A Case Study for Developing a C Based Module in Pike

Andreas Finnman

February 26, 2003

Department of Computer and
Information Science (IDA)
University of Linkoping

Abstract

The Pike C module interfaces were evaluated by implementing a Bzip2 compression module twice. One implementation was done using the old C module API and one was done using the newer CMOD API. A comparison to Java (JNI) was also done in order to get an idea of how convenient the use of C code is in Pike. The results of the implementations of the Bzip2 module led to a thorough evaluation on developing C based modules in Pike. This in its turn yielded written information on how to extend Pike with C based modules.

Contents

1 Introduction **1**
1.1 Purpose and Problem Definition 1
1.1.1 In General 1
1.1.2 Introduction 1
1.1.3 The Bz2 Module 2
1.1.4 Evaluation 2
1.1.5 Appendix A - Tutorials 2
1.2 Approach 2
1.3 Delimitations 3
1.4 Reading Instructions 3
1.5 Pike in Abstract 4
1.5.1 What is Pike? 4
1.5.2 Pike Compared to Other Languages 4
1.5.3 Pike Programs and the Program Pike 5
1.5.4 Reasons for Using Pike 5
1.6 Extendibility 6

2 The Bz2 Module **9**
2.1 Background 9
2.1.1 Data Compression 9
2.1.2 The Program Bzip2 and the Library libbzip2 11
2.1.3 The Compression Algorithm 11
2.1.4 The libbzip2 API 12
2.1.5 Bz2 Compression in Pike Based on libbzip2 13
2.2 Requirements 13
2.2.1 Requirements Overview 13
2.2.2 List of Requirements 13
2.3 Architecture 14

2.3.1 Design Overview 14
2.3.2 Detailed Design Description 16
2.3.3 From Design to Implementation 21
2.3.4 Implementation Using the C Interface 23
2.3.5 Implementation Using the CMOD Interface 24
2.3.6 Description of Some C Level Functions 25
2.3.7 Test 28

3 Evaluation of the Pike C Module APIs **32**
3.1 Overview 32
3.2 The two Interfaces 32
3.3 CMOD and the Old C Interface 33
3.4 The JNI Interface 35
3.4.1 What is the JNI? 35
3.4.2 How is the JNI Used? 35
3.5 Comparative Evaluation 36
3.5.1 Overview 36
3.5.2 Functionality 38
3.5.3 Performance/Efficiency 38
3.5.4 Platform Dependency 39
3.5.5 Getting Started 39
3.5.6 Programming with the Interface 40
3.5.7 Argument Handling 40
3.5.8 Readability of Code 41
3.5.9 Summary of Comparative Evaluation 42

4 Results and Future Work **44**
4.1 Results 44
4.2 Future Work 45
4.3 Acknowledgements 46

Glossary **47**

Bibliography **49**

Appendix A **A–1**

Appendix B **B–1**

Appendix C C–1

Appendix D D–1

Chapter 1

Introduction

1.1 Purpose and Problem Definition

1.1.1 In General

The goal of this master's thesis was to evaluate the Pike C module system by implementing a Bzip2 module. The whole master's thesis project did not focus on one specific task. The idea behind the project was to have a closer look at the Pike module system and make some evaluation of it. At least one implementation of a new module for Pike was also to be done. This implementation was then to be evaluated from a more general point of view, in order to see what could be found and brought to paper regarding the Pike C module system.

The report consists of four major chapters; this introduction, a chapter on the Bz2 module, a chapter containing evaluation of the Pike C module APIs, and a chapter that summarizes the results of the case study. The report also has an extensive appendix. Appendix A contains tutorials on how to extend Pike with C modules. The purpose of each part of the report is described closer below.

1.1.2 Introduction

The 'introduction' chapter aims to provide the reader with the background information necessary to benefit from the report. It will state the problem definition and the approach that was used to solve it. The introduction also

aims to make the reader aware of the topics that are discussed in the report and thus give the reader an overview of it.

1.1.3 The Bz2 Module

The purpose of this chapter is to document the part of the work that concerned the Bz2 module. Bz2 is a module to handle Bz(ip)2 compression and decompression. The purpose of making the Bz2 module was to provide Pike with Bz2 support, but it was also to serve as the special case of development of a C based module, that this entire case study was based on.

1.1.4 Evaluation

The purpose of this chapter is to give the reader an evaluation of the two Pike C module APIs. The two APIs are to be compared to each other so a reader can get an idea of their differences. Furthermore the section will contain a comparison of the Pike C APIs and the JNI. That way the reader can get an idea of the differences and similarities between Pike and Java, and also get an idea of the strengths and weaknesses of Pike and Java respectively, in the aspect of extending them with C code.

1.1.5 Appendix A - Tutorials

The purpose of this appendix is to give the reader information on how to work with the Pike C module interfaces. This is done on the basis of the Bz2 module implementations.

1.2 Approach

The approach used to reach to the results was a case study of writing a C module in Pike. The special case that served as the foundation was the development of the Bz2 module. The new module was implemented twice, the first implementation was done using the traditional C interface. The second implementation was done using the newer and more high level CMOD interface. During this development notes on interesting topics were taken continuously and then later on put together to this report.

1.3 Delimitations

In order to fit the limited amount of time, the following delimitations were added to the project:

- The project focused on documentation and evaluation of the Pike C module APIs. The finding of bugs, doing optimization of code, etc in the Bz2 module was therefore only done to the extent that time allowed.

- The documentation of the development process of the Bz2 module was done briefly, since this was not the main goal with the project.

- Since the project was of the type case study, it treats only a special case. The covering of the Pike C module system was thus not exhaustive in any way. That is why project was not to result in complete reference manuals of the C module APIs.

