

Argot Programmer's Guide

Version 1.3.beta

Table of Contents

1.Introduction.....	4
2.Quick Start.....	5
Booklist Data Dictionary.....	5
Java Language Implementation.....	6
3.About Argot.....	9
Specific Uses For Argot.....	10
Approaching Argot for the First Time.....	12
Environments and Versioning.....	13
Latest Versions.....	13
License	13
Getting Help.....	14
4.Binary Information.....	15
Introduction.....	15
Basic Types.....	15
Sequences.....	16
Arrays.....	16
Data Encoding.....	17
Choices.....	17
Hexadecimal Notation.....	18
Conclusion.....	18
5.Argot Meta Data.	19
Introduction.....	19
meta.sequence.....	20
meta.array.....	21
meta.reference.....	21
meta.tag.....	22
meta.encoding.....	23
meta.abstract and meta.abstract.map (choices).....	24
meta.identified (any).....	25
Type Naming.....	25
6.Creating new meta types.....	26
meta.choice (extension).....	26
7.Argot Type System.....	28
Meta Schema.....	29
Library Structure.....	29
Versioning.....	31
Type Mapping.....	32
8.Argot Libraries and Maps.....	33
Argot Type Libraries.....	33
Argot Type Maps.....	33
Type Mappers.....	35
9.Argot Message Files & Dictionaries.....	36
10.Argot Network Protocol (Dynamic Type Agreement).....	38
Check Core Type Map.....	38
Map Type.....	38
Map Reserve.....	38
Map Reverse.....	39
Get Base Object (Optional).....	39
Send Message.....	39
Error Message.....	39
11.Programming Argot.....	40
TypeLibrary.....	40
TypeMap.....	41
ReferenceTypeMap.....	42
TypeMapperCore, TypeMapperDynamic, TypeMapperError.....	43

TypeInputStream.....	43
TypeOutputStream.....	43
TypeReader.....	44
TypeWriter.....	44
TypeBound.....	44
TypeReaderAuto.....	44
TypeBeanMarshaller.....	44
TypeArrayMarshaller.....	44
12.Meta Dictionary Reference.....	45
empty.....	45
uint8.....	45
uint16.....	45
meta.id.....	45
meta.fixed_width.....	45
u8utf8.....	46
meta.name_part & meta.name.....	46
meta.version.....	46
meta.abstract.....	46
meta.abstract_map.....	47
meta.encoding.....	47
meta.reference.....	47
meta.tag.....	47
meta.sequence.....	47
meta.array.....	48
meta.envelope.....	48
meta.expression.....	48
meta.definition.....	48
meta.fixed_width.attribute.....	49
meta.fixed_width.attribute.size.....	49
meta.fixed_width.attribute.integer.....	49
meta.fixed_width.attribute.unsigned.....	49
meta.fixed_width.attribute.bigendian.....	50
dictionary.name.....	50
dictionary.definition.....	50
dictionary.relation.....	50
dictionary.location.....	50
meta.definition.envelope.....	51
dictionary.entry.....	51
dictionary.entry.list.....	51
Core Type Map.....	51
Meta Dictionary Pending Changes.....	52
Core Meta Dictionary.....	53
13.Common Data Type Reference.....	54
14.Dictionary & Argot Message File Format.....	58
15.Argot Compiler Syntax Reference.....	59

1. Introduction

The Argot library provides a programming language neutral method of describing, reading and writing information in a binary format. Argot is currently available for Java, C and .Net(C#), allowing for easy data integration between heterogeneous systems. (Note: Argot 1.3.beta is currently only available in Java.)

Argot provides binary meta data able to describe, read and write any well structured binary data. Argot's ability to work directly with binary data makes it an ideal solution where bandwidth and computing power need to be conserved. With a unique dictionary and data mapping system, it can be used to build flexible heterogeneous distributed systems. It is a perfect alternative to XML where speed, data size, and data integrity are required.

Argot features include:

- A data serialisation method which is totally language and system independent.
- Self descriptive binary data, allowing software to confirm the format of the data before attempting to read it. (Argot message format)
- Ability to define and create binary data specifications for any binary file format. This is system and machine architecture independent.
- Extensible meta data concepts which can capture any binary encoding format.
- Meta Data versioning which allows multiple versions of data structures to be defined.
- Abstract data types which allow multiple representations for a single data type to be defined.
- Ability to extended data dictionaries without modifying existing definitions.
- Definitions from multiple dictionaries to be combined and intersected to provide a single dictionary source of data types.
- Argot Network Resolution, which allows client and server to negotiate the specific data structures before attempting to send/receive them.

To get a broad overview of Argot and how Argot can be used please read the Argot white paper. It is available from the Einet web site - <http://www.einet.com.au/Articles>.

This document provides a developer with the architecture of Argot and how to use the Argot library to build Argot enabled applications.

2.Quick Start

The following example provides a simplistic data dictionary and Java implementation. It shows all the required elements for an Argot implementation that reads and writes a simple binary data file format. The example source code is included in the Argot download.

Booklist Data Dictionary

Argot uses a format similar to S-Expressions to define the structure of binary data. The following provides a very simplistic example of how a list of books could be defined using Argot Data syntax. (This format is new in Argot 1.3)

```
!import u8ascii;
!import meta.array;
!import meta.reference;
!import meta.sequence;
!import uint16;
!import uint8;
!import meta.expression;
!import meta.name;

(library.list [

(library.entry
  (library.definition meta.name:"book" meta.version:"1.0")
  (meta.sequence [
    (meta.tag u8ascii:"ISBN" (meta.reference #u8ascii))
    (meta.tag u8ascii:"title" (meta.reference #u8ascii))
    (meta.tag u8ascii:"description" (meta.reference #u8ascii))
    (meta.tag u8ascii:"author" (meta.reference #u8ascii))
  ]))

/*
 * A list of books.
 */

(library.entry
  (library.definition meta.name:"booklist" meta.version:"1.0")
  (meta.sequence [
    (meta.array
      (meta.reference #uint8)
      (meta.reference #book)
    )
  ]))

])
```

This meta data describes a very simple file format to store a list of book details. The file format is simply an array of books and is defined by the “booklist” element. This “booklist” element will be used as the top level data structure of the file. The booklist defines an array with a size element of up to 255 books (uint8 – unsigned 8-bit integer).

After an Argot file is defined with description of the required meta data, the file can be compiled to a dictionary file. The dictionary file is the argot encoded version which is loaded by the Argot runtime library. The Argot compiler is invoked using:

```
ac <file.argot> [ -o <file.dictionary> ]
```

The argot compiler can also be used from Ant in Java.

Java Language Implementation

The Argot runtime system uses a `TypeLibrary` object as the central meta data repository. When the library is first initiated it contains no type definitions by default (including the base meta data types). The easiest method to add meta data is to pass to the `TypeLibrary` constructor an array of library loaders. The following method for setting up the bookstore `TypeLibrary` is taken from the bookstore example.

```
TypeLibrary setupArgot()
throws TypeException, IOException
{
    TypeLibraryLoader libraryLoaders[] = {
        new MetaLoader(),
        new DictionaryLoader(),
        new CommonLoader(),
        new BookstoreLoader()
    };

    return TypeLibrary( libraryLoaders );
}
```

Each loader will read a dictionary file and bind the languages data types to the dictionary data types. Each binder resolves the location of the dictionary file (usually within the jar) and loads the dictionary data. Once loaded, the `TypeLibrary` will contain the type name and meta data associated with each type in the dictionary file.

The other task each loader must perform is bind each type in the dictionary to a function to read and write the data. The binding process can optionally configure one or more classes in language. The `BookstoreLoader` bind method is as follows:

```
public void bind( TypeLibrary library )
throws TypeException
{
    library.bind( library.getTypeId("book","1.0"),
        new TypeBeanMarshaller(), /* reader method */
        new TypeBeanMarshaller(), /* writer method */
        Book.class );
    library.bind( library.getTypeId("booklist","1.0"),
        new TypeArrayMarshaller(),
        new TypeArrayMarshaller(),
        Book[].class );
}
```

As you can see the book type is bound to an instance of `TypeBeanMarshaller`. This marshaller is able to read and write sequences directly into a java bean. It uses the `meta.tag` meta data to match the method names for getters and setters. The class `Book.class` has been bound to the type `book`. The `booklist` type is bound to an instance of `TypeArrayMarshaller` to perform reading and writing of arrays. This style of binding each type allows maximum flexibility when developing readers and writers for reading any binary data.

Before information can be read or written a `TypeMap` is required. A `TypeMap` is an important aspect of the Argot Library and is required when reading or writing any data. The `TypeMap` is used to limit the number of types that can be read or written to a data stream. A `TypeMap` maps a local set of type identifiers to the central `TypeLibrary` type identifiers.

```
Public TypeMap getBookTypeMap( TypeLibrary library )
throws TypeException
{
```

```

    TypeMap map = new TypeMap( library, new TypeMapperError() );
    map.map( 1, library.getId( "book" ) );
    map.map( 2, library.getId( "booklist" ) );
    map.map( 3, library.getId( "u8ascii" ) );
    map.map( 4, library.getId( "uint8" ) );
    return map;
}

```

In the example above the type identifiers 1-4 are mapped to the library identifiers.

```

public Book[] loadBooks( TypeLibrary library, String filename)
throws IOException, FileNotFoundException, TypeException
{
    FileInputStream inputStream = new FileInputStream( filename );
    TypeInputStream typeInputStream =
        new TypeInputStream( inputStream, getBookTypeMap( library ) );
    return (Book[]) typeInputStream.readObject( "booklist" );
}

```

With the TypeLibrary loaded and the TypeMap configured it is simply a matter of creating a TypeInputStream using the InputStream and TypeMap parameters. Reading data from the stream is done by identifying the data type in the readObject method. To write books to a file a similar approach is taken:

```

public void saveBooks( TypeLibrary library,
    String filename, Book[] books )
throws IOException, FileNotFoundException, TypeException
{
    FileOutputStream outputStream = new FileOutputStream( filename );
    TypeMimeOutputStream typeInputStream =
        new TypeMimeOutputStream( outputStream, getBookTypeMap( library ) );
    typeInputStream.writeObject( "booklist", books );
}

```

The TypeMap is an important aspect of Argot. It allows us to map and constrain the data types we use in a file. It will be covered in more detail later. The complete bookstore application is included in the Argot download.

As an example of how Argot encodes information, given two books:

```

[ (book u8ascii:"123" u8ascii:"the book" u8ascii:"good book" u8ascii:"me")
  (book u8ascii:"222" u8ascii:"book me" u8ascii:"bad book" u8ascii:"me") ]

```

The data would be encoded to file as shown below (hex notation on first line and comments on second):

```

0x02 0x03 0x?? 0x?? 0x?? 0x08 0x74 0x68 0x65 0x20 0x62 0x6F 0x6F 0x6B
(count) (len 3)--"123"--- -(len 8)-----"the book"-----

0x09 0x67 0x6F 0x6F 0x64 0x20 0x62 0x6F 0x6F 0x6B 0x02 0x6D 0x65
-(len 9)----"good book"----- -(len 2)--"me"--

0x03 0x?? 0x?? 0x?? 0x07 0x62 0x6F 0x6F 0x6B 0x20 0x6D 0x65
-(len 3)--"222"--- -(length 7)-----"book me"-----

```

```
0x08 0x62 0x61 0x64 0x20 0x62 0x6F 0x6F 0x6B 0x02 0x6D 0x65  
-(length 8)-----"bad book"----- -(len 2)--"me"--
```

Only the minimal information is stored. Argot uses the meta data to know which fields follow in order.

3.About Argot

Argot's design philosophy is fundamentally different to the human readability notion of XML. It is however similar to XML Schema in that it is designed to provide flexible methods of defining information using well defined meta-data. Argot meta-data is designed to reflect directly binary information to allow information to be encoded in a small well defined manner. Where XML makes it "easier" for humans to read, Argot is making it more efficient and flexible for the computer to read information.

We describe that Argot meta-data reflects directly binary encoding. Binary data is the base format heterogeneous computer systems share. It is also the most compact and succinct form of transferring information between computers. The information Argot encodes can be sent across any transport medium: sockets, http email, file, message oriented middle-ware, embedded in XML, or any other place you can transfer data. Argot can fit in the smallest of embedded devices and extend to large heterogeneous corporate systems.

Information exchange in computer science is one of the most, if not the most fundamental problems. The problem is faced when exchanging files, sending messages, email and service oriented architectures (SOA) to name a few. As distributed and network computing become central to all applications, it is surprising how few tools we have to solve the problem.

The current trend in solving the problem of information exchange is XML. We expect many people will look at Argot because XML is failing in some areas; specifically embedding binary data, large data sets, performance, version management and handling small devices. However, other than XML, what other tools are available? ASN1, CORBA and XDR are some of the more popular cross platform solutions.

