Argot Programmer's Guide
Version 0.4
by David Ryan.

Copyright 2005-2007 © Live Media Pty Ltd.

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 hetrogenous systems.

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 heterogenous distributed systems. It is a perfect alternative to XML where speed, data size, and data integrity are required.

Argot features include:

- A data serialization method which is totally language and system independant.
- 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.
- 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/recieve them. (not available with core package).

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.

Argot's design philosopy is the 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 hetrogenous computer systems share. It is also the most compact and succint form of tranfering information between computers. The information Argot encodes can be sent across any transport medium: sockets, http email, file, message oriented middleware, embedded in XML, or any other place you can transfer data. Argot can fit in the smallest of embedded devices and extend to large hetrogenous corporate systems.

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

Why Argot?

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 Noation 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 interopable.
- 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 21 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 21 meta data statements are called the meta dictionary. It is this these concepts 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.

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 transfering the Argot meta data for each type; allowing the server to confirm that it uses the same data type structure and interface defintions 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 tranfered; 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 contraints and battery power consumption. These systems are also often written using differing enviornments 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.

Evolveable Programming Languages & Byte Code.

A more expiremental 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 disect 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. 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.

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 recently 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 a base 'alphabet' which should not be changed. If we allow modifications to be distributed, we risk companies and individuals 'inovating' 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 for

assistance.

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 dimenstional 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 uaually 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:

u8 : unsigned 8-bit integer.
s8 : signed 8-bit integer using two's complement encoding.
u16: unsigned big-endian 16-bit integer.
s16: signed big-endian 16-bit integer using two's complement encoding.
u32: unsigned big-endian 32-bit integer.
s32: signed big-endian 32-bit integer using two's complement encoding.

u64: unsigned big-endian 64-bit integer.
s64: signed big-endian 64-bit integer using two's complement encoding.

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 u16 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 u8 values, a single u32 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 indivisable.

Argot is able to support other types of basic data types. It is up to us to define these as required.

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 u16 values, we were able to say that the data was a sequence of two u16 values.

To demonstrate a sequence in another way, lets say that our original binary data is a sequence of: one u8, one s8 and one u16. 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 indivudal 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 u8 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 u8 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 transfering 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 informatio 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 differents ways of representing a book identifier; the first is with its catalogue number represented by a u16 (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 conept.

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 inneficient 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 hexidecimal 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.

Argot Meta Data.

The previous 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 dictionary entry is:

```
NAME :  DEFINITION;
```

Each entry in the dictionary has a name and a definition. Each definition is made up of a previously defined name. A simple book definition might be as follows:

```
book: meta.sequence([
            meta.reference( #u8ascii, "ISBN" ),
            meta.reference( #u8ascii, "title"),
            meta.reference( #u8ascii, "description"),
            meta.reference( #u8ascii, "author")
      ]);
```

This book example defines a book as a sequence of ISBN, title, description and author. Each element of the book is stored using a u8ascii type. The u8ascii definition is defined as a common dictionary element. It is defined as:

```
u8ascii: meta.sequence([
            meta.encoding(
                  meta.array(
                        meta.reference( #u8, "size" ),
                        meta.reference( #u8, "data" )
                  ),
                  "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”.

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 u8, s8 and u16 values.

```
mytype: meta.sequence([
      meta.reference( #u8, “value1” ),
      meta.reference( #s8, “value2” ),
      meta.reference( #u16, “value3” )
]);
```

As a meta sequence is a very common declaration, it can be written using a shorter notation with the Argot compiler. This is done using { }, for example:

```
mytype:
{
      meta.reference( #u8, “value1” ),
      meta.reference( #s8, “value2” ),
      meta.reference( #u16, “value3” )
};
```

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:

```
meta.sequence:
{
      meta.array(
            @u8["size"],
            @meta.expression["type"]
      )
};
```

This describes that a sequence is an array of up to 255 elements (u8). Each element is of type meta.expression. This 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.