1.4 Reading Instructions

There are mainly two kinds of intended readers that this report is written for; readers interested in writing modules for Pike, and readers interested in the Bz2 module. The report is written in a way that minimizes the knowledge prerequisites of the reader. It is however assumed that the reader has some fundamental knowledge of programming. Further it is assumed that the reader has some general knowledge of computer systems as well as the terminology. At the very end of this report a glossary can be found where the most subject specific terms are explained. If a word is not recognized, or its meaning is not distinct then there might be an explanation or a definition of it in the glossary.

Appendix A is the tutorial part of the report. Although it is placed in the appendix, it is a very central part of the results of this project. The tutorials are put in an appendix because they are separate documents and is therefore hard to fit to the report. The other appendices are the documents that were written during the development of the Bz2 module. Most of their content is also found in the Bz2 chapter of the report. Some of it is left out though, and that is why these documents (Appendix B, C, and D) were appended.

Even though attempts have been done to keep the text of this entire document as much like one report as possible, everything in later sections is not always directly consequences, from earlier sections. Since the sections cover rather different topics that is natural. It is recommended to read the whole report, but there are some shortcuts that can be taken:

- Even though the section on Bz2 illustrates a good example of a module implementation, there is no direct need to read it in order to benefit from the 'tutorial' or the 'evaluation' part.

- The subsection on implementation in the chapter on Bz2 has some rather detailed technical content that does not necessarily need to be read.

- The chapter on evaluation is an analysis of the current Pike C module system and does not include any further information on how to write modules. It can therefore be skipped by readers not interested in this analysis.

1.5 Pike in Abstract

1.5.1 What is Pike?

Pike is a modern general purpose high level programming language, originally developed by Roxen Internet Software AB. It is currently maintained by the Department of Computer and Information Science at Linkoping University.

1.5.2 Pike Compared to Other Languages

Pike is a modern high level programming language. It is in many aspects similar to C, C++, and Java. For instance is the Pike syntax similar to the C syntax. Methods in Pike are called in the same way as functions are called in C. C and Pike share many arithmetic operators such as '+', '-', '*', '/' and '%'. Pike and C also share many logical and comparison operators such as '==', '!=', '&&', '||', and so on. Since Pike is totally based on C, the list of similarities between C and Pike can be made long. Therefore it is probably more interesting to have a look at some of the differences between Pike and C.[4] [5]

Pike has a lot of nice features such as automatic memory management with garbage collection. Due to that there is no reason for a programmer to use pointers the way they are used in C. In this aspect Pike is, from a Pike programmer's point of view a lot like Java. The way that Objects are created, accessed, and destroyed in Pike is also similar to the way it is done in Java.

1.5.3 Pike Programs and the Program Pike

Pike is an interpreted language. That means that the Pike compiler compiles Pike code to an intermediate format called byte code upon program execution. This byte code is then interpreted by the Pike virtual machine. In order to run Pike programs on a workstation, a Pike compiler and a Pike virtual machine of course need to be installed. The process of compiling and installing these utilities is in common speech referred to as 'to build a Pike'.

In other words, 'to build a Pike' actually means to make an installation of the program Pike. The program Pike then runs Pike programs (programs written in the language Pike). Run in this case means to process the source files of the program so that it is compiled to byte code and then executed by the virtual machine.

1.5.4 Reasons for Using Pike

The main reason for using Pike is that it has all the features that a powerful modern programming language should have. It is object oriented, fast, portable, extendible, and free of charge. According to Pike developers and Pike users, Pike has a lot of advantages compared to many other programming languages. [4]

One of them is that Pike supports multiple inheritance. Multiple inheritance means that a class can inherit more than just one class. Java for instance does not support that.

Pike is also a true high level language. That means that it has high level features that other languages (such as C++) might lack. The garbage collection in Pike is a good example of that. Pike has full garbage collection, so that the Pike programmer can create objects without having to worry about issues like memory leaks.

The possibility to handle strings in a dynamic way and use all kinds of operations on them is another good example of the Pike high level features.

Pike has a lot of operations on strings that simplify programming a lot. Since Pike has all these high level features it allows a less rigid programming style than for instance C++. [6]

Pike is also portable. Nowadays Pike compiles on almost every flavor of Unix and also on MAC/OS, Amiga OS, and Windows.[6] During the writing of this report the Pikefarm project was made public[1]. The Pikefarm project is about building Pikes on different software and hardware architectures and monitor it. That way changes and improvements in the field of platform independence can be done continuously while Pike develops.

Last but not least, Pike is fast. To compare the execution times of programs written in C++ to the ones of programs written in Pike is not that interesting since C++ is a compiled language and Pike is interpreted. A more relevant comparison would be to compare Pike to for instance Java which is interpreted/JIT compiled. Compared to Java, Pike has proven to be remarkably faster in many applications. [11]

1.6 Extendibility

In order to make programming languages more powerful it is common that they are extended with modules that add certain functionality to them. Normally, the functionality in a module is of such kind that it is used only in applications of a certain kind and therefore should not be on the top level. For instance would all programs that do not handle images of course never use the functionality of an image handling module. That is also why image handling, in Pike 7.5, is in a module. One advantage of putting odd functionality in modules rather than on the top level is that the user can choose what modules he/she wants to have in his/her Pike upon building it. Thus the user can shrink his/her Pike to fit his/her needs better.

As mentioned in the section above the extendibility of Pike is one of the main advantages of the language. A module for Pike can be written in either Pike or C.

Today, there are two APIs for writing C modules in Pike. In this report they are referred to as the 'C interface' and the 'CMOD interface'. The C interface is the oldest of the two C interfaces. The newer CMOD is an attempt to build a more intuitive API and thus facilitate development of new modules. Actually, CMOD is no interface of its own. It is a wrapper around the old C

[1] http://pike.ida.liu.se/development/pikefarm/

interface. CMOD code, when it is compiled, is first translated to plain C by a precompilation program before it is compiled. Further descriptions of this can be found in chapter 4.

