pres_c_mapping`, or `pres_c_inline` for a STRUCT mapping.) The target mapping for another mapping (e.g., mapping pointer, optional pointer, ...).

`PMA_ElementMapping`
: (Required tag values: `pres_c_mapping`.) The element mapping for an internal array.

`PMA_Kind`
: (Required tag values: `pres_c_reference_kind`.) The kind for a mapping reference.

`PMA_RefCount`
: (Required tag values: `int`.) The reference count for a mapping reference.

`PMA_Value`
: (Required tag values: `cast_expr`.) The value for a mapping initialize.

`PMA_CType`
: (Required tag values: `cast_type`.) The ctype for a parameter root mapping or temporary variable.

`PMA_Name`
: (Required tag values: `char *`.) The name of a temporary variable.

`PMA_PreHandler`
: (Required tag values: `char *`.) The name of a run-time handler function that must be executed prior to marshaling or unmarshaling the temporary variable.

`PMA_PostHandler`
: (Required tag values: `char *`.) The name of a run-time handler function that must be executed after marshaling or unmarshaling the temporary variable.

`PMA_IsConst`
: (Required tag values: `int`.) The flag indicating whether the temporary's initialization is constant.

`PMA_TempType`
: (Required tag values: `pres_c_temporary_type`.) The type of temporary (associated with the presentation or the encoding).

`PMA_ArgList`
: (Required tag values: `char *`.) The name of the associated argument list.

`PMA_InternalCType`
: (Required tag values: `cast_type`.) The internal type (as viewed by the generated stubs) specified by a `PRES_C_MAPPING_XLATE` node.

`PMA_InternalMapping`
: (Required tag values: `pres_c_mapping`.) The mapping of the internal type for a `PRES_C_MAPPING_XLATE` node.

`PMA_Translator`
: (Required tag values: `char *`.) The translator used to convert between the presented and internal types in a `PRES_C_MAPPING_XLATE` node.

`PMA_Destructor`
: (Required tag values: `char *`.) The destructor of the internal type, if any, of a `PRES_C_MAPPING_XLATE` node.

`int pres_c_add_inline_struct_slot(pres_c_inline inl)`
: Add a new structure slot to the given `inl` (must be an inline struct of some kind).

`int pres_c_add_inline_union_case(pres_c_inline inl)`
: Add a new union case to the given `inl` (must be an inline union of some kind).

`char *mint_to_ctype_name(mint_1 *mint, mint_ref itype)`
: Given a simple mint type this will return a name corresponding to the C type.

`cast_type mint_to_ctype(mint_1 *mint, mint_ref itype)`
: This will create a `cast_type` capable of holding a value of the simple mint type.

`void descend_collapsed_union(pres_c_1 *pres, mint_ref *itypep, pres_c_inline *inlp)`
: Given an initial `itype` which must be a `MINT_UNION` and a pres_c_inline which must be a `PRES_C_INLINE_COLLAPSED_UNION`, find the appropriate case in the union selected by the presentation and return the new `itype` and pres_c_inline corresponding to it. Note that 'itypep' and 'inlp' are in-out parameters.

`int pres_c_find_mu_stub(pres_c_1 *pres, mint_ref itype, cast_type ctype, pres_c_mapping map, pres_c_stub_kind stub_kind)`
: Upon encountering a `PRES_C_MAPPING_STUB` node while descending a mapping tree, this routine finds a stub that can marshal/unmarshal between the `itype` and the `ctype`. Returns -1 if none found.

`void pres_descend_mapping_stub(pres_c_1 *pres, mint_ref itype, cast_type *ctype, pres_c_mapping *map)`

Given an `itype`, `ctype`, and a `PRES_C_MAPPING_STUB`, this routine finds a stub that can marshal/unmarshal between the `itype` and `ctype`, and bumps the `map` and `ctype` pointers to point into the stub's mapping and type, essentially dereferencing through the level of indirection. Note that this shouldn't be done blindly and unconditionally, because there can be recursion through stub mappings.

`void pres_c_interpose_pointer(cast_type *inout_ctype, pres_c_mapping *inout_mapping, const char *arglist_name)`

Given a `ctype` and `map`, return a new type and mapping which is identical except that it imposes one additional level of pointer indirection.

`void pres_c_interpose_indirection_pointer(cast_type *inout_ctype, pres_c_mapping *inout_mapping, pres_c_allocation ptr_alloc)`

This is like 'pres_c_interpose_pointer', except that the mapping for the imposed pointer is an entire allocation context (through a singleton mapping), instead of just a pointer mapping. This is most useful for constructing simple pointer indirection without going through the mundane process of creating a specialized allocation context for it.

`void pres_c_interpose_internal_array(cast_type *inout_ctype, pres_c_mapping *inout_mapping, const char *arglist_name)`

This is like 'pres_c_interpose_pointer', except that the mapping for the imposed pointer is `PRES_C_MAPPING_INTERNAL_ARRAY`, not `PRES_C_MAPPING_POINTER`. The special `INTERNAL_ARRAY` mapping is used to identify it as an array instead of a simple pointer.

`void pres_c_interpose_var_reference(cast_type *inout_ctype, pres_c_mapping *inout_mapping)`

This is like `pres_c_interpose_pointer` except that it interposes a C++ style reference.

`void pres_c_interpose_direction(pres_c_mapping *inout_maping, aoi_direction aoi_dir)`

Given a mapping, return a new mapping that encapsulates the original mapping but which tells the back end what kind of parameter the mapping corresponds to: 'in', 'out', etc.

`void pres_c_interpose_param_root(pres_c_mapping *inout_mapping, cast_type ctype, cast_init init)`

Given a mapping, return a new mapping that encapsulates the original mapping but which tells the back end that this PRES_C tree point corresponds to the "root" of a client stub or server work function formal parameter. This lets the back end take special processing steps, e.g., allocating a local variable to serve as the actual parameter in a call to a server work function.

`void pres_c_interpose_argument(pres_c_mapping *inout_mapping, const char *arglist, const char *name)`

Given a mapping, return a new mapping that encapsulates the original mapping but which tells the back end to save information about the CAST expression and type being processed. This information is then used by some parent of the PRES_C mapping node.

`void pres_c_interpose_temporary(cast_type type, const char *name, cast_expr init, const char *prehandler, const char *posthandler, int is_const, pres_c_tempoarary_type temp_type, pres_c_mapping *inout_mapping)`

Given a mapping, return a new mapping that encapsulates the original so that all operations are done on a temporary variable with the given hints. This is useful for C++ or any other situations where the data being marshaled/unmarshaled doesn't exactly match the structures/control flow of the mapping.

`void pres_c_add_unpresented_channel(pres_c_1 *pres, data_channel_mask dcm)`
: Adds a new, unpresented channel.

`char *pres_c_make_arglist_name(const char *hint)`
: Creates a unique new string (based on the given string 'hint') to be used for an argument list name.

`const char *pres_c_dir_name(pres_c_direction dir)`
: Returns the string name of the given PRES_C direction.

`tag_list *create_tag_list(int list_size)`
: This will allocate a `tag_list` of the given size.

`void delete_tag_list(tag_list *tl)`
: This will free the `tag_list` and all of its tags.

`tag_list *copy_tag_list(tag_list *tl)`
: This will do a deep copy of the given `tag_list`.

`void concat_tag_list(tag_list *tl1, tag_list *tl2)`
: This will move any tags from `tl2` to `tl1` and then delete `tl2`.

`void relink_tag_list(tag_list *tl_root)`
: This function will set all of the `parent` pointers in `tag_lists` to their proper parents. This is usually run after a PRES_C file has been read in because the `parents` are always null because the cycles have to be broken before writing.

`tag_item *create_tag_item(const char *tag, tag_data *data)`
: This will allocate a `tag_item` and initialize it with the arguments.

`void delete_tag_item(tag_item *ti)`
: This will free a `tag_item`.

`tag_item *add_tag(tag_list *tl, const char *tag, tag_data_kind kind, ...)`
: This will add a tag to a `tag_list` with the given arguments. The last argument is the data to be put into the tag, its C type is determined by the `tag_data_kind`.

`void rem_tag(tag_list *tl, const char *tag)`
: This will remove the tag from the `tag_list`.

`tag_item *find_tag(tag_list *tl, const char *tag)`
: Finds a tag matching the given name in the `tag_list`.

`tag_item copy_tag_item(tag_item *ti)`
: Deep copies a `tag_item`.

`tag_data create_tag_data(tag_data_kind kind, int length)`
: Initializes and allocates any memory for the `tag_data`.

`void delete_tag_data(tag_data *td)`
: Frees any memory allocated for the `tag_data`.

`tag_data copy_tag_data(tag_data *td)`
: Deep copies the given `tag_data`.

`void set_tag_data(tag_data *td, int index, union tag_data_u data)`
: Sets a `tag_data` to the given value. If the `tag_data` is an array, then `index` specifies which slot to set.

`union tag_data_u get_tag_data(tag_data *td, unsigned int index)`
: Returns the value of the `tag_data`, if its an array then `index` specifies the slot in the array.

`unsigned int tag_data_length(tag_data *td)`
: Returns one if the `tag_data` isn't an array, otherwise, it returns the length of the array.

`int append_tag_data(tag_data *td, union tag_data_u data)`
: Appends a value to a `tag_data` array.

`tag_data_kind get_base_tag_data_kind(tag_data_kind kind)`
: Returns the type, or element type, of a `tag_data`.

`int tag_data_kind_size(tag_data_kind kind)`
: Returns the size of the `tag_data` type, or element type.

`char *ptr_to_tag_ref(const char *type, void *ptr)`
: Converts a pointer and type string into a string usable by a `TAG_REF`.

`void *tag_ref_to_ptr(const char *type, const char *ref)`
: Converts a `TAG_REF` string into a pointer, if the `ref` string doesn't match the given type, it will return null.

`int cmp_tag_ref_class(char *ref_class, char *ref)`
: Compare the class of a `TAG_REF` string against the given one.

`void print_tag_data(int indent, tag_data_kind kind, union_tag_data_u data)`
: Prints tag values.

`void print_tag(int indent, tag_item *ti)`
: Prints a `tag_item` at a certain indentation. The format is '<tag_name>:<tag_type>=<tag_value>'.

`int pres_function(pres_c_1 *out_pres, tag_list *parent_tl, cast_scoped_name current_scope_name, cast_scoped_name fname, int tag, ...), int vpres_function(pres_c_1 *out_pres, tag_list *parent_tl, cast_scoped_name current_scope_name, cast_scoped_name fname, int tag, va_list arg_addr)`
: This function is used to create presentation only functions. These 'presentation only' functions are simple functions that exist solely for the sake of the presentation and don't need to be optimized, processed or whatever. So, to make it easier to produce these functions, this tag based function was created that will take care of most of the grunt work. `pres_function` basically takes a description of the presentation for the function through the tags and then creates the appropriate CAST, and adds a number of tags to the `parent_tl` `tag_list` describing the function. These tags are then used by the back end to actually produce the function body.

`PFA_TAG_DONE`
: (No required tag values.) The terminator.

`PFA_Scope`
: (Required tag values: `cast_scope`.) Specifies the scope the function should be declared in.

`PFA_ReturnType`
: (Required tag values: `cast_type`.) The return type of the function.

`PFA_Spec`
: (Required tag values: `int`.) The spec for the parameter.

`PFA_Parameter`
: (Required tag values: `cast_type`, `char *`, and `cast_init`.) Adds a parameter to the function with the given attributes.

`PFA_StorageClass`
: (Required tag values: `cast_storage_class`.) The storage class for the declaration.

`PFA_Template`
: (No required tag values.) Specifies that this function should be in a template.

`PFA_TemplateParameter`
: (Required tag values: `cast_template_arg_kind`, `char *`, and `cast_template_arg`.) Adds a parameter to the template.

`PFA_FunctionKind`
: (Required tag values: `char *`.) The description of what this function is supposed to do. Its just a semantically loaded name that will get matched up to something in the back end.

`PFA_Constructor`
: (No required tag values.) Specifies that this function is some form of a C++ constructor.

`PFA_Included`
: (Required tag values: `inclusion`.) The inclusion state for the function declaration.

`PFA_Tag`
: (Required tag values: `char *`, `tag_data_kind`, and `union tag_data_u`.) Adds an arbitrary tag to the function's tag list.

`PFA_Model`
: (No required tag values.) Indicates that the function is not explicitly defined, but, rather follows some model definition. For example, a getter or setter in a union is a model, while a constructor isn't.

`PFA_Protection`
: (Required tag values: `cast_def_protection`.) The protection for the function declaration.

`PFA_GlobalFunction`
: (No required tag values.) Indicates that the function is not a method in a C++ class. This causes any references to this function to be fully scoped rather than implicitly scoped due it being in a class.

7.4 presd, the PRES_C Printing Utility

The presd program is a simple pretty-printer that takes PRES_C input from a file or standard input and outputs a human-readable, ASCII translation. If the PRES_C input comes from a file, the output is a corresponding '.prd' (PRES_C dump) file. Otherwise, if the PRES_C input comes from stdin, the human-readable output is sent to stdout. Note that as the PRES_C format changes, this program must also be changed to support any new constructs, or modifications to existing ones.

Chapter 8

Source Code Markup Language (SCML)

The Source Code Markup Language (SCML) is a formatting language that can be used to create templates for source code that needs to be output by Flick. For example, the CORBA C++ mapping requires a number of C++ classes and functions to be generated alongside the actual defined types. These extra classes and functions are only there for the sake of presentation and do not need any special analysis or optimization so generating them is usually just a matter of plugging the right strings into a template. This phase of code generation can be done with a series of `printf`s, however, a more data oriented and externalized approach is more flexible and possibly easier to maintain. Therefore, SCML was created so that the user could write the implementations of these functions as a template and data created by Flick would fill in the holes.

The use of SCML code in Flick is currently limited to the functions and methods for the CORBA C++ classes created for the TAO implementation, although it would be nice to begin using it for other code which is currently done with `printf`s. This code is read in and executed at start up so that it can define a number of new commands that are used to generate the appropriate output. As a Flick back end then processes the PRES_C and defers to the SCML code whenever it encounters a presentation function declaration.

8.1 The Language

8.1.1 Introduction

The syntax of the language was based on SGML in an attempt to make it more accessible to people in general since many are already familiar with its most popular descendent, HTML. An example code section might be the best way to start.

The following SCML code is taken from `runtime/headers/flick/pres/std_defines.scml`:

```
<ifndef name="STD_DEFINES_SCML">
<define name="STD_DEFINES_SCML" kind="static-variable" value=true>

<!-- These are the standard set of defines used in SCML -->

<define name="scope" kind="bracket-tag" value="c-scope-handler"
        rparams={name:string}>
<define name="macro" kind="bracket-tag" value="c-macro-handler"
        rparams={name:string} oparams={close:bool=false
                                       rparams:tag_list={}
                                       oparams:tag_list={}}>
```

```
...
</ifndef>
```

Anything within angle brackets is a command name followed by the arguments for the command. So the first command is `ifndef`, which is basically the same thing as a C preprocessor's '#ifndef', it checks to see if name has been declared or not and executes the block if it is not defined. The second command, `define`, creates a static variable in the current scope with the given name and value. From there we have a comment and then create some more commands for the system using define and then we terminate the `ifndef` block with `</ifndef>`. So this basically corresponds to the organization of a header file in C.

8.1.2 Types

Now that we know basically what a command looks like lets look a little deeper at the rest of the command. When executing a command there are a series of assignments after it that set the values of some variables. These assignments are setting the values of the parameters for the command. Each parameter has a name, which is the name used in the assignment, a type, which is specified in the definition, and finally a default value. A name can be any string, but only C style identifiers can be used regularly, otherwise one must surround the name with single quotes ('). The type of a parameter is currently limited to:

- **any** any supported type
- **arrays** simple array types
- **bool** a boolean value (e.g., "true" or "false")
- **int** a signed integer
- **float** a single precision floating point number
- **string** a string
- **tag_list** a tag_list
- **cmd** a scoped reference to a command name
- **scml** scml code
- **stream** An I/O stream, this is a bit of a hack at the moment, try and avoid using it

8.1.3 Values

The format for declaring the tag that represents this parameter is the tag name, a colon, and an equal sign followed by a literal value to set the default. For example, `num:int=5` makes an integer tag named 'num' and sets its value to 5, of course if this tag is not added to a list, it cannot be accessed and is lost, which is why all parameter declarations are tag_lists.

The set of literal values for these types are:

- **bool** true, false
- **int** regular and hex, uh, do not try to negate anything...
- **float** as one would expect, again do not negate

- **strings** "put some text here", it also supports C style escapes, except for octal values. (Note: one can also build strings with CAST stuff from Flick-generated tags, but do not try to use it for tag names or anything other than printing. It will barf.)
- **tag_list** `{ fname:string="John" lname:string="Doe" }`
- **cmd** struct::var::'T operator=(T &)'
- **scml** Nothing other than the builtin contents tag which is created for block commands.