```
myarray:
{
      meta.array(
            meta.reference( #u32, “size” ),
            meta.reference( #u16, “data” )
      )
};
```

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 u32 value, and then following is three u16 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 two values. The first is the type identifier of the referenced type, and the second is a short ascii string (u8ascii) value to describe how the value will be used.

A meta.reference is defined as:

```
meta.reference:
{
      meta.reference( #u16, "type" ),
      meta.reference( #u8ascii, "name" )
};
```

The meta.reference “type” value is an unsigned 16bit value. This is how other types are refered 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 identifers are not used.

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:

```
mytype:
      {
            meta.reference( #u16, “value” )
      };
```

The type “mytype” is a single #u16 value. If we had a value of 5467 and encoded “mytype” we would produce:

```
0x15 0x5B
```

Like meta.sequence, the meta.reference type is used often in defining data types. It also has a short hand notation. The above “mytype” could also be describes as:

```
mytype:
      {
            @u16[“value”]
      };
```

meta.basic

The meta.basic 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. Some of these are:

```
empty: meta.basic( 0, 0 );

/*
 *  Unsigned data types (big endian network order).
 */
u8:  meta.basic( 8, 0 );
u16: meta.basic( 16, 0 );
u32: meta.basic( 32, 0 );
u64: meta.basic( 64, 0 );

/*
 *  Signed data types (big endian network order).
 */
s8:  meta.basic( 8, 1 );
s16: meta.basic( 16, 1 );
s32: meta.basic( 32, 1 );
s64: meta.basic( 64, 1 );
```

The meta.basic value has two values. The first value is the number of bits the data type uses. The second value is for flags. A basic value can be up to 255 bits in width. The flags provide additional information about the data type. Currently only one bit is used to describe if an integer is signed or unsigned.

The meta.basic type is defined as:

```
meta.basic:
      {
            @u8["size"],
            @u8["flags"]
      };
```

Currently Argot does not describe the flags at the bit level. It also does not describe variable width data types. These may change in a future version of Argot.

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:

```
u8ascii:
      {
            meta.encoding(
                  meta.array(
                        @u8["size"],
                        @u8["data"]
                  ),
                  "ISO646-US"
            )
      };
```

As you can see from the u8ascii definition, the meta.encoding tag is used to add the encoding format to a meta.array using u8 for size and u8 for data. The meta.encoding type is defined as:

```
meta.encoding:
      {
            @meta.expression["data"],
            @u8ascii["encoding"]
      };
```

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.map (choices)

The meta.abstract and meta.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:

```
bookid: meta.abstract();
book.isbn: meta.reference( #u8ascii, "isbn" );
book.catno: meta.reference( #u32, "id" );
bookid#isbn: meta.map( #bookid, #book.isbn );
bookid#catno: meta.map( #bookid, #book.catno );
```

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

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 identifer 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.

There are various ways to describe the concept of choices at a binary level. The default method in Argot uses the meta.abstract concept as it provides a way to extend choices without requiring changes to previous data type definitions. For instance if we wished to provide a new method of representing a “bookid” we create two new data types.

```
/* from dewey decimal system */
book.callno:  meta.reference( #u8ascii, "callno" );
bookid#callno: meta.map( #bookid, #book.callno );
```

This method also allows us to remove mappings we wish to no longer support. The newer types continue to be supported, while the older types would result in an error if found in data.

The name provided to the mapping type names is not required. For example “bookid#callno” could be any name. Using the # symbol is just a convention we have used to clearly identify mapping data types.

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:

meta.identified: meta.reference(#empty, “any”);

No other information is required other than to mark a value as identified. The meta.identified value is encoded in the same way meta.abstract is done. A u16 type id is recorded which specifies the type to follow. The meta.identified places no contraint other than that the type id is a valid type identifier.

Type Naming

The current Argot library uses a flat naming structure for all names. In future proper name spaces may be used. Name spaces have currently been broken up using the . (period) syntax.

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

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.

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

Book List Example

The following example provides a simplistic data dictionary and java implementation. It shows all the required elements for an Argot implementation.

Booklist Data Dictionary

The following provides a very simplistic example of how a list of books could be defined using Argot Data syntax.