The central aspect of all information exchange is agreement of how the information to be exchanged is encoded:

- In early solutions this agreement was handled by documented specifications that were then implemented by hand. This is still the case today for many file formats and other information exchange. Any change to a specification can cause ripple effects through multiple applications.
- ASN1 or Abstract Syntax Notation provides an abstract description (or contracts) of the information to be exchanged. This is then encoded using a number of different encodings. There are also ASN1 tools which use the ASN1 data to generate source code to read and write the encoded information. Formal specifications provide a much better solution to ensuring multiple parties can create software that is interoperable.
- CORBA was intended to be the final solution for building distributed systems; it would create a network of applications that could work together. However, CORBA only described services using the formal Interface Definition Language (IDL). Information external to CORBA required a different design and strategy. Using CORBA tools, the IDL specifications can be used to generate software to allow clients to access services.
- XML and XML Schema were born out of the Standard Generalized Markup Language. The W3C standards organisation defines XML as "a simple, very flexible text format derived from SGML (ISO 8879)." XML's main feature of being an easily readable text based format is its biggest advantage and in some cases its

biggest disadvantage.

One issue common to all of these methods of information exchange is that of versioning. Changes to the file format, ASN1 definition, CORBA interface, or XML schema can create high costs to maintain and update a distributed computing system. This is because the information about the format of the information is stored external to the application.

Argot was created because we see a need for a solution which provided the extensibility provided by XML and XML Schema, the ability to describe and encode binary information like ASN1, and the ability to create remote procedure call services like CORBA. The solution was also required to be aware of change to ensure that a change to one aspect of the system did not create cost across the whole network.

The Argot solution to handling change is to have each application in the network to contain all the meta data of the information it understands. This allows each Argot application to negotiate directly with other peers the information they are able to exchange. It is this fundamental change of moving the meta data knowledge internal to the application or file, rather than external which is the most important aspect of Argot.

By embedding the meta data knowledge of communications into applications we have created a system which mimics human communication. Like two people who speak multiple languages meeting for the first time and negotiating which language to speak, Argot allows two computers to negotiate directly what data they can transfer. This creates a more flexible system that is open to change.

To ensure Argot is able to provide flexible and extensible data encodings it uses binary data. By providing meta data that describes the fundamental concepts of binary encoding we are able to create the most flexible and extensible system.

Argot only deals with the syntax and meta data associated with encoding information. It does not specify the constructs required to handle service oriented architectures or the transport of the encoded information. This allows Argot to be used in any method of information exchange.

At the core of Argot is a set of 32 meta data statements which are themselves used to define the same statements. This creates a sort of rosetta stone from which all other information can be defined. These core 32 meta data statements are called the meta dictionary or meta schema. It is this concept which allows two peers to discover from first principles if the information they wish to communicate matches.

Embedding the meta dictionary, a data dictionary and data into files allows an application to ensure the structure of the data before attempting to read the data. This concept is embodied in the Argot Message Format. The Argot programming library also uses the same code used to read and write data in either file or peer to peer communications.

Argot is for all communications ranging from the smallest embedded device to large message oriented corporate systems. Our aim is to create a world that every device and applications can describe how it is able to communicate with other applications. We have proven that an Argot enabled device can shrink to as small as 3kb, or grow to service oriented architectures. Argot is specifically designed for the Internet of Things. The concept that every electronically device will eventually mesh across our homes and businesses and create a global network of interconnected devices.

Specific Uses For Argot

Argot can be used for a very wide range of problems; in-fact any time an application requires input or output. The following provide some problems where Argot can be utilised:

Service Oriented Architectures, Remote Procedure Call, Remote Method Invocation

Argot was specifically created for this purpose. Using Argot meta data, a client and server can ensure that the information to be transferred has matching schemas. This is performed by transferring the Argot meta data for each type; allowing the server to confirm that it uses the same data type structure and interface definitions as the client. This creates a strong data and interface binding between client and server. This method picks up errors faster; allows more dynamic data to be transferred; uses binary encoding to reduce network cost, is cross platform, and can be shrunk to tiny 3kb devices.

Our RPC solution is Colony which uses the Argot encoding format for all communications. Colony uses the Argot Network Resolution feature, which performs all the data type matching between client and server. It is a small and simple protocol, and can be used for a wider range of client/server interaction; for instance Object Streaming.

Message Oriented Middleware(MOM), Interprocess communication.

There are many advantages in using message based systems over remote procedure calls. Products such as MQ Series provide guaranteed delivery of messages in a flexible network. In these networks it is up to the application to decide how to encode information.

Argot provides a compact format where large data sets needs to be transferred. It can be embedded into XML documents using Base64 encoding to provide a hybrid data set.

Where an application uses MOM and RPC in a single application Argot shares the same code. This reduces the amount of code and therefor development time.

Binary XML

The combination of XML and XML Schema are very similar in nature to that of binary and Argot metadata. We have developed a prototype called Axle which provides XML to Argot encoding. In this way people are able to use familiar XML Schema tools to develop their data dictionaries. Axle then provides the ability to encode valid XML documents into binary encoded Argot equivalents.

Data Files and Argot Message Format

If you need to create data files which are not well suited for other encodings such as XML, then Argot provides an easy way to create binary encoded data files. Argot also provides the Argot Message Format which stores the meta data to describe the format of the data with the data itself. This can be useful where the types of data stored is changing or long term storage (ie 5+ years) is required.

Mobile Applications

There is a huge growth potential for mobile applications which use Wifi and other remote methods of interacting with the internet. These applications have a range of problems including high latency, low bandwidth, memory constraints and battery power consumption. These systems are also often written using differing environments from their internet servers.

Argot provides a small footprint library which creates language neutral binary encodings for client server interactions. Creating small data overheads ensures the best utilisation of bandwidth, memory and battery power.

Gaming Applications

The requirements for games differs again greatly from other environments. Games require flexible methods of communication, covering TCP/IP, UDP and sometimes their own IP based protocol. The encoding of the in-game data must be fast and open to change quickly as the game develops. Argot handles a well defined methods to specify and encode game data.

Binary Data Specifications

The main strength behind Argot is its ability to describe binary data formats. The meta data can be extended to meet the needs of any specification. Argot meta data can be used to describe data formats where the Argot programming libraries are not being used. This creates a formal language which can be used to share and describe any binary data.

Tiny Devices and Sensor Networks

In very small devices such as sensor networks there is usually a computational gap between the devices receiving the data and the computer collating and storing the data. In these situations specific code can be written in the small device to encode the binary data. In larger device the Argot library can be used to make it easier and faster to consume and collate the client device data.

Evolvable Programming Languages & Byte Code.

A more experimental use for Argot is in evolvable programming languages and byte code. As the Argot Message Format allows the definition of the data to be stored with the data, it allows the concepts stored in data to change as required. By storing the source code of a programming language as Argot, the concepts used in the language can grow and change as needed. This is discussed in a short paper on creating evolvable languages on the Eient web site. In virtual machine byte code it would allow the virtual machine to grow and change for specific needs (eg. 3D graphics, etc). Our Colony service oriented architecture system uses Argot byte code and simple virtual machine for creating very flexible communication.

Database communications

A database is itself a store of data and meta data describing how the information was stored. Instead of files or communication channels the storage is tables. Mapping the Argot meta data to database tables could allow much easier object to relational database mappings and management. We have not yet started to investigate the implementation of such a mechanism.

All of the Above

Each type of meta data and code can be used across all communications. This means that the more channels that Argot is used to communicate the greater the benefit to the developer.

Approaching Argot for the First Time

Argot is all about Information and how to encode it to a binary format. There are two possible ways to tackle learning about encoding with Argot: top down and bottom up. From the top down perspective you start with some concept of information like the parts that make up a mailing address. You dissect the various parts until you form the binary data required. From the bottom up you start with binary data and decide how you encode

various data types, building up the concepts which eventually form mailing addresses and other concepts.

We have taken the bottom up approach in this manual. It gets your hands dirty and gives you a feel for working with all those 0's and 1's. This is because after you have a general understanding of the low level concepts, it is unlikely you will need to work with them directly again. As you understand the higher level meta data concepts of Argot, you will not be required to perform the bit wrangling yourself.

Environments and Versioning

The Argot library has currently been made available for the Java, C and .Net(C#) environments. Argot 1.3 is currently only available in Java. Additional environments will be made available when there is demand. Each environment has a very similar set of functionality. Modern environments like Java and .Net(C#) provide some additional functionality for objects not available for C.

All Argot libraries provide a Major.Minor.Revision#patch level for versioning. An Argot library must stay compatible between Revision changes. A Minor change must not change the base meta dictionary. A change to the base meta dictionary will result in a change in the Major version level.

Note: Prior to version 2.0.0 a Minor change may change the base meta dictionary. This provides us with the opportunity to ensure that the meta data concepts we use create the most flexible base for future revisions. There have been considerable changes in the meta dictionary between Argot 1.2 and Argot 1.3.

Latest Versions

To gain access to the latest stable releases visit www.einet.com.au/download. Alternatively you can access the unstable and development versions by accessing our Subversion repository. The URL of the repository is `svn://www.einet.com.au/einet` or browsable at `svn.einet.com.au`

License

We have made Argot available under the Argot Public License. This is designed to be compatible with Open Source software. The LICENSE file distributed with the software contains the software license.

Please read the LICENSE file for full details of the license, however, this provides a quick list of how the software can be used.

- Argot can be used for any internal use, either commercially or non-commercially. This can be in either non-modified or modified form.
- Argot can be distributed freely only in Open Source software that has a license that has been approved by the Open Source Initiative.
- If you distribute Argot in Open Source software you may not modify the source code.
- In any use of the software you may not modify the meta dictionary elements as provided in the distributed package.

We have tried our best in developing a license which is flexible for use, yet still protecting the underlying concept of argot. Argot is based on the base of the meta dictionary which

should not be changed. If we allow modifications to be distributed, we risk companies and individuals 'innovating' on the concept, and making incompatible Argot dialects. To ensure that this does not happen we have used a half open license model.

We have shared the source of Argot to encourage use and study by individuals, universities and companies. We believe the underlying concept of Argot is an important step in the right direction for new and interesting methods of software development and communication. Making Argot source code available ensures that people can learn and study the underlying concepts.

We are always open to suggestions on how to improve the License model.

If you would like to use Argot in commercial software that will be distributed to clients or third parties, please contact us for commercial license arrangements.

Getting Help

To get any information on Argot, please visit www.einet.com.au. There are currently no forums available for Argot, however, you may contact david at [einet.com.au](mailto:david@einet.com.au) for assistance.

4. Binary Information.

Argot was created to describe binary information. Before the concepts of Argot can be understood it is important to recap some of the properties of what binary information contains. If you are familiar with binary encoding concepts, I strongly suggest you at least skim this section. It will provide a good basis for understanding Argot.

Introduction.

Binary information at its most basic is a series of ones and zeros. Binary information is one dimensional. This is a very important aspect and can't be forgotten whenever looking at binary data. To start, lets look at a section of binary data in a stream.

... 00000011000000100010001100110100 ...

What can be said about this data? It's a series of 32 bits of binary data. Other than that, we can't deduce much else. It is impossible to say if this is a section of an image, a single 32bit number (either signed or unsigned). It could be a 32bit floating point number. It could be some data from a program. What is important is that we require other information (meta data) to tell us how to interpret this data. Argot provides the meta information to allow applications to decode binary data.

Given that binary data is a one dimensional stream of bits, it is up to software engineers to group the bits together into useful information. Computers already group the information to some extent by default. When reading data from a stream it is usually read 8bits (or a byte) at a time. This makes our stream above look like:

... 00000011 00000010 00100011 00110100 ...

Argot's current implementation uses byte aligned units for all information. This makes it easier to implement and easier to decode.

Basic Types.

Basic types provide the ground work for all binary information. A basic type could also be described as atomic. It is the smallest unit of divisible information in data. Argot provides a number of basic integer data types:

- uint8 : unsigned 8-bit integer.
- int8 : signed 8-bit integer using two's complement encoding.
- uint16: unsigned big-endian 16-bit integer.
- int16: signed big-endian 16-bit integer using two's complement encoding.
- uint32: unsigned big-endian 32-bit integer.
- int32: signed big-endian 32-bit integer using two's complement encoding.
- uint64: unsigned big-endian 64-bit integer.

int64: signed big-endian 64-bit integer using two's complement encoding.
float: IEEE754 signed floating point number (32 bits).
double: IEEE754 signed double precision floating point number (64 bits).

Each size of integer is able to provide a differing range. By default Argot uses big-endian or network ordering for byte ordering. Argot can be extended to handle little-endian values if required. For more information on Integer encodings, see:

<http://en.wikipedia.org/wiki/Endianness>
http://en.wikipedia.org/wiki/Two's_complement

Returning to the example above we could say that the information encoded as two uint16 values. This would give a sequence of:

... [00000011 00000010] [00100011 00110100] ...

This can be represented using their decimal integer values:

... [770] [8996] ...

Alternatively, this data could be described as four uint8 values, a single uint32 value or a number of other things. The important thing to remember is that at some point we must reach a point that data becomes indivisible.

Argot is able to support other types of basic data types. The Argot language is extensible and allows new data types to be defined to describe any data format.

Sequences.

A sequence is one of the most important aspects of describing binary data. The example above provides a good example. After deciding that the data was two uint16 values, we were able to say that the data was a sequence of two uint16 values.

To demonstrate a sequence in another way, lets say that our original binary data is a sequence of: one uint8, one int8 and one uint16. Our data now looks as follows:

... [00000011] [00000010] [00100011 00110100] ...