SCML also supports simple expressions for most of its types:

- **arrays** [] (indexing, as in a[5])
- **bool** &&, ||, ==, !=, !
- **int/float** +, -, /, *, %, (), !, ==, !=, <, <=, >, >=, <<, >>
- **strings** + (concat)
- **tag_list** . (dereference), | (append)
- **cmd** none
- **scml** none

8.1.4 Commands

In order to allow this language to be flexible enough to handle any type of presentation it has only a minor set of builtin commands from which all others are created. They are `define`, `undef`, `rename`, `ifdef`, `ifndef`, and `include`, although define is really the only one that is needed, the others are just there to be nice. Define is basically the hook that is used to add other objects into the system. It lets one create local variables, static variables, commands, and escape sequence for regular text. The parameters of the `define` command are:

- *`name:string`* the name of the object
- *`kind:string`* the type of the object, needs to be one of:
  - **"variable"** a local variable
  - **"static-variable"** a static variable
  - **"tag"** a command (e.g., `<define name....`)
  - **"bracket-tag"** a block command (e.g., `<ifndef name="..."></ifndef>`)
  - **"escape"** an escape sequence
- *`value:any`* The value for this object. If it is a variable then it is the type and value of the variable; if it is a command than it is a name corresponding to the internal C handler code; and if it is an escape it is the string that the escape resolves too.
- *`rparams:tag_list`* The list of required parameters for this command
- *`oparams:tag_list`* The list of optional parameters for this command

The name, kind, and value parameters are required in order for define to work properly. From define there comes a standard set of commands that are useful, and cannot really be used to implement each other since they modify some SCML execution state, they are:

- `scope` *name:string* Creates a new lexical scope with given the given name
- `macro` *name:string close:bool=false rparams:tag_list={} oparams:tag_list={}* macro is able to create commands from SCML code to allow for code reuse, recursion, and other nice things. Close specifies whether or not this is a block command and the rparams and oparams parameters are similar to define's.
- `if` *'test expression'* This is a block command that will partition its contents around an `else` command if there is one, and then execute the one block or the other based on the expr. (Note: this is sort of a special case. One does not specify a parameter name because it is silly and annoying, so the first thing after the `if` needs to be some valid boolean/integer expression.)
- `for` *iter:string each:string length:int*

  For will iterate over an array, specified with the each parameter as the tag name, or will loop 'length' times. The 'iter' parameter is the name of the tag you want created to hold the iterator number.
- `ignore` *'expressions'* This will just evaluate the expressions inside the command and not print anything out.
- `pre` *indented:bool=true* This tells the execution environment to go into preformatting mode. The placement of this tag is important, if it is indented than any of its contents will be shifted to the left by that much. This is to allow users to give structure to their SCML code without affecting the output. The *indented* parameter is used to indicate whether any macro calls or SCML code should be indented relative to the indentation of where they are executed. For example, if indented is set to true than any output from a macro call will be shifted to the right so that all lines will line up with the beginning of the macro call. If it was set to false then any lines would be indented based only on the indentation inside of the macro.
- `defvar` *'tag defs'* This will go through the list of tags following the defvar id and declare those tags in the local scope
- `aliascmd` *name:string handler:cmd* This is used to make a command in a scope without having to actually define it, rather the command is simply a reference to the one specified by handler.
- `create_stream` *name:string path:string* This is used to create an output stream so that one can send SCML output to a different file. The 'name' argument is used as the name of the static variable to be created. The 'path' argument should be a valid path for the file or empty for standard out.
- `retarget` *output:stream* This is used to redirect any SCML output to the stream referred to by 'output', which was created by `create_stream`.

In addition to regular style commands with names, there are a number of "special" commands which are denoted by some symbol. They are:

- `<!-- -->` This is a comment

- `<|Some text|>` This is a verbatim, it prints whatever is between the vertical bars without doing any processing
- `<(some_expr)>` This will print out the value of the expression inside the parentheses
- `&escape_name;` This is an escape sequence, it can be used in regular text to print some special character.

There is also a special tag which gets added to every block command. It is called ‘contents’ and contains the scml code that is between its start and terminating commands.

8.1.5 Larger Example

Now lets look at some useful code:

```
<macro name="idl" close=true rparams={type:string}>

  <scope name=type>

  <macro name="pres" close=true rparams={type:string}>

    <scope name=type>

    <macro name="func" close=true rparams={kind:string}>

      <macro name=kind oparams={sub_contents:scml=contents}>
        <pre><(sub_contents)></pre>
      </macro>

    </macro>

    <(contents)> <!-- put the contents of idl here so it can see 'func'-->

    </scope>

  </macro>

</macro>
```

This code is used to setup some infrastructure that is used to actually define function bodies. The ‘idl’ macro specifies an IDL type (e.g., struct, union, etc.) and adds a new scope with that as the name. Inside of this scope it creates the ‘pres’ macro which corresponds to a presentation type (e.g., var, out, etc.). Again a new scope is created with this name and we make the ‘func’ macro which will be used to define a function body. Since just putting the contents of ‘func’ right inside of the macro would print them out when the file is first being evaluated we make a new macro and pass our contents to it. This way when the Flick back end needs to output a function body it just looks up the right scope, grabs the macro and then executes it, thus printing out the function body. Now let’s see this being used:

```
<idl type="struct">

  <pres type="out">

    <func kind="T_out(T_ptr &)">
      : ptr_(<(parameter[0])>)
    {
      this->ptr_ = 0;
    }
    </func>

</idl>
```

This code makes a macro named ‘T_out(T_ptr &)’ inside of the struct::out scope. The presentation_impl inside of the back end now writes out the function type which it found in the cast and then looks up and executes this macro, resulting in the following output.

```
blah_out::blah_out(blah_ptr &p)
  : ptr_(p)
{
  this->ptr_ = 0;
}
```

Notice that everything is spaced just fine. That is because `func` created the macro with ‘sub_contents’ surrounded by the preformatting tag. Since preformatting causes indentation artifacts to be ignored they were ignored here, the important thing to note is that the indentation is based on the indentation of the block command that created the contants, not the one that used it.

8.2 The SCML interpreter

8.2.1 Introduction

The SCML interpreter is contained in the C presentation back end library so all of the relevant sources are in `c/pbe/lib/scml_*.cc` and the header file is `mom/c/scml.hh`. The interpreter is primarily written in C++ with classes for the lexxer, parser, execution environment, and support for these. Using the interpreter is simply a matter of constructing lexxer, parser, and EE objects with an input file or string and then setting them in motion.

8.2.2 scml_stream

This class is used to hold any information about IO streams used in the code. The primary use of the class is for holding the input SCML code in a string which is then used by the lexxer. However, these stream objects can also be used in SCML code for setting the output stream with the `create_stream` and `retarget` commands.

`char *tag_ref()`
: Make a TAG_REF string that refers to this object

`static struct scml_stream *ptr(char *ref)`
: Convert a TAG_REF string into a `scml_stream` pointer

`void set_flags(unsigned int the_flags), unsigned int get_flags()`
: Set/get the flags for the stream. Currently, the only flags are SSF_INPUT and SSF_OUTPUT, which correspond to whether the stream will be used for input or output.

`void set_data(char *str), char *get_data()`
: Set/get a C string as the stream, this does not work for an output file.

`void set_desc(char *desc), char *get_desc()`
: Set/get the description of the stream (e.g., file name)

`void set_include_directory_list(char **dir_list), char **get_include_directory_list()`
: Set/get the list of directories in which to find any include files

`int get_length()`
: Get the length of the stream

`void set_file(FILE *file), FILE *get_file()`
: Set/get a file as the stream, if this is an input stream then it will read the whole file into memory for processing, otherwise it is just stored for later reference.

8.2.3 scml_string

The scml_string class is a bit of a hack to get around problems with using the PRES_C tags as the representation of variables. The problem has to do with the tags being able to hold CAST and SCML not wanting to deal with anything language specific: it is only interested in strings, numbers, and the like. So the solution is to treat CAST as "computed strings" and just place all strings inside of scml_strings so that we can still process regular strings and CAST without too many problems.

`char *tag_ref(), static struct scml_string *ptr(char *ref)`
: Encode/decode this scml_string object as a TAG_REF string.

`static char *tag_string(tag_item *ti)`
: Attempts to construct a C string from a tag, this is done by getting a string directly from a TAG_STRING or doing a make_chars on an encoded scml_string object.

`tag_data *add_component()`
: Adds another component to this strings. It is just like concatenating a string except it needs to be placed inside of a tag_data

`void concat(struct scml_string *ss)`
: Concatenate an entire scml_string onto another one

`int cmp(struct scml_string *ss)`
: Just like `strcmp` except it compares scml_strings

`char *make_chars()`
: Tries to make a C string from this scml_string, it will fail if any of the components are "computed strings" (e.g., CAST structures)

`void print()`
Print the string using w_printf

8.2.4 scml_token

The scml_token class is the primitive used to represent SCML code and is used by almost every class that is a part of the interpreter. A token can be one of many kinds which may or may not have a value associated with it. Any values are contained within the `value` field of scml_token. The possible values it may hold are:

`b`
A boolean

`i`
An integer

`f`
A float

`id`
An identifier

`str`
An scml_string

`text`
Regular text

`escape`
The name of an escape sequence

`ti`
A tag_item

`tl`
A tag_list

`children`
The list of children for a non-terminal token

`void strip()`
If the token contains a tag this will try to make the token the same value as the tag. It can only fail if the tag does not have a corresponding SCML type. Any other type of demotions, like float to int, need to be done explicitly by the user since information can be lost

`void promote(int required_kind)`
This will promote a literal value to a more complex SCML type. For example, it can promote an integer to a float to make it easier for later stages to process.

`int is_expr()`
This will return true if the token represents some expression. For example, if it were an addition, subtraction, or just a plain value.

`int is_value()`
: This will return true if the token contains a literal value

`int is_operator()`
: This will return true if the token represents a mathematical operator

`int is_container()`
: This will return true if the token represents some container, like a tag list, or an expression in parentheses.

`int get_prec()`
: If the token represents an operator this will return its precedence

`char *get_identifier()`
: If the token represents an identifier this will return a C string version. The possible type of identifiers include regular variable name, escape sequence names, and scoped names.

`void print()`
: Used for debugging, just prints out the token

Each token also has a `kind` member which indicates what the token represents:

`SCML_NONE`
: A null token, often used to indicate the end of a list of tokens

`SCML_IGNORE`
: A null token, perhaps it once contained something meaningful, but should be ignored now. Or it is used when a token must be returned, but nothing meaningful was done.

`SCML_ERROR`
: An error token, processing has resulted in an error

`SCML_DONE`
: Indicates the end of the stream

`SCML_COL_POS`
: The column position in the stream is in `i` so that it can be tracked and used in error reporting.

`SCML_ROW_POS`
: The row position in the stream is in `i` so that it can be tracked and used in error reporting.

`SCML_TERM_BOOL`
: A literal boolean value in `b`

`SCML_TERM_INT`
: A literal integer value in `i`

`SCML_TERM_FLOAT`
: A literal float value in `f`

`SCML_TERM_STRING`
: A literal string value in `str`

`SCML_TERM_TAG`
: A tag value in `ti`

`SCML_TERM_TAG_LIST`
: A tag list in `tl`

`SCML_TERM_TEXT`
: Any text from SCML code that is not an escape sequence or a command is held in `text`

`SCML_TERM_ESCAPE`
: The name of an escape sequence is in `escape`

`SCML_TERM_ID`
: An identifier string is in `id`

`SCML_TERM_VERBATIM`
: Any text within a verbatim command is in `text`

`SCML_TERM_LPAREN`
: A left parenthesis

`SCML_TERM_RPAREN`
: A right parenthesis

`SCML_TERM_LBRACE`
: A left bracket ‘[’

`SCML_TERM_RBRACE`
: A right bracket ‘]’

`SCML_TERM_LCURLY`
: A left curly brace

`SCML_TERM_RCURLY`
: A right curly brace

`SCML_TERM_SLASH`
: A forward slash, ‘/’. This token is only created for the forward slash immediately after an ‘<’, otherwise an SCML_NT_DIV is created.

`SCML_TERM_LT`
: A single less than sign, ‘<’. This token is used to signify the opening of a command, it is not the same as a less than inside an expression in the command.

`SCML_TERM_GT`
: A single greater than sign, ‘>’. This token is used to signify the closing of a command, it is not the same as a greater than inside an expression in the command.

`SCML_NT_COMMAND`
: A command with `children` pointing to the contents of the command. The token is a call to a tag if the first token is a `SCML_NT_NAME` followed by an `SCML_NONE` terminated array of expressions corresponding to the arguments. If the first token is an `SCML_TERM_SLASH` followed by an `SCML_`

`NT_NAME` then the command is a terminator for the tag with that name. Otherwise the token can be just be an expression that will be printed later.

`SCML_NT_SET`
A converted `SCML_NT_ASSIGN`. It is a binary expression that indicates that the value of `children[1]` should be set in `children[0]`

`SCML_NT_ASSIGN`
An equal sign, '='. This token will be converted to an `SCML_NT_SET` for no real reason; it is legacy and the set should just be killed in favor of this token.

`SCML_NT_PLUS`
A plus sign, '+'. It is a binary expression so `children[0]` is the left side and `children[1]` is the right.

`SCML_NT_MINUS`
A minus sign, '-'. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_DIV`
A division sign, '/'. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_MOD`
A modulus sign, '%'. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_MULT`
A multiplication sign, '*'. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_COLON`
A colon, ':'. This token will immediately be converted into an `SCML_NT_TAG` when parsed.

`SCML_NT_EQUAL`
A C style equal sign, '=='. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_NOT_EQUAL`
A C style not equal sign, '!='. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_LT`
A double less than sign, '<<', used in expressions since a single one is already used to demarcate a command.

`SCML_NT_GT`
A double greater than sign, '>>', used in expressions since a single one is already used to demarcate a command.

`SCML_NT_LE`
A less than or equal sign, '<='. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_GE`
A greater than or equal sign, '>='. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_LAND`
A C style logical and sign, '&&'. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_OR`
: An or sign, ‘|’, used for doing union operations on tag lists.

`SCML_NT_LOR`
: A C style logical or sign, ‘-’. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_NOT`
: A exclamation mark, ‘!’, that signifies a logical not on `children[0]`

`SCML_NT_DOT`
: A dot, ‘.’, used for referencing tags within a tag list.

`SCML_NT_SCOPE_RES`
: A C style scope resolution operator, ‘::’. Also a binary expression so it is structured like `SCML_NT_PLUS`

`SCML_NT_NAME`
: A null token that indicates the token in `children[0]` can be used as a name.

`SCML_NT_EXPR`
: A null token that indicates the token in `children[0]` is some kind of an expression

`SCML_NT_TAG`
: A tag declaration converted from an `SCML_NT_COLON`. The tag name is contained in `children[0]` and the name of the type is in `children[1]`

`SCML_NT_SEL`
: A slot selection on an array variable. This token is created from a pair of brackets with `children[0]` expecting to resolve to a tag and `children[1]` evaluating to an integer.

`SCML_NT_COMMA`
: A comma, ‘,’, used when separating the elements of a tag list.

`SCML_NT_INIT`
: An initialization contained in curly braces. This is a bit hacky since one can create an array initializer, or a tag list, depending on what kind of data is in the initialization.

`SCML_NT_COND`
: A conditional expression similar to the one in C, ‘<test expr> ? <expr> : <expr>’, except it is not implemented yet.

`SCML_NT_AND`
: An ampersand, ‘&’, used for to do intersection on a tag list, if it was implemented.

`SCML_NT_XOR`
: A carat, ‘^’, for doing exclusive ors, except it is not implemented.

8.2.5 scml_token_sequence

An `scml_token_sequence` is used to reference sections of SCML code. The class tracks the position and the origin of the code so it is possible to format the code correctly, and point out where any errors are in the source.

`char *tag_ref(), static struct scml_token_sequence *ptr(char *ref)`
: These are used to encode and decode the token sequence as a `TAG_REF` style string. This allows the sequence to be passed around and used in SCML code

`void print()`
: Used for debugging, just prints the contents of the sequence out

`void set_value(struct scml_token *st), struct scml_token *get_value()`
: Set/get the array of the tokens that represent the value of this sequence

`void set_length(int len), int get_length()`
: Set/get the length of the array of tokens in this sequence

`void set_stream_pos(struct scml_stream_pos *ssp), struct scml_stream_pos *get_stream_pos()`
: Set/get the stream position object used when parsing the tokens in the sequence. This is mainly here for error reporting when the tokens get processed

`void set_indent(int offset), int get_indent()`
: Set/get the column based indentation of the tokens. This allows users of SCML to structure their code with indents without having it affect the way it will eventually be printed, even when preformatted.

8.2.6 scml_token_stack

The `scml_token_stack` is a support class for the parser that just maintains a stack of tokens.

`void push(struct scml_token &st), void pop()`
: Push/pop a token

`struct scml_token *index(int offset)`
: Access the token at the given offset from the top of the stack

`int count()`
: Returns the number of tokens on the stack

`void print()`
: Used for debugging, just prints out the contents of the stack

8.2.7 scml_stream_pos

The `scml_stream_pos` is used to lex the SCML source and for tracking the position during execution. The class is slightly different from a regular lexxer since it needs to operate in two modes, one for the raw text, and one for the command tokens. Additionally, the class also needs to return position tokens so that it is possible to accurately track position in the source during execution. Otherwise, the operation of the class is like one would expect, it translates text into tokens for others components to process.

`void set_stream(struct scml_stream *ss), struct scml_stream *get_stream()`
: Set/get the data stream for lexxing

`void set_flags(unsigned int the_flags), unsigned int get_flags()`
: Set/get the flags

`void set_state(int state), int get_state()`
: Set/get the state

`void set_cursor(char *cursor), char *get_cursor()`
: Set/get the current position in the `scml_stream` buffer

`void set_row(int row), int get_row()`
: Set/get the current row in the stream

`void set_column(int column), int get_column()`
: Set/get the current column in the stream

`int get_last_row(), int get_last_column()`
: Get the last row/column that was lexxed in the stream. This is used in error messages to give the user a range to look for the error

`char *munge_string(char *str)`
: Convert a string with escape sequences into a standard C string.