Writing the module in Pike is probably the easiest way to write a module. The programmer can then take advantage of the whole Pike language and write the whole module on a higher level.

The other way to extend Pike is, as stated earlier, to write modules for it in C. Even though it might be easier to write the module in Pike, writing the module in C still has at least the following advantages:

1. **Convenience:** Many programmers have C as there 'native language' and therefore find it convenient to program in C

2. **Performance:** Since C is on a low level it allows the programmer to write fast programs.

3. **Useful libraries:** C has a lot of useful libraries that the programmer can take advantage of if he/she chooses to use C

Because of those reasons many modules in Pike 7.5 are C modules. Especially point 2 and 3 need to be considered carefully. Point 2, performance, is of course always an issue since programs written in Pike never will perform better than the Pike that runs them makes them perform. In other words, if a module can be implemented in both C and Pike, and the C implementation will be remarkably faster than the Pike implementation, then a C implementation ought to be done.

Using C libraries, as suggested in point 3, can save the programmer a lot of work. Taking advantage of work done by others prevents the programmer from spending a lot of time reinventing the wheel. Good modules for Pike can be written by simply writing a C module that implements a Pike level interface. That module then uses C libraries that have the functionality that the C module is to provide Pike with. The 'only' task of the C module itself is to implement a high level interface for the underlying C library. This is in common speech referred to as 'implementing a glue' between Pike and that C library. Implementing a glue can be done more or less thoroughly. The easiest way of doing it is to just map the methods in the module to a function in the C library. However, that is not a good way of making a module for Pike. The glue should of course implement the common Pike high level features. That is, the methods in the module should be called in the

same way as any other Pike method and it should also take care of memory management, and error handling. In other words, new modules ought to be just as high level as the rest of Pike.

Chapter 2

The Bz2 Module

2.1 Background

2.1.1 Data Compression

This section gives a little background of data compression so the reader can understand the functionality of the Bz2 module. How compression algorithms work and how they are implemented, lies outside the scope of this report.

In order to shrink data, it can be compressed using a compression algorithm. Nowadays different kinds of data compression is used frequently. There are endless utilities that use all varieties of compression algorithms in order to achieve some goal. The first differentiation one can make among these algorithms is to divide them into 'lossy' and 'lossless' algorithms. As one can probably tell by the name, lossy algorithms may lead to some loss of data. That means that the original data might not be recreated from the compressed data generated by a lossy algorithm. The advantage of these algorithms is that their compressed output is much smaller than the corresponding output of a lossless compression algorithm. Lossy algorithms are well suited for image and audio compression where the noise level is high, and the differences between the recreation and the original only need to be as small as not to be detected by human eyes and ears. Since the human senses are rather limited, a recreated compressed image or sound will be interpreted as its original even if the loss of data has been rather large. A computer on the other hand of course interprets the recreation different than its original even if the loss is as small as one single bit. Therefore, when compressing for instance executable files, algorithms that shrink the original data without

loss, also known as lossless algorithms, are used. [8]

Compressing data is a good way to save storage space, when storing data, and a good way to save bandwidth when transmitting data. When talking about the usage of data compression, it is not so bad to think of an inflatable/deflatable life boat as an analogy. If we compare our data to an inflatable life boat we can say the following. The minimum size of our data is when we have taken out all the unnecessary 'air' of it, i.e. deflated it, or in other words compressed it. Both the life boat and the data take a lot of unnecessary space if we store it without letting out the air. When we move our life boat around, it is kept deflated because that makes the transport go easier. But as soon as we want to use it, we will have to inflate it.

That analogy illustrates how data compression should be used. Data that is mainly stored and transmitted and not that often used is well suited for compression. Compression is a good tool in all cases where storage space and bandwidth is of much greater importance than processing time and memory consumption. [9]

There are lots of different compression programs such as PKZIP,Gzip and Bzip2. They all use different algorithm and have different implementations, but they all depend on this fundamental fact:

On the lowest level, data is stored in zeros and ones. In order to store data in strings of bits, data needs to be coded in some kind of way. ASCII characters is probably the easiest example of this. Every character gets a unique string of 8 bits. If we then have a text file containing ASCII characters we know for sure that some characters will appear a lot more frequent than others. If the text file for instance contains some normal English text, then the characters e, r, s will be a lot more frequent than for example q, j, and d. Some characters are thus more likely to appear than others. That is why the file can shrink if we change the coding. The frequent characters are then coded in strings with less than 8 bits and infrequent characters are coded in strings longer than 8 bits. That way, if the recoding is done in a smart way, the average amount of bits that a character in the file is coded with, will be less than 8.

In general this means that compression can be done on all kinds of data, no matter what kind of data it is, since all data represents a coding of something. So compression algorithms in general aim to find ways to code frequent bit patterns with less bits than they are coded with originally. In order for that to work some bit patterns, infrequent ones, need to be coded with more bits than they originally were.[8]

2.1.2 The Program Bzip2 and the Library libbzip2

Bzip2 is a program for data compression [9]. It is very similar to the more common Gzip program. As a matter of fact, in a Unix environment, Bzip2 is used exactly in the same way as Gzip. The main difference between them is that they use different algorithms to compress data. Bzip2 and libbzip2 use an algorithm called the Burrows-Wheeler algorithm combined with Huffman coding for data compression [7]. This is referred to as Bz2 compression in this report. The algorithm used in Bzip2 is slower than the one used in Gzip, but it compresses better[9].

Just as Gzip, Bzip2 has its own file format. These files usually have the suffix '.bz2' (compare to Gzip files '*.gz'). Libbzip2 is the Bzip2 C programming library.