As decimal values:

... [3] [2] [8996] ...

We can now say that a sequence is a list of individual types of data. This is one of the most important types of meta data we require for describing binary data. It allows us to describe any sequence of complex data.

Arrays.

The next very important concept required for binary data is repeating elements. An array is just a list of values. Arrays can be represented by any number of ways. In English you might use a colon for a list of days: Monday, Tuesday, Wednesday, Thursday, etc. In a programming language you might use square braces. A list of numbers [10, 12, 7, 2, 4].

In both of these lists we can count how many items there are in the list. There is designators to say where the start and end of the list are, however, if we have a binary file we don't get these designators.

If our data above was a list of u8 integers and part of a larger set of information, we would need a way of marking the beginning and end of the list. One method would be to use a special number to mark the beginning and the end. Let's say 00000001 was the start and 11111111 was the end. The problem with this is that as this is a list of uint8 integer values we wouldn't be able to use the number 1 or 255 in the actual list.

The solution most often used is to use a value at the start of the list saying how many items in the list/array. Looking again at the data example above, we can say that it is an array of s8 integers, with the first 8bit byte designating the length:

```
... [ 00000011 ] { [ 00000010 ] [ 00100011 ] [ 00110100 ] } ...  
           length      value      value      value
```

The first number is 00000011 which is the decimal number 3. What follows is the list of 3 8-bit values, containing the values [2, 35, 52]. This array can only be upto 255 values in length as the first uint8 byte is only capable of representing values between 0 and 255. For an array then we need two bits of meta information. We require the type of data being used to represent the length, and the type of data we are reading. Using these two parts of meta data we are able to read any array data.

Data Encoding.

A very important concept in transferring information between computers is the concept of text strings. A text string has the same encoding as an array, however, we are missing the information about which encoding it is using. The encoding tells the computer how the text should be displayed for the reader. This is very important for displaying text in different languages.

Argot allows an encoding name to be specified for any encoded data. This does not change the actual data in any way. By providing the encoding name in the meta data we are able to ensure that it is correctly read and written by the host machine.

For more information on common encodings view:

```
http://en.wikipedia.org/wiki/ASCII  
http://en.wikipedia.org/wiki/ISO\_8859  
http://en.wikipedia.org/wiki/UTF-8
```

Choices.

So far we have covered basic types, sequences and arrays. These alone provide a good set of meta data for describing binary data. A concept not yet covered is choices. A choice concept is required because we may need to choose between a number of different data types for a given concept. This fits closely to the concept of polymorphic data types in object oriented computing.

Imagine we have two different ways of representing a book identifier; the first is with its catalogue number represented by a uint16 (unsigned 16bit integer), the second is with an ISBN string.

A simple way to solve this problem is to provide an identifier which tells the computer the type of data to follow. For example.

Type 1: catalogue number.

Type 2: ISBN string.

This would allow us to encode a book identifier that is catalogue number 8 as follows:

```
[ 00000001 ] [ 00000000 00001000 ]
```

The first byte tells the computer that the type following is a catalogue number type and it should be read as such. This type of choice system can be expanded to include up to 255 different ways of representing the same concept.

Argot uses a similar technique called abstract data types. These methods are described later in this document.

Hexadecimal Notation.

So far we have demonstrated the various concepts of binary encoding using only binary notation. This becomes very inefficient as we need to demonstrate larger amounts of data. The most common form of writing binary data is using hexadecimal notation(or Hex). Hex breaks up binary values into 4bits per value. Each 4bits is represented by a value between 0-F. Hex usually has 0x at the start of each value to show that it is a hex value rather than decimal. (eg 10 is a different value to 0x10. 0x10 converted to decimal is 16).

To learn more about hexadecimal notation visit:

<http://en.wikipedia.org/wiki/Hexadecimal>

Conclusion.

This short introduction to binary encoding provides the basics for how all binary data is encoded. Argot provides the methods to describe these concepts and more. The next chapter provides details on the Argot dictionary concept and how to quickly get started.

5.Argot Meta Data.

The chapter on binary information provides the ground work for the type of information Argot meta data describes. This chapter looks at the constructs used in Argot Meta Data and how they can be applied to create dictionaries.

Introduction.

An Argot Dictionary provides a collection of meta data which is used in reading and writing binary data. Each entry in a dictionary contains a unique identifier, a unique name, and a data structure representing its meta data definition. In this chapter we will provide examples of binary data and how Argot provides definitions for them.

The first step in developing an Argot based application is developing a data dictionary for the data you wish to store/transfer/etc. The Argot library provides methods to read and write dictionary files. These files are themselves binary and stored in an Argot encoded format. You can learn more about dictionary files in the Argot message format section.

We do not currently provide a native editor for Argot dictionary files. To overcome this we have created an Argot compiler and programming syntax for creating Argot dictionaries.

The general form of a Argot data a form of S-Expressions:

S-Expressions are commonly used in Lisp. Argot uses these in a slightly different way. Each data type is defined using a type and arguments. eg.

```
(type arg1 arg2 arg3)
```

Each type in Argot has a set number of arguments as defined by its structure. Argot data can also include array data (ie repeating elements). These are defined as:

```
[ element1 element2 element3 ]
```

Any primitive type in argot is defined as:

```
<type>:<value> eg uint16:23 or u8ascii:"hello"
```

The above demonstrates defining an unsigned 16-bit integer with the value 23 and an ascii string with a length defined using an 8-bit unsigned with the value hello. This allows argot data to be very specific about how to parse and encode all data defined.

Each entry in the dictionary has a location in the dictionary and a definition. A simple book definition might be as follows:

```
(library.entry
  (library.definition meta.name:"book" meta.version:"1.0")
  (meta.sequence [
    (meta.tag u8ascii:"ISBN" (meta.reference #u8ascii))
    (meta.tag u8ascii:"title" (meta.reference #u8ascii))
    (meta.tag u8ascii:"description" (meta.reference #u8ascii))
    (meta.tag u8ascii:"author" (meta.reference #u8ascii))
  ])
```

)

This book example defines a book with the values of ISBN, title, description and author. These elements are defined using a “meta.sequence” type. The dictionary location is a “library.definition” type and labels the type name as “book” and its version as “1.0”. Each element of the book is stored using a u8ascii type as defined by the “meta.reference” type. Each element is tagged with a name using the “meta.tag” type.

As described earlier, all basic elements are also defined in Argot. For example, the u8ascii definition is defined as a common dictionary element. It is defined as:

```
(library.entry
  (library.definition meta.name:"u8ascii" meta.version:"1.3")
  (meta.encoding
    (meta.array
      (meta.reference #uint8)
      (meta.reference #uint8))
    u8utf8:"ISO646-US"))
```

A u8ascii is an array which has been encoded using the ISO646-US standard; commonly known as ASCII. If we were to have a title of a book “The Hobbit”, it would be encoded as u8ascii as:

```
0x0A 0x54 0x68 0x65 0x20 0x48 0x6F 0x62 0x62 0x69 0x74
 10   T   h   e           H   o   b   b   i   t
```

This encoding uses the array concept described in the binary information introduction. The first value describes the length of the array (ie 0x0A is 10 characters). The following 10 characters is the ASCII values for “The Hobbit”. As the length is a single byte the maximum length is 255 characters.

Each meta data type captures the details of the binary encoding we wish to perform on the data. The following provides details of each of the meta data types and the type of encoding they describe.

meta.sequence

A meta.sequence type captures information about a sequence of data types. A simple example would be the following, which defines a sequence of a single uint8, int8 and uint16 values.

```
(library.entry
  (library.definition meta.name:"mytype" meta.version:"1.0")
  (meta.sequence [
    (meta.reference #uint8 )
    (meta.reference #int8 )
    (meta.reference #uint16 )
  ])
)
```

The type “mytype”, with the values value1 = 10, value2 = -1, value3 = 5463 would be encoded to binary as:

```
0x0A 0xFF 0x15 0x57
```

The sequence simply encodes each value following the other. The definition of `meta.sequence` is:

```
(library.entry
  (library.definition meta.name:"meta.sequence"
    meta.version:"1.3")
  (meta.array
    (meta.reference #uint8)
    (meta.reference #meta.expression)))
```

This describes that a sequence is an array of up to 255 elements (`uint8`). Each element is of type `meta.expression`. `Meta.expression` is an abstract type which will be described later.

meta.array

A `meta.array` captures meta data about repeating elements. We have already seen a number of examples of this. Both the `meta.sequence` and `u8ascii` use the `meta.array` type. The `meta.array` type requires two pieces of information. The first provides how the length of the array is described, and the second is the type of data to be repeated.

As another example of `meta.array`, lets define an array of unsigned 16bit values which we can have a large number of entries.

```
(library.entry
  (library.definition meta.name:"mytype" meta.version:"1.0")
  (meta.array
    (meta.reference #u32)
    (meta.reference #u16)
  )
)
```

As an example of encoding this, lets provide a small array of the values: 413, 12, 5467.

```
0x00 0x00 0x00 0x03  0x01 0x9D  0x00 0x0C  0x15 0x5B
```

The first four bytes provide the length of the array as a `uint32` value, and then following is three `uint16` values.

meta.reference

We have already seen the `meta.reference` type being used numerous times. A `meta.reference` is used to refer to a type that has already been defined as part of the dictionary. A `meta.reference` requires the type identifier of the referenced type.

A `meta.reference` is defined as:

```
(library.entry
  (library.definition meta.name:"meta.reference"
    meta.version:"1.3")
  (meta.sequence [(meta.reference #meta.id)]))
```

The `meta.reference` uses a `meta.id` value which is an unsigned 16bit value. This is how other types are referred to in Argot. The `#` is used to signify that the identifier of the data type name should be used. Data type identifiers are assigned dynamically which is why the actual identifiers are not used.

Note: The observant readers might wonder if 65535 values is enough. This limit is only relevant at a per file or per communications channel level. The Argot library uses a 32bit integer for data types, allowing a much larger number of types when required. As each data type identifier is assigned dynamically this limit is difficult to reach. This concept is explained further when describing TypeMaps.

A meta.reference is not encoded directly, instead the type it refers to is. A very simple example of this is:

```
(meta.entry
  (meta.definition meta.name:"mytype" meta.version:"1.0")
  (meta.sequence [
    (meta.reference #uint16)
  ])
)
```

The type "mytype" in the above example is sequence of a single #uint16 value. If we had a value of 5467 and encoded "mytype" we would produce:

```
0x15 0x5B
```

meta.tag

The meta.tag type is used to provide descriptive information to elements in sequences or other parts of a structure. It does not change the underlying data encoding. In some cases it can be used to assist the binding of elements to a specific language. For instance the java TypeBeanMarshaller matches the meta.tag descriptions with a java bean's getter and setter names.

In the mytype example given for meta.sequence we defined the structure as:

```
(library.entry
  (library.definition meta.name:"mytype" meta.version:"1.0")
  (meta.sequence [
    (meta.reference #u8 )
    (meta.reference #s8 )
    (meta.reference #u16 )
  ])
)
```

An obvious problem here is that each element in the sequence is not named. This can be fixed using meta.tag:

```
(library.entry
  (library.definition meta.name:"mytype" meta.version:"1.0")
  (meta.sequence [
    (meta.tag u8ascii:"dayOfMonth" (meta.reference #u8) )
    (meta.tag u8ascii:"changeFactor" (meta.reference #s8) )
    (meta.tag u8ascii:"age" (meta.reference #u16) )
  ])
)
```

meta.fixed_width

The meta.fixed_width type is used to describing atomic or basic binary values. These are data types which can not be broken into smaller parts. Argot provides a number of

common data types which are already described. Two examples of these are “int32” (32-bit signed integer) and “uint8” (8-bit unsigned integer):

```
(library.entry
  (library.definition u8ascii:"int32" u8ascii:"1.3")
  (meta.fixed_width uint16:32
    [ (meta.fixed_width.attribute.size uint16:32)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.signed)
      (meta.fixed_width.attribute.bigendian) ] ))

(library.entry
  (library.definition u8ascii:"uint8" u8ascii:"1.3")
  (meta.fixed_width uint16:8
    [ (meta.fixed_width.attribute.size uint16:8)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.unsigned)
      (meta.fixed_width.attribute.bigendian) ] ))
```

The `meta.fixed_width` value has two parts. The first value is the number of bits the data type uses. The second part is an array of attributes. A basic value can be up to 255 bits in width. The attributes provide additional information about the data type.