`int get_number(struct scml_token *st)`
: Convert a number in the text stream and store its value in `st`.

`int scan(int kind, const char *str)`
: Scan forward in the stream until the condition is satisfied, the cursor will then be placed one character after the cause of the stop.

`struct scml_token get_token()`
: This is the main function for lexxing, it will walk through the stream returning tokens until it hits the end of the stream and returns `SCML_DONE`

8.2.8 scml_parser

The `scml_parser` is a simple (and not very good) parser for SCML tokens returned by an `scml_stream_pos`. It works by giving the class an `scml_stream_pos` and then calling `parse`. If the parse was successful a token sequence will be returned which can then be executed.

`void set_stream_pos(struct scml_stream_pos *ssp), struct scml_stream_pos *get_stream_pos()`
: Set the stream that the parser should get tokens from.

`struct scml_token_sequence *parse()`
: The main parsing function, it will ask the stream position object for tokens and then construct an `scml_token_sequence` from them.

`struct scml_token collapse(struct scml_token_stack *values, struct scml_token *oper)`
: An internal function used when parsing. Basically, it just looks at the operator and pulls any operands off of the stack, and then pushes the operator connected to the operands onto the values stack.

8.2.9 scml_handler

The `scml_handler` is a structure for binding a name to a C/C++ function, making it easy to bind to a function from an SCML level.

8.2.10 scml_handler_table

The `scml_handler_table` class is used for tracking and indexing all of the functions available.

`void add_handler(struct scml_handler *sh), void rem_handler(cosnt char *name)`
: Add/remove a handler in the table

`struct scml_handler *find_handler(const char *name)`
: Find a handler in the table with the given name

8.2.11 scml_cmd_definition

The `scml_cmd_definition` class is used for describing an SCML command, its name, parameters, and definition.

`char *tag_ref(), static struct scml_cmd_definition *ptr(char *ref)`
: Encode/decode the object as a TAG_REF compatible string

`void set_name(const char *name), const char *get_name()`
: Set/get the name of the command

`void set_opt_params(tag_list *tl), tag_list *get_opt_params()`
: Set/get the list of optional parameters

`void set_req_params(tag_list *tl), tag_list *get_req_params()`
: Set/get the list of required parameters

`void set_flags(int flags), int get_flags()`
: Set/get the flags for the object

`void set_handler(struct scml_handler *sh), struct scml_handler *get_handler()`
: Set/get the handler for this command.

`void set_token_sequence(struct scml_token_sequence *sts), struct scml_token_sequence *get_token_sequence()`
: Set/get the token sequence that defines this command

`int execute(struct scml_token *st, struct scml_context *sc)`
: Execute the handler with the token and context

8.2.12 scml_escape/scml_escape_table

The `scml_escape` class is used for describing an SCML escape, the name and value. The escapes are then tracked in an `scml_escape_table`.

`void add_escape(struct scml_escape *se), void rem_escape(cnst char *name)`
: Add/remove an escape definition in the table

`void find_escape(const char *name)`
Find an escape definition with the given name

8.2.13 scml_scope

The `scml_scope` class is used to represent a scope in SCML, it holds definitions, child scopes, and escapes. Static variables and command definitions are held in the values tag list, but escapes are held separately.

`struct scml_scope *get_parent()`
Return the parent scope

`void set_name(const char *name), const char *get_name()`
Set/get the name of the scope

`void set_escape_table(struct scml_escape_table *setable), struct scml_escape_table *get_escape_table()`
Set/get the escape table associated with this scope

`void add_child(struct scml_scope *ss)`
Add a child scope

`struct scml_scope *find_child(const char *name)`
Find a child scope

`void add_cmd_definition(struct scml_cmd_definition *scd), void rem_cmd_definition(struct scml_cmd_definition *scd)`
Add/remove a command definition in the scope

`struct scml_cmd_definition *find_cmd_definition(struct scml_scope **scope, const char *name)`
Return the command definition with the given name or NULL if it was not found. The scope it was actually found in is set in `scope`.

`tag_list *get_values()`
Return the `tag_list` with the list of global variables

`void print()`
Debug printf

`static struct scml_scope *make_root_scope()`
Construct an `scml_scope` that is suitable for use as the root scope. Basically, it just adds all the builtin command definitions

8.2.14 scml_context

The `scml_context` class is used to execute SCML code and actually generate the resulting output. The primary function is `format_sequence` which takes a token sequence (e.g., from an `scml_parser::parse()`) and then executes the contents, printing out text tokens and evaluating expressions. Execution of SCML commands will create a child context for which the command executes in, of course, if the command is implemented by a C function it can manipulate its parent context, allowing for great flexibility.

`void set_flags(int flags), int get_flags()`
: Set/get the flags for the context

`void set_parent(struct scml_context *parent), struct scml_context *get_parent()`
: Set/get the parent context

`void set_stream_pos(struct scml_stream_pos *ss), struct scml_stream_pos *get_stream_pos()`
: Set/get the `scml_stream_pos` object which was used in lexxing the code

`void set_cmd_def(struct scml_cmd_definition *def), struct scml_cmd_definition *get_cmd_def()`
: Set/get the command definition associated with this context

`void set_lvalues(tag_list *tl), tag_list *get_lvalues()`
: Set/get the left hand side values

`void set_rvalues(tag_list *tl), tag_list *get_rvalues()`
: Set/get the right hand side values

`void set_scope(struct scml_scope *sc), struct scml_scope *get_scope()`
: Set/get the current scope

`void set_indent_size(int size), int get_indent_size()`
: Set/get the indent size. This size corresponds to how much indentation there is before a macro call, so that any preformatted text in the macro will be aligned with the beginning of the call instead completely left justified.

`void set_offset_size(int size), int get_offset_size()`
: Set/get the offset size. This size corresponds to the difference between the start of a line and the position of a `pre` command. The size is then used to determine when to start printing leading spaces in a line, thus, it is possible to indent SCML code with having the indentation affect the output.

`int print_token(struct scml_token *st)`
: Prints out the contents of a token. The token should already have been evaluated so only tokens with literal values should be passed to the function.

`void print_indent(int size)`
: A simple function to print an indentation of the given size.

`void locate_scope(struct scml_token *st, struct scml_scope **inout_scope, const char **out_id)`

`int equalize_tokens(struct scml_token *dest, struct scml_token *src, int count)`
: Converts the `count` sized array of tokens into the simplest common type. The process is to strip all of the tokens in `src` and then try to promote them all to the same type. If a common type was found, it will be returned and dest will have the converted tokens, otherwise `SCML_ERROR` is returned.

`struct scml_token eval_token(struct scml_token *st)`
: Evaluate a token expression to produce a token that has a printable value, except for commands, which have to be handled by `exec_token`.

`void token_lvalue(struct scml_token *st, tag_item **out_ti, int *out_index)`
: Get the lvalue of a token. Since all variables are really just tags, the result is a pointer to the `tag_item` and, if it is an array, the index into the array.

`int handle_params(struct scml_token *cmd)`
: Processes the `SCML_NT_COMMAND` token to evaluate each argument and set its value in the context.

`void tag_type(struct scml_token *t_type, tag_data_kind *out_kind, int *out_size)`
: Determine the PRES_C tag type needed to hold an SCML value.

`struct scml_cmd_definition *find_cmd(struct scml_scope **cmd_scope, struct scml_token *st)`
: Find the command definition referenced by the token

`void define(struct scml_context *sc)`
: Handles the `define` command which is used to define other commands, escapes, and variables. This is executed on the parent context and a child context is passed in which contains all of the arguments. The method will then pull any arguments of interest from the `sc` context and define the object.

`void undefine(struct scml_context *sc)`
: Undefine an object that was previously defined.

`void rename(struct scml_context *sc)`
: Rename an object

`int ifdef(struct scml_context *sc, struct scml_token_sequence *sub, int defined)`
: Similar to the `define` method, this will handle the `ifdef` and `ifndef` builtin commands. If the `defined` parameter is true then it will act like `ifdef` and execute the contents if the object is defined, otherwise it acts like `ifndef` and executes when it is not defined.

`struct scml_token_sequence *get_contents(struct scml_cmd_definition *def, struct scml_token_sequence *sts, int start)`
: Return the contents of a bracketed command in a token sequence. The passed in token sequence and `start` parameter indicate where the start of the command contents is located which the function then scans through looking for the terminator. Once the terminator is located it will create a new token sequence and set it to contain the contents.

`struct scml_token_sequence *partition_sequence(struct scml_cmd_definition *def, struct scml_cmd_definition *partition, struct scml_token_sequence *sts)`
: Partitions a token sequence around a given command. The passed in sequence will be adjusted to contain the first part of the partition and a new sequence will be created with the second part. This is useful for bracketed commands that have several different sections. For example, it can be used to separate the true and false branches of an if test, or for the cases in a switch.

`int exec_token(struct scml_token_sequence *sts, int *index, struct scml_token *st)`
: Execute a token, if the token is a command it will consume the tokens of the body, if it is bracketed, and then call the handler for the command. If the token is a simple value then it will be printed.

`int format_sequence(struct scml_token_sequence *sts)`
: Executes the commands and prints out any text in a token sequence.

`int exec_cmd(const char *cmd_name, tag_list *locals, ...)`
Manually execute a command. The `locals` argument is set as the parent tag list of the context that the command is executed in so it is possible to get at the values. The varargs portion of the call is used to add values directly to the context and is a formatted set of values. The first value is a string corresponding to the name of a variable, next there is a `tag_kind` specifying the type of the variable, and finally a `tag_data_u` which contains the value of the variable.

`static void set_handler_table(struct scml_handler_table *sht), static struct scml_handler_table *get_handler_table()`
Set/get the handler table where handlers specified in `defines` will be found. This table is global to all contexts so only one table needs to be constructed.

`void print(int level)`
Print a stack trace for debugging

8.2.15 Command Handler Functions

Any builtin commands not handled by the `scml_context` are handled by C functions. These functions are defined in `c/pbe/lib/scml_context.cc` and then a number of `scml_handlers` are created for each and added to the handler table in the `scml_context`.

`int c_defvar_handler(struct scml_token *st, struct scml_context *sc)`
Handles the `defvar` command by walking the child tokens of `st` and evaluating them and adding the tags they create to the context.

`int c_ignore_handler(struct scml_token *st, struct scml_context *sc)`
Handles the `ignore` command by walking the list of child tokens and evaluating them, without doing anything else. The evaluated expressions are expected to create some kind of side effect.

`int c_scope_handler(struct scml_token *st, struct scml_context *sc)`
Handles the `scope` command by changing to, or creating the scope with the given name. The parent scope is set to the new scope and then the parent context executes contents of the `scope` on behalf of the child.

`int c_macro_handler(struct scml_token *st, struct scml_context *sc)`
Handles the `macro` command by creating a new command that will execute the contents of the `macro` when used.

`int c_macro_executer(struct scml_token *st, struct scml_context *sc)`
Handles commands that have been created using the `macro` command. It will evaluate the arguments and then get the macro body from the command definition and execute the SCML code.

`int c_if_handler(struct scml_token *st, struct scml_context *sc)`
Handles the `if` command by evaluating `children[1]` of `st` to get a value to test for being true or false. Then it will try to partition the contents of the command around the `else` command. Finally, it tests the expression and executes the true or false branch accordingly.

`int c_else_handler(struct scml_token *st, struct scml_context *sc)`
Handles the `else` command by signalling a runtime error since the command is only meant for use as a partition inside of an `if`.

`int c_for_handler(struct scml_token *st, struct scml_context *sc)`
: Handles the `for` command by creating the loop variable in the context and then executing the contents for however long has been specified.

`int c_pre_handler(struct scml_token *st, struct scml_context *sc)`
: Handles the `pre` command by setting the `SCF_PREFORMATTED` flag and the offset of size of the contents block in the parent before using the parent to execute the contents.

`int c_aliascmd_handler(struct scml_token *st, struct scml_context *sc)`
: Handles the `aliascmd` command by building a new command with the same definition as the command specified by the *`handler`* argument.

`int c_include_handler(struct scml_token *st, struct scml_context *sc)`
: Handles the `include` command by executing the file specified by the *`file`* argument. The file is searched for in the include directory list of the stream the current file was read.

`int c_retarget_handler(struct scml_token *st, struct scml_context *sc)`
: Change the output file to the one specified by the 'output' parameter, which should resolve to an `scml_stream`.

`int c_create_stream_handler(struct scml_token *st, struct scml_context *sc)`
: Create an `scml_stream` object and set its file to the one specified by the 'path' argument, or stdout if path is empty. The object will then be added as a static variable to the current scope with the name specified by the 'name' argument. This is a big hack.

8.2.16 Miscellaneous Functions

These functions are just simple helper functions to make life a little easier.

`void scml_alert(struct scml_stream_pos *ssp, int alert_flags, const char *format, ...)`
: A printf like function for reporting errors/warnings when executing SCML code. The stream position argument is optional and currently only used to print out the column and line number of code that is causing the problem. The `alert_flags` argument is a bitfield used to describe the alert. The possible flags are:

  `SAF_WARNING`
  : This is a warning

  `SAF_ERROR`
  : This is an error

  `SAF_INTERNAL`
  : The problem is internal

  `SAF_GENERAL`
  : The problem cannot be classified

  `SAF_IO`
  : The problem is with I/O

  `SAF_LEXICAL`
  : The problem is with lexxing

`SAF_PARSE`
: The problem is with parsing

`SAF_TYPE`
: The problem is with bad/inconsistent types

`SAF_RUNTIME`
: The problem is with the execution environment

`int init_scml()`

Does any initialization of SCML structures. Currently, this just initializes the handler table with all of the builtin command handlers.

`const char **scml_std_include_dirs()`

Returns an array of strings which are paths that it can find include files. The array is the same kind that `fopen_search` so it can be used directly with that functions.

`int scml_hash_name(const char *name, int table_size)`

Produces a hash number for `name` in a table the size of `table_size`

`int scml_parse_cmdline_defines(struct scml_scope *root_scope, flag_value_seq *defs)`

Evaluates the flag value sequence to create the desired SCML variables. The flags are expected to be formatted strings of the form, 'var=expr', where 'var' is the variable name, and expr is the typed value of the expr. If there is no expression then a boolean variable is defined and set to true.

`int scml_execute_str(struct scml_scope *root_scope, char *code_desc, char *code, tag_list *tl)`

Execute SCML 'code' in the 'root_scope' with 'tl' holding any implicit values. The 'code_desc' argument is used as the description for the `scml_stream` so that one can figure out what SCML code caused an error.

`struct scml_stream_pos *scml_execute_defs_file(struct scml_scope *root_scope, const char **include_dirs, const char *file_desc, const char *file_name)`

Execute a file that will only define objects and not produce any output.

`int scml_code_cast_handler(int indent, cast_handler *ch)`

A routine for executing SCML code that is encapsulated by a `cast_handler` object. The `cast_handler` is set up with this routine as the handler and four arguments for the SCML code. The arguments in order are the code description for the `scml_stream`, the SCML code itself, a TAG_REF encoded pointer to an `scml_scope` to execute in, and finally a `tag_list` of implicit arguments.

Part III

The Programs and Libraries

Chapter 9

Libcompiler

The libcompiler library contains a number of general utility functions that may be used by Flick in any compiler stage. Some of the functionality is only relevant to Flick's front ends (e.g., calls to the C preprocessor), mainly because there is no front end library in which it can be contained. It would be reasonable (and probably trivial) to combine these front end specific routines into a front end library, leaving libcompiler a pure library of general utility functions.

9.1 Errors and Warnings

Some basic functions for reportings errors and warnings:

`void panic(const char *format, ...)`
: Reports an internal error and then `exit`s.

`void warn(const char *format, ...)`
: Prints out a warning message to the user and returns.

Both `panic` and `warn` use `printf`-style arguments.

9.2 Memory and Strings

Memory allocation in Flick is just assumed to work, so the `must*` allocators either return allocated memory or `panic` if it fails. Some simple string functions are also provided:

`void *mustmalloc(long size)`

`void *mustcalloc(long size)`

`void *mustrealloc(void *orig_buf, long new_size)`
: Memory allocators that always return allocated memory. If no memory is available, it is considered a fatal error, and they internally `panic` to prevent returning a NULL pointer.

`char *muststrdup(const char *str)`
: Duplicates a string into a freshly allocated buffer, or `panic`s if it cannot allocate the necessary memory.

`char *flick_asprintf(const char *fmt, ...)`
: Like `sprintf`, however this automatically allocates a sufficient buffer for the resulting string. Like the functions above, this `panics` if allocation fails.

`int flick_strcasecmp(const char *s1, const char *s2)`
: Case insensitive string comparison; return value is similar to `strcmp`.

`ir_strlit(str)`
: Convert a string literal for use in an intermediate representation structure. This is just a `#define` that type casts the argument, helpful for removing warnings/errors about violating constness.

9.3 Files

A number of functions are provided for managing and generating files. All file output is generally done through a set of functions that refer to a globally set file. It is a little restrictive, but at least the file pointer doesn't have to be passed around everywhere...

`FILE *call_c_preprocessor(const char *filename, const char *flags)`

`FILE *call_cxx_preprocessor(const char *filename, const char *flags)`
: Calls the C or C++ preprocessor, respectively, on `filename` using the given set of preprocessor flags.

`FILE *fopen_search(const char *path, const char * const *dir_list, const char *mode, char **out_path)`
: Opens a file from a list of source directories. The `dir_list` argument is a null terminated array of strings representing paths to search for the file indicated by the `path` argument. If the file is found then it will open it and return the `FILE` object and set the `out_path` argument to the directory where the file was found.