2.1.3 The Compression Algorithm

The algorithm that libbzip2 uses is a lossless compression algorithm based on block sorting. This means that by sorting the data in a block in a certain way the block can be rearranged to a set of data which is suited for compression. In order to achieve this a transformation, originally discovered by David Wheeler in 1983, is used. Such transformation and retransformation however is based on a lot of heavy mathematical theory and is beyond the scope of this paper. [7]

The important thing to know about the algorithm in order to use libbzip2 and Bzip2 properly, is what the block sorting thing actually means. The user needs to be aware of that the block size that he/she is using in his/her programs, is the size of the blocks that data is being sorted in before it is compressed.

Sorting in bigger blocks is of course more memory and processing time consuming than sorting in smaller blocks. On the other hand, the bigger the blocks are, the better sorted is the data. Therefore, data sorted in bigger blocks can be compressed better than data sorted in smaller blocks. That is why block size often also is referred to as compression rate. When calling functions in libbzip2 one often has to pass the block size (compression rate) as a parameter. The block size parameter is then an integer between 1 and 9, and the actual block size used is 100000 bytes times this number.

2.1.4 The libbzip2 API

The libbzip2 C library has an API that provides functions for operations on both Bzip2 files and Bz2 compressed data streams. The interface for the files consists of some functions that operate on Bzip2 files as if they were normal files. That means that if a programmer for instance wants to read data from a Bzip2 file, he/she can simply call the read function in the library and specify how many bytes of uncompressed data he wants the library to read. The library will then read the compressed data from the file, uncompress it, and, if possible, 'return' (call by reference, no real return) the specified number of uncompressed bytes. Similarly a call to the write function will take the uncompressed data, and send it to compression. The actual compression is not done until the file is closed. As one can see, the interface is very convenient. A programmer only needs to keep track of file pointers. There is no need to deal with buffers because that is already taken care of by the library[9].

As stated above the file interface is high level and straight forward. In opposite to that, the interface for handling data streams is a little bit more low level and thus complicated to use. There are three main functions for compression and three for decompression. The programmer has to maintain a data structure called a stream. This stream contains, among other things, in and out buffers for the compression and decompression functions. Each function needs to be called at the right time, with the right parameters in order to compress and decompress properly. If the programmer for instance should happen to call functions in an illegal sequence, the stream will most certainly no longer work. The interface for streaming data therefore, with all maintenance that has to be done, renders a lot of more potential sources of error than the file interface. On the other hand it gives the programmer great possibilities to control the data flow. Both when compressing and decompressing the programmer is able to supply the compression/decompression functions with as little or as much data as he/she likes. The buffer space where the output is to be written can also be of arbitrarily size. By maintaining the in and out buffers the programmer also controls part of the memory usage of libbzip2[9].

There are also some high level functions to compress and decompress data without involving '*.bz2' files. These functions are called utility functions. These functions take and return entire streams so they can not be used for implementing streaming compression and decompression.

2.1.5 Bz2 Compression in Pike Based on libbzip2

Since Bz2 compression is a useful way of compressing data, and since it compresses better than Gzip, it is desirable to support Bz2 compression in Pike. That is why a module to handle Bz2 compression was made in the first place. The module was called Bz2 and was based entirely on the libbzip2 library. The Bz2 module is a glue between libbzip2 and Pike that accesses the functions in the libbzip2 library for data compression and decompression.

2.2 Requirements

2.2.1 Requirements Overview

The requirements section in this report summarizes the requirements described in the requirements specification of the Bz2 module. The requirements specification can be found in the appendix. It is somewhat more detailed and can be read if this summary of the requirements is not clear enough.

What the Bz2 module should do, was to implement a glue between libbzip2 and Pike. It was to provide high level methods that accessed the functions in the libbzip2 in order to realize certain functionalities. These methods should follow the Pike style, i.e. they should be called, take arguments, return values, etc, the same way as similar Pike methods do. A short list of the functionalities that the module was to implement follows below.

2.2.2 List of Requirements

1. The module should be able to read data from a Bzip2 file.

2. The module should be able to write data to a Bzip2 file.

3. The module should be able to do other normal file operations on Bzip2 files such as open, close, and end of file.

4. The module should be able to handle streaming Bz2 compression of a data stream.

5. The module should be able to handle streaming Bz2 decompression of a Bz2 data stream.

6. The module should implement a high level interface that features memory management and error detection.

7. All functionality should be realized in classes and methods so that the Bz2 module's objects were as similar as possible to other Pike objects that handle similar tasks. That meant that in this case the file part of the Bz2 module was to resemble Pike file objects. The streaming part of the module was to resemble other Pike buffer objects.

8. The module may not have any impact on the rest of the Pike system. It had to peacefully coexist with the rest of the system, and not render any new platform dependency.

2.3 Architecture

2.3.1 Design Overview

Since the part of the module where implementation work had to be done was the glue, most of the design work concerned the final user level interface of the module. By defining the interface much of the design was done implicitly. As always, designing was much about compromising. There are always contradicting interests when software is designed. In the case of Bz2 they were rather few:

Intuitive VS similar to other applications (Gz) In Pike there is one more module that handles data compression, namely the Gz module. The Gz module is a module to handle data that has been compressed using Gzip. The to Gzip corresponding C programming library is called zlib[10]. The Gz module is a glue to the zlib library and the APIs of zlib and libbzip2 are in most respects almost identical[9][10]. Therefore, it would be reasonable to make the interface of the Bz2 module as similar as possible to the already existing interface of the Gz module. On the other hand, there were some things in the Gz module which were rather unintuitive. For example are compress and decompress, more intuitive names for functions that handle compression and decompression than deflate and inflate are. The way that the buffer objects were mapped to Pike level in the Gz module was not too beautifully done either. In the final design, the names inflate and deflate

were kept, the buffer methods though were not represented in exactly the same way.