The `meta.fixed_width` type is defined as:

```
(library.entry
  (library.definition meta.name:"meta.fixed_width"
    meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"size" (meta.reference #uint16))
    (meta.tag u8utf8:"attributes"
      (meta.array
        (meta.reference #uint8)
        (meta.reference #meta.fixed_width.attribute))))))
```

meta.encoding

The `meta.encoding` type is purely a meta type. It does not describe the method used to encode the binary data. It adds additional information to other types. We have already seen this used in the definition of `u8ascii`:

```
(library.entry
  (library.definition meta.name:"u8utf8" meta.version:"1.3")
  (meta.encoding
    (meta.array
      (meta.reference #uint8)
      (meta.reference #uint8))
    u8utf8:"UTF-8"))
```

As you can see from the `u8ascii` definition, the `meta.encoding` tag is used to add the encoding format to a `meta.array` using `uint8` for size and `uint8` for data. The `meta.encoding` type is defined as:

```
(library.entry
```

```
(library.definition meta.name:"meta.encoding"
  meta.version:"1.3")
(meta.sequence [
  (meta.tag u8utf8:"data" (meta.reference #meta.expression))
  (meta.tag u8utf8:"encoding" (meta.reference #u8utf8))]))
```

The encoding type is defined using a short u8ascii name. Argot does not specify the actual encoding names, however, common standard names are encouraged.

meta.abstract and meta.abstract.map (choices)

The meta.abstract and meta.abstract.map types describe a type of choice. In the discussion of choices in the binary information section, we gave the example of a book being able to be described using either an ISBN or catalogue number. Using Argot this would be done as follows:

```
(library.entry
  (library.definition meta.name:"book.isbn" meta.version:"1.0")
  (meta.reference #u8ascii))

(library.entry
  (library.definition meta.name:"book.catno" meta.version:"1.0")
  (meta.reference #u32))

(library.entry
  (library.definition meta.name:"bookid" meta.version:"1.0")
  (meta.abstract [
    (meta.abstract_map #book.isbn)
    (meta.abstract_map #book.catno)]))
```

This requires three different Argot data types. The first two data types, “book.isbn” and “book.catno” define to two concrete ways of representing a “bookid”. The “bookid” is then defined as an abstract data types with mappings to the “book.isbn” and “book.catno” types. This means that the “bookid” has no default encoding. The two concrete data types are used to map or bind the concrete data types to the abstract data type.

Abstract data types can be mapped to concrete data types at the time the abstract type is declared (as is the case above), or mapped later. In this way the abstract type can be extended with new types. For instance, a new type “book.supplierid” could be mapped to bookid:

```
(library.entry
  (library.relation meta.name:"bookid" meta.version:"1.0"
    meta.tag:"supplierid")
  (meta.abstract_map #book.supplierid))
```

This entry into the type library uses a “library.relation” location. It attaches the mapping of “book.supplierid” to the “bookid” abstract definition. Allowing these definitions to be declared separately is how Argot is able to extend the schema language.

If we had an example of a book id that was a catno type with value 23, we would encode it as a bookid using the following:

```
0x00 0x12 0x00 0x00 0x00 0x17
```

The first two bytes are used to identify the data type being used to represent the “bookid”. In this case the value is 0x00 0x12 or type id 18. In this instance type id 18 is the “book.catno” type id. When reading the type, the Argot library will check to ensure that only a valid identifier that has been mapped is used. The following data 0x00 0x00 0x00 0x17 provides the actual data, in this case the “book.catno” value of 23. The allocation of typeid's is discussed in the TypeMap section of the document.

meta.identified (any)

The meta.identified tag can be used to specify that any valid data type can follow. This is useful for elements which can be any data type. The meta.identified is defined as:

```
(library.entry
  (library.definition u8ascii:"meta.identified" u8ascii:"1.3")
  (meta.sequence [
    (meta.tag u8ascii:"description" (meta.reference #u8utf8))
  ]))
```

The meta.identified value is encoded in the same way meta.abstract is done. A uint16 type id is recorded which specifies the type to follow. The meta.identified places no constraint other than that the type id is a valid type identifier.

Type Naming

The Argot library uses a hierarchical naming structure for all names. The definition of a type name in Argot is an array of utf8 strings. The Argot language uses “.” to separate name collections when defined as text.

```
(library.entry
  (library.definition meta.name:"meta.name_part"
    meta.version:"1.3")
  (meta.reference #u8utf8))

(library.entry
  (library.definition meta.name:"meta.name" meta.version:"1.3")
  (meta.array
    (meta.reference #uint8)
    (meta.reference #meta.name_part)))
```

Please keep any names you create in their own collections. Only common data types have been provided in the base collection.

6. Creating new meta types

The meta types that Argot provides do not cover every aspect of encoding methods of binary data. However, Argot is designed to be extended. If you have an encoding or meta data which is not currently provided you are able to create a new meta data type. There are two abstract data types from which you can add in additional types; meta.definition and meta.expression.

meta.choice (extension)

A common requirement for data encoding is the concept of a choice. A choice is a more restricted version of a meta.abstract type used in the meta dictionary. A value will be used to select the type of encoding that follows.

```
0X01 0x0A 0x54 0x68 0x65 0x20 0x48 0x6F 0x62 0x62 0x69 0x74
1    10    T    h    e                H    o    b    b    i    t

0X02 0x15 0x5B
2    4567
```

In the example above a single uint8 value is used as the selector of what data follows. The value 1 requires a u8ascii string to follow, while a value of 2 requires a uint16 to follow.

One way to represent this in an argot definition which includes all the possible choices as follows:

```
(meta.entry
  (meta.definition meta.name:"mychoice" meta.version:"1.0"
    (meta.choice #uint8
      [ (meta.choice_option uint8:1 (meta.reference #u8ascii)
        (meta.choice_option uint8:2 (meta.reference #uint16) )
      ]
    )
  )
)
```

The “choice” data type could also be defined in a similar way to the meta.abstract data type allowing it to be extended using multiple statements that can be validated individually by a client.

Using this method a client is able to validate each individual choice statement as required. This also allows a third choice to be added in the future without requiring older clients to be upgraded to understand the new choice. For example the following choice map could add without changing the above:

```
(meta.entry
  (meta.relation meta.name:"mychoice" meta.version:"1.0"
    meta.tag:"option3")
  (meta.choice_option uint8:3
    (meta.array
      (meta.reference #uint8)
      (meta.reference #u8ascii)
    )
  )
)
```

In this scenario, choice three will require an array of u8ascii strings. To define the structure of meta.choice and meta.choice_option requires the following:

```

(library.entry
  (library.definition meta.name:"meta.choice"
    meta.version:"1.0")
  (meta.sequence [
    (meta.tag u8ascii:"type" (meta.reference #meta.id))
    (meta.array
      (meta.reference #uint8)
      (meta.reference #meta.choice_option))
  ])

(library.entry
  (library.definition meta.name:"meta.choice_option"
    meta.version:"1.0")
  (meta.sequence [
    (meta.tag u8ascii:"selector" (meta.identified))
    (meta.tag u8ascii:"expression" (meta.reference
      #meta.expression))
  ])
)

```

A choice and choice map must be mapped to the abstract type meta.definition element so that it can be created in dictionary files:

```

(library.entry
  (library.relation meta.name:"meta.definition"
    meta.version:"1.3"
    meta.tag:"choice")
  (meta.abstract_map #meta.choice)

(library.entry
  (library.relation meta.name:"meta.definition"
    meta.version:"1.3"
    meta.tag:"choice_option")
  (meta.abstract_map #meta.choice_option)
)

```

Note: The meta.choice concept is not currently implemented in Argot 1.3.

Creating new meta types requires a good understanding of the meta dictionary concepts. Please see the meta dictionary reference section for more details.

7. Argot Type System

The Argot type system is specifically designed to describe the format and structure of information encoded in raw binary formats. The type system, however, has a more specific requirements which require it to be able to compare and agree on data types between systems. This requirement dictates many of the features of Argot and how the type system must be created and maintained.

As explained in the introduction, Argot is designed around the idea that every device, program or file have their own type system. The advantage of every part of a system having its own type system is that tools can be developed to discover how to communicate with devices, read files or send commands to programs.

This concept has analogies in the real world where every person has their own language and a unique vocabulary. When people meet from different fields or cultures there is often an ongoing discovery process of working out a common vocabulary so that discussion can take place. Argot is designed to allow that same ongoing discovery of language. This analogy is used in Argot with a collection of data types being called a dictionary. Each device, program or file has a collection of types called a type library.

To allow Argot to perform the concept of communication discovery it requires the following:

- Every type must be defined using other types defined by the type system. This ensures that the type system has no external definitions which require special definitions. Allowing externally defined types would be like having an English dictionary with some words used in the dictionary not defined in the book.
- Every type must be able to be compared individually with the same type on a remote system. Every entry in an Argot type library or dictionary is unique. This allows a single type to be used by any device. The type is not contained by a schema file as used in XML Schema.

The result of this system is that each device in a system may have a different number of data types in its type system. Take the example below where three devices each contain a different number of type definitions in their type library.

Device A

- 200 data types.

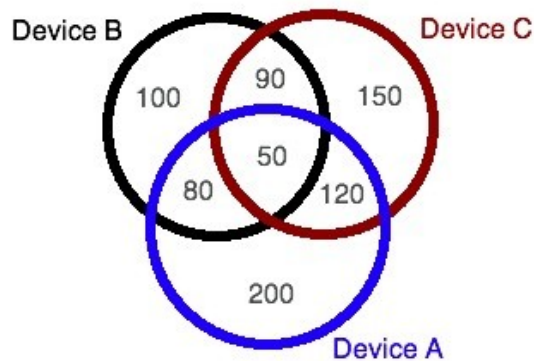
Device B

- 100 data types.

Device C

- 150 data types.

For "Device A" and "Device B" to communicate they must find the types which a common to both of them. In this hypothetical situation they have 80 data types in common. Although "Device A" has 200 data types defined most of them are not understood by "Device B"; the formats are different. However "Device A" and "Device C" have 120 data types in common so can have a more complex communication. Additionally "Device B" and "Device C" have 90 data types in common. Finally "Device A", "Device B" and "Device C" only have 50 data types in common between them. This may be useful to know if "Device A" is attempting to send a message to "Device C" via "Device B". This can all be represented using traditional Venn diagrams.



As more devices join the network with different type libraries the ability to negotiate and find the common data types becomes more and more important. This complex issue is compounded when data type versioning is introduced. Devices must be capable of distinguishing between different versions of data types.

This concept is unique to Argot and creates a method where if adopted by standards bodies allows a standard set of data types to be defined that like devices can share. However, it also allows for commercial products to extend the data types for their devices to introduce special features without compromising the base common functionality.

This concept also allows programs or devices to discover in an ad-hoc manner that they have a common set of data types where it was not expected or initially designed. The chance for serendipity between devices grows as the number of common data types between devices grows.

As all devices contain their unique data types it also allows tools to be developed which are able to build bridges between devices that may not have a common set of data types. The bridge is able to discover the data set of two devices and then provide translations between similar data types.

Meta Schema

As mentioned previously, every data type in the type system must be defined using other data types in the type system. This is true for even the basic data types such as 'uint8 – unsigned 8-bit integer'. To ensure that all type libraries contain the same set of basic data types which are used to define other types, all type libraries start with an initial set of 32 types. These types are known as the meta schema and form a self describing set of data types where each of the 32 types are defined using a combination of the other 32 data types.

The meta schema or “meta dictionary” as it is often referred is the base from which all other data types are defined. Each element of the meta dictionary is defined in a later chapter.

Library Structure

The structure of the type library is an important element of the Argot design. Each definition contained in the type system must be able to be compared against remote systems. In some situations a single data type can have both multiple versions and be made up of multiple parts. This is similar to a real world dictionary which can have multiple

entries under each word.

An example of a multi-part definition is an abstract type. Abstract data types are those that initially have no concrete representation. Concrete data types must be mapped to them so that data can be encoded. For example the following shows the “bookid” data type defined earlier.

```
(library.entry
  (library.definition meta.name:"bookid" meta.version:"1.0")
  (meta.abstract [
    (meta.abstract_map #book.isbn)
    (meta.abstract_map #book.catno)]))
```

The “bookid” data type could also be defined using the following three library entries:

```
(library.entry
  (library.definition meta.name:"bookid" meta.version:"1.0")
  (meta.abstract []))

(library.entry
  (library.definition meta.name:"bookid" meta.version:"1.0"
    meta.tag:"isbn")
  (meta.abstract_map #book.isbn))

(library.entry
  (library.definition meta.name:"bookid" meta.version:"1.0"
    meta.tag:"catno")
  (meta.abstract_map #book.catno))
```

This is designed this way to allow new entries to be added to the “bookid” abstract type over time. It allows different devices to contain different concrete representations of the “bookid” type while still allowing them to discover which concrete types are in common. For instance a device may only have one mapping to “book.isbn” while a second device may have both concrete mappings (ie to “book.isbn” and “book.catno”). These two devices are able to negotiate that they may only communicate the “book.isbn” type.

To cater for both versioning and data types with multiple parts the type library specifies each entry in the system using two parts:

Location: The location specifies a unique location in the type library. The location includes the type name, the type version and possibly additional tags required to specify how a multipart definition relates to the base type.

Definition: The definition of the data type or extension of a base type.

This two part style of definitions is designed to allow the type library to support both multiple versions of a single type and complex types that are made up of multiple parts as shown.

The type library can be considered a type of directed tree where every aspect of the tree is described or made up from another part of the tree. Each definition is inserted into a location in the tree and references other parts of the same tree. No item can be inserted into the tree while holding a reference which is not contained in the same tree.

Each location in the tree also has a unique integer identifier to assist faster negotiation, references and data type resolution. Data type identifiers require mappings from internal identifiers to external identifiers; this is covered by the section on Type Maps.