`void w_set_fh(FILE *out_fh)`
: Sets the file handle for writing. Each of the following `w_*` commands output to the file specified by the most recent `w_set_fh`.

`void w_putc(char c)`
: Writes out the given character.

`void w_vprintf(const char *format, va_list vl)`
: `va_list` version of `w_printf`.

`void w_printf(const char *format, ...)`
: Writes out the formatted string.

`void w_indent(int indent)`
: Writes out an indentation of the given size.

`void w_i_vprintf(int indent, const char *format, va_list vl)`
: Indented form of `w_vprintf`.

`void w_i_printf(int indent, const char *format, ...)`
: Indented form of `w_printf`.

There are also several functions for manipulating file names, thus allowing for some indirection for different OS file name formats.

`char *resuffix(const char *orig, const char *newsuffix)`
: Returns a new string (filename) with the extension portion of `orig` changed to `newsuffix`.

`const char *file_part(const char *filename)`
: Returns a pointer into `filename` that refers to the file only.

`const char *dir_part(const char *filename)`
: Returns a new string that contains the directory part of `filename`.

`const char *add_file_part(const char *pathname, const char *filename)`
: Append `filename` onto `pathname` intelligently (i.e., adds the '/' if needed).

`int absolute_path(const char *filename)`
: Returns true if `filename` is an absolute path.

`int current_dir(const char *filename)`
: Returns true if `filename` refers to the current directory. (Note: This just recognizes '.' at the moment.)

`void filename_to_c_id(char *filename)`
: Convert, in place, a filename to a legal C identifier. This is useful for producing the symbol for an `#ifdef` in a header file, or for use in distinguishing a function name.

9.4 Command Line Arguments

The command line arguments for a module all follow similar forms so `libcompiler` provides a flexible infrastructure for dealing with them. It is really quite capable and able to handle flags, strings, numbers, and arrays of these types.

`flag_value`
: A union for holding the value of a flag, either a boolean, integer, or string.

`flag_value_seq`
: Holds a sequence of `flag_values` so that we can store all instances of a flag on the command line. If a flag can only occur once, or if only the last one is used, then this sequence will be of size one.

`flags_in`
: A structure that describes a single flag. An array of these is constructed and then used to figure out how to parse a command line.

`flags_out`
: Holds the results of a command line parse. The flag values are stored in an array of `flag_value_seqs` that has the same ordering as the `flags_in` array. Extra arguments are stored as an array of strings and any errors are indicated by the `error` slot.

`fe_flags`
: This holds the front end flags after they have been pulled out of the `flags_out`.

There are also a number of functions for manipulating these structures and the actual arguments.

`flags_out parse_args(int argc, char **argv, int flag_count, flags_in *flags)`
: Parse the given arguments, based on the input flags, into a flags_out and return.

`void print_args_usage(const char *progname, int flag_count, flags_in *flgs, const char *cl_extra, const char *extra)`
: Print the usage statement for a collection of flags.

`void print_args_flags(flags_out res, int flag_count, flags_in *flags)`
: Debugging function that prints out the parsed flags.

`void set_def_flags(flags_in *flgs)`
: Add the standard flags set to the passed in array.

`void std_handler(flags_out out, int flag_count, flags_in *flgs, const char *cl_info, const char *info)`
: This will handle version, usage, and errors.

`fe_flags front_end_args(int argc, char **argv, const char *info)`
: Handle the front end args here since we have no front end library.

9.5 Stacks

A simple pointer stack is provided to store pointers when dealing with nested objects and such.

```
struct ptr_stack {
        void **ptrs;
        int top;
        int size;
};
```

`struct ptr_stack *create_ptr_stack()`
: Allocates and initializes a `struct ptr_stack`.

`void delete_ptr_stack(struct ptr_stack *stack)`
: Frees a `struct ptr_stack`.

`void push_ptr(struct ptr_stack *stack, void *ptr)`
: Pushes a pointer onto the stack.

`void pop_ptr(struct ptr_stack *stack)`
: Pops the top pointer off the stack.

`void *top_ptr(struct ptr_stack *stack)`
: Returns the top pointer on the stack.

9.6 Hash Tables

A simple hash table that hashes on a string.

```
struct h_entry {
        struct h_entry *next;
        const char *name;
};

struct h_table {
        int table_size;
        struct h_entry **entries;
};
```

`struct h_table *create_hash_table(int size)`
: Allocate and return a hash table of with the given table size.

`void delete_hash_table(struct h_table *ht)`
: Deallocate a hash table.

`void add_entry(struct h_table *ht, struct h_entry *he)`
: Add an entry to the table.

`void rem_entry(struct h_table *ht, const char *name)`
: Remove an entry from the table.

`struct h_entry *find_entry(struct h_table *ht, const char *name)`
: Find and return an entry in the table, if it couldn't be found it returns `NULL`.

9.7 Linked Lists

A simple doubly linked list implementation is provided for use by other data structures. Simply place the `list_node` structure at the beginning of another structure definition and then you can add that larger structure to any `dl_list`.

```
struct list_node {
        struct list_node *succ;
        struct list_node *pred;
};

struct dl_list {
        struct list_node *head; /* points to the head node */
        struct list_node *tail; /* always == 0 */
        struct list_node *tail_pred; /* points to the tail node */
};
```

`void remove_node(struct list_node *ln)`
: Removes a node from a list.

`void new_list(struct dl_list *list)`
Initializes a list structure.

`void add_head(struct dl_list *list, struct list_node *ln)`
Add a list node to the head of a list.

`void add_tail(struct dl_list *list, struct list_node *ln)`
Add a list node to the tail of a list.

`struct list_node *rem_head(struct dl_list *list)`
Remove and return the list node at the head of the list.

`struct list_node *rem_tail(struct dl_list *list)`
Remove and return the list node at the tail of the list.

`int empty_list(struct dl_list *list)`
Returns true if the list is empty.

`void insert_node(struct list_node *pred, struct list_node *ln)`
Insert a node after `pred`.

Chapter 10

Front Ends

A Flick front end is a program that reads an IDL source file and from that produces an internal representation of the described interfaces. There is a separate front end for each IDL that Flick understands: one for the CORBA IDL, a second for the ONC RPC (a.k.a. Sun RPC) IDL, and a third for the Mach MIG IDL.

The CORBA and ONC RPC front ends translate IDL files into Flick's AOI representation, whereas the MIG front end translates its inputs into Flick's PRES_C representation. The difference is due to the nature of the three languages. The CORBA and ONC RPC IDLs are "pure" interface description languages in that they describe interfaces only in abstract terms. In other words, a CORBA or ONC RPC IDL file does not specify how its IDL constructs are to be translated into constructs of another programming language (e.g., C or C++). Because CORBA and ONC RPC interface descriptions are free from translation details, a CORBA or ONC RPC IDL file can be completely translated — without loss of information — into a representation in Flick's AOI format. In contrast, a MIG IDL specification commonly describes how the interface is to be translated into C functions and data types. For example, the MIG IDL keywords `cusertype` and `cservertype` specify how a MIG IDL type will be translated into the generated client-side and server-side code! Because MIG interface specifications are littered with these kinds of mapping-detail "impurities," MIG IDL cannot generally be translated into AOI without loss of information. Therefore, Flick's MIG front end skips over the AOI format and generates PRES_C directly from MIG IDL.

10.1 Front End Libraries

The nature of any compiler front end is that it is structured around the compiler's input language. Because input languages are often quite different from each other, and because there are many tools for creating compiler front ends (e.g., `lex` and `yacc`), Flick does not provide any kind of shared language-processing library for its front ends. (The situation is quite different for later stages of IDL compilation. As described in Chapters 11 and 12, Flick's separate "presentation generators" and back ends are quite similar to one another, and therefore, Flick can provide a great deal of library support for all implementations of those compilation steps.)

The library support for Flick's front ends is therefore limited to manipulation of common data types, command line parsing, and a few utility functions:

`libflick-aoi.a`	The AOI library provides common functions for manipulating AOI data structures, as described in Chapter 4.
`libflick-meta.a`	This library contains functions for manipulating file metadata, as described in Chapter 3.

`libflick-compiler.a` This library provides general utility functions that are used by all Flick compiler steps. Additionally, because there is no library just for front ends, this library also holds a few functions that are really only useful to front ends: e.g., functions to invoke the C and C++ preprocessors (`call_c_preprocessor` and `call_cxx_preprocessor`) and a function to handle common command line arguments (`front_end_args`). These front-end specific functions should probably be moved into a separate library in the future.

Because the MIG front end produces PRES_C, it must link with `libflick-pres_c.a`, `libflick-mint.a`, and `libflick-cast.a` in addition to the libraries listed above.

The next three sections describe each of Flick's front ends.

10.2 CORBA Front End

The CORBA front end, contained in the `fe/newcorba` directory, was written especially for Flick. It is a more or less straightforward `lex`- and `yacc`-based program that translates CORBA IDL specifications into their AOI equivalents. The front end generally parses CORBA 2.0 IDL, with the addition of the CORBA 2.1 "`long long`" data type.

(Prior to August 1996, Flick's CORBA front end was based on Sun's freely available CORBA compiler front end. The Sun-based CORBA front end was contained in the `fe/corba` directory; hence, the new front end went into `fe/newcorba`. Although the Sun-based CORBA front end is still available in the Flick CVS archives, it is not distributed as part of Flick, and it has not been maintained since 1996.)

The mapping from CORBA IDL to AOI is straightforward, since many of the notions in AOI were in fact based on their CORBA equivalents. Table 10.1 summarizes the translation of top-level CORBA IDL declarations into AOI definitions, and of specific CORBA IDL types into AOI types. IDL constant values (e.g., `7`, `TRUE`) are of course translated into equivalent AOI literal constants. Most IDL declarations translate into AOI `aoi_defs` that associate the appropriate names and definitions. IDL operations and attributes, for historical reasons, are different, and are translated into special `aoi_operation` and `aoi_attribute` structures. Note that certain IDL constructs and types, introduced after CORBA 2.0, are not yet supported. For further details about AOI, refer to Chapter 4.

A few additional comments about the CORBA front end:

Scoping. Although Flick generally accepted the CORBA 2.0 IDL, the front end implements some scoping rules from later versions of the language (CORBA 2.3). For instance, the name of an interface cannot be redefined within the immediate scope of that interface. Many other features of CORBA 2.3, such as escaped identifiers, are *not* yet supported.

`#pragma` **Directives.** Flick does not yet support the `pragma` directives defined for CORBA IDL.

`context` **Clauses.** Flick's CORBA front end does not support `context` clauses. It would be relatively simple to extend AOI to support context variables, but nobody uses them anyway.

10.3 ONC RPC (Sun) Front End

The ONC RPC front end, contained in the `fe/sun` directory, is derived from the 1988 "RPCSRC 4.0" version of Sun's `rpcgen` program. The principal modification is that original `rpcgen` back end has been removed and replaced with a custom "back end" that generates AOI. Other minor modifications have updated the 1988

CORBA IDL Declaration	AOI Definition (`aoi_def`) Type
`module`	`AOI_NAMESPACE`
`interface`	`AOI_INTERFACE` or `AOI_FWD_INTRFC`
operation	an `aoi_operation` within an `AOI_INTERFACE`
attribute	an `aoi_attribute` within an `AOI_INTERFACE`
`typedef`	*see specific types, below*
`struct`	`AOI_STRUCT`
`union`	`AOI_UNION`
`enum`	`AOI_ENUM`
`native`	*not supported*
`const`	`AOI_CONST`
`exception`	`AOI_EXCEPTION`

CORBA IDL Type	AOI Type
`void`	`AOI_VOID`
`boolean`	`AOI_INTEGER`, min 0, range 1
`char`	`AOI_CHAR`, 8 bits, no flags
`wchar`	*not supported*
`enum`	*see declarations, above*
`octet`	`AOI_INTEGER`, min 0, range $2^8 - 1$
`short`	`AOI_INTEGER`, min -2^{15}, range $2^{16} - 1$
`unsigned short`	`AOI_INTEGER`, min 0, range $2^{16} - 1$
`long`	`AOI_INTEGER`, min -2^{31}, range $2^{32} - 1$
`unsigned long`	`AOI_INTEGER`, min 0, range $2^{32} - 1$
`long long`	`AOI_SCALAR`, 64 bits, no flags
`unsigned long long`	`AOI_SCALAR`, 64 bits, unsigned
`float`	`AOI_FLOAT`, 32 bits
`double`	`AOI_FLOAT`, 64 bits
`long double`	*not supported*
`fixed`	*not supported*
fixed-length array	`AOI_ARRAY` of element type
`sequence`	`AOI_ARRAY` of element type
`string`	`AOI_ARRAY` of `AOI_CHAR`, with null-termination flag
`wstring`	*not supported*
`struct`	*see declarations, above*
`union`	*see declarations, above*
`Object`	`AOI_INDIRECT` to built-in `CORBA::Object` definition
`any`	`AOI_TYPED`
`CORBA::TypeCode`	`AOI_TYPE_TAG`
use of named type	`AOI_INDIRECT` to `aoi_def` that defines the type

Table 10.1: Mapping of CORBA IDL Declarations and Types to AOI.

code for ANSI C, fixed bugs, and so on. The ONC RPC front end's scanner and parser are hand-coded — largely unmodified from the original `rpcgen` code — and produce a linked list of `definition` structures. The Flick-specific "back end," contained in `fe/sun/xlate.c`, simply translates this list of definitions into the equivalent AOI structures.

As with CORBA IDL, the mapping from ONC RPC IDL to AOI is more or less straightforward, and is summarized in Table 10.2. Because Flick's ONC RPC front end is based on a relatively old version of `rpcgen`, it parses an old version of the IDL that predates the current language defined by RFC 1831. In practice, this means:

- Flick's ONC RPC front end allows at most one parameter to be passed to any procedure. (The newer IDL allows multiple parameters.)
- Flick does not recognize the `hyper` and `quadruple` types, which were introduced in later versions of `rpcgen`.

Flick accepts the various IDL "extensions" that are generally accepted by `rpcgen`, as outlined in Table 10.2. Flick does not, however, support '`%`' directives in ONC RPC IDL files. These directives tell `rpcgen` to pass text directly into the generated output file. Since Flick's ONC RPC front end produces AOI, there is nowhere to store literal code, and so Flick simply discards these directives.

Finally, here are a few interesting observations about the ONC RPC front end:

`program` and `version` Declarations. A `program` declaration is represented in AOI as an (empty) `AOI_INTERFACE`, not as an `AOI_NAMESPACE`. This is partly historical — the ONC RPC front end was created before namespaces were represented in AOI — and partly because there is nowhere to store an ONC RPC program identifier in an `AOI_NAMESPACE` definition. A `version` declaration is represented as an `AOI_INTERFACE` that inherits from the interface corresponding to the containing `program` declaration. The AOI representation of ONC RPC `program` and `version` declarations should be changed in the future; this would for example allow one to make a CORBA-style presentation of an ONC RPC IDL-defined interface.

`long` Versus `int`. Although `long` is not properly part of the ONC RPC IDL, `rpcgen` accepts `long` as a built-in type and maps it to the C language `long` type. Flick, however, is unable to treat `long` and `int` declarations separately. Flick's ONC RPC presentation generator (described in Section 11.3.4) must map both of these types to C `int`, because Flick's ONC RPC front end maps both IDL types to a single AOI type. Fixing this would require a change to AOI, i.e., extending `AOI_INTEGER` definitions to keep track of whether the type was explicitly long (or short, or unsigned, or...).

`const` Declarations. RFC 1832 is unclear about the types of constants; more recent ONC documentation states that "symbolic constants may be used wherever an integer constant is used." Nevertheless, `rpcgen` and Flick's ONC RPC front end both allow literal string constants to be defined. Unlike `rpcgen`, Flick disallows forward and undefined-but-unused constant declarations; the value of a constant must be determinable when the constant is declared. Thus, the following declarations will be accepted by `rpcgen` but rejected by Flick:

```
const a = b; const b = 1; /* ``Forward'' declaration of `b'.  */
const bar = foo; /* `foo' undefined, but `bar' is never used. */
```

`rpcgen` accepts these kinds of weird declarations simply because it doesn't do anything interesting with them.

ONC RPC IDL Declaration	AOI Definition (`aoi_def`) Type
`program`	`AOI_INTERFACE` (*not* `AOI_NAMESPACE`)
`version`	`AOI_INTERFACE`
procedure	an `aoi_operation` within an `AOI_INTERFACE`
`typedef`	*see specific types, below*
`struct`	`AOI_STRUCT`
`union`	`AOI_UNION`
`enum`	`AOI_ENUM`
`const`	`AOI_CONST`

ONC RPC IDL Type	AOI Type
`void`	`AOI_VOID`
`bool`	`AOI_INTEGER`, min 0, range 1
`char`[a]	`AOI_CHAR`
`unsigned char`[a], `u_char`[b]	`AOI_CHAR`, with unsigned flag
`enum`	*see declarations, above*
`int`	`AOI_INTEGER`, min -2^{31}, range $2^{32}-1$
`unsigned int`, `u_int`[b]	`AOI_INTEGER`, min 0, range $2^{32}-1$
`short`[b]	`AOI_INTEGER`, min -2^{15}, range $2^{16}-1$
`unsigned short`[b], `u_short`[b]	`AOI_INTEGER`, min 0, range $2^{16}-1$
`long`[b]	`AOI_INTEGER`, min -2^{31}, range $2^{32}-1$
`unsigned long`[b], `u_long`[b]	`AOI_INTEGER`, min 0, range $2^{32}-1$
`hyper`	*not supported*
`unsigned hyper`	*not supported*
`float`	`AOI_FLOAT`, 32 bits
`double`	`AOI_FLOAT`, 64 bits
`quadruple`	*not supported*
fixed-length array	`AOI_ARRAY` of element type
variable-length array	`AOI_ARRAY` of element type
fixed-length `opaque`	`AOI_ARRAY` of `AOI_INTEGER` (min 0, range 2^8-1)
variable-length `opaque`	`AOI_ARRAY` of `AOI_INTEGER` (min 0, range 2^8-1)
`string`	`AOI_ARRAY` of `AOI_CHAR`, with null-termination flag
pointer (a.k.a. optional)	`AOI_OPTIONAL` of member type
`struct`	*see declarations, above*
`union`	*see declarations, above*
use of named type	`AOI_INDIRECT` to `aoi_def` that defines the type

[a]This type is not part of the ONC RPC IDL defined by RFC 1831, but is recognized as a keyword by `rpcgen`.
[b]This type is not part of the ONC RPC IDL defined by RFC 1831, but is recognized as a built-in alias by `rpcgen`.