User friendliness VS possibilities to control? Must an increase of control possibilities always lead to a product that is harder to use? In many cases, unfortunately yes. Increasing control means often to increase the settings the user has to do by lowering the level of abstraction where the user is interfaced. More settings makes the software harder to use. One example from Bz2 is the choice of buffer size in the calls to the functions in libbzip2. Shall we let the programmer himself/herself maintain the buffers or shall the glue do that? On the one hand is it nice for the programmer if he/she does not need to worry about it. On the other hand, a skilled programmer that knows what kind of data and how much of it he/she is going to compress or decompress, could make his application perform better if he/she could control the buffers manually.

What features are needed? Is it worth the work it takes? Another compromise is what features that should be supported versus the amount of work it will take to realize that. Libbzip2 has both a low level interface and a high level interface. The question was: Do we need to access the library at a low level, or can adequate functionality be implemented using only the high level interface (utility functions). How great will the loss be in functionality and performance? How great will the gain be in less work? As one can see this is not only an issue during the design, but it also needs to be considered already in the requirements phase.

As it will be seen in the design description below, much of the design was kept as it is in the Gz module. Since it was the same functionality, only using another compression algorithm, that was implemented, there was no reason to make any bigger changes. That way, a Pike programmer that has used the Gz module will easily use the Bz2 module. Some minor changes were done. For instance were the methods feed, read, and finish added to the Deflate class. One reason for that is that buffer classes in Pike tend to have these methods. A second reason for that is that the old solution, with a deflate method that takes an additional integer parameter, and then decides if feed, read or finish should be done, is unnecessary complex.

A design was done for the library to access libbzip2 with its low level interface. The reason for that is that streaming compression and decompression

can not be supported by only using the high level interface.

2.3.2 Detailed Design Description

The following chapter is a listing of the components that the Bz2 module consists of. It describes in detail the composition and responsibilities of the implemented Pike level interface. Figure 2.1 is a UML diagram that illustrates the design. Note that the word string is wrongly capitalized. That is because the UML tool that was used was a Java tool and would not allow to spell string in any other way.

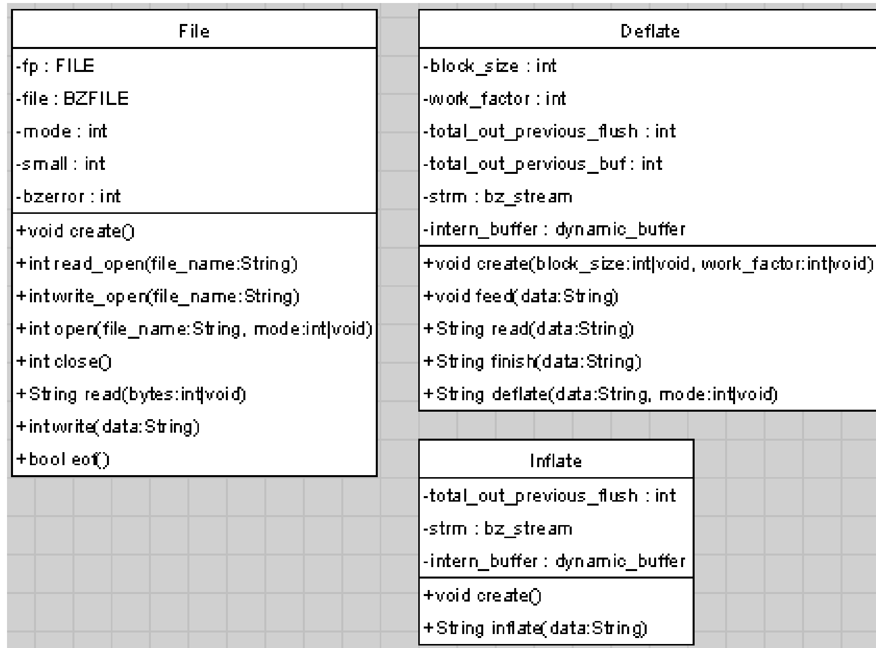

Figure 2.1: UML chart of the Bz2 module

As it is seen in the picture the module consists of three classes. The Inflate class is a class to handle streaming decompression and the Deflate class is a class to handle streaming compression. The File class provides methods for reading and writing Bzip2 files. Note that streaming compression/decompression is not compatible with compressed files. Therefore an additional class to handle Bzip2 files is needed.

Composition

The module consists of the following components

1. Constant BZ_RUN
2. Constant BZ_FLUSH
3. Constant BZ_FINISH
4. The Inflate class
5. The Deflate class
6. The File class

Constant BZ_RUN

This is an integer constant. It is equal to 0.

Constant BZ_FLUSH

This is an integer constant. It is equal to 1.

Constant BZ_FINISH

This is an integer constant. It is equal to 2.

The Inflate Class - Purpose

The purpose of this class is to provide methods for inflating (decompressing) data. The data fed to an Inflate object is a stream of Bz2 compressed data. The object is able to handle data being fed to it in arbitrarily large/small chunks.

The Inflate Class - Composition

The composition of the Inflate class is the following:
Class name: Bz2.Inflate
Class method 1: void create()
Class method 2: string inflate(string data)

The Inflate Class - Purposes of the Class Methods

Class method create: This is the constructor of the Inflate class. It creates an Inflate object. That means that it takes care of all necessary initialization so that the inflating of a data stream can be performed.

Class method inflate: This is the method that is used to do the inflating. The argument data is a string of compressed data from a data stream. The method decompresses (inflates) this string as much as possible and returns the unpacked data in a string. If no data was decompressed, the empty string "" is returned, and if something goes really wrong, the method returns 0. Data that could not be decompressed because the end of the compression block was still missing, is stored internally and a new attempt to decompress it is done the next time this method is called and fed with more data.

The Deflate Class - Purpose

The purpose of this class is to provide methods for deflating (packing) data. It provides methods for streaming Bz2 compression. The data can be fed in arbitrarily large/small strings.