```
import u8ascii;
import meta.array;
import meta.reference;
import meta.sequence;
import u16;
import u8;
import meta.expression;
import meta.name;
```

```
/*
 *  A very simple book description.
 */

book:
      {
            @u8ascii["ISBN"],
            @u8ascii["title"],
            @u8ascii["description"],
            @u8ascii["author"]
      };

/*
 *  A list of books.
 */

booklist:
      {
            meta.array(
                  @u8["size"],
                  @book["book"]
            )
      };
```

The bookstore we are developing will use a very simple file format to store all the details of the books we use. 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. It is simply an array of book descriptions using the following description:

```
booklist:
{
     meta.array(
        @u8["size"],
        @book["book"]
     )
};
```

The meta.array data description takes two values. The first value specifies the size of the array, and the second value specifies what type of data is used in the array. The size of the array is an unsigned 8bit. This constrains the array to a maximum of 256 books in the file. For the purposes of an example this is sufficient.

The Argot specification for a book looks like:

```
book:
   {
        @u8ascii["ISBN"],
        @u8ascii["title"],
        @u8ascii["description"],
        @u8ascii["author"]
   };
```

Once again this is an overly simplified example of the structure of a book. The specification describes a book as a sequence of u8ascii elements for ISBN, title, description and author. We are using a short u8ascii description, allowing each field to use a maximum of 256 characters.

The file format description is now complete. Using the Argot compiler we are able to generate a dictionary file which describes these formats in Argot's native binary format.

Compiling Argot Files

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 what is loaded by the Argot library.

AC <file.argot> [-o <file.dictionary>]

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

Java Language Implementation

Now we have defined the definition of a book, its time to write the code that will be used to marshall the book from and to file format.

```
public class BookMarshaller implements TypeWriter
{
   public void write(TypeMimeOutputStream out,
      Object o, TypeElement element)
   throws TypeException, IOException
   {
        Book book = (Book) o;

        out.writeObject("u8ascii", book.getISBN() );
        out.writeObject("u8ascii", book.getTitle() );
        out.writeObject("u8ascii", book.getDescription());
        out.writeObject("u8ascii", book.getAuthor());
    }
}
```

That completes the code required to read and write a book to and from file. Only the write method needed to be implemented. The Argot Java implementation is able to automatically read the book data using the Argot meta descriptions. The Book object will be instantiated using Java reflection. It will find a constructor which takes four String arguments and instatiate the object.

The BooklistMarshaller requries that both read and write be implemented.

```
public class BooklistMarshaller
implements TypeWriter, TypeReader
{
    public void write(TypeMimeOutputStream out,
      Object o, TypeElement element)
    throws TypeException, IOException
    {
        Book[] books = (Book[]) o;

        out.writeObject( "u8", new Integer(books.length));

        for (int x=0; x<books.length; x++)
        {
            out.writeObject("book", books[x]);
        }
    }

    public Object read(TypeMimeInputStream in,
      TypeElement element)
    throws TypeException, IOException
    {
        Short size = (Short) in.readObject( "u8");
        Book[] books = new Book[size.intValue()];
        for(int x=0; x<size.intValue(); x++)
        {
```

```
                books[x] = (Book) in.readObject("book" );
            }
            return books;
        }
    }
```

Notice that in the read method, when reading the size using a u8, that it returns a Short. Java only has signed primatives, so unsigned datatypes get scaled up to a value that can hold the range of values.

With both marshallers complete, we can now start reading and writing our file format.