Here is an example of how the above “bookid” data type made up with three entries would be stored in the tree:

Identity: "bookid"

Version: "1.0" (id 10)

Meta.Abstract

"catno" - (id 11) meta.abstract_map #book.catno

"isbn" - (id 12) meta.abstract_map #book.isbn

The identifiers 10, 11 & 12 uniquely identify each part of the "bookid" type. The name of the data type is called the Identity. The Identity must be unique in the Type Library and does not directly relate to a type definition. Type definitions must be attached to version information. The data type "bookid" version 1.0 has the definition of a Meta.abstract type in the above entry. This entry has two sub parts (id 11 & 12). Each sub-part has a unique tag to allow the entry to be found (ie "catno" and "isbn"). Each sub-part may only be a meta.abstract_map type which maps the abstract type to concrete types or other abstract types.

Versioning

An important restriction of the Argot library is that when writing a data type to a stream only a single version of a type may be used. This restriction is by design and ensures that a developer can write data types using the type identity only and is not required to specify both identity and version. For example,

```
outStream.write( "book.isbn" , book.isbn() );
```

This requires the the correct version of the "book.isbn" type is mapped to the stream being written to. Mappings can be static or dynamic. In a distributed environment, peers may negotiate dynamically the correct version of a data type without having hard coded versions in the software.

This design also has consequences for how changes in a version can effect the stream encoding. This can be shown in the following example of an address data types:

```
/* street version 1.0 - single string */
(library.entry
  (library.definition meta.name:"street" meta.version:"1.0")
  (meta.sequence [
    (meta.tag u8utf8:"street" (meta.reference #u8utf8))
  ]))

/* street version 2.0 - number, name & type fields */
(library.entry
  (library.definition meta.name:"street" meta.version:"2.0")
  (meta.sequence [
    (meta.tag u8utf8:"number" (meta.reference #u8utf8))
    (meta.tag u8utf8:"name" (meta.reference #u8utf8))
    (meta.tag u8utf8:"type" (meta.reference #u8utf8))
  ]))
```

```

/* address version 1.0 */
(library.entry
  (library.definition meta.name:"address" meta.version:"1.0")
  (meta.sequence [
    (meta.tag u8ascii:"number" (meta.reference #u8utf8))
    (meta.tag u8ascii:"street" (meta.reference #street))
    (meta.tag u8ascii:"city" (meta.reference #u8utf8))
  ])
)

```

The three data types define two versions of the “street” data type and a single “address” data type. The encoding used on a particular stream for the address type will depend on which version of the “street” type that has been mapped (see next section on type mapping). Decoupling the version specification from the referenced types ensures that ripple effects do not occur. For example, an update to “street” would require an update to “address” and so on.

Type Mapping

Data types must be mapped to a stream before the type can be used. This mapping binds a specific version of a type to a stream. Different mapping mechanisms can be used for different types of data streams.

- **Static Mappings** – A static mapping can be used for protocols that use specific data types for the structure of data. This type of mapping would be used for protocols that do not rely on any features present in Argot as part of the data encoding.
- **Version Selected Mappings** – This style of mapping is similar to static mapping. A selected version of specific data types are chosen based on a key identifier in the protocol. Version selected mappings might be used where a protocol has gone through multiple revisions and is selected by a header parameter.
- **Dynamic Mappings** – Dynamic mappings involve the client dynamically discovering the data types used in the stream or in communication with a peer. A secondary protocol must be used to perform the type and version agreement between the peers.
- **Peer Dictated Dynamic Mappings** – This form of dynamic mapping is a variant of the dynamic mapping type. In this scenario the server or file dictates the data types and selected versions of the data used on the stream. This is used by the Argot Message Format where the file contains the data dictionary of the types used in the file. The reader must ensure that all data types and versions are understood before attempting to read the file. In a communications based system the peer dictating the types may be a small device that is not capable of doing complex data management. In these situations a simple dynamic mapping protocol may be used.
- **Peer Agreed Dynamic Mappings** – A more complex form of dynamic mapping allows two peers to dynamically find the common data types and common version of those data types that best suites the communication. This form of dynamic mapping requires that either peer may send a request to the other to perform type agreement.

Other forms of mapping can be developed by the programmer. The details of how mappings occur are covered by the following section.

8.Argot Libraries and Maps.

Each Argot application uses the concept of a type library to hold and manipulate each of the meta data concepts defined. Type maps are used to create mappings between internal libraries to external applications or files.

Argot Type Libraries.

Type Libraries provide the core meta data of any Argot application. An application can read one or more dictionaries into a single Type Library. Each data type when loaded into a Type Library is assigned a unique identifier; the type id. Each type id matches an individual Type Name. Every application is likely to contain different data types with differently assigned type identifiers.

Each type library must provide every data type referenced within the type library. The type library is usually primed with the data types which define the meta dictionary and then additional common data types are added. After the meta and basic data types are added the users data dictionaries can be loaded.

During the process of loading data types into a library, a type can be in a number of different states. They are:

TYPE_RESERVED: A type name has been reserved and type identifier assigned, but no definition has been registered. This is used when loading dictionary files.

TYPE_REGISTERED: The type contains a name, id and definition. All the information about the type is available. At this point no binding has been made to the local environment.

TYPE_COMPLETE: The type contains a name, id and definition. It has been bound to the local environment. This means that methods to read and construct objects or data in the local language have been associated with the type.

After a data dictionary has been loaded into the type library it will be in the **TYPE_REGISTERED** state. It is up to the developer to bind methods to create the designed software objects from the data to the data type. After this is done the type will be in the **TYPE_COMPLETE** state and be ready to use.

Argot Type Maps.

Type maps form a core feature of the Argot system. Type maps provide the facility of creating strong data binding between an Argot system and external Argot systems and data. Every time information is read or written a type map is used.

To create a strong data binding between systems, two systems must agree on the information they will be transferring. Traditionally this is done using an externally agreed contract. The external contract might have been the software specification or in web services an XML Schema document. In Argot the agreement can be formed dynamically between the two participants communicating. The two participants can be either in client/server interactions or between an application and file. The agreement formed is the same for both.

Type Maps are created because each Argot enabled application or file contain different

type libraries. Within these type libraries the application uses different type identifiers to uniquely find each type. For dynamic agreement to take place we must create a map from the applications internal library identifiers to the external application or files identifiers.

Take for example two Argot enabled applications communicating over the internet. The first application contains the following items in its Type Library.

Type ID	Type Name	Type Description
33	bookid	(meta.abstract)
34	book.catno	(meta.reference #uint32);
35	bookid#catno	(meta.map #bookid #book.catno)
36	booklist	(meta.array (meta.reference #u8) (meta.reference #bookid))

Note: All other data types referenced are also contained in the Type Library. They are not shown for compactness.

A second remote application contains a similar type library:

Type ID	Type Name	Type Definition
41	bookid	(meta.abstract);
42	book.catno	(meta.reference #uint32);
43	bookid#catno	(meta.map #bookid, #book.catno)
44	book.isbn	(meta.reference #u8ascii);
45	bookid#isbn	(meta.map #bookid, #book.isbn)
46	booklist	(meta.array (meta.reference #uint8) (meta.reference #bookid))

The second type library contains additional entries and each entry uses different type identifiers. For these two systems to communicate they must both create a Type Map which maps their internal identifiers to agreed identifiers. For example:

First System		Second System.	
Type ID	Map ID	Map ID	Type ID
33 (bookid)	22	22	41(bookid)
34 (book.catno)	23	23	42 (book.catno)
35 (bookid#catno)	24	24	43 (bookid#catno)
36 (booklist)	25	25	46 (booklist)

Each system contains a type map which provides a mapping of identifiers between their own internal identifiers to the agreed external id.

For an agreement to take place, it is required that both the Type Name and the Type

Definition match. If either the Type Name or Type definition do not match the specific type can not be mapped. In the scenario above the second system contains additional types of “book.isbn” and “bookid#isbn”. These can not be mapped or used by the client system as it does not define them.

When an application is reading an Argot enabled file the file is not able to define a Type Map. In these scenarios the Application creates a type map to match the identifiers used in the file. For example:

First System Type ID	Map ID	Argot File. Type ID
33 (bookid)	35	35 (bookid)
34 (book.catno)	36	36 (book.catno)
35 (bookid#catno)	37	37 (bookid#catno)
36 (booklist)	38	38 (booklist)

The same rules apply for files. If the file defines a data type which the reading application does not contain it will not be able to read the file. For more information on Argot enabled files read the “Argot Message Files & Dictionaries” section.

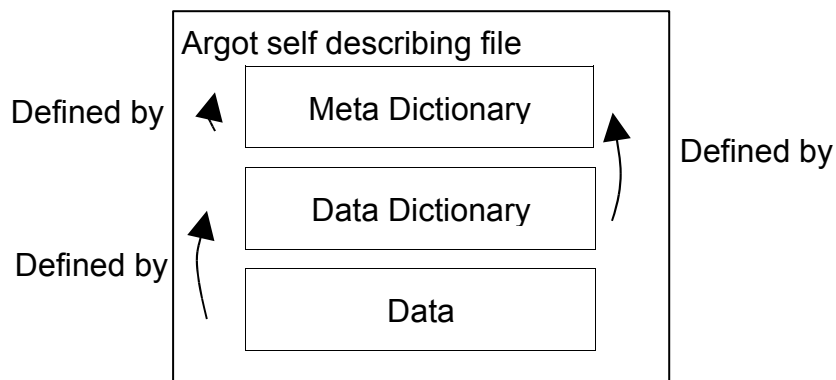
Type Mappers

Argot uses TypeMappers to create the mappings between a type name used on the stream to the corresponding data type and version in the Type Library. A TypeMapper may be static or dynamic; a dynamic TypeMapper will automatically negotiate the specific types and version with a peer or file.

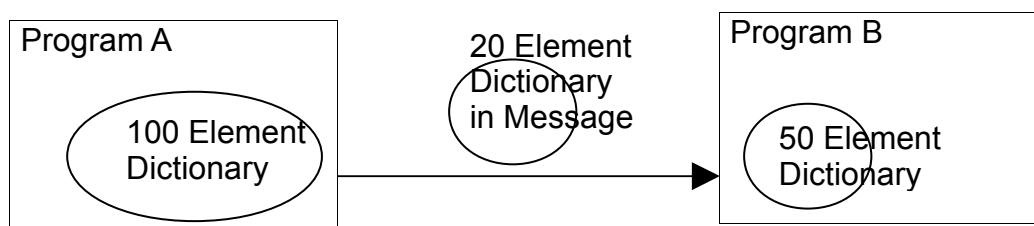
9. Argot Message Files & Dictionaries

Argot message files are binary encoded files that provide the specification of their data with the data. An Argot file contains three parts; a meta dictionary, a data dictionary and the data. Argot dictionary files use this format to store dictionary definitions.

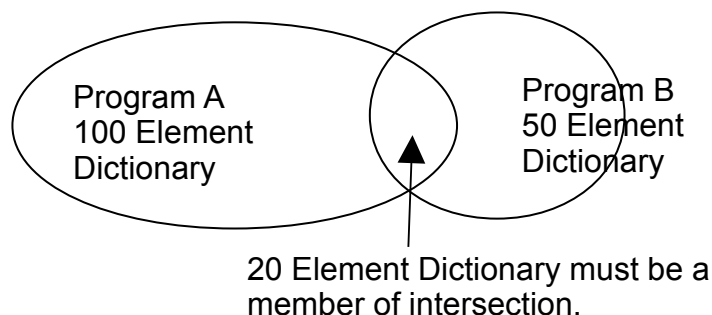
The Argot Message format allows the full specification of the data to be transferred with the data. This requires no external definition of the data. For an application to be able to read the file its type library must contain all the data types used in the file. A Type Map is generated from the data dictionary portion of the file to read the data. The general format of the file is:



The receiver of an Argot enabled file is able to read the dictionary and compare the data types of its own dictionary with that of the files. Once the types of the file dictionary have been matched with that of the application reading the file, the data can be read. This completely removes the need for a static common domain schema. Each application and file in effect contains its own schema.



This can be re-illustrated using the following set theory:



The process of reading a file involves:

1. Binary compare of meta dictionary map. The very first dictionary map of the meta dictionary is the core meta dictionary. The only way to read this entry is by performing a binary compare. These are the base dictionary items used to describe new items. Please refer to the meta dictionary reference section for details of the core meta dictionary.
2. Build and read Meta dictionary. The rest of the meta dictionary is read and mapped between the application and file.
3. Read the Data dictionary. Using the Type Map produced from entries in the Meta Dictionary the Data dictionary is read. A Data dictionary type map is created based on the types identified.
4. Read the Data. Using the Data dictionary type map the actual data of the file is read.

The argot message format can be used anywhere that a data buffer can be transferred. In files, message oriented middleware, email, etc.

10. Argot Network Protocol (Dynamic Type Agreement)

Argot uses the same Type Map agreement to communicate between client and server as it does using message files and dictionaries. However, in the case of Argot network agreement the Type Map is agreed dynamically. To perform these agreements the Argot network protocol is used. The protocol has a small number of request/response pairs which allow any client and server to agree on the data structures they plan on transferring to each other. The Argot Network Protocol can be completely separate to the actual data transfer channel.