The Deflate Class - Composition

The composition of the Deflate class is the following:
Class name: Bz2.Deflate
Class method 1: void create(int(1..9)|void block_size, int(1..250)|void work_factor)
Class method 2: void feed(string data)
Class method 3: string read(string data)
Class method 4: string finish(string data)
Class method 5: string deflate(string data,int(0..2)|void mode)
mode is element of BZ_RUN, BZ_FLUSH, BZ_FINISH

The Deflate Class - Purposes of the Class Methods

Class method create: This is the constructor of the Deflate class. It creates a Deflate object. That means that it takes care of all necessary initialization so that the deflating (compressing) of a data stream can be

performed. The first optional argument for the method sets what block size (the larger the blocks, the higher the compression) the deflate methods will use when compressing. The block size actually used is 100000*blocksize bytes. The second optional argument specifies how quickly the compression library will turn to a fallback algorithm when running into difficulties during compression.

Class method feed: This method feeds the string data to the internal buffers of the Deflate object. Errors are rather unlikely and since they are exceptional they will be handled by throwing exceptions.

Class method read: This method feeds the string data to the internal buffers of the Deflate object. Then it compresses all in the object buffered data, returns the compressed data as a string, and empties the internal buffers of the object. It does not terminate the stream, but just does some partial packing. It returns the compressed chunk of data as a string. If there is no data to return it returns the empty string "".

Class method finish: This method feeds the string data to the internal buffers of the object and compresses all buffered data, just like read. The difference between read and finish is that finish also adds an end of data stream marker to the compressed data and resets the Deflate object. It terminates the old data stream and initializes a new one so that more deflate calls to this object can follow. The return value is then as usual the packed data as a string. The method returns 0 if there is no data to return.

Class method deflate: This method is nothing more than method 2, 3, and 4 in one. The first argument is the string that is to be compressed, the second argument specifies which one of feed, read, and finish that will be run. The second argument is optional and if it is omitted finish will be run by default. The reason for that is that the output from finish can be streamingly unpacked without adding anything to it. The constants BZ_RUN, BZ_FLUSH, and BZ_FINISH can be used to specify the mode. They are taken directly from the libbzip2 library, and are defined on the top level of the Bz2 module. The method returns the resulting string. If the method is run in run mode, it does a feed and returns the empty string. If it is run in

flush or finish mode the returned string will be the same as it would have been if read or finish would have been run respectively instead.

The File Class - Purpose

The purpose of this class is to provide the methods for handling Bzip2 files. It provides methods to read from and write to Bzip2 files, doing the decompression and compression for the programmer.

The File Class - Composition

The composition of the File class is the following:
Class name: Bz2.File
Class method 1: void create()
Class method 2: int(0..1) read_open(string file)
Class method 3: int(0..1) write_open(string file)
Class method 4: int(0..1) open(string file, void|string mode)
Class method 5: int(0..1) close()
Class method 6: int|string read(int|void len)
Class method 7: int write(string data)
Class method 8: int(0..1) eof()

The File class - Purposes of the class methods

Class method create: This method is the constructor of the class. It takes no arguments and does only the basic initialization.

Class method read_open: This method opens a Bzip2 file for reading and associates it with the object. The argument is a string containing the name of the file that should be opened. The return value is 1 if the open was successful and 0 otherwise.

Class method write_open: This method opens a Bzip2 file for writing and associates it with the object. The argument is a string containing the name of the file that should be opened. The return value is 1 if the open was successful, 0 otherwise.

Class method open: This method opens a Bzip2 file for either writing or reading and associates it with the object. The first argument is a string specifying the name of the file that should be opened. The second optional argument should either be the string "r" or "w" depending on if the file should be opened for reading or writing. The mode argument will by default be set to "r" if it is omitted. Due to the methods read_open and write_open this method is redundant. The only reason for keeping it is that this is the way files are opened in the Gz module. The method returns 1 upon success otherwise 0.

Class method close: This method closes the file associated with the object. It returns 1 if the close was successful, 0 otherwise.

Class method read: This method reads uncompressed data from a Bzip2 file. The optional argument len specifies how many bytes of uncompressed data that is to be read. If len is omitted the whole file is read. The read data is returned as a string. If the read is unsuccessful the method returns 0.

Class method write: This method writes the string data to the file associated with the object. The method returns the number of compressed bytes written. It returns 0 if the write is unsuccessful (or if the method was invoked with an empty string).

Class method eof: This method returns 1 if the end of the file has been reached, 0 otherwise.

2.3.3 From Design to Implementation

Taking the step from a design model to an implementation is in the software development process always a crucial and in many cases a not too easy task. It involves making something concrete out of the abstract model. The module is designed for the Pike level, but it is implemented on the C level. The design of the user interface and the design of the Pike level of the module is more or less the same thing. The whole object orientation including models describing the components of the module is done on Pike level. The interesting question then is how to implement this on the C level. A possibility could be to make

a new design model for the C level. Still in this case that would not have helped the implementation as much as it would have rendered more work.

Instead, a C skeleton was made out of the design model. The skeleton consisted of declarations of the functions and data structures needed in order to implement the classes. In order to build this skeleton some kind of mapping between Pike level and C level had to be done.

The mapping of classes and objects to the C level is a little fuzzy and Pike specific since classes with class variables and class methods do not exist in common C. The C API in Pike provides functions and macros to create Pike classes. There are three kinds of C level entities that play a central role when a class is being implemented and they are called program, function, and storage.

Function

A function is nothing more than just a normal C function. Functions are used for the implementation of class methods. In the Bz2 module the methods in the design model were all assigned one underlying C function. That function is then called each time the overlying Pike method is invoked, and is then responsible for doing all the 'work' of the overlying method. This means that each method in the Pike level design more or less "became" a function in its C level implementation.