```
public Book[] loadBooks( TypeLibrary library, String filename)
throws IOException, FileNotFoundException, TypeException
{
   FileInputStream inputStream = new FileInputStream( filename );
   TypeMap map = new TypeMap( library );
   map.map( 1, library.getId( "book" ) );
   map.map( 2, library.getId( "booklist" ));
   map.map( 3, library.getId( "u8ascii"));
   map.map( 4, library.getId( "u8"));

   TypeMimeInputStream typeInputStream =
        new TypeMimeInputStream( inputStream, map );
   return (Book[]) typeInputStream.readObject( "booklist" );
}
```

And to write books to a file, our method will look like:

```
public void saveBooks( TypeLibrary library,
      String filename, Book[] books )
throws IOException, FileNotFoundException, TypeException
{
   FileOutputStream outputStream = new FileOutputStream( filename );
   TypeMap map = new TypeMap( library );
   map.map( 1, library.getId( "book" ) );
   map.map( 2, library.getId( "booklist" ));
   map.map( 3, library.getId( "u8ascii"));
   map.map( 4, library.getId( "u8"));

   TypeMimeOutputStream typeInputStream =
        new TypeMimeOutputStream( outputStream, map );
   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.

Before we can use these methods, we need to load our dictionary into the Argot runtime system and bind the marshallers we defined to the matching java environment objects.

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

      return TypeLibrary( libraryLoaders );
}
```

We create a TypeLibrary by passing in an array of loaders. A loader will read a dictionary file and bind the languages data types to the dictionary data types. The MetaLoader is

used for the meta data types. The DictionaryLoader binds the format of dictionary files. The CommonLoader binds common data types. And finally the BookstoreLoader binds the data types used in the bookstore dictionary. For example the BookstoreLoader binds the following:

```
public void bind( TypeLibrary library )
throws TypeException
{
      library.bind( BookArgot.TYPENAME, new TypeReaderAuto( Book.class ),
            new BookArgot(), Book.class );
      library.bind( BookArrayArgot.TYPENAME, new BookArrayArgot(),
            new BookArrayArgot(), Book[].class );
}
```

We now have a complete Argot solution for reading and writing the booklist to a file format which is completely system independant. The format we used contains just the array of books as we defined.

The complete bookstore application is included in the Argot download.

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 addded 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 regsitered. 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 transfering. 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 is formed dynamically between the two participants communicating. The two participants can be either in client/server interactions or between an application and file. The agrement 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(#u32 , “catno”);
35	bookid#catno	meta.map(#bookid, #book.catno);

36	booklist	meta.array(meta.reference(#u8, “size”),
		meta.reference(#bookid, “bookids”));

Note: *All other data types refernced 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(#u32 , “catno”);
43	bookid#catno	meta.map(#bookid, #book.catno);
44	book.isbn	meta.reference(#u8ascii, “isbn”);
45	bookid#isbn	meta.map(#bookid, #book.isbn);

46	booklist	meta.array(meta.reference(#u8, “size”),
		meta.reference(#bookid, “bookids”));

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 Definінition 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.

Note: *The base Argot library does not provide the Argot Network Resolution protocol for remote systems. This will be released as a seperate module in the future.*

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:

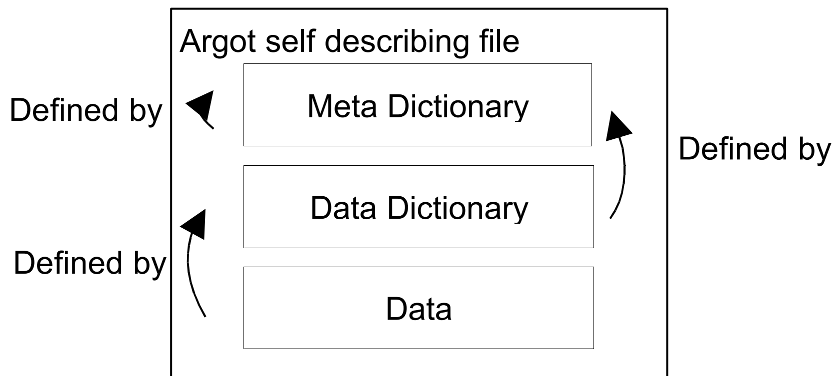
First System		Argot File.
Type ID	Map ID	Type ID
33 (bookid)	35	35(bookid)
34 (book.catno)	36	36 (book.catno)
35 (bookid#catno)	37	37 (bookid#catno)
36 (booklist)	38	38 (booklist)

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.

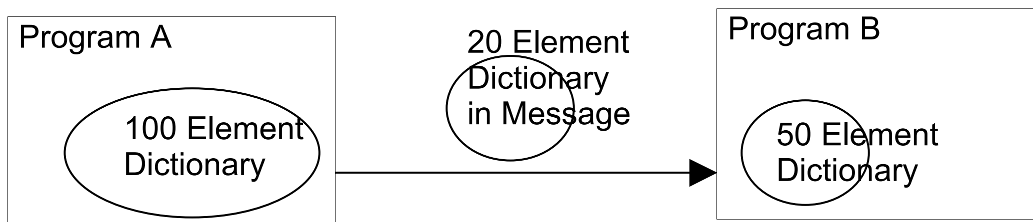
Argot Message Files & Dictionaries

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, the data. Argot dictionary files use this format to store dictionary definitions.

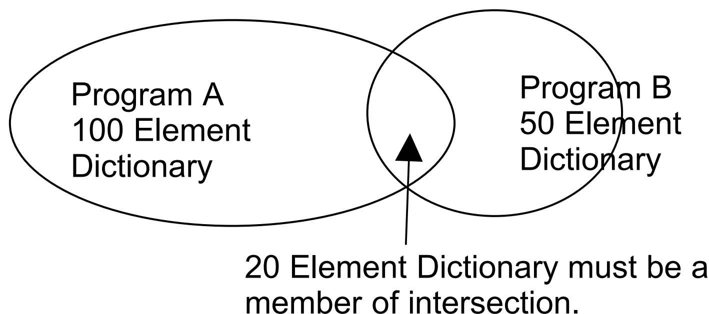
The Argot Message format allows the full specification of the data to be transferred with the data. This requires no external definition of the data. For an application to be able to read the file its type library must contain all the data types used in the file. A Type Map is generated from the data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

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 clss);***
>
> 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 clss);

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 clss);

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);

To construct a TypeMap a specific TypeLibrary must be supplied. This can 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 getId(String name);

Returns the external id of the selected type name. An exception will be thrown if the name is not registered in the type library or the name has not been mapped in the type map.

String getName(int id);

Returns the type name given an external mapped id. An exception will be thrown if the id is not mapped in the type map.

TypeReader getReader(int id);

Returns the type reader given an external mapped id. An exception will be thrown if the id is not mapped in the type map.

TypeElement getStructure(int id);

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

TypeWriter getWriter(int id);

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

Class getClass(int id);

Returns the environment class given an external mapped id. An exception will be thrown if the id is not mapped. Only available on environments that support object reflection.

int getId(Class clss);

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 getSystemId(int id);

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

int getId(int systemId);

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 id);

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.

DynamicTypeMap & ReferenceTypeMap

Argot comes with three different Type Maps. 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. Dynamic Type Maps and Reference Type Maps provide additional features required for the Argot Message Files and Dynamic Data agreement between client and server.

The DynamicTypeMap 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 ReferenceTypeMap 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 interfer with the TypeMap that the data dictionary is refering. This is used when reading and writing information in the Argot Message Format.

TypeInputStream

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

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 outStream, 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, TypeElement element);***
>
> Each implementation of TypeReader can read directly from the InputStream. The actual definition of the element is provided by the element variable. 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, TypeElement element, Object o);***
>
> Each implementation of TypeWriter can write directly to the OutputStream. The actual definition of the element is provided by the element variable. 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.

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.

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 meta dictionary. From the meta dictionary elements all other types are created. The following provides the definitions of each type in the meta dictionary:

empty: meta.basic(0, 0);

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.

u8: meta.basic(8, 0);

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

u16: meta.basic(16, 0);

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

```
meta.basic:
	{
		@u8["size"],
		@u8["flags"]
	};
```

This defines how the meta data associated with a basic data definition is serialized. Two values of unsigned 8 bit values. The first value is the size if available and the second value is additional flags. Basic data types are atomic values which require specific function to read the values.