Table 10.2: Mapping of ONC RPC IDL Declarations and Types to AOI.

opaque **Array Flags.** The front end sets the AOI_ARRAY_FLAG_OPAQUE flag on opaque arrays, although this flag is obsolete in the current version of Flick.

10.4 MIG Front End

The MIG front end, contained in the fe/mig directory, is derived from the original MIG IDL compiler written by Carnegie Mellon University. Many files are copyrighted by CMU:

```
/*
 * Mach Operating System
 * Copyright (c) 1991,1990 Carnegie Mellon University
 * All Rights Reserved.
 *
 * Permission to use, copy, modify and distribute this software and its
 * documentation is hereby granted, provided that both the copyright
 * notice and this permission notice appear in all copies of the
 * software, derivative works or modified versions, and any portions
 * thereof, and that both notices appear in supporting documentation.
 *
 * CARNEGIE MELLON ALLOWS FREE USE OF THIS SOFTWARE IN ITS
 * CONDITION.  CARNEGIE MELLON DISCLAIMS ANY LIABILITY OF ANY KIND FOR
 * ANY DAMAGES WHATSOEVER RESULTING FROM THE USE OF THIS SOFTWARE.
 *
 * Carnegie Mellon requests users of this software to return to
 *
 *  Software Distribution Coordinator  or  Software.Distribution@CS.CMU.EDU
 *  School of Computer Science
 *  Carnegie Mellon University
 *  Pittsburgh PA 15213-3890
 *
 * any improvements or extensions that they make and grant Carnegie the
 * rights to redistribute these changes.
 */
```

Flick's MIG front end was created by replacing MIG's original back end with a new "back end" that produces PRES_C files instead of C language stubs. As described previously, Flick's MIG front end produces PRES_C instead of AOI because the MIG IDL allows a programmer to specify how IDL constructs should be mapped onto the C language. In essence, this means that Flick's MIG front end is also a "presentation generator" as described in Chapter 11.

Except for minor maintenance (e.g., updates for ANSI C), the basic scanning and parsing functions are essentially unchanged from the original MIG source code. The presentation generator functions of Flick's MIG front end are described in Section 11.3.6.

10.5 Summary and Comments

The differences between Flick's CORBA, ONC RPC, and MIG front ends illustrate an important strength: namely, that by providing multiple intermediate representations with differing levels of detail, Flick is able to accommodate several very different IDLs. The differences also highlight a potential improvement to Flick: in particular, the need to extend AOI to accommodate some sort of "annotation" facility that could express MIG's "impure" constructs. The AOI format is critical in order for Flick to provide a large, reusable infrastructure for presentation generation, the second stage of IDL compilation. The common presentation generator library is designed to read and process AOI descriptions. Since Flick's MIG front end does not produce AOI, it cannot directly benefit from Flick's common presentation generation library, and therefore, the MIG front end has its own, mostly separate infrastructure for producing PRES_C. This duplication of effort could be greatly reduced if it were possible to divide the current MIG front end into two programs: a front end producing some sort of annotated AOI, and a separate presentation generator (based on Flick's common library) that read and processed the annotations. An annotatable AOI format would also be useful for other purposes, such as attaching quality-of-service attributes to normally "pure" CORBA interface descriptions.

Chapter 11

Presentation Generators

Presentation generation is an intermediate stage of the Flick IDL compiler process. Presentation generator modules (typically) take as input an AOI interface representation, and output an interface representation that is specific to a programming language. The output also specifies a default presentation (see Section 11.1) for stubs and data types. This chapter explains the Flick presentation generation system in detail for the C and C++ languages.

The translation from AOI to PRES_C moves the interface representation toward stub code generation by introducing a message format (MINT) and a language-specific representation (CAST) of data types and functions in the interface. As described in Chapter 7, PRES_C structures contain MINT and CAST structures. Utilizing these MINT and CAST structures is a third part of PRES_C, an array of stub definitions.

11.1 Presentation of an Interface

The *presentation* of an interface encompasses everything the application programmer must know to program to and implement the interface. This includes the stub signatures, data type names and structure (if transparent), and specialized support functions directly related to the interface. This also includes any semantics that may be associated with passing values in the system (by reference or value), runtime routines for initialization/destruction of the data types, and when/where/how allocation and deallocation is expected to take place. The presentation generator is responsible for completely describing these requirements without dictating an implementation for them. The implementation is left to the back end which takes into account the message encoding, link-layer details, and the target runtime's API.

11.1.1 Standard RPC Presentation

The typical RPC presentation follows a general pattern, regardless of the specific presentation. This pattern is a set of automatically-generated stubs that execute communication between a client program and a server program. Each operation includes a request, or message sent by the client to the server with any data to be acted upon, and a reply, or message sent by the server back to the client with any data that resulted from the action performed. There is almost always some sort of object model, where the operations belong to a specific object (or server entity), and they are invoked as methods on an object instance (via a reference to the object instance). These object references contain the information necessary for the runtime to direct messages to the appropriate object for invocation.

In general, there are four major types of parameters that may be a part of the request or reply messages. Note that not all presentation use all of these major parameter types. The following list briefly describes each type.

- IN - Describes a parameter that is sent in the request message from the client to the server for processing.
- OUT - Describes a parameter that is sent in the reply message from the server to the client as a result of processing.
- INOUT - Describes a single parameter whose data is sent in the request to the server, and whose data is overwritten with data from the reply message from the server.
- RETURN - Describes the return value of an operation, sent in the reply message from the server. This is similar to an OUT parameter, except for the semantic distinction: it is the return value of the operation rather than a formal parameter.

In addition to the major types of parameters, there are a set of communication stubs produced for every function declared in the input IDL file. These include:

- *Client stubs* to marshal the request, send it to the target object, wait for a reply, and unmarshal into the out parameters and return value.
- *Server skeletons* to demultiplex a message on the server side, unmarshal the request, dispatch to the work function, and marshal and send the reply to the client.
- *Marshaling stubs* to marshal specific data types into the message stream. These are only generated and called from client stubs and server skeletons if the code cannot be directly inlined (e.g., a recursive data type).
- *Unmarshaling stubs* to unmarshal specific data types from a message stream. Again, these are only generated and called when their code cannot be inlined directly.

In addition to these generated functions, there are prototypes generated for the *work functions* that implement the operations defined in the IDL.

11.1.2 Decomposed Stubs Presentation

Some work has been done on specializing Flick to generate non-traditional communication stubs for a specific class of distributed applications. These stubs reflect a message-passing methodology rather than RPC or RMI, however they still have notions of requests and replies. We call them "decomposed" because we have split apart the separate main tasks of a typical RPC stub into their own stubs, namely marshaling, unmarshaling, sending, and receiving. Note that while the presentation of these stubs is significantly different, the definition of the interfaces and operations (e.g., CORBA IDL) has not been modified. We can produce either regular RPC stubs or our decomposed stubs from a single `.IDL` file.

By decomposing the separate tasks of an RPC call, we gain finer grain control over when and how the system creates and exchanges messages. For example, one could construct a single message and send it to multiple servers without paying for the marshaling cost multiple times. Further, when a message is sent, it is not necessary to idly wait for its reply (thus allowing for asynchronous communications). Messages can be forwarded (by resending them) to another server without paying the cost of decoding and reencoding the message contents. Finally, the handling of a received message (on either side) can be postponed until a later time (e.g., if other data is not yet available).

The decomposed stubs presentation generates the following types of communication stubs:

- *Message marshaling stubs* to marshal an entire message for an operation. Two are generated: one to marshal the request, and one to marshal the reply. They take the operations parameters and return an opaque marshaled message that can later be sent.
- *Message unmarshaling stubs* to unmarshal an entire message for an operation. Two are generated: one to unmarshal the request, and one to unmarshal the reply. These take a marshaled message as a parameter, and unmarshal it into the appropriate parameter set defined by the operation.
- *Send stubs* to send a message to an appropriate destination. Two are generated: one to send the request message to a server, and one to send the reply message to the client. Both take the target reference, a marshaled message to be sent, an invocation identifier, and the client reference.
- *Continuation stubs* to postpone computation for a received message. Two are generated: one to setup a request to be continued, and one to setup a reply to be continued. One can think of these as a type of send-to-self stub, except that the application has control over when the system "notices" it has been "received" (again). These stubs also allow the application to save arbitrary state with the continuation. The parameters are the same as those for a send stub, with the addition of a `void *` pointer to the state to save.
- *Marshaling stubs* and *unmarshaling stubs* are generated as in the standard presentation.

As in the default presentation, there are prototypes generated for the *work functions* that implement the operations described in the IDL. The prototypes are significantly different, however, in that they take a marshaled message rather than the unmarshaled parameters. This allows the application control over when (or even if) the message is unmarshaled. Another difference is that work function prototypes are generated for *replies* (on the client side) in addition to the expected ones for requests (on the server side). This completes the symmetric handling of requests and replies on each side of communication (thus the relation to message-passing versus standard RPC) and allows for complete asynchronous communication.[1]

The decomposed stub presentation is currently only supported in the CORBA C presentation generator and the IIOP back end and runtime. The Khazana runtime, when finished, will fully support the decomposed presentation as well. Support exists (in fact, much is implemented) in the main presentation generator library, but the presentation itself was a modification of the C mapping for CORBA, and thus was not carried into other presentations. To try them out, issue the command-line option "-a" (for asynchronous) to the presentation generator.

Finally, it was intended that the resulting stubs for the decomposed stubs presentation be compatible and interchangeable with the standard CORBA presentation stubs. Unfortunately, our goal could not be fully met, as the decomposed stubs required some additional information be sent in both request and reply messages, and the communication model is slightly different (the server contacts the client to send the reply message, rather than replying on the communication socket initiated by the client. A sufficiently intelligent runtime system would iron out the details and allow a standard/decomposed stub mismatch between client and server.

11.2 AOI to PRES_C Translation

The process of translating AOI to PRES_C involves the generation of MINT, CAST and stub definitions. This section discusses these data translation steps in more detail. For detailed information regarding the structure

[1] Although the presentation allows for complete asynchrony, the runtime system thus far developed only supports a polling interface for handling operation replies. To gain full asynchrony, it would be necessary to have a separate thread waiting for replies to be received (much as the server waits for requests), or at least sufficiently frequent hooks into the runtime to allow for prompt detection and dispatch of reply messages.

of the internal representations themselves, see their respective chapters in this guide.

11.2.1 Overview of the Translation Algorithm

Translation of AOI to PRES_C involves the translation of each element of `aoi.defs<>`, the sequence of AOI definitions. This is accomplished in two main stages. The first stage is building the MINT. The PG starts by creating a list of "primitive" MINT definitions, corresponding to the primitive types supported by the presentation (those that are expected to exist without being defined, such as "int" or "float"). It then makes an optional initial "preprocess" of the AOI, possibly translating or modifying information to make the PRES_C conversion easier or more straightforward. For example, CORBA's interface attributes are presented as accessor functions, so the preprocess phase in the CORBA PGs translate these attributes into operations, so the second stage can do the obvious translation on an operation. Finally, it traverses the entire AOI tree making a one-to-one translation of the AOI definitions into their MINT counterparts.

The second stage requires building the necessary CAST and linking it to the MINT through the PRES_C structures. This is (broadly) the function of `pg_state::gen`. First, the CAST is initialized similar to MINT with the primitive definitions that are expected to exist by default in the presentation. Second, the AOI definitions are walked, building the PRES_C definitions hierarchically.

The PRES_C definitions generated are specific to either the client or server side of the communication, based on parameters given to the presentation generator. Currently the infrastructure is not sufficient to generate both the client and server presentations at the same time. Perhaps this is a future direction that can be explored.

Other parameters that can provide information, direction, and guidance to a PG belong to the `pg_flags` class, which is shared by all presentation generators. These include the flags to choose client or server presentation (as described above), those for inhibiting generation of inherited operations (for when they are generated separately), one to enable the use of security identifiers, and finally one to enable the "decomposed" stub presentation style. In addition to the `pg_flags`, there is a large set of options that can modify the naming conversion between the IDL and the generated code.

11.2.2 Naming

One of the differences between presentations is the names assigned to data types and operations defined in the IDL. There is typically a well defined pattern which can be used to determine the name of an operation or data type one can use in the implementation from that described in the IDL specification. For example, the CORBA C mapping dictates that a data type `bar` inside of an interface `foo` can be used by the implementation as `foo_bar`. Under the CORBA C++ mapping, this becomes `foo::bar`. Flick has a relatively simple method of describing and handling these name conversions, very similar to C's `printf`: `calc_name`.

For more information on how to redefine or define new `calc_name` mappings, see the code documentation available in `mom/c/pfe.hh` and `c/pfe/lib/p_calc_name.cc`. The default set of name mappings can be found in `c/pfe/lib/pg_state.cc` (and likewise the sets of mappings for each presentation can be found in the presentation's `.cc` file).

11.2.3 Operations

Another common difference between presentations is the mapping from the IDL-defined operations to the actual functions that are created for use by the IDL compiler. In general, there is a one-to-one correspondence between the two sets; however that need not be the case (e.g., the decomposed stubs presentation). In addition, there are often "special" parameters that need to be added before and/or after the normal parameters given in the IDL (e.g., CORBA's object reference and environment parameters). These differences cannot be expressed solely by name translation.

There are high-level generation functions that allow one to control the stubs or skeletons that are generated. `p_client_stubs` handles all of the client-side stubs that are to be generated. For the decomposed stubs, this is modified to produce a request marshaler and reply unmarshaler for every operation, rather than just a single client stub. `p_skel` is the counterpart that produces a skeleton, or dispatch routine, again modified for the decomposed stubs to also produce request unmarshalers and reply marshalers for every operation on the server side. The decomposed presentation also uses `p_skel` to generate the client-side dispatch routine for replies.

Other high-level generation function include `p_marshal_stub` and `p_unmarshal_stub` to generate marshal and unmarshal functions for a specific data type, and `p_server_func` to generate the prototype for a server work function. More exist that are currently specific to the decomposed presentation, including `p_continue_stub`, `p_send_stub`, and `p_receive_func`.

Each of the high-level functions are similar in that they create a new stub description in the CAST and PRES_C (simultaneously) and establish the link between the the description and the MINT. As part of creating the function description, one of the `process_*_params` functions is used to create and handle each parameter of the function, including special parameters (ones that must exist for the presentation but do not appear explicitly in the IDL) such as the object reference or the CORBA environment. From there, parameters are handled individually by `p_param`, and dispatched to a specific data type presentation via `p_type`.

Most parameters are explicit, meaning they appear in the presentation. However, there are some cases where a parameter is implicit, meaning it may be constant or exist at the global level, but it isn't a formal parameter to the stub. An example of this is `errno` in the ONC RPC presentation: it is a global error property that must be set within the stub, but from the standpoint of the stubs, it can be viewed as an implicit parameter.

11.2.4 Data Types and Type Collections

Data types are generated by the `p_*_type` family of functions, some of which may recursively call each other to construct more complex types. Each operates on a *type collection* that potentially holds more than a single (default) CAST and PRES_C mapping tree for the type. This is particularly necessary for the CORBA C++ presentation that yields multiple presentation types from a single IDL type. In general, however, only the default type is used.

Following are brief descriptions of the `p_type_collection` and related interfaces.

p_type_collection

`void set_id(const char *id), const char *get_id()`

Set/get the identifier for this IDL type. Also adds the "id" tag to the tag list.

`void set_included(inclusion included), inclusion get_included()`

Set/get the inclusion for this collection, this is then passed down to be used as the default for other inclusions.

`void set_protection(cast_def_protection protection), cast_def_protection get_protection()`

Set/get the CAST protection for this collection, this is then passed down to be used as the default for other protections.

`void set_ref(aoi_ref ref), aoi_ref get_ref()`
: Set/get the AOI reference from which this collection was created.

`void set_attr_index(int idx), int get_attr_index()`
: Set/get the index used to find the `tag_list` in the presentation attributes.

`void set_name(const char *name), const char *get_name()`
: Set/get the name of the type.

`void set_tag_list(tag_list *tl), tag_list *get_tag_list()`
: Set/get the `tag_list` used to store any type information.

`void set_collection_ref(struct p_type_collection *ptc), struct p_type_collection *get_collection_ref()`
: Set/get the collection reference for this type.

`void define_types()`
: Define all of the types in the collection.

`struct p_type_node *find_type(const char *name)`
: Find a type with the given name.

`struct p_scope_node *find_scope(const char *name)`
: Find a scope with the given name.