The Argot Network Protocol has a very simple structure. It uses a single uint8 byte to identify a message type and then the actual message data. There are:

- 0x01 – Map Type
- 0x02 – Map Reserve
- 0x03 – Map Reverse
- 0x04 – Get Base Object
- 0x05 – Error
- 0x06 – Message
- 0x07 – Check Core

Each of these are described further in the following sections. It is important to note that the current Argot Network Protocol uses server dictated data type version selection.

Check Core Type Map.

The very first message that must be sent from the client is the check core type map message. It contains the binary data for the core meta dictionary. The server is able to use this to confirm that both client and server are using the same version of Argot. This also establishes the base data types required to agree on other data types.

Request: (uint8:0x07 dictionary.words /* core type map */)

The server response with true or false based on if the map matches.

Response: (uint8:0x07 u8bool: true/false }

If the core meta dictionary type map does not match the client and server can not continue. There is no basis for on-going data agreement.

Map Type.

The first time a client requires to send a specific type to the server it must perform a map type request. The request sends the name and the definition to the server.

Request: (uint8:0x01 u8ascii:name meta.definition)

The server responds with the mapped identifier:

Response: (uint8:0x01 int32:mapped_id)

In some cases the meta definition will use data structures that itself has not been mapped and these will need to be mapped before a request can be completed.

Map Reserve.

The Map reserve request is used for the special case that a type data structure's definition is self referencing or creates a dependency loop with-in its definition. In these cases the client requires an identifier for a data structure before it can finish writing a definition. In this case only the type name is sent to the server.

Request: (uint8:0x02 u8ascii:name)

The server responds with the mapped identifier:

Response: (uint8:0x02 int32:mapped_id)

Map Reverse.

In some cases the server will send a new data type that the client has not yet mapped. In this situation the client performs a reverse mapping. The client sends the server the mapped identifier given to it by the server.

Request: (uint8:0x03 int32:mapped_id)

The server responds with the name and definition

Response: (uint8:0x03 u8ascii:name meta.definition:definition)

Get Base Object (Optional).

The get base object is a simple bootstrap method for a client to discover the capabilities of a server. This request allows the server to respond with a single remote.object definition. The server can respond with false if there is no remote.object defined. A remote.object definition includes a location and a type. The location is an abstract type and can be defined by various methods depending on the environment. The type is expected to be a remote.interface, however, can be any other Argot data type.

Request: (uint8:0x04)

Response: (uint8:0x04 u8bool (remote.object meta.location meta.id))

Send Message.

The send message request is to allow using the same base protocol stack for other requests. This allows a single socket endpoint or other transport to be used. A request and reply simply include a u32binary buffer as the message envelop.

Request: (uint8:0x06 u32binary)

Response: (uint8:0x06 u32binary)

Error Message.

Any request that results in an error on the server returns an error message. An error message includes no error details.

Response: (uint8:0x05)

11. Programming Argot.

There are only a few interfaces required to learn to effectively use Argot for reading binary data. This section introduces the various interfaces and how to use them.

TypeLibrary

The type library provides the core resource of any data type definition for an application. In general an application will require only a single type library instance. In this situation the TypeLibrarySingleton(Java & .Net) is available which ensures only a single instance is created.

The type library provides the following functionality.

int getTypeState(String name);
int getTypeState(int id);

Returns the current state of a type. Will return TYPE_NOT_DEFINED, TYPE_RESERVED, TYPE_REGISTERED, TYPE_COMPLETE. Two versions exist for supplying a name or id.

int reserve(String name);

Reserves the name of a type. If the name is already reserved or registered a TypeNameDefinedException will be raised. Returns the type id assigned.

int register(String name, TypeElement structure);

Registers a name and type definition. The type must be either in the TYPE_NOT_DEFINED or TYPE_RESERVED state. If the type is in another state a TypeNameDefinedException will be raised. On success the type will be in the TYPE_REGISTERED state. Returns the type id assigned.

int register(String name, TypeElement structure, TypeReader, TypeWriter, Class class);

Registers and binds a name, definition and type readers and writers. The type must be either in the TYPE_NOT_DEFINED or TYPE_RESERVED state. If the type is in another state a TypeNameDefinedException will be raised. On success the type will be in the TYPE_COMPLETE state. Returns the type id assigned.

The Class variable is only relevant on environments which support class meta information. A class may only be registered with a single type. This value may be null.

int bind(String name, TypeReader reader, TypeWriter writer, Class class);

Binds type readers and writers to the specified data type. The type must be in the TYPE_REGISTERED state. An exception will be raised if the type is in a different state. On success the type will be in the TYPE_COMPLETE state. Returns the type id assigned.

Int getId(String name);

Returns the library type id given the type name. If the name is not registered throws a TypeNotDefinedException.

String getName(int id);

Returns the library type name given a type id. If the type id is not registered throws a TypeNotDefinedException.

TypeElement getStructure(int id);

Returns the type definition given a type id. If the type id is not registered, throws a `TypeNotDefinedException`. If the type is in an invalid state a `TypeException` is thrown.

TypeReader getReader(int id);

Returns the type reader given a type id. If the type id is not registered, throws a `TypeNotDefinedException`. If the type is in an invalid state a `TypeException` is thrown.

TypeWriter getWriter(int id);

Returns the type writer given a type id. If the type id is not registered, throws a `TypeNotDefinedException`. If the type is in an invalid state a `TypeException` is thrown.

Class getClass(int id);

Returns the native `Class` for a given type id. This is only available on environments which have reflection support (eg. Java, .Net, etc).

int getId(Class class);

Returns the type id given a native class. This is only available on environments which have reflection support.

TypeMap

A `TypeMap` maps the type identifiers from the `TypeLibrary` to external type identifiers. This is required when reading any data from a stream.

The type map provide the following functionality:

TypeMap(TypeLibrary library, TypeMapper mapper);

To construct a `TypeMap` a specific `TypeLibrary` and `TypeMapper` must be supplied. These values must not be null.

void map(int id, int libraryId);

Creates a mapping between an external id and the library id. If the id has already been mapped to a different library id a `TypeException` will be thrown.

int getStreamId(String name);

Returns the external id of the selected type name. If the name has not been mapped the `TypeMapper.mapDefault` method will be called to attempt to map the name.

String getName(int id);

Returns the type name given an external stream id. The `TypeMapper.mapReverse` method will be called if the identifier is not mapped.

TypeReader getReader(int id);

Returns the type reader given an external mapped id. The `TypeMapper.mapReverse` method will be called if the identifier is not mapped.

TypeElement getStructure(int id);

Returns the type definition given an external mapped id. The

TypeMapper.mapReverse method will be called if the identifier is not mapped.

TypeWriter getWriter(int id);

Returns the type writer given an external mapped id. The TypeMapper.mapReverse method will be called if the identifier is not mapped.

Class getClass(int id);

Returns the environment class given an external mapped id. The TypeMapper.mapReverse method will be called if the identifier is not mapped. Only available on environments that support object reflection.

int getIid(Class class);

Returns the external mapped id given the environment class. An exception will be thrown if the class is not mapped to a type or if the type has not been mapped.

int getDefinitionId(int id);

Returns the library type id given an external stream id. An exception will be thrown if the id is not mapped.

int getIid(int streamId);

Returns the external mapped id given a type library id. An exception will be thrown if the id is not mapped.

TypeLibrary getLibrary();

Returns the library this type map is associated with.

boolean isValid(int streamId);

Returns true if an external mapped id is valid. Does not throw an exception.

boolean isValid(String name);

Returns true if the name of a type has been mapped. Does not throw an exception.

boolean isMapped(int definitionId);

Returns true if the type library identifier has been mapped to a stream id. Does not throw an exception.

The standard type map requires that the user defines the mappings from the TypeLibrary to external identifiers. This is perfect when reading basic information from a file.

ReferenceTypeMap

The ReferenceTypeMap extends the TypeMap to provide a secondary reference TypeMap. This is required when writing data dictionary meta information. In this situation the ReferenceTypeMap keeps a reference to the TypeMap that the data refers. This ensures that the TypeMap used to write the meta data does not interfere with the TypeMap that the data dictionary is referring. This is used when reading and writing information in the Argot Message Format.

Note: The ReferenceTypeMap is likely to be removed in a later version. This functionality will be incorporated in a more generic way to the TypeMap allowing any additional data to be attached to the TypeMap which may be used by the stream during encoding and decoding.

TypeMapperCore, TypeMapperDynamic, TypeMapperError

Argot comes with a number of TypeMapper implementations. The TypeMapper interface performs any automated mappings between data types used on the stream and data types available in the Type Library.

The TypeMapperCore maps the meta dictionary types of the current type map. This mapper can be extended with other mappers.

The TypeMapperDynamic will automatically create identifier mappings as types are utilised. This is used to map types as used when writing files in the Argot Message Format. When the file's data is finished being written the type map contains only those types that were utilised. Using this information, the file's data dictionary is written.

The TypeMapperError does not perform any mappings and will throw an exception if any attempt is made to map a data type.

TypeInputStream

A type input stream is the interface used to read information from an input stream. It provides the following interfaces.

TypeInputStream(InputStream inStream, TypeMap map);

This is constructed with a reference to an input stream and a type map. The form of the input stream is dependent on the environment. In C a function pointer must be supplied which is able to read a buffer from the stream.

Object readObject(String name);

Object readObject(int id);

Reads the specified data type from the input stream. The type name must be mapped via the type map. The Object class type returned is dependent on the implementation of the specific type reader.

InputStream getStream();

Returns the underlying input stream passed to the constructor.

TypeOutputStream

The type output stream is the interface used to write information to a stream. It provides the following interfaces.

TypeOutputStream(OutputStream outputStream, TypeMap map);

This is constructed with a reference to an output stream and a type map. The form of the output stream is dependent on the environment. In C a function pointer must be supplied which is able to write a buffer to a stream.

void writeObject(String name, Object o);

void writeObject(int id);

Writes the specified data type to the output stream. The type name must be mapped via the type map. The Object class is required and is dependent on the implementation of the type reader of the specified name.

OutputStream getStream();

Returns the underlying output stream passed to the constructor.

TypeReader

The type reader interface provides a delegate to perform reading of each data type. The interface all type readers must implement is:

Object read(TypeInputStream in);

Each implementation of TypeReader can read directly from the InputStream. The actual definition of the element can be provided to the reader by implementing the TypeBound interface. This can be used to read the meta data of the type to change the behaviour of the type reader.

TypeWriter

The type writer interface provides a delegate to perform writing of each data type. The interface all type writers must implement is:

void write(TypeOutputStream out, Object o);

Each implementation of TypeWriter can write directly to the OutputStream. The actual definition of the element can be provided to the writer by implementing the TypeBound interface. This can be used to read the meta data of the type to change the behaviour of how the data is written to the stream.

TypeBound

The type bound interface allows readers and writers to receive additional information about the data type it is writing at the time it is bound to the TypeLibrary.

void bind(TypeLibrary library, int definitionId, TypeElement definition);

The TypeLibrary definition id and the actual structure of the definition is provided.

TypeReaderAuto

The type reader auto class provides automatic reading and construction of data objects using reflection. This is only available on environments such as Java and .Net which offer reflection. This implements the TypeReader interface and automatically reads the data using the type definition meta data. It constructs an object in the environment by searching the provided objects constructors for a constructor that matches the elements of the data type.

TypeBeanMarshaller

The type bean marshaller is similar to the TypeReaderAuto class. This class assumes a bean interface to the bound class. An empty constructor must be available and getter/setter names must match the meta.tag names used in the type definition.

TypeArrayMarshaller

The array marshaller automatically reads/writes array data types.