```
meta.name:
	{
		meta.encoding(
			meta.array(
				@u8["size"],
				@u8["data"]
			),
			"ISO646-US"
		)
	};
```

A name is a simple ASCII encoded string of maxiumum length 255 characters. This is used to describe the names of each type in the Argot.

```
meta.abstract:
	{
		@empty["abstract"]
	};
```

A type defined as abstract has an empty definition. 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.

```
meta.map:
	{
		@u16["abstract"],
		@u16["concrete"]
	};
```

A map is used to map an abstract data type to a concrete data type. Two u16 values are used to specify identifiers of the data types in the dictionary being mapped.

```
meta.expression: meta.abstract();
```

An expression is an abstract type. It allows different expressions to be used to define data types. New expressions can be added to expand Argot's specification language.

```
meta.encoding:
	{
		@meta.expression["data"],
		@meta.name["encoding"]
	};
```

Encoding specifies the data encoding used on a character string. The data expression must return an array that can have encoding applied.

```
meta.reference:
	{
		@u16["type"],
		@meta.name["name"]
	};
```

A reference delcares a usage of another data type in the system. The name data type is used to define a description of usage of that data type. A reference is defined using the @ type "[" name "]" syntax in the definition given.

```
meta.sequence:
	{
		meta.array(
			@u8["size"],
			@meta.expression["type"]
		)
	};
```

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. A sequence is defined as "{" "}" in the grammar defined to describe argot definitions.

```
meta.array:
	{
		@meta.expression["size"],
		@meta.expression["type"]
	};
```

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.

```
meta.expression#reference: meta.map( #meta.expression, #meta.reference );
meta.expression#sequence: meta.map( #meta.expression, #meta.sequence );
meta.expression#encoding: meta.map( #meta.expression, #meta.encoding );
```

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

```
meta.definition: meta.abstract();
```

The definition of a type is abstract. Specific concrete types are mapped to the abstract type.

meta.definition#basic: meta.map(#meta.definition, #meta.basic);
meta.definition#sequence: meta.map(#meta.definition, #meta.sequence);
meta.definition#map: meta.map(#meta.definition, #meta.map);
meta.definition#abstract: meta.map(#meta.definition, #meta.abstract);

A definition is defined as being basic, sequence, map or abstract.

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 = 1
u8 = 2
u16 = 3
meta.basic = 4
meta.abstract = 5
meta.map = 6
meta.expression = 7
meta.sequence = 8
meta.reference = 9
meta.name = 10
meta.encoding = 11
meta.array = 12
meta.expression#reference = 13
meta.expression#sequence = 14
meta.expression#array = 15
meta.expression#encoding = 16
meta.definition = 17
meta.definition#basic = 18
meta.definition#map = 19
meta.definition#sequence = 20
meta.definition#abstract = 21
```