`struct p_type_node *add_type(const char *name, struct p_type_node *ptn, int add_to_tail = 1, int add_to_ref = 1)`
: Add a type to the collection in the scope given by `name`. The `add_to_tail` flag indicates whether the type should be added to the head or tail of the list of types, which allows you to add definitions before the other rest will be defined. The `add_to_ref` flag indicates whether the type should also be added to the referenced collection, which allows you to add types that are only to be defined in the referencer collection.

`void add_scope(struct p_scope_node *psn)`
: Add a scope to the collection.

`static p_type_collection *find_collection(struct dl_list *list, aoi_ref ref), static p_type_collection *find_collection(struct dl_list *list, const char *name, cast_scope *scope)`
: Find a collection in a linked list based on its AOI reference or by matching the name of the collection and the default CAST scope.

p_scope_node

The `p_scope_node` class is used with the `p_type_collection` class to allow type declarations to be separated into separate C++ scopes. The scope node holds a reference to the `cast_scope` and the list of types in this scope.

`void set_collection(p_type_collection *ptc), p_type_collection *get_collection()`
: Set/get the collection that holds this scope node.

`void set_name(const char *name), const char *get_name()`
: Set/get the semantic name of this scope.

`void set_prefix(const char *prefix), const char *get_prefix()`
: Set/get the prefix string. This string is prepended to the formatted name of any types if this is the root scope. Otherwise, the prefix is assumed to already be a part of the scoped name.

`void set_scope_name(cast_scoped_name name), cast_scoped_name get_scope_name()`
: Set/get the actual C/C++ name of this scope.

`void set_scope(cast_scope *scope), cast_scope *get_scope()`
: Set/get the CAST scope.

`void add_type(struct p_type_node *ptn)`
: Add a type to the scope.

`struct p_type_node *find(const char *name)`
: Find a type in the scope with the given name.

`void ref_types(struct p_scope_node *psn)`
: Add references to this scope node to the types in the given scope node.

`void add_defs()`
: Add definitions of the types to the scope.

`void set_included(inclusion the_included), inclusion get_included()`
: Set/get the inclusion for this scope.

p_type_node

The `p_type_node` class is used to hold any information about a type in the presentation generator. The types it represents can be real (i.e., they end up defined in CAST) or just a different version of another type (i.e., it has a different interpretation than the standard version).

`void set_flags(unsigned int flags), unsigned int get_flags()`
: Set/get the flags for this type:

  `PTF_NAME_REF`
  : Forces the name in any reference to just be a plain name rather than one with struct/union/etc.

  `PTF_REF_ONLY`
  : This flag is set when a type is added to a collection without adding it to its referenced collection. It signifies that the type is defined only for the referencer collection and should be defined alongside the referenced types.

  `PTF_NO_REF`
  : This flag is also usually set when a type is added only to a referencer collection and signifies that it shouldn't be referenced when the collection is referenced.

`void set_name(const char *name), const char *get_name()`
: Set/get the semantic name of the type.

`void set_format(const char *name), const char *get_format()`
: Set/get the format of the type name.

`void set_type(cast_type type), cast_type get_type()`
: Set/get the CAST type.

`void set_mapping(pres_c_mapping map), pres_c_mapping get_mapping()`
: Set/get the mapping for the type.

`void set_included(inclusion included), inclusion get_included()`
: Set/get the inclusion for the type.

`const char *format_name(struct p_scope_node *psn)`
: Format the name of the type as it will be in the given scope.

`void add_def(struct p_scope_node *psn)`
: Define the type in the given scope.

`struct p_type_node *ref_type(struct p_scope_node *psn)`
: Return a new type that references this type.

`void remove(struct p_scope_node *psn)`
: Remove this type from the given scope.

11.2.5 Metadata Associations

An additional aspect of presentation generation is the semantic relationship between the entities of the presentation (at least as far as organization). For example, an IDL file may include other IDL files for definitions of common data types or interfaces from which to inherit. Because they are defined, the presentation generator creates the necessary PRES_C for them. However, it may be useful to know the origin of these inherited operations or included data structures, especially if they are compiled separately and should not be included with the native interfaces.

For this purpose, the *metadata* provided in the AOI is copied into the PRES_C, and associations between AOI entities and the metadata are preserved when generating corresponding PRES_C entities. Further, presentation-only entities (necessary constructions that are not based on the input) can be linked appropriately to the metadata structures by the presentation generator. For more information on metadata, see Chapter 3.

11.3 Flick Presentation Generator Architecture

Presentation generation programs in Flick are built using a base library containing the main procedure for the program and a C++ class, `pg_state`, which maintains the presentation generation state information. The top-level member function in `pg_state` is `gen`, which starts presentation generation for the AOI stored in the class member `in_aoi` (input AOI data structure).

IDL-specific presentation generation functionality is achieved through inheritance of `pg_state` and overriding some or all of its virtual functions as needed. Two additional libraries exist that have specialized the base presentation generator library to the standard CORBA C and C++ mappings, providing a starting ground for generators that implement or extend these CORBA standards. These are named `pg_corba` and `pg_corbaxx`, respectively.

Presentation generation code can make use of library routines in the `libaoi`, `libmeta`, `libmint`, `libcompiler`, `c/libcast`, and `c/libpres_c` directories.

11.3.1 Base Presentation Generator Library

The base Presentation Generator Library for the C language is in the directory `c/pfe/lib`, and the class declaration can be found in `mom/c/pfe.hh`. To extend the base library, one must inherit from the `pg_state` and write a simple `getGenerator` function to return an instance of the new class. By overriding the `p_*` methods, one can change the structure of the generated PRES_C. The other methods are higher-level and either create or return additional state information or guide the PRES_C generation at a higher level (e.g., `process_client_params` guides the generation of all the parameters of a client stub).

In general, this base library and other presentation generators are specific to the C language (the exception being the CORBA C++ PG library). Typically, support for a language other than C will require the development of a language-specific abstract syntax tree (analogous to CAST) and presentation structure (analogous to PRES_C). Since C++ is not far distant (structurally) from C, it was possible to extend parts of CAST and PRES_C to include the necessary information to describe and generate C++ code. However, it is not in harmony with the original intention of either CAST or PRES_C to be extended in this way.

11.3.2 CORBA C Presentation Generator

The CORBA C PG implements the standard CORBA mapping for the C language.

CORBA C PG Organization

As mentioned above, the CORBA C mapping is implemented as a library and can be found in `c/pfe/libcorba`. The class declaration is in `mom/c/pg_corba.hh`. The CORBA C PG implements the `pg_corba` class.

Notes

CORBA-style typed allocators are not yet implemented in the CORBA C language presentation. The CORBA C language presentation generator creates presentations in which heap memory is allocated with the type-independent functions `CORBA_flick_alloc` and `CORBA_flick_free`. (`CORBA_alloc` and `CORBA_free` are aliases for these functions.) To release an object reference, one must explicitly call `CORBA_object_release`.

11.3.3 CORBA C++ Presentation Generator

The CORBA C++ PG implements the standard CORBA mapping for the C++ language. It should be noted that exception handling is performed as in the CORBA C mapping, rather than using C++ exceptions. Since the C-style mapping for CORBA exceptions is permitted by the CORBA C++ mapping specification, it was decided to reuse the current infrastructure rather than spend the time developing new infrastructure to perform C++ exception handling.

CORBA C++ PG Organization

Like the CORBA C mapping, the C++ mapping is also implemented as a library in `c/pfe/libcorbaxx`. The class declaration is in `mom/c/pg_corbaxx.hh`. The CORBA C++ PG implements the `pg_corbaxx` class, and is currently the only generator that handles C++. At a high level, this PG modifies the base library in much the same way that the CORBA C PG does, since the two presentations are closely related. The specific details, however, are different for C++, and are thus reflected in the specific CAST and PRES_C generated.

Notes

Ideally, a presentation generator should not be specific to or customized for any particular runtime; that's the job of the back end. However, there may be a few things we have overlooked that are specific to the TAO runtime, since we support no other C++ runtime.

As mentioned above, C++ exception handling cannot currently be expressed in Flick. Adding this infrastructure is not trivial, but may simplify some of the other code generation, such as handling communication or allocation errors. It would be nice, in the future, to make use of C++ exceptions, not only for the presentation, but in the stub and runtime layers as well.

11.3.4 ONC RPC (`rpcgen`) Presentation Generator

ONC RPC PG Organization

The ONC RPC presentation is provided by `pg_sun` that specializes `pg_state`. The class declaration is in `c/pfe/sun`. This is a relatively simple PG in that all operations take at most one "in" parameter, and return at most one value. The only errors possible are those defined by the system and allocation and deallocation are straightforward.

Notes

The ONC RPC presentation generator does not produce `rpcgen`-like marshal and unmarshal stubs (XDR translation routines). It would likely not be hard to implement such stubs, but it has not been a priority for the development team.[2]

11.3.5 Fluke Presentation Generator

Fluke is the operating system being developed by the Flux project at the University of Utah. All of Fluke's communication stubs are produced by Flick.

Fluke PG Organization

The class declaration for the Fluke presentation generator is in `c/pfe/fluke/pg_fluke.hh`. The Fluke PG specializes the `pg_corba` class to produce `pg_fluke`).

Notes

The Fluke (MOM) presentation generator may produce bad code for recursive data types. This is because all references are "unqualified"; for example, we always refer to `struct foo` as `foo`, never `struct foo`. We do this for esoteric reasons. Eventually, we can fix self-referential data types by changing the PG library to separate the name definition from the structure definition:

[2] It has been noted that if Flick could produce such stubs, we could then use Flick instead of `rpcgen` to build the storage routines for Flick's IRs.

```
typedef struct foo foo;
struct foo {
  ...
};
```

11.3.6 MIG Presentation Generator

The Mach Interface Generator (MIG) for the Mach OS provides a special presentation that is closely related to the input IDL. Or rather, the IDL allows the user some control over the presentation in addition to defining interfaces. As such, it is not possible to simulate MIG's behavior through the normal function of the presentation generator. For this reason, Flick's MIG PG is implemented as a combined front-end and presentation generator to enable the presentation control the MIG IDL allows. Rather than cluttering our AOI representation with hints and hacks to allow for MIG, the MIG PG directly translates a representation of the input IDL (`.defs`) file to a PRES_C file. The MIG presentation generator can be found in `fe/mig`.

MIG PG Organization

The MIG PG is based on MIG's `lex`- and `yacc` based parser that converts MIG IDL into an internal parse tree. The Flick-specific "back end" then walks over this parse tree in order to produce the appropriate MINT representations of the messages and PRES_C representations of the to-be-generated client stubs and server skeletons. The Flick-specific code is contained in the following files:

`main.c`	Contains the `main` function that one would expect: parse the command line, invoke the `yacc`-generated parser, invoke the `translate` function to produce PRES_C, and then output the PRES_C file.
`xlate.c`	Contains the `translate` function, which sets up the standard MINT definitions and then invokes `translate_routines`. That function, also contained in this file, is the top-level iterator over the parsed MIG IDL statements.
`xlate_pres.c`	Contains the functions that process specific MIG IDL constructs. This is by far the largest file in Flick's MIG front end. The general organization and design of these functions is described below.
`xlate_util.c`	Contains a few utility functions to aid in the translation process, such as one to emit a `#include` statement, one to build up the initial CAST, and a debugging `printf` that indicates the filename and line number (used for error reporting).
`iheader.c`	Contains a function to produce a header file redefining the routine names to `\emph{routine}_external`.
`mom_routines.c`	Contains small utility functions for accessing and generating MINT.
`gen_error_mappings.c`	Contains the mapping of Flick's error codes to the named constants expected in the MIG presentation. This is analogous to the file of the same name in the other presentation generator directories.

Translation from MIG IDL to PRES_C

Presentation generation for MIG is organized differently from the other presentation generators, although the algorithm is the similar. The main difference is that we generate PRES_C and MINT simultaneously from the special MIG IDL parse tree, rather than separately from an AOI tree. The bulk of this translation is coded in `xlate_pres.c`.

The guiding function is `make_routine` which handles the creation of the MINT and PRES_C for exactly one RPC routine. It fills in several "out" data structures that are used by the higher-level `translate_routines` to create a `pres_c_stub`.

The high-level MINT and PRES_C is constructed by `make_top_inlines`, which produces everything from the union of all IDL's (the top-most MINT union) down to the union of the operation's request and reply, and returns the necessary hooks to produce the request or reply parameters for the operation. The PRES_C for each parameter is then processed by `make_arg_inlines`, which is in charge of expanding a single IDL argument into multiple presented arguments if needed, e.g., an array and its length. Note that these multiple presented arguments are still contained within a single inline because they are related in some fashion. The MIG presentation is unique in this regard, where IDL entities are expressed through multiple arguments (as opposed to CORBA or ONC RPC where they are encapsulated within a single C structure). Finally, `make_arg` guides the specific PRES_C construction for a single presented parameter.

There are three routines which make the lowest-level `pres_c_mapping` for a particular parameter type, as follows:

`make_ref_arg()`
: Creates the appropriate mapping for a port reference (similar to a CORBA object reference).

`make_string_arg()`
: Creates the appropriate mapping for a string parameter.

`make_simple_arg()`
: Creates the appropriate mapping for a simple, native data type (e.g., an integer, floating point, char, or boolean).

If the parameter type is an array, the element type will be created, and an array will be "interposed" on top of the element's mapping. Other things may be interposed on the mapping, such as the parameter direction, translation (between user-defined and compatible native types), allocation semantics, and the parameter's root tag. These functions are named `interpose_*` or `pres_c_interpose_*`, depending on whether or not they belong to the PRES_C library.

Two important functions regarding allocation and deallocation are `make_arg_server_dealloc_in` (to deallocate "in" parameters on the server side) and `make_arg_server_alloc_out` (to allocate "out" parameters on the server side). The latter corresponds directly to `pg_state::p_param_server_alloc_out` from the PG library. There should be a corresponding `pg_state::p_param_server_dealloc_in`, however most other presentations rely on stack-based allocation of "in" parameters on the server side and thus require no explicit deallocation. MIG however has the capability to send data *out-of-line*, and such data will be allocated automatically by the system in the receiving process's address space. In some cases the RPC system determines whether data should be sent inline or out-of-line, but must properly deallocate it to preserve a consistent presentation. There are also a few `get_*_allocation` functions to return the common `pres_c_allocation` structures describing MIG's allocation semantics.

Notes

Flick's MIG PG still cannot handle everything MIG itself supports. Most of the unsupported features, however, shouldn't take long to implement; it simply hasn't been a high priority for the development team. Following is a list of unsupported MIGisms, and a brief description of how Flick might be extended to implement them. Note that many of the unsupported features will likely require modification to both the

MIG PG and the Mach3/MIG back end. This list is also contained (briefly) in the file `doc/BUGS`. For more information regarding these concepts, refer to the MIG Server Writer's guide

- `waittime` and `nowaittime` are properties at the global level of the IDL file, affecting all operations after their use (they can be used multiple times). It seems the easiest way to implement this is add the property to the internal representation, and pick it out when generating the operation. For functions that do not have their own special `waittime` parameter (which is currently supported), an "implied" parameter may be used for the same purpose. The mapping would be similar to the supported explicit `waittime` parameter: a `PRES_C_MAPPING_MESSAGE_ATTRIBUTE` with the kind `PRES_C_MESSAGE_ATTRIBUTE_TIMEOUT`.

- `msgoption` is another global property affecting all operations that can be implemented similar to `waittime` and `nowaittime` above. Again, the parameter option version of this is currently supported.

- `serversecid` and `usersecid` indicate security identifier parameters. These can most likely be supported in the same manner that security identifiers are supported in the CORBA presentation. See `process_params` from the base library and `p_client_stub_special_params` from the CORBA PG for more information.

- `countinout` can be implemented by adding the array count parameter to the request message PRES_C structure. The catch is that the array in the reply must be rejected if it exceeds the count passed into the stub. It may be possible to accomplish this in the allocation context structure, but it may require more infrastructure to allow data to be passed between the `mu_states` of the request and reply in the back end.

- `samecount` should be relatively straightforward; the previous array's count parameter can be reused in the allocation context.

- `physicalcopy` and `overwrite` would likely be handled as a properties of the out-of-line allocator, or part of the allocation semantics.

- `retcode` probably would cause the associated parameter to be ordered first in the list of parameters, regardless of its actual position. Ideally, it would indicate that the message was actually a reply message instead of a request, and place the return code in the proper location; however, MIG's message formats for requests and replies are similar enough it could be simulated by just a parameter reordering.

- Asynchronous RPC is possible using two `simpleroutines`, one with a `retcode` parameter, and appropriate numbering of the "reply" operation. Once `retcode` is implemented, this should "just work", but it has never been tested or thoroughly thought out.

- MIG allows one to pass in buffers for 'out' parameters, so the data may be copied directly into preallocated space (if the buffer isn't large enough, it will be reallocated to an appropriate size). Unfortunately, Flick doesn't currently support this notion of buffer management. A similar situation arises in the CORBA presentation with 'inout' sequences, where we deallocate the 'in' version and allocate for the 'out' version instead of reusing the space or reallocating as needed.

- Passing arrays of polymorphic types hasn't been tested much. At one point it was believed to work, so use at your own risk.

11.4 Summary and Comments

Presentation generation is only half of the story. In many instances, a given presentation is *almost* what the user wants, with the exception of only a few minor tweaks, such as name changes or slight variations on semantics. For example, a CORBA-like presentation might be desired, but with the allocation semantics such that a zero-copy environment is possible (e.g., out parameters are always preallocated by the caller on the client side, so the data can be placed exactly where it needs to be rather than in freshly allocated space). While it is relatively simple to implement this functionality by deriving a new class from the CORBA PG library, it would be far easier and more useful to have a presentation *modification* system, where (at least some) pieces of the presentation can be tweaked according to some description. Ideally, even more complex presentations such as decomposed stubs could be expressible in a presentation modification language.