12. Meta Dictionary Reference.

The meta dictionary is an important aspect of Argot. It is the core of Argot and defines the core set of definitions from which all other data types are created. Every element in the meta dictionary is defined using other definitions from the same meta dictionary. The following provides the definitions of each type in the meta dictionary:

empty

The empty type is a place holder for a type that has no data associated. This can be useful for identifying specific data which has no additional information associated other than the identifier itself.

```
(library.entry
  (library.definition meta.name:"empty" meta.version:"1.3")
  (meta.fixed_width uint16:0
    [ (meta.fixed_width.attribute.size uint16:0) ]))
```

uint8

This is a basic element which reads an unsigned eight bit number between 0 and 255.

```
(library.entry
  (library.definition meta.name:"uint8" meta.version:"1.3")
  (meta.fixed_width uint16:8
    [ (meta.fixed_width.attribute.size uint16:8)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.unsigned)
      (meta.fixed_width.attribute.bigendian) ] ))
```

uint16

This is a basic element which reads an unsigned sixteen bit big endian number between 0 and 65536.

```
(library.entry
  (library.definition meta.name:"uint16" meta.version:"1.3")
  (meta.fixed_width uint16:16
    [ (meta.fixed_width.attribute.size uint16:16)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.unsigned)
      (meta.fixed_width.attribute.bigendian) ] ))
```

meta.id

All data types are referenced or identified using a meta.id type. As this is a 16bit integer the largest single dictionary can be 65536 data type definitions.

```
(library.entry
  (library.definition meta.name:"meta.id" meta.version:"1.3")
  (meta.reference #uint16))
```

meta.fixed_width

This defines how the meta data associated with a basic data such as integers and floats are defined. The first value is the size if available and the second value is additional attributes. Fixed width data types are atomic values which require specific function to read

the values.

```
(library.entry
  (library.definition meta.name:"meta.fixed_width"
    meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"size" (meta.reference #uint16))
    (meta.tag u8utf8:"flags"
      (meta.array
        (meta.reference #uint8)
        (meta.reference #meta.fixed_width.attribute))))])
```

u8utf8

An array of utf8 encoded strings of maximum length 255 characters. This is used to describe the names of each type in the Argot.

```
(library.entry
  (library.definition meta.name:"u8utf8" meta.version:"1.3")
  (meta.encoding
    (meta.array
      (meta.reference #uint8)
      (meta.reference #uint8))
    u8utf8:"UTF-8"))
```

meta.name_part & meta.name

A name in argot is defined as an array of name parts. This allows definitions to be grouped into different packages.

```
(library.entry
  (library.definition meta.name:"meta.name_part" meta.version:"1.3")
  (meta.reference #u8utf8))

(library.entry
  (library.definition meta.name:"meta.name" meta.version:"1.3")
  (meta.array
    (meta.reference #uint8)
    (meta.reference #meta.name_part)))
```

meta.version

The version of a data type is identified with a major and minor field.

```
(library.entry
  (library.definition meta.name:"meta.version" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"major" (meta.reference #uint8))
    (meta.tag u8utf8:"minor" (meta.reference #uint8))
  ]))
```

meta.abstract

The abstract data type is an important concept in Argot. It allows a concept to be defined which has no definition associated. A map is used to associate a concrete representation with an abstract data type. This allows the Argot to be expanded by the user without

modifying types previously defined. Concrete types may be mapped as part of the abstract definition or added later in separate definitions.

```
(library.entry
  (library.definition meta.name:"meta.abstract" meta.version:"1.3")
  (meta.sequence [
    (meta.array
      (meta.reference #uint8)
      (meta.reference #meta.abstract_map))]))
```

meta.abstract_map

A `abstract_map` is used to map an abstract data type to a concrete data type or other abstract type.

```
(library.entry
  (library.definition meta.name:"meta.abstract_map" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"id" (meta.reference #meta.id))
  ]))
```

meta.encoding

Encoding specifies the data encoding used on a character string or other type expression.

```
(library.entry
  (library.definition meta.name:"meta.encoding" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"data" (meta.reference #meta.expression))
    (meta.tag u8utf8:"encoding" (meta.reference #u8utf8))]))
```

meta.reference

A reference declares the usage of another data type in the system.

```
(library.entry
  (library.definition meta.name:"meta.reference" meta.version:"1.3")
  (meta.sequence [(meta.reference #meta.id)]))
```

meta.tag

A tag is used to associate a name to a sub-expression of a data type.

```
(library.entry
  (library.definition meta.name:"meta.tag" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"name"
      (meta.reference #u8utf8))
    (meta.tag u8utf8:"data"
      (meta.reference #meta.expression))]))
```

meta.sequence

A sequence defines a set of expressions which are executed in order. In the most normal case, it defines an ordered set of types in a data buffer.

```
((library.entry
  (library.definition meta.name:"meta.sequence" meta.version:"1.3")
  (meta.array
    (meta.reference #uint8)
    (meta.reference #meta.expression)))
```

meta.array

An Array is used to define any collection of data with a size and a type. The abstract type `exprssion` is used to define how the size and type is specified.

```
(library.entry
  (library.definition meta.name:"meta.array" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"size" (meta.reference #meta.expression))
    (meta.tag u8utf8:"data" (meta.reference #meta.expression)) ]))
```

meta.envelope

The envelope type is used to wrap a data type in an array. This allows the data type to be read without decoding it.

```
(library.entry
  (library.definition meta.name:"meta.envelop" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"size" (meta.reference #meta.expression))
    (meta.tag u8utf8:"type" (meta.reference #meta.expression)) ]))
```

meta.expression

The abstract expression type is mapped to reference, tag, sequence, array, envelop and encoding. Any of the concrete types can be used in place of the expression type.

```
(library.entry
  (library.definition meta.name:"meta.expression" meta.version:"1.3")
  (meta.abstract [
    (meta.abstract_map #meta.reference)
    (meta.abstract_map #meta.tag)
    (meta.abstract_map #meta.sequence)
    (meta.abstract_map #meta.array)
    (meta.abstract_map #meta.envelop)
    (meta.abstract_map #meta.encoding)
  ]))
```

meta.definition

The definition of a type is abstract. Specific concrete types are mapped to the abstract type. The abstract type “`meta.expression`” is also mapped allowing any concrete mapped to that type to be used as a “`meta.definition`”.

```
(library.entry
  (library.definition meta.name:"meta.definition" meta.version:"1.3")
  (meta.abstract [
    (meta.abstract_map #meta.fixed_width)
    (meta.abstract_map #meta.abstract)
```

```
(meta.abstract_map #meta.abstract_map)
(meta.abstract_map #meta.expression)
(meta.abstract_map #meta.identity)
)))
```

meta.fixed_width.attribute

A `fixed_width.attribute` is an abstract type which is mapped to attributes for describing basic data types. Only those attributes which are actively used in the meta dictionary are mapped. Additional attributes are defined and mapped in the common dictionary.

```
(library.entry
  (library.definition meta.name:"meta.fixed_width.attribute"
    meta.version:"1.3")
  (meta.abstract [
    (meta.abstract_map #meta.fixed_width.attribute.size)
    (meta.abstract_map #meta.fixed_width.attribute.integer)
    (meta.abstract_map #meta.fixed_width.attribute.unsigned)
    (meta.abstract_map #meta.fixed_width.attribute.bigendian)
  ]))
```

meta.fixed_width.attribute.size

Specifies the size of a `fixed_width` data type.

```
(library.entry
  (library.definition meta.name:"meta.fixed_width.attribute.size"
    meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"size" (meta.reference #uint16))
  ]))
```

meta.fixed_width.attribute.integer

Specifies that the `fixed_width` data type being defined is an integer type.

```
(library.entry
  (library.definition meta.name:"meta.fixed_width.attribute.integer"
    meta.version:"1.3")
  (meta.sequence []))
```

meta.fixed_width.attribute.unsigned

Specifies that the `fixed_width` data type being defined is an unsigned type.

```
(library.entry
  (library.definition meta.name:"meta.fixed_width.attribute.unsigned"
    meta.version:"1.3")
  (meta.sequence []))
```

meta.fixed_width.attribute.bigendian

Specifies that the fixed_width data type being defined is encoded in bigendian network order.

```
(library.entry
  (library.definition meta.name:"meta.fixed_width.attribute.bigendian"
    meta.version:"1.3")
  (meta.sequence[]))
```

dictionary.name

```
(library.entry
  (library.definition meta.name:"dictionary.name" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"name" (meta.reference #meta.name))
  ]))
```

dictionary.definition

The dictionary definition type is a location specifier. It is the encoded version of the “library.definition” type. The difference between the two is that the “library.definition” type uses the “meta.name” for the type while the “dictionary.definition” uses the “meta.id”.

```
(library.entry
  (library.definition meta.name:"dictionary.definition" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"id" (meta.reference #meta.id))
    (meta.tag u8utf8:"version" (meta.reference #meta.version))
  ]))
```

dictionary.relation

The dictionary relation type is a location specifier. It is the encoded version of the “library.relation” type. The “library.relation” specifies the name, version and tag while the encoded version replaces the name and version with a identifier.

```
(library.entry
  (library.definition meta.name:"dictionary.relation" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"id" (meta.reference #meta.id))
    (meta.tag u8utf8:"tag" (meta.reference #u8utf8))
  ]))
```

dictionary.location

The dictionary location type is the abstract type for all location specifiers.

```
(library.entry
  (library.definition meta.name:"dictionary.location" meta.version:"1.3")
  (meta.abstract [
    (meta.abstract_map #dictionary.name)
    (meta.abstract_map #dictionary.definition)
    (meta.abstract_map #dictionary.relation)
  ]))
```

```
]))
```

meta.definition.envelope

The envelope is used to wrap a definition. This allows the dictionary to read all data types before attempting to process the actual definition.

```
(library.entry
  (library.definition meta.name:"meta.definition.envelop"
    meta.version:"1.3")
  (meta.envelope
    (meta.reference #uint16)
    (meta.reference #meta.definition)))
```

dictionary.entry

A dictionary entry is the encoded version of a "library.entry". The encoded version specifies a unique identifier in addition to the location and definition of the data type.

```
(library.entry
  (library.definition meta.name:"dictionary.entry" meta.version:"1.3")
  (meta.sequence [
    (meta.tag u8utf8:"id" (meta.reference #meta.id))
    (meta.tag u8utf8:"location" (meta.reference #dictionary.location))
    (meta.tag u8utf8:"definition"
      (meta.reference #meta.definition.envelop))]))
```

dictionary.entry.list

An array of dictionary.entry items.

```
(library.entry
  (library.definition meta.name:"dictionary.entry.list" meta.version:"1.3")
  (meta.array
    (meta.reference #uint16)
    (meta.reference #dictionary.entry )))
```

Core Type Map

In order to negotiate the meta dictionary between file or remote systems the core set of type identifiers must be defined. The following are the core identifiers required for the meta dictionary.

```
EMPTY_ID = 1;
UINT8_ID = 2;
UINT16_ID = 3;

META_ID = 4;
ABSTRACT_ID = 5;
ABSTRACT_MAP_ID = 6;
U8UTF8_ID = 7;
NAME_PART_ID = 8;
NAME_ID = 9;
VERSION_ID = 10;
DEFINITION_ID = 11;
IDENTITY_ID = 12;
```

```
EXPRESSION_ID = 13;
REFERENCE_ID = 14;
TAG_ID = 15;
SEQUENCE_ID = 16;
ARRAY_ID = 17;
ENVELOPE_ID = 18;
ENCODED_ID = 19;

FIXED_WIDTH_ID = 20;
FIXED_WIDTH_ATTRIBUTE_ID = 21;
FIXED_WIDTH_ATTRIBUTE_SIZE_ID = 22;
FIXED_WIDTH_ATTRIBUTE_UNSIGNED_ID = 23;
FIXED_WIDTH_ATTRIBUTE_INTEGER_ID = 24;
FIXED_WIDTH_ATTRIBUTE_BIGENDIAN_ID = 25;

DICTIONARY_NAME_ID = 26;
DICTIONARY_DEFINITION_ID = 27;
DICTIONARY_RELATION_ID = 28;
DICTIONARY_LOCATION_ID = 29;

DEFINITION_ENVELOPE_ID = 30;
DICTIONARY_ENTRY_ID = 31;
DICTIONARY_ENTRY_LIST_ID = 32;

META_SIZE = 32;
```

Meta Dictionary Pending Changes

Changes will be held off until Argot 1.4

The following data types are not required to be defined by the meta schema.

- Remove empty from meta schema and define in common dictionary. Empty can be replaced with a meta.sequence with no elements.
- Remove dictionary.name from meta schema.
- Remove meta.identity from meta schema.

Possibly ideas to explore:

- Investigate adding addition fields to dictionary.entry for comments or other semantic information. Information would be imported into Type Library, however, would not be used for any dynamic type agreement/mapping.