Meta Dictionary Pending Changes

We are still attempting to improve the meta dictionary to ensure that base concepts we use are flexible and open to change. There are currently some changes to the meta dictionary pending to be added in the next release. The new meta dictionary may looking something like:

```
empty: meta.basic( 0, 0 );

uint8: meta.basic( 8, 0 );

uint16: meta.basic( 16, 0 );

meta.id: meta.reference( #uint16 );

meta.basic: meta.sequence
        ([
                meta.tag( “size”, meta.reference( #uint8 ) ),
                meta.tag( “flags”, meta.reference( #uint8 ) )
        ]);

meta.abstract: meta.reference( #empty );

meta.map: meta.sequence
```

```
        ([
                meta.tag( “abstract”, meta.reference( #meta.id )),
                meta.tag( “concrete”, meta.reference( #meta.id ))
        ]);
meta.expression: meta.reference( #abstract );
meta.sequence: meta.array(
                meta.reference( #uint8 ),
                meta.reference( #meta.expression )
        );
meta.array: meta.sequence([
                meta.tag( “size”, meta.reference( #meta.expression ),
                meta.tag( “data”, meta.reference( #meta.expression ),
        );
meta.reference: meta.reference( #meta.id );
meta.tag: meta.sequence([
                meta.tag( “name”, meta.reference( #u8ascii ),
                meta.tag( “data”, meta.reference( #meta.expression )
        ]);
u8ascii: meta.encoding(
                meta.array(
                        meta.reference( #uint8 ),
                        meta.reference( #uint8 )
                ),
                “ISO646-US”
        );
meta.encoding: meta.sequence([
                meta.tag( “data”, meta.reference( #meta.expression )),
                meta.tag( “encoding”, meta.reference( #u8ascii ))
        ]);
meta.expression#reference: meta.map( #meta.expression, #meta.reference );
meta.expression#sequence: meta.map( #meta.expression, #meta.sequence );
meta.expression#array: meta.map( #meta.expression, #meta.array );
meta.expression#tag: meta.map( #meta.expression, #meta.tag );
meta.expression#encoding: meta.map( #meta.expression, #meta.encoding );
meta.definition: meta.abstract();
meta.definition#basic: meta.map( #meta.definition, #meta.basic );
meta.definiton#map: meta.map( #meta.definition, #meta.map );
meta.definition#abstract: meta.map( #meta.definition, #meta.abstract );
meta.definition#expression: meta.map( #meta.definition, #meta.expresison );
```

The following changes have been made to the meta dictionary in this instance:

1. Abstract types could previously only be mapped to concrete types. This version allows abstract types to be mapped to other abstract types. This allows meta.definition to be mapped to meta.expression. Previously all definitions had to be wrapped using meta.sequence. This changes no longer requires this.

2. Meta.reference has been seperated into meta.reference and meta.tag. This allows the tag name to be only used where necessary in meta data. This has an effect on most data types.
3. Meta.name renamed to its more correct name of u8ascii.
4. U8 and u16 have been renamed to uint8 and uint16 to better describe the data types.

Before finalising these changes to the meta dictionary we expect to update the meta.basic type. The use of size and flags does not adequately describe or provide enough flexiblity for basic data types. We also expect to introduce a more formal method of specifying name spaces for data types. This may result in a change to the argot message format.

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.basic;
import meta.expression;
import meta.sequence;
import meta.reference;
import meta.name;

/*
 *  The empty data type does not read any data. Like a NOP for Argot.
 */
empty: meta.basic( 0, 0 );

/*
 *  Unsigned data types (big endian network order).
 */
u8:  meta.basic( 8, 0 );
u16: meta.basic( 16, 0 );
u32: meta.basic( 32, 0 );
u64: meta.basic( 64, 0 );

/*
 *  Signed data types (big endian network order).
 */
s8:  meta.basic( 8, 1 );
s16: meta.basic( 16, 1 );
s32: meta.basic( 32, 1 );
s64: meta.basic( 64, 1 );

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

bool: {
                  @u8[ "bool" ]
      };


/*
 *  A UTF8 encoded string.  Maximum size 255 bytes.
 */
u8ascii: {
                  meta.encoding(
                        meta.array(
                              @u8[ "size" ],
                              @u8[ "data" ]
```

```
                        ),
                        "ISO646-US"
                  )
            };

/*
 *  A UTF8 encoded string.  Maximum size u32.max bytes.
 */
u32utf8: {
                  meta.encoding(
                        meta.array(
                              @u32[ "size" ],
                              @u8[ "data"]
                        ),
                        "UTF8"
                  )
            };