At one time, Flick had such a utility, named `pdl`, which can be found in the `c/pdl` directory. It even served as a driving utility, invoking the other stages of Flick as needed to produce stubs in a single step. The code is severely out-of-date, however, and has been largely ignored as advancements in Flick have been made. In the future, it is intended to resurrect this relic (or at least its intention) to provide a simple means whereby application writers may modify a presentation to their needs and liking, rather than limiting extension only to developers familiar with Flick's internal design.

Chapter 12

Back Ends

The presentation created by the presentation generators contains only the outside view, the language presentation and behavior, of a series of functions. To create the actual implementations of these presentations they need to be fed into a back end code generator. There are several distinct back ends which are based on a common library that does most of the work. The library and specific back ends are located under the `c/pbe` directory.

12.1 The Back End Library

The back end library is the base implementation for code generation which is then specialized to an encoding scheme and runtime. The library provides a great deal of infrastructure for dealing with various things, like command line arguments, and loading the PRES_C input file. However, the primary job of the library is to handle the translation and optimization of the input PRES_C descriptions into C or C++ code. The process of interpreting the PRES_C tree falls to the `mu_state` family of classes which provide a number of functions for walking the tree and interpreting its nodes. The `mu_state` can then be overridden by specific back ends to implement their own functionality. In addition to generating code from the PRES_C description of stubs, the back end can also employ SCML code to generate formatted C and C++ source code. This SCML code is usually defined externally and then called explicitly by the back end. The process of calling SCML macros can be done manually as it is for creating the header and footer for output files, or through the presentation implementation and collection classes described later.

The declarations for the library classes and functions can be found in the `mom/c/be` directory and the source in `c/pbe/lib`.

12.1.1 Back End State

The first set of classes we'll look at are used for maintaining state in the back end and managing the top level flow of control. The classes and functions work together based on a simple event based methodology to allow for easy expansion. The core of this model is a handler object which can process an event, it does not necessarily have to process every event, just the ones it is interested in. These handlers are then prioritized and collected into an object which is able to distribute a single event amongst its set of handlers. The resulting system allows for easy addition of code into the root control path without having to tamper directly with the library.

Types:

`be_event`

A simple structure which has an event identifier and can be subclassed to contain extra information about an event.

`be_handler`
: Contains a function pointer which is expected to handle any `be_event`s passed to it.

`be_looper`
: A subclass of `be_handler` that maintains a list of target `be_handler`s. Whenever an event is given to the looper it will distribute it to the target handlers.

`be_file`
: A subclass of `be_looper` that is used to track any files in the back end as well as generate their output. The handlers that are added to the `be_file` object are expected to generate the output for that file.

`be_state`
: A subclass of `be_looper` that acts as the primary event handler and dumping ground for anything a back end might be interested in. Handlers added to a `be_state` object are used for initializing variables, reacting to command line arguments and directing the `be_files` to generate themselves.

Functions:

`be_state *get_be_state()`
: Retrieve the `be_state` object for a specific back end

`int main(int argc, char **argv)`
: The library provided `main` simply gets a `be_state` from `get_be_state()` and then installs any handlers. Next, the command line arguments are passed to the `be_state` for processing and finally it is all set in motion by sending the initialize, command line args, and shutdown events to the `be_state`.

The current set of handler functions can be found in `c/pbe/lib/arg_handlers.cc`, `c/pbe/lib/state_handlers.cc`, and `c/pbe/lib/file_handlers.cc`. The handlers are all relatively simple, they just react to whatever events they are interested in and can then choose what to do from there. Currently, only the IIOPXX back end installs its own handler: it squelches the output of any definitions from the `orb.idl` file after the PRES has been read in by the earlier handlers.

12.1.2 Stub Generators

Once the flow of control comes down from the high level handler functions, we finally get to generate the contents of a file. The handlers in `file_handlers.cc` take care of this by calling stub generation functions for each stub described in the PRES_C. The functions provided by the back end library are empty for the most part since they are expected to be implemented by the specific back end: information is required about the runtime in order to generate correct code.

Type marshaling and unmarshaling stubs are generated a bit different from regular client and server stubs. Since they are rarely used, we generate them on demand to prevent excessive output. Whenever one of these stubs is needed (a call to the stub is generated in the code), we create an `mu_stub_info_node` and store a reference to the stub and how it is going to be used (parameter direction and byteswap flag). This node is then put into a list kept by the `be_state`. After all other stubs have been generated, the set of required marshal/unmarshal stubs are generated by traversing the list.

`void w_stub(pres_c_1 *pres, int idx)`
: This will dispatch to one of the functions below depending on the type of stub. The stubs below are meant to be overridden by the back end since they tend to be specific to the runtime.

`void w_marshal_stub(pres_c_1 *pres, int idx)`
: Writes out a marshaling stub for a specific data type. The base library defines this to do nothing.

`void w_unmarshal_stub(pres_c_1 *pres, int idx)`
: Writes out an unmarshaling stub for a specific data type. The base library defines this to do nothing.

`void w_client_stub(pres_c_1 *pres, int stub_idx)`
: Writes out a client RPC stub. This is left undefined in the base library; the responsibility belongs entirely to the specific back end how a client stub is generated.

`void w_skel(pres_c_1 *pres, int stub_idx)`
: Writes out a server or client skeleton (dispatch function). This is left undefined in the base library; the responsibility belongs entirely to the specific back end how a skeleton is generated.

`void w_send_stub(pres_c_1 *pres, int stub_idx)`
: Writes out a stub meant only for sending a specific message type (part of the decomposed stub model). The base library version causes a warning to be emitted, and no code is generated.

`void w_recv_stub(pres_c_1 *pres, int stub_idx)`
: Writes out a stub meant only for receiving a specific message type (part of the decomposed stub model). The base library version causes a warning to be emitted, and no code is generated.

`void w_msg_marshal_stub(pres_c_1 *pres, int idx)`
: Writes out a stub meant for marshaling a complete, specific message type (part of the decomposed stub model). The base library version causes a warning to be emitted, and no code is generated.

`void w_msg_unmarshal_stub(pres_c_1 *pres, int idx)`
: Writes out a stub meant for unmarshaling a complete, specific message type (part of the decomposed stub model). The base library version causes a warning to be emitted, and no code is generated.

`void w_continue_stub(pres_c_1 *pres, int idx)`
: Writes out a stub meant for continuing a delayed message (part of the decomposed stub model). The base library version causes a warning to be emitted, and no code is generated.

`void do_main_output(pres_c_1 *pres)`
: This can be used to write out a main function for a server. By default, nothing is generated.

`void make_interface_graph(pres_c_1 *pres)`
: This is a bit of a hack used to get information about runtime type information into source code. This is only really used to help implement CORBA's `is_a` operation. The function will write out a data structure containing information about interface inheritance which is then used by the runtime `is_a` function.

12.1.3 mu_state

The `mu_state` C++ class is the primary mechanism for creating marshaling and unmarshaling code. An object of this class is created and initialized with the necessary data structures from the PRES_C and then set in motion to create the code. Basically, this code creation is done by interpreting the intentions of the PRES_C nodes in the presence of specific state options, set at initialization or by context, to produce the actual code. The class itself is simply made up of a set of functions, each handling a single PRES_C node. Each function does some processing on the node, which may include using the `mu_state` to process any child nodes. Thus, flexibility comes from overriding these functions to specialize them to the needs of a back end. Any code produced is stored in CAST blocks rather than directly writing to the file, since we we may want to perform some post-processing on the generated code. Finally, the user of the `mu_state` takes whatever CAST is left after processing and wraps it with whatever boilerplate code is necessary for a complete construct.

Walking the PRES_C tree

Processing a PRES_C node usually requires more information than is contained in the node itself. This is due to the structure of the PRES_C tree; since it does not always encode references to CAST and MINT structures, these need to be walked in parallel. This is why some functions require arguments for CAST and MINT structures, however, a PRES_C node does not necessarily have to traverse the structures. For example, a `PRES_C_MAPPING_POINTER` will pass down the C type that the pointer is referring to, but the MINT structure is not descended since it has no representation of pointers. The functions that do not require a CAST object instead work off of a collection of CAST objects accessed through an `inline_state`. The `inline_state` is used to map indices from a `PRES_C_INLINE_ATOM` to a C structure, union, or function encoded in CAST. Once, the `PRES_C_INLINE_ATOM` is executed the `mu_state` goes into "mapping" mode and the selected CAST object is passed down to the mapping nodes. Once we've figured out what objects we're trying to process we need to know what we're supposed to do with them. The op slot in the `mu_state` is used for this; it can be set with several flags that will influence the code to do what is needed by the user. The current set of flags is:

`MUST_ALLOCATE`
: Memory should be allocated whenever it is needed. For a server stub, the space for the unmarshaled 'in' parameters and "root" space for the 'out' parameters that will be filled in by the work function should be allocated at this point. For a client stub, the results contained in the reply message may need to have space allocated so they can be returned to the client program. In addition, this iterator should deal with any setup necessary to begin an RPC.

`MUST_ENCODE`
: Appropriate values should be marshaled into the message stream. Given the data values as parameters to the client stub or as return parameters from the work function, they are serialized into a data stream according to the requirements of the message format and data encoding scheme in use.

`MUST_DECODE`
: Appropriate values should be unmarshaled from the message stream. The serialized data stream is converted into the data values that can be returned by the client stub or handed to the work function, according to the message format and data encoding scheme in use.

`MUST_DEALLOCATE`
: Memory should be deallocated when it is no longer needed (where appropriate). More generally, this is for any type of cleanup necessary to complete an RPC.

`MUST_REQUEST, MUST_REPLY`
: These indicate that the current state is handling a request or reply message, respectively. This

knowledge is implicitly held by the other flags and whether client or server code is being generated; however, it is more convenient to have an explicit flag to represent this. Naturally, these flags are mutually exclusive.

Marshaling Buffer Management

Stubs generated for non-trivial RPCs generally need some kind of *marshaling buffer* or *stream*, into which messages are marshaled or from which messages are unmarshaled. Sometimes a message can use multiple marshaling buffers. The format of the marshaled data in the buffer(s) depends on the encoding scheme being used by the back end (e.g., XDR is an encoding scheme), and by the MINT interface definition for the RPC in question. The format of the marshaled data does *not* usually depend on any aspects of presentation.

Some transport mechanisms can transfer an "unlimited" amount of data in one message (e.g., Mach 3); others impose some arbitrary limit (e.g., Mach 4). Flick stubs are generally expected to be able to handle an unlimited amount of data, regardless of any limitations of the transport mechanism. Fixed-length arrays are generally handled as merely a degenerate case of variable-length arrays: the two types of arrays are identical except that fixed-length arrays use a "degenerate integer" data type with only one possible value as their length data type. Thus, fixed and variable-length arrays can usually be marshaled in exactly the same way; the simple-integer-marshaling code will automatically handle the degenerate length "variable" in fixed-length arrays (see `mu_mapping_direct.cc`).

The back end support library assumes that code generation can be done by traversing the type trees in a certain "natural order":

`MINT_STRUCT`
: Marshal each element of the struct in the same order in which the elements appear in the presented structure. Note that the element ordering in the `MINT_STRUCT` may be different from that in the presented structure, which may in turn be different from that which is encoded on the wire.

`MINT_UNION`
: Marshal the union discriminator first, then marshal the selected union case.

`MINT_ARRAY`
: First marshal the array's length (if non-constant), then marshal each element in order, lowest-indexed element first.

This ordering is not mandated; it is the responsibility of the back end to choose an appropriate on-the-wire layout for the data, as the needs of a particular transport mechanism dictate. For example, just because the type graphs are traversed in the order specified above does not mean the data must always appear in memory or on the wire in exactly this order; however, it is generally easier to write the back end if the layouts match.

Generated Code

The actual stub code that gets generated is made up of a number of macro calls and some control flow code. The macros are all defined in the runtime header files and follow similar naming and calling conventions across implementations.

`runtime/headers/flick/pres`	These headers are specific to the presentation implementation; they provide macros for things like handling exceptions and memory management. This is also where any SCML files should be located.

`runtime/headers/flick/encode`	These headers provide macros for encoding and decoding data into and out of the message buffer.
`runtime/headers/flick/link`	These headers provide macros for managing the message buffers and adding any link level message data.

When the `mu_state` generates these macros it uses the `op` flags and names for the presentation, encoding scheme, and protocol (link) to determine the names of the macros. For example, the macro to encode an 8-bit character in CDR would be `flick_cdr_encode_char8()`, and the macro to decode would be `flick_cdr_decode_char8()`. Unfortunately, this approach can cause some problems because it restricts what parameters we can pass to the macros, especially presentation specific macros which can vary greatly.

Also note that much of Flick's flexibility is actually due to the implementation of the macro calls. While the names change between back ends, often the series of macro calls is very similar, and thus a significant amount of the implementation comes from the definitions of the macros.

12.1.4 client_mu_state and target_mu_state

The `client_mu_state` and `target_mu_state` C++ classes inherit from the base `mu_state` class and describe how to marshal the client and target object references, respectively. Most presentations do not have a client reference, and most protocols do not have an explicit representation for them. The client reference is necessary in the decomposed stub presentation, and thus must be handled by the protocols and runtimes that support decomposed stubs.

The reason these are special states is that the client and target references are often encoded differently than other object references; for instance, they may be placed at a known location within the message rather than encoded in the midst of other parameter data. Since a single `mu_state` can only specify one method for handling object references (e.g., the handling of an arbitrary object reference parameter), the special client and server states provide the mechanism in which the client and target references can be handled specially and separately.

12.1.5 mem_mu_state

The `mem_mu_state` class is an extension of the basic `mu_state` class, intended for use by typical back ends where parameters are marshaled (at least partly) into variable-length memory buffers of some kind. It deals with things like buffer management, alignment, endian conversion, etc.

For code optimization purposes, marshaled messages are logically divided into *globs*, then subdivided into *chunks*.

A chunk is a sequence of bytes in the marshaled message whose length and internal format is known. For example, a fixed-length array of bytes could be one big chunk; an array with variably sized elements would not be a chunk, but each individual element in that array could be. Chunks are used primarily for optimization of data packing/unpacking code: once a chunk is started, no alignment checks or pointer adjustments need to be done between successive primitives in the chunk.

A glob is part of a message whose *maximum possible length* is a "smallish" compile-time constant, even though the *actual* length may not be constant. For example, a variable-length array of bytes with a maximum length of 32 bytes would be a good candidate for being lumped into a single glob, whereas an array with variably sized elements and an unlimited length would not be: instead each individual element of the array could be a separate glob in that case. (The definition of "smallish" is defined by the back end: each back end defines some maximum glob size, usually on the order of a few kilobytes.) Globs are used to optimize marshaling buffer management: once "enough" buffer memory is allocated at the beginning of a glob, the marshaling code within the glob can simply bump through with a pointer without having to worry about buffer space again until the next glob.

12.1.6 mu_state_arglist

The `mu_state_arglist` class is used to help process some PRES_C nodes by coalescing information from their children. For example, an allocation context node has children representing the length, buffer pointer, and possibly other attributes which need to be used together in order to do the correct allocation. However, since this information is only known by the children, the arglist provides a way of passing it back up to the parent (and/or subsequently down to other siblings) so it can be used. The actual filling of the arglist is taken care of by a `PRES_C_MAPPING_ARGUMENT` node on the path to the child which will capture the current CAST expression and type and store it in a particular arglist and argument.

`mu_state_arglist(const char *name, mu_state_arglist *aparent = 0)`
: Constructs an arglist with a name and optionally connected to a parent (the new arglist must have a unique name (it cannot clash with a parent, grandparent, etc.).

`~mu_state_arglist()`
: Destructs the arglist by freeing the list of arguments and any other information.

`void add(const char *aname, const char *arg)`
: Adds an argument to the named list.

`void remove(const char *aname, const char *arg)`
: Removes an argument from the named list.

`int getargs(const char *aname, const char *arg, cast_expr *expr, cast_type *type)`
: This will attempt to find the argument, `arg` in the arglist, `aname`. If the argument is found, the associated CAST expression and type are returned with a nonzero result. If the argument or arglist cannot be found, the method will return zero, and `*expr` and `*type` will be set to `NULL`.

`int setargs(const char *aname, const char *arg, cast_expr expr, cast_type type)`
: Similar to `getargs` except this will set the value of argument. The return value is true if the argument was found and set, zero otherwise.

`argument *findarg(const char *arg)`
: This is a private function used to locate an argument inside a single list (it does not look at parents).

12.1.7 mu_abort_block

The `mu_abort_block` class is used for tracking any error handling code in a stub. Thus, we're able to do a proper recovery from any possible errors that occur during stub execution, such as a lack of memory or runtime error. To accomplish this task the class provides functionality to add code for handling an error, and then getting an "abort label" which the stub can jump to during execution if the error occurs. The abort handling code is then dumped at the end of blocks and then linked together with `goto`s. Finally, the code is reduced to only contain those blocks that are reachable.

`void set_kind(int kind), int get_kind()`
: Set/get the kind of this abort block. The possible kinds are:

`MABK_NONE`
: No kind, this is illegal.

`MABK_THREAD`
: The abort block is a simple thread of execution, one statement after another.

`MABK_CONTROL`
: The block has a unique control flow amongst its children.

`MABK_CONTROL_IF`
: The block is an if statement, with the only child representing the true branch.

`MABK_CONTROL_IF_ELSE`
: The block is an if/else statement, where the child with an expression is the true arm and the one without is the false arm.