Core Meta Dictionary

The Argot 1.3 meta dictionary is 1325 bytes long when encoded. Here is the meta dictionary encoded to a dictionary.entry.list in hex encoding:

```
00 20 00 01 00 1b 01 05 65 6d 70 74 79 01 03 00 09 00 14 00 00 01 00 16 00 00 00 02 00 1b 01
05 75 69 6e 74 38 01 03 00 0f 00 14 00 08 04 00 16 00 08 00 18 00 17 00 19 00 03 00 1b 01 06
75 69 6e 74 31 36 01 03 00 0f 00 14 00 10 04 00 16 00 10 00 18 00 17 00 19 00 04 00 1b 02 04
6d 65 74 61 02 69 64 01 03 00 04 00 0e 00 03 00 05 00 1b 02 04 6d 65 74 61 08 61 62 73 74 72
61 63 74 01 03 00 0d 00 10 01 00 11 00 0e 00 02 00 0e 00 06 00 06 00 1b 02 04 6d 65 74 61 0c
61 62 73 74 72 61 63 74 5f 6d 61 70 01 03 00 0c 00 10 01 00 0f 02 69 64 00 0e 00 04 00 07 00
1b 01 06 75 38 75 74 66 38 01 03 00 12 00 13 00 11 00 0e 00 02 00 0e 00 02 05 55 54 46 2d 38
00 08 00 1b 02 04 6d 65 74 61 09 6e 61 6d 65 5f 70 61 72 74 01 03 00 04 00 0e 00 07 00 09 00
1b 02 04 6d 65 74 61 04 6e 61 6d 65 01 03 00 0a 00 11 00 0e 00 02 00 0e 00 08 00 0a 00 1b 02
04 6d 65 74 61 07 76 65 72 73 69 6f 6e 01 03 00 1b 00 10 02 00 0f 05 6d 61 6a 6f 72 00 0e 00
02 00 0f 05 6d 69 6e 6f 72 00 0e 00 02 00 0b 00 1b 02 04 6d 65 74 61 0a 64 65 66 69 6e 69 74
69 6f 6e 01 03 00 0d 00 05 05 00 14 00 05 00 06 00 0d 00 0c 00 0c 00 1b 02 04 6d 65 74 61 08
69 64 65 6e 74 69 74 79 01 03 00 03 00 10 00 00 0d 00 1b 02 04 6d 65 74 61 0a 65 78 70 72 65
73 73 69 6f 6e 01 03 00 0f 00 05 06 00 0e 00 0f 00 10 00 11 00 12 00 13 00 0e 00 1b 02 04 6d
65 74 61 09 72 65 66 65 72 65 6e 63 65 01 03 00 07 00 10 01 00 0e 00 04 00 0f 00 1b 02 04 6d
65 74 61 03 74 61 67 01 03 00 19 00 10 02 00 0f 04 6e 61 6d 65 00 0e 00 07 00 0f 04 64 61 74
61 00 0e 00 0d 00 10 00 1b 02 04 6d 65 74 61 08 73 65 71 75 65 6e 63 65 01 03 00 0d 00 10 01
00 11 00 0e 00 02 00 0e 00 0d 00 11 00 1b 02 04 6d 65 74 61 05 61 72 72 61 79 01 03 00 19 00
10 02 00 0f 04 73 69 7a 65 00 0e 00 0d 00 0f 04 74 79 70 65 00 0e 00 0d 00 12 00 1b 02 04 6d
65 74 61 07 65 6e 76 65 6c 6f 70 01 03 00 19 00 10 02 00 0f 04 73 69 7a 65 00 0e 00 0d 00 0f
04 74 79 70 65 00 0e 00 0d 00 13 00 1b 02 04 6d 65 74 61 08 65 6e 63 6f 64 69 6e 67 01 03 00
1d 00 10 02 00 0f 04 64 61 74 61 00 0e 00 0d 00 0f 08 65 6e 63 6f 64 69 6e 67 00 0e 00 07 00
14 00 1b 02 04 6d 65 74 61 0b 66 69 78 65 64 5f 77 69 64 74 68 01 03 00 20 00 10 02 00 0f 04
73 69 7a 65 00 0e 00 03 00 0f 05 66 6c 61 67 73 00 11 00 0e 00 02 00 0e 00 15 00 15 00 1b 03
04 6d 65 74 61 0b 66 69 78 65 64 5f 77 69 64 74 68 09 61 74 74 72 69 62 75 74 65 01 03 00 0b
00 05 04 00 16 00 18 00 17 00 19 00 16 00 1b 04 04 6d 65 74 61 0b 66 69 78 65 64 5f 77 69 64
74 68 09 61 74 74 72 69 62 75 74 65 04 73 69 7a 65 01 03 00 0e 00 10 01 00 0f 04 73 69 7a 65
00 0e 00 03 00 17 00 1b 04 04 6d 65 74 61 0b 66 69 78 65 64 5f 77 69 64 74 68 09 61 74 74
69 62 75 74 65 08 75 6e 73 69 67 6e 65 64 01 03 00 03 00 10 00 00 18 00 1b 04 04 6d 65 74 61
0b 66 69 78 65 64 5f 77 69 64 74 68 09 61 74 74 72 69 62 75 74 65 07 69 6e 74 65 67 65 72 01
03 00 03 00 10 00 00 19 00 1b 04 04 6d 65 74 61 0b 66 69 78 65 64 5f 77 69 64 74 68 09 61 74
74 72 69 62 75 74 65 09 62 69 67 65 6e 64 69 61 6e 01 03 00 03 00 10 00 00 1a 00 1b 02 0a 64
69 63 74 69 6f 6e 61 72 79 04 6e 61 6d 65 01 03 00 0e 00 10 01 00 0f 04 6e 61 6d 65 00 0e 00
09 00 1b 00 1b 02 0a 64 69 63 74 69 6f 6e 61 72 79 0a 64 65 66 69 6e 69 74 69 6f 6e 01 03 00
1c 00 10 02 00 0f 04 6e 61 6d 65 00 0e 00 09 00 0f 07 76 65 72 73 69 6f 6e 00 0e 00 0a 00 1c
00 1b 02 0a 64 69 63 74 69 6f 6e 61 72 79 08 72 65 6c 61 74 69 6f 6e 01 03 00 16 00 10 02 00
0f 02 69 64 00 0e 00 04 00 0f 03 74 61 67 00 0e 00 07 00 1d 00 1b 02 0a 64 69 63 74 69 6f 6e
61 72 79 08 6c 6f 63 61 74 69 6f 6e 01 03 00 09 00 05 03 00 1a 00 1b 00 1c 00 1e 00 1b 03 04
6d 65 74 61 0a 64 65 66 69 6e 69 74 69 6f 6e 07 65 6e 76 65 6c 6f 70 01 03 00 0a 00 12 00 0e
00 03 00 0e 00 0b 00 1f 00 1b 02 0a 64 69 63 74 69 6f 6e 61 72 79 05 65 6e 74 72 79 01 03 00
28 00 10 03 00 0f 02 69 64 00 0e 00 03 00 0f 04 6e 61 6d 65 00 0e 00 1d 00 0f 0a 64 65 66 69
6e 69 74 69 6f 6e 00 0e 00 1e 00 20 00 1b 03 0a 64 69 63 74 69 6f 6e 61 72 79 05 65 6e 74 72
79 04 6c 69 73 74 01 03 00 0d 00 10 01 00 11 00 0e 00 03 00 0e 00 1f
```

To conserve space on small embedded devices the meta dictionary may be replaced with a mnemonic value specifying the Argot version.

13. Common Data Type Reference.

The meta dictionary provide the base meta data from which new types can be formed. The following are the common data types Argot supports directly. This is taken directly from the common dictionary.

```
!import meta.fixed_width;
!import meta.fixed_width.attribute;
!import meta.expression;
!import meta.sequence;
!import meta.reference;
!import meta.name;

(library.list [

/*
 * The empty data type does not read any data. Like a NOP for Argot.
 */

(library.entry
  (library.definition u8ascii:"empty" u8ascii:"1.3")
  (meta.fixed_width uint16:0
    [ (meta.fixed_width.attribute.size uint16:0) ]))

/*
 * Unsigned data types (big endian network order).
 */

(library.entry
  (library.definition u8ascii:"uint8" u8ascii:"1.3")
  (meta.fixed_width uint16:8
    [ (meta.fixed_width.attribute.size uint16:8)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.unsigned)
      (meta.fixed_width.attribute.bigendian) ] ))

(library.entry
  (library.definition u8ascii:"uint16" u8ascii:"1.3")
  (meta.fixed_width uint16:16
    [ (meta.fixed_width.attribute.size uint16:16)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.unsigned)
      (meta.fixed_width.attribute.bigendian) ] ))

(library.entry
  (library.definition u8ascii:"uint32" u8ascii:"1.3")
  (meta.fixed_width uint16:32
    [ (meta.fixed_width.attribute.size uint16:32)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.unsigned)
      (meta.fixed_width.attribute.bigendian) ] ))

(library.entry
  (library.definition u8ascii:"uint64" u8ascii:"1.3")
  (meta.fixed_width uint16:64
```

```

    [ (meta.fixed_width.attribute.size uint16:64)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.unsigned)
      (meta.fixed_width.attribute.bigendian) ] ))

/*
 * Signed data types (big endian network order).
 */

(library.entry
  (library.definition u8ascii:"int8" u8ascii:"1.3")
  (meta.fixed_width uint16:8
    [ (meta.fixed_width.attribute.size uint16:8)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.signed)
      (meta.fixed_width.attribute.bigendian) ] ))

(library.entry
  (library.definition u8ascii:"int16" u8ascii:"1.3")
  (meta.fixed_width uint16:16
    [ (meta.fixed_width.attribute.size uint16:16)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.signed)
      (meta.fixed_width.attribute.bigendian) ] ))

(library.entry
  (library.definition u8ascii:"int32" u8ascii:"1.3")
  (meta.fixed_width uint16:32
    [ (meta.fixed_width.attribute.size uint16:32)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.signed)
      (meta.fixed_width.attribute.bigendian) ] ))

(library.entry
  (library.definition u8ascii:"int64" u8ascii:"1.3")
  (meta.fixed_width uint16:64
    [ (meta.fixed_width.attribute.size uint16:64)
      (meta.fixed_width.attribute.integer)
      (meta.fixed_width.attribute.signed)
      (meta.fixed_width.attribute.bigendian) ] ))

/*
 * Floats and Double.
 */

(library.entry
  (library.definition u8ascii:"float" u8ascii:"1.3")
  (meta.fixed_width uint16:32
    [ (meta.fixed_width.attribute.size uint16:32)
      (meta.fixed_width.attribute.IEEE756)
      (meta.fixed_width.attribute.signed) ]))

(library.entry
  (library.definition u8ascii:"double" u8ascii:"1.3")
  (meta.fixed_width uint16:64
    [ (meta.fixed_width.attribute.size uint16:64)
      (meta.fixed_width.attribute.IEEE756)
      (meta.fixed_width.attribute.signed) ]))

/*
 * Boolean value is a byte that can be 0 false.. !0 true.

```

```

*/

(library.entry
  (library.definition u8ascii:"bool" u8ascii:"1.3")
  (meta.reference #uint8))

/*
* An ascii encoded string. Maximum size 255 bytes.
*/

(library.entry
  (library.definition u8ascii:"u8ascii" u8ascii:"1.3")
  (meta.encoding
    (meta.array
      (meta.reference #uint8)
      (meta.reference #uint8))
    u8ascii:"ISO646-US"))

(library.entry
  (library.definition u8ascii:"u8utf8" u8ascii:"1.3")
  (meta.encoding
    (meta.array
      (meta.reference #uint8)
      (meta.reference #uint8))
    u8ascii:"UTF-8"))

/*
* A UTF8 encoded string. Maximum size u32.max bytes.
*/

(library.entry
  (library.definition u8ascii:"u32utf8" u8ascii:"1.3")
  (meta.encoding
    (meta.array
      (meta.reference #uint32)
      (meta.reference #uint8))
    u8ascii:"UTF-8"))

/*
* A binary data block. Maximum size u32.max.
*/

(library.entry
  (library.definition u8ascii:"u32binary" u8ascii:"1.3")
  (meta.array
    (meta.reference #uint32)
    (meta.reference #uint8)))

/*
* A binary data block. Maximum size u16.max
*/

(library.entry
  (library.definition u8ascii:"u16binary" u8ascii:"1.3")
  (meta.array
    (meta.reference #uint16)
    (meta.reference #uint8)))

/*
* Allows any data to be loaded.
*/

```

```

(library.entry
  (library.definition u8ascii:"meta.identified" u8ascii:"1.3")
  (meta.sequence [
    (meta.tag u8ascii:"description" (meta.reference #u8utf8))
  ]))

(library.entry
  (library.relation #meta.expression u8ascii:"1.3" u8ascii:"identified")
  (meta.abstract_map #meta.identified))

/*
 * Date is an abstract type that can be defined using various methods.
 */

(library.entry
  (library.definition u8ascii:"date" u8ascii:"1.3")
  (meta.abstract []))

/*
 * A Java data is the number of milliseconds (or is it seconds) from 1st of
 Janurary 1970.
 */

(library.entry
  (library.definition u8ascii:"date.java" u8ascii:"1.3")
  (meta.sequence [ (meta.reference #int64) ]))

(library.entry
  (library.relation #date u8ascii:"1.3" u8ascii:"java")
  (meta.abstract_map #date.java))

])

```

14. Dictionary & Argot Message File Format

The following provides the dictionary.argot file which specifies the format of an Argot dictionary.

```
!import uint8;
!import u8ascii;
!import dictionary.entry.list;

(library.list [

(library.entry
  (library.definition u8ascii:"dictionary.file" u8ascii:"1.3")
  (meta.sequence [
    (meta.tag u8ascii:"core"
      (meta.array
        (meta.reference #uint8)
        (meta.reference #dictionary.entry.list)))
    (meta.tag u8ascii:"meta"
      (meta.array
        (meta.reference #uint8)
        (meta.reference #dictionary.entry.list)))
    (meta.tag u8ascii:"message"
      (meta.identified u8ascii:"message"))
  ]))
])
```

15. Argot Compiler Syntax Reference.

The Argot Compiler is used to convert text descriptions of Argot data types into their native dictionary format. The Argot Compiler is a temporary measure until a functioning Argot editor is created.

The syntax of the Argot data language is quite simple. The following is taken from the ANTLR grammar file.

```
file : (headers)? expression
      ;

headers: headerline (headers)?
      {
      }
      ;

headerline: import1 | load | reserve | COMMENT
          {
          }
          ;

reserve: '!!! 'reserve'^ IDENTIFIER ';'!!
        ;

load: '!!! 'load'^ QSTRING ';'!!
      ;

import1: '!!! 'import'^ IDENTIFIER ( 'as'! IDENTIFIER)? ';'!!
        ;

expression: '('^ IDENTIFIER (primary)* ')''!
           ;

primary: expression
        | INT^
        | QSTRING^
        | '['^ ( primary)* ']''!
        | '#'^ IDENTIFIER
        | IDENTIFIER^ ':'! value
        | '$'^ IDENTIFIER
        ;

value: INT^
      | QSTRING^
      ;
```