/*
 *  A binary data block.  Maximum size u32.max.
 */
u32binary: {
                  meta.array(
                        @u32["size"],
                        @u8[ "data" ]
                  )
            };

/*
 * A binary data block.  Maximum size u16.max
 */
u16binary: {
                  meta.array(
                        @u16["size"],
                        @u8[ "data" ]
                  )
            };

/*
 * Allows any data to be loaded.
 */
meta.identified: {
                  @meta.name["description"]
            };

meta.expression#identified: meta.map( #meta.expression, #meta.identified )
;

date: meta.abstract();

/*
 *  A Java date is the number of milliseconds (or is it seconds) from 1st
of Janurary 1970.
 */
date.java: {
      @s64["date"]
};

date#java: meta.map( #date, #date.java );
```

Dictionary & Argot Message File Format

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

dictionary.

```
import u8;
import u16;

import meta.name;
import meta.expression;
import meta.definition;
import meta.identified;

/* An envelop is an array of bytes specified with a size.
   The type is the data that the envelop contains.
*/

meta.envelop:
      {
            @meta.expression["size"],
            @meta.expression["type"]
      };

/*
 Make an evelop available as a base definition.
*/
meta.definition#envelop: meta.map( #meta.definition, #meta.envelop );

/* Dictionary definition.
*/
dictionary.definition:
      meta.envelop(
            @u16["size"],
            @meta.definition["definition"]
      );

dictionary.entry:
      {
            @u16["id"],
            @meta.name["name"],
            @dictionary.definition["definition"]
      };


dictionary.map:
      {
            meta.array(
                  @u16["size"],
                  @dictionary.entry["word"]
            )
      };

dictionary.words:
      meta.envelop(
            @u16["size"],
            @dictionary.map["words"]
      );

dictionary.file:
      {
            meta.array(
                  @u8["core"],
                @dictionary.map[ "words"]
            ),
            meta.array(
                  @u8["message"],
                  @dictionary.map["words"]
            ),
            @meta.identified["message"]
      };
```

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: statementlist
      ;

statementlist: statement (statementlist)?
      ;

statement: import | load | entry | reserve | COMMENT
      ;

import: IMPORT IDENTIFIER SEMI
      ;

load: LOAD QSTRING SEMI
      ;

reserve: RESERVE IDENTIFIER SEMI
      ;

entry :  IDENTIFIER COLON specification SEMI
      ;

specification : expression
      ;

expression
      : IDENTIFIER LBRACE primarylist RBRACE
      | LCBRACE primarylist RCBRACE
      ;

primarylist : ( primary ( COMA primary )* )?
      ;

primary : expression
      | INT
      | QSTRING
      | SLBRACK primary ( COMA primary )* SRBRACK
      | HASH IDENTIFIER
      | ATSYM IDENTIFIER SLBRACK QSTRING SRBRACK
      ;



tokens
{
      IMPORT="import";
      RESERVE="reserve";
      LOAD="load";
}

DQUOTE: '"';
PIPE: '|';
COMA: ',';
LBRACK: '<';
```

```
RBRACK: '>';
LBRACE: '(';
RBRACE: ')';
LCBRACE: '{';
RCBRACE: '}';
ATSYM: '@';
HASH: '#';
COLON : ':';
SEMI: ';';
SLBRACK: '[';
SRBRACK : ']';
PERIOD: '.';

DIGIT: '0'..'9';
INT : (DIGIT)+;
QSTRING:    '"'! (.)* '"'!;

IDENTIFIER: ('a'..'z'|'A'..'Z')
('a'..'z'|'A'..'Z'|'0'..'9'|'.'|'_'|'#')*;

COMMENT: "/*"
                ( '*'
                | "\r\n"
                | ( '\r' | '\n' )
                | ~( '*'| '\r' | '\n' )
                )*
                '*' '/';

WS:   (' ' | '\t' | '\n' | '\r' )+;
```