`MABK_CONTROL_SWITCH`
: The block is a switch statement with each child representing a case.

`void set_expr(cast_expr expr), cast_expr get_expr()`
: Set/get the expression associated with this block. The expression is used by the parent control block to make a link to this child (for example, the expression used in a case).

`void set_reaper_label(cast_stmt reaper_label), cast_stmt get_reaper_label()`
: Set/get the reaper label for this block. The reaper label is the label to jump to at the end of the control flow for an abort block.

`cast_stmt get_block_label()`
: Get the `cast_label` associated with the abort block's `cast_block`. The label's statement is actually the block, and is used for external references to the start of the block. Any internal references will just be the first label in the block.

`char *use_block_label()`
: Use the block label, this will increment the use count in the `cast_label` and return the label string.

`void drop_block_label()`
: The opposite of `use_block_label`, this will decrement the use count in the `cast_label`.

`void add_stmt(cast_stmt stmt)`
: Add a statement to the abort block.

`int stmt_count()`
: Return the number of statements in the block. This is really meant for internal use to figure out if a child block really does anything and needs to be linked to by the parent.

`void add_child(struct mu_abort_block *mab, int connection)`
: Add a child to the abort block using a certain type of connection, which can be:

`MABF_INLINE`
: This indicates that the child's CAST block should be inlined directly inside the parents.

`MABF_OUT_OF_LINE`
: This indicates that the child's CAST block is located somewhere else in the code and a `goto` should be used to reach the block.

`struct mu_abort_block *find_child(char *child_label)`
: This will locate the child with the given label.

`cast_stmt get_current_label()`
: Returns the current label string.

`char *use_current_label()`
: This will return the string corresponding to the current label name and increment the use count in the `cast_label`.

`void grab_label(char *label)`
: Locates the given label name and increments its use count.

`void drop_label(char *label)`
: Locates the given label name and decrements its use count.

`void begin(), void end()`
: These perform initialization and finalization around any additions of statements and children.

`int rollback(int has_path = 0)`
: This function will walk through the abort block and all of its children trying to remove any dead code. The `has_path` argument signifies whether or not a path begins with a flow of control, and should almost always be left at zero when called externally. Basically, this function goes through each statement and tries to figure out if there is a path to that statement. If there is no path, then it turns the statement to a `CAST_STMT_EMPTY`; otherwise the statement is left alone. Determining the possibility of a path is done by looking for any `cast_label`s that are used, a prior statement that has a path, or a path from a parent.

`int rollback_block(cast_block *block, int start, int len, int has_path)`
: This is a helper function for `rollback`; this just works on `cast_block`s. The return value indicates the possibility of a control flow path.

`int rollback_stmt(cast_block *block, int pos, cast_stmt curr_stmt, int has_path)`
: This is another helper function for `rollback`, and works on `cast_stmt`s. Again, the return value indicates the possibility of a control flow path.

`void make_dispatch()`
: This is a private function used to create any code for MABK_CONTROL_* kinds. Depending on the kind it will produce the statements needed to do the test and dispatch to the correct child block.

12.1.8 mu_msg_span

The `mu_msg_span` class is used for putting buffer space checks into the stub code so it does not try to decode something that is not there. The term "span" is used to describe a message segment of an exact size that can be larger than a chunk, but not an approximation like a glob. The class is used to describe each span in the message and join them together into a tree which can then be reduced in a second pass. This reduction pass is used to merge checks together and to lift them out of array loops, if possible. For example, a union where each arm is the same size can be reduced to a single check before unmarshaling the union, rather than a check for each arm.

Using the spans depends on the format of the message that is being unmarshaled at the time. Normally, simple message segments are handled automatically by piggy-backing on the chunk functions in the `mem_mu_state`. This allows us to figure out how big a segment is, but we do not necessarily know how to join it into the rest of the message. This merging is done by creating another `mu_msg_span` which acts as a parent to a set of segments, and then setting the kind of the span to one of the formats described below.

`void set_flags(int flags), int get_flags()`
: Set/get any flags. Currently, the only flag is `MSF_ALIGN` which indicates that the span follows an alignment so it cannot be absorbed by any previous spans.

`void set_kind(int kind), int get_kind()`
: Set/get the kind. The possible kinds are:

  `MSK_NONE`
  : These contain the actual lengths of a span and are usually added to one of the other kinds.

  `MSK_SEQUENTIAL`
  : These contain other spans which are organized in a sequential fashion.

  `MSK_UNION`
  : These contain other spans which are organized in a union, so each child is a different possibility in the union. Usually, the children are `MSK_SEQUENTIAL` spans which then contain the actual span lengths.

  `MSK_ARRAY`
  : These contain another span which indicates the size of each element in the array, and the size of the array itself is contained in the this object's `size_expr`.

`void set_block(cast_stmt block), cast_stmt get_block()`
: Set/get the block where any checks should be added.

`void set_abort(struct mu_abort_block *mab), struct mu_abort_block *get_abort()`
: Set/get the abort block to use for getting labels.

`void grow(int size)`
: Grow the size of the span by a fixed size.

`void shrink(int size)`
: Shrink the size of the span by a fixed size.

`void set_size(cast_expr size)`
: Set the size of the span, which may, or may not be fixed size. If this is a `MSK_ARRAY`, then this expression is used as the length of the array.

`void add_child(struct mu_msg_span *mms)`
: Add a child span to this one.

`void begin(), void end()`
: These perform initialization and finalization around any size modifications (growing, shrinking, adding children).

`void drop()`
: Removes the check and drops the used abort label.

`void collapse()`

This is called on the root span object to go back through all of the children to optimize the sizes of the spans. For sequential spans it will walk backwards through its children trying to absorb the sizes into the previous span nodes, until it hits a variable sized node which cannot be absorbed by the previous child. For unions, it will find the minimum size of all the cases and use that as the size for the entire union. For arrays, if the length is constant it will pull all the checks out of the body of the loop and do a check at the start, otherwise it will just leave the checks in the body. Any check statements that reduce to zero are changed to `CAST_STMT_EMPTY`.

`void commit()`

This is done after the call to collapse, it basically does the final lock-down, any checks that have a size are set, empty checks are dropped.

`static void set_be_name(const char *name)`

This sets the back end name that the spans will use when producing macro calls.

12.1.9 Implementing Presentation Functions

The presentation generator is able to create "presentation functions" which have simple implementations and do not require optimization. However, the presentation generator cannot provide the implementations since they may be dependent on the runtime or other things specific to a back end. The current solution is to store a simple description of the presentation function, a semantically loaded string, in the presentation attributes tag lists. These tag lists can then be processed by the back end to produce the needed code. However, writing `printf`s for each function can become complicated and tedious, so the back end library provides a set of classes for semi-automatically passing this work off to SCML.

`presentation_impl`

This class is used to bind a tag in the presentation attributes to a set of C++ functions and/or an SCML macro. The SCML macro is determined by the `idl_type` and `pres_type` which are nested scopes in SCML, and finally the macro is selected based on the string used to describe it in the presentation generator.

`presentation_collection`

This class walks over the tag lists in the presentation attributes and then asks a set of `presentation_impl` objects to implement whatever is described.

This approach is not always necessary since the SCML interpreter can also be called explicitly to execute macro's. See file prologue and epilogue handlers in `file_handlers.cc` for an example.

12.2 Specific Back Ends

The back end library is not able to produce working code by itself, so a separate back end program must be created which provides its own functionality in addition to the library, specializing the generic code generated by the library into the specific code necessary to implement the encoding and transport layers required by a specific protocol and runtime system. These back ends are located in the `c/pbe` directory and are all structured very similarly.

- Fluke - Produces stubs for the Fluke operating system.

- IIOP - Produces CORBA IIOP stubs for the Flick-provided C runtime.
- IIOPXX - Produces CORBA IIOP stubs for the TAO C++ runtime.
- Khazana - Produces stubs for use with the Khazana distributed memory system.
- Mach3Mig - Produces stubs for use in the Mach 3 operating system using MIG-compatible messages.
- Sun - Produces ONC RPC stubs that can replace those generated by `rpcgen`.
- Trapeze - Produces stubs for the Flick-provided Trapeze based runtime using either XDR or CDR encoding (based on the originating presentation[1]).

12.2.1 The Basics

The common tasks done in a back end are to create subclasses for the `be_state` and the various `mu_states` that implement the correct behavior for the back end's encoding and runtime. Once these classes have been filled out the stub writers are overridden to do some actual work. Coding these stub generators is simply a matter of using the `mu_states` to generate CAST blocks which are then bracketed by the appropriate runtime code. Unfortunately, since all of this is relatively similar across all back ends, it has become common practice to copy an existing back end and then change all the names and `printfs` to your needs. This results in a fast start up time; however, this quickly turns into a maintenance nightmare. Hopefully in the future the redundant code in the back ends will be consolidated so that maintenance will be easier.

12.2.2 Decomposed Stubs in IIOP and Khazana

Decomposed stub generation is an option of the presentation generator that causes regular client stubs to be split into separate stubs (see Section 11.1.2 for more information). Handling this kind of presentation is done simply through implementation of the appropriate stub writers (for a detailed list, refer back to Section 12.1.2, assuming the runtime can support it. They work basically the same as the regular client and server stubs, with the functionality of the stubs split into separate functions. However, only the IIOP and Khazana back ends are currently able to handle this presentation.

12.2.3 Presentation Implementation in IIOPXX

The IIOPXX back end is responsible for generating code specific to TAO, so anything that is implementation dependent must be handled before outputting any code. For example, a string inside of a structure needs to be of the "TAO_Managed_String" type and not a "String_var" like the PG will create since it does not know the implementation. The solution is to use the presentation implementation facilities to do any preprocessing. This type of code has all been dumped in the file `c/pbe/iiopxx/tao_impl.cc`. Currently, this code ranges from generating `CORBA::TypeCode` to modifying the structs and classes created by the presentation generator. Any associated SCML code is located in `runtime/headers/flick/pres/tao_*.scml`. Eventually, it would be nice to organize the mess present in `tao_impl.cc`, but exactly how best to do it is unknown.

[1]Basing the encoding scheme on the originating presentation is certainly not an ideal solution: the choice of XDR versus CDR should be independent of which presentation is desired. See Section 12.3 for a more detailed description of the problem and proposed solution.

12.3 Summary and Comments

The back end code generators are one of the most complex pieces of Flick, mainly because of the number of choices that must (finally) be made. While the current suite of back ends is impressive, particularly in their variety and capabilities, there are a few design changes that might make them even more flexible and useful. Following are a few ideas for improvement:

Reduce/Remove Dependence on Macros. Although a lot of flexibility arises through our use of macros, the macros themselves can be hard to write and become a burden to maintain. Ideally, the back end should generate *code* and not a list of macro calls.

Split the Encoding Mechanism, the Protocol, and the Transport Mechanism. Ideally, these could all be independently chosen (and independent of the presentation as well) for maximum flexibility. For example, it would be conceivable to want ONC RPC-style messages using CDR encoding over a Trapeze/Myrinet link. To some extent, the current design allows some flexibility: the Trapeze back end and runtime support either IIOP-style messages with the CDR encoding or ONC RPC-style messages with the XDR encoding. However, most of the details are taken care of through macro magic in the runtime header files, and the encoding/message format binding is still fixed. It may be possible in some cases to change the encoding or transport protocol simply by redefining the macros for a particular back end. However, this is a rather daunting task and often requires at least some tweaks to the back end code generator. In the future, it would be nice to have each aspect of the generated code in some form of encapsulated module that could be either statically linked in as part of the back end, or perhaps even chosen at runtime by the user.

Better Define the Separation of Presentation From Implementation. While it is clear how a CORBA exception might be encoded using IIOP and CDR, it is not defined how such a construct might be sent in an ONC RPC-style message. Some sort of conversion must be done between the presentation and the encoding or message format layers to allow for the abstract presentation entities to be encoded according to the link-layer requirements. See Section 13.4 for more discussion on this as it relates to the runtime.

Chapter 13

Runtime

The Flick compiler cannot stand on its own, it needs a runtime to implement some basic functionality for the stubs. This functionality includes things like transmitting messages, receiving messages, and managing objects. In several cases, Flick's back ends use Flick's minimal runtime systems — they are designed to get the job done, but usually do not implement the full range of features that a commercial runtime might. However, a few of Flick's back ends are geared toward externally developed runtime systems, such as TAO.

13.1 Flick Runtimes

The set of runtimes located in the `runtime/libraries/link` are minimal in functionality. They provide the absolute bare-bones infrastructure needed to get an object/server and send messages back and forth. The various back ends provided with Flick are keyed to use these libraries; however, one can always adapt the back ends to one's needs.

Flick's runtime libraries are contained under the `runtime/libraries` directory of the Flick distribution:

`libflick-iiop.a`	The IIOP runtime library for C stubs, to be using with stubs generated by Flick's IIOP back end, `flick-c-pbe-iiop`.
`libflick-suntcp.a`	The ONC/TCP runtime library, to be used with stubs generated by Flick's ONC/TCP back end, `flick-c-pbe-sun`.
`libflick-mach3mig.a`	The Mach 3 runtime library, to be used with stubs generated by Flick's Mach 3 back end, `flick-c-pbe-mach3mig`.
`libflick-trapeze.a`	The Trapeze runtime library, to be used with stubs generated by Flick's Trapeze back end, `flick-c-pbe-trapeze`.

13.2 Khazana, Fluke

Khazana and Fluke are separate projects that maintain their own runtimes, so they are included directly in Flick. See `http://www.cs.utah.edu/flux/` for information about Fluke and `http://www.cs.utah.edu/flux/` for information about Khazana.

13.3 The ACE ORB (TAO)

The IIOPXX back end targets the TAO real-time ORB from Washington University in St. Louis. This ORB implements the majority of the CORBA spec; however, the spec doesn't include requirements for transmitting a raw message to another object and other low level functionality. Fortunately, TAO exports enough of this functionality for Flick to do its job. Unfortunately, this is not part of the public interface and like the internal implementation details of any system, it is is likely to change over time. As such, Flick must be adapted to these changes as they occur to take advantage of newer versions of TAO. As of this writing, the IIOPXX back end has been designed for and tested with with TAO version 1.0.

13.4 Logical Runtime and Header Separation for Flick

Flick's support of multiple presentations over multiple "transports" brings with it many real problems:

- What interface to the "runtime" does the user expect?
- What interface does Flick need to the link layer?
- At what level is the presentation independent of the link layer?
- Where should we separate the link layer from the message format?

This section is an attempt to deal with those problems in a manner that will allow Flick to grow to support additional link layers, message formats, and presentation without serious changes to the runtime interface.

13.4.1 Definitions

Here are the terms I'm going to be using, and what I mean by them:

Presentation The way an IDL maps to a given language. This includes everything the user of the stubs must know in order to use them properly.

Message Format The way the bits look in the buffer being transmitted. XDR and CDR are message formats.

Link Layer The way a buffer is transmitted between client and server; the interface to the network.

Trouble occurs when you really try and divide these three things as completely independent entities. Link Layer influences message format and presentation. Presentation influences Link Layer (and to some small degree, message format), etc. That is the real problem.

13.4.2 Presentation Issues

Presentation primarily consists of the basic language mapping. Exception/error mapping is a presentation issue. All errors that occur in stubs generated by Flick should call presentation dependent code that deals with the error accordingly. This code should be found in the header `<flick/pres/`*`presentation`*`.h>`. This header DOES concern the user, because it should contain all information about how to extract information about exceptions and errors.

13.4.3 Link Layer Issues

The link layer is primarily used by Flick internally — send request to server, get request from client, send reply to client, get request from server. The link layer also specifies what the *object reference* data structure should be. For example, Sun's RPC link layer (TCP/UDP) specifies that the object reference needs to contain an IP address, and a port number to be created. Once the object reference is created, it only really needs a socket number, though.

This is not as separable as it may initially seem, though. Object reference construction and destruction is the job of the user, generally (except for object references as parameters), so this needs to be exposed to the user. The code for the normal "Create a client object reference" or "Destroy a Client object reference" function should be kept separate from the code that Flick uses internally. So, as a user/presentation issue, these functions *should* be found in the header `<flick/pres/`*`presentation`*`.h>`. They are not, at this time. Continue reading...

We should have presentation level object references that translate to link layer object references. This can be accomplished fairly efficiently by simply tagging the presentation object reference with an intermediate object reference representation, then a conversion to the link layer object reference would be made when the connection is actually created. The user would simply link their files to the appropriate link layer library, which would implement `flick_create_obj_ref`. This would prevent the N by M pairing of presentations and link layers. When the information supplied is simply insufficient to create a valid object reference on the selected link layer, an alternative call could be made, using specific information. An example of this problem might be attempting to use the Sun RPC presentation with the Mach 3 link layer. An IP address and a server identifier are insufficient to actually find the correct Mach port. In this case, the user could manually call `flick_create_obj_ref` with the Mach port they're using.

This approach should be sufficient for the problem, but would require more time to implement. As an initial pass, there will be a header, found in the `flick/pres` directory that is named *`pres`*`_on_`*`link_layer`*. The link layer object reference information, as well as the information for using the link layer from within Flick stubs and skeletons will be located in `<flick/runtime/`*`link_layer`*`.h>`.

13.4.4 Message Format Issues

The message format is almost completely separable from the link layer. The `flick/format/`*`encoding`*`.h` header should implement the macros for globbing, chunking, and bit layout. While there are plenty of arguments against this division (we really do need a USC-like IR for this thing...) this is the way it is going to be for now. The header may contain some assumptions about the buffer layout, at least for now. This buffer information should eventually be encoded and dealt with appropriately, but for the initial cut, this overlap is acceptable.