Storage

Storage is a piece of memory where the programmer can store data necessary to implement an object. The data needed to be stored in storage are the kind of data that needs to be "remembered" over several function calls. That is variables that in a way represent the state of the object. In a high level object oriented programming language this would be object variables. In the Bz2 module each class has a storage for a struct called container. The struct contains all variables needed in order to store the state of either one of the possible objects of the Bz2 module.

Program

Classes in Pike are implemented on the C level using an entity called a program. Storages and functions can be added to a program using macros (see tutorial section). When a Pike level object's method is invoked the program

runs the function that was added to that program to be the underlying C function for that invoked method. The program also provides a pointer to the storage where the object variables of that object are found. To sum up and generalize, the mapping from Pike level design to C level implementation is as follows:

- Classes become programs.
- Class methods become static functions that are added to their corresponding program.
- Object variables become storage that is added to its corresponding program.

2.3.4 Implementation Using the C Interface

Once the skeleton was built, and the final step from high level design to a C level implementation model was taken, the actual implementation work could start. The implementation was much more time consuming work than it was first estimated to be. Even though libbzip2 has a well defined API, with a good manual, streaming compression and decompression took a lot of time to implement.

Maybe the time could have been shortened a bit if the manual of the libbzip2 had been read more carefully. The problem is often that one need to fully understand almost every detail in the manual in order to use that specific software right. One also often need to do in order to fully understand what is written in the manual. Unfortunately it is not unusual for manuals to contain ambiguous statements. Also it is not unusual for manuals to describe complicated topics that are learned easiest by doing. Therefore, taking the starting conditions into account, some time of unsuccessful trial and error was needed in this particular case.

Most of the problems with the implementation were with the streaming compression and in particular decompression. Since the glue was to implement some memory management there were some common problems with buffers such as broken pointers and memory leaks. The low level interface of the libbzip2 was used for the implementation of the streaming methods. Basically what the functions did was to take the data it was fed with, and compress/decompress it as far as possible. Then it returned the output as a string

and buffered all data necessary to carry on the compression/decompression the next time the function was invoked.

2.3.5 Implementation Using the CMOD Interface

Since one reason for this project was to evaluate both C module interfaces, a second implementation was done. This time the CMOD interface was used. In comparison to how much time that was spent on the first implementation, very little was spent on the second. That is of course because no actual second implementation was done in the meaning solving all problems over again. It was more a transformation of the already existing C implementation into a CMOD implementation. The only things that needed to be redone were the class variables, the classes and methods, and the handling of arguments for the methods.

Still the second implementation was not only a matter of translating code into CMOD. During this part of the implementation the module was tested further and some bugs(from the C implementation) were found and fixed. The code was worked through once again, and so the module was improved.

The final CMOD implementation had some advantages compared to the earlier C implementation:

- Less code
- More consistent argument checking
- Nicer looking code
- Less bugs

Less Code

By inspection of the from CMOD generated C files, one easily discovers that CMOD automizes argument checking. In this case that reduced the amount of code needed.

More Consequent Argument Checking

Since the argument checking is automized it is always performed in the same way. That way errors occurring due to sloppy written argument checking code are avoided.

Nicer Looking Code

Actually, what is nice and not is subjective. Therefore it is just my personal opinion that the code that was written in CMOD looked a lot nicer than the pure C code. I think the CMOD code has a better structure so that it is a lot easier to see what it does.

Less Bugs

During the 'transformation' from the C implementation to the CMOD implementation two bugs were found in the function that reads data from Bzip2 files. The bugs were found by inspection of the code and were corrected in the final CMOD implementation.

2.3.6 Description of Some C Level Functions

The rest of this section contains short descriptions of what some functions (that might need a little explanation) of the Bz2 module do. A reader without the intention to look at the source code of the module can skip this and jump to 2.3.7 . The descriptions only mention the most essential of the functions. Error check and such is omitted from the description. Note also that this section is for both the CMOD and the C implementation. The only thing that differs concerning this part is the name of the functions. The first name in each header below is the name the function had in the C implementation, the second is the name the function had in the CMOD implementation.

```
bz2_inflate_inflate() / inflate()
/* implementation of Bz2.Inflate()->inflate() */
```

This function is the underlying C function for the method **inflate()**. The function decompresses as much as possible from its input, returns the decompressed data and buffers the rest in the current storage. The function works as follows:

- If there is any data from the previous call to this function that has not been compressed, it copies that data from the previous input buffer to a new input buffer.

- The whole previous input buffer is freed

- The input is appended to the new input buffer (string concatenation)
- Output buffer space is allocated
- bzDecompress is called, if output buffer is filled, more space is allocated and bzDecompress is called again, and again .....
- All bytes that bzDecompress has written to the output buffer is pushed to the stack as a Pike string.
- All output buffer space is freed.

```
bz2_deflate_deflate_run() / feed()
/* implementation of Bz2.Deflate()->deflate() */
```

This function is the underlying C function for the method **feed()**. It runs bzCompress(BZ_RUN...) with its input. Depending on how many bytes of input it gets, bzCompress will either just feed it to its internal buffers, or it will compress parts of it and write it to its output buffer (weird!). The function works as follows:

- If the internal buffer has not been initialized then it is initialized.
- An output buffer is allocated
- The input is taken as input buffer for bzCompress
- bzCompress is called with mode == BZ_RUN
- If the output buffer is full, more space is added to the output buffer and bzCompress is called repeatedly until all input has been used.
- If the output buffer is not empty, then the data in the output buffer is appended to the internal buffer of the glue.
- The whole output buffer is freed.

```
do_deflate()
/* help function for:  Bz2.Deflate()->read()
                       Bz2.Deflate()->finish() */
```

This function is called by the following to functions:

```
bz2_deflate_deflate_flush() / read()
bz2_deflate_deflate_finish() / finish()
```

It handles the calls to bzCompress(). The function compresses all of the input data and returns a buffer containing the compressed data. It works as follows:

- It takes an output buffer and the flush mode as argument. The input data, which is read from the stack is the input buffer.
- bzCompress is called with either BZ_FLUSH or BZ_FINISH as argument, depending on with which mode parameter do_deflate() was called.
- If output buffer is full, more output buffer space is allocated and bzCompress is run repeatedly until the whole output buffer is not filled.
- The whole output buffer is returned to the calling function.

```
bz2_deflate_deflate_flush() / read()
/* implementation of Bz2.Deflate()->read() */
```

This function is the underlying C function for the method **read()**. It does a flush on the stream. It compresses all of its input and all of the internally buffered uncompressed data. It concatenates this compressed data, with the internally buffered compressed data and pushes it all to the stack. It works as follows:

- Initializes an output buffer and calls do_deflate().
- Checks the internal buffer, if it is not empty the result from the call to do_deflate() is appended to the internal buffer and becomes the buffer that is to be returned. If the internal buffer is empty the result of the call to do_deflate() becomes the buffer to return.

- The buffer to return is pushed to the stack.
- The internal buffer and the initialized output buffer are freed.

`bz2_deflate_deflate_finish() / finish()`

This function is the underlying C function for the method **finish()**. It is similar to bz2_deflate_deflate_flush(). Instead of flushing the stream, it finishes it and initializes a new one. Another difference between finish and flush is that finish adds an end of stream marker at the end of the compressed data. It works as follows:

- First four steps are the same as for bz2_deflate_deflate_flush()
- bzCompressEnd is called to terminate the stream
- The settings of the corresponding object are pushed to the stack as arguments for the create function, then bz2_deflate_create() is called in order to reinitialize the object.

The rest of the functions are rather easy to understand how they work and do not need that much explanation. The create functions of the Inflate and the Deflate class initialize the internal data structures needed for decompression and compression respectively. In practice that means that they reset all 'class variables' and run bzDecompressInit() and bzCompressInit() respectively. The whole file part of the implementation interacts with libbzip2 on a higher level than the streaming part and has therefore a much easier implementation.

2.3.7 Test

Test Items

The items to be tested were the components of the Bz2 module as they are described in the design description. Every subcomponent became so to say one test item.

Test item 1: The 'Inflate' class
Item 1.1: The 'create' method
Item 1.2: The 'inflate' method

Test item 2: The 'Deflate' class
Item 2.1: The 'create' method
Item 2.2: The 'feed' method
Item 2.3: The 'read' method
Item 2,4: The 'finish' method
Item 2.5: The 'deflate' method

Test item 3: The 'File' class
Item 3.1: The 'create' method
Item 3,2: The 'read_open' method
Item 3.3: The 'write_open' method
Item 3.4: The 'open' method
Item 3.5: The 'close' method
Item 3,6: The 'read' method
Item 3.7: The 'write' method
Item 3.8: The 'eof' method

Test Strategy

There is one keyword that summarizes the whole test process and that is 'continuous'. While the module evolved, subcomponent by subcomponent, it was also tested subcomponent by subcomponent. That means that when a method was written it was also immediately tested as far as possible. Once that method fulfilled its test cases a new subcomponent was made an tested. With time the subcomponents formed components, and so entire components were tested and finally the entire module. So the test strategy was to begin with the smallest subcomponents and then work the way up.

Tools

One tool that was used for test was 'Hilfe'. 'Hilfe' stands for 'Hubbe's incremental LPC front end'. Hilfe can be run to interactively evaluate Pike statements. The advantage with using Hilfe is that one during development can run the current Pike without having to make a new install of it. It takes a lot of time to install a Pike. Therefore, the time from a change has been made until the consequences of it an be tested, can be shortened significantly by running Hilfe instead.

Another tool that was used for test was the Bzip2 program. It is used to compress data to *.bz2 files and to extract data from *.bz2 files. Thus the output from the read and write methods in the File class could be evaluated more easily.

Test Robustness

The whole idea behind testing robustness is to run the program doing everything in one's power to make it crash, and see how the program reacts on that. In the case of the Bz2 module the following was done:

- Methods were invoked with bad number of arguments
- Methods were invoked with argument of bad type or containing bad data
- Streams were broken calling feed(), read(), finish(), and create respectively inflate() and create() at unexpected times
- Attempts to open already open File objects in new modes were done
- Attempts to write to File objects opened for reading were done
- Attempts to read from File objects opened for writing were done
- Pike programs were exited without closing the open files
- The internal buffers used in the glue were made as small as one byte in order to let it run out of memory.

Look for Memory Leaks

In order to find all memory leaks Pike can be run in dmalloc mode. Dmalloc stands for debug malloc and is a feature that keeps track of all memory allocations and deallocation that a Pike program does when it is run. That way Pike can tell upon termination what memory has been returned and what memory not. By running test programs in dmalloc mode memory leaks that occur during normal program execution will reveal.

It is a little bit tricky to find memory leaks that happen when exceptions are thrown, because they will never be found as long as the program executes normally. One needs to keep in mind that when the exception handler is

invoked the execution takes another way. Therefore it is of great importance to release all memory that is held before throwing an exception.

Faults Discovered on the Way

The most common errors that were found were memory leaks and broken pointers. There were also some minor faults with the usage of libbzip2. All found faults were corrected.

Results from Final Tests

The final tests that were run are the test programs that can be found in the source of the Bz2 module. These programs replace the smaller test cases that were used during development. The four final test programs all together test each test item at least once. The test programs were all run in dmalloc mode and no errors were found.

Chapter 3

Evaluation of the Pike C Module APIs

3.1 Overview

This chapter is an evaluation of the Pike C module APIs. The C and CMOD interface are first compared to each other. After that a comparison to the JNI is done based on some criteria. The development of the Bz2 module formed the basis of the Pike parts of this chapter. The Java parts of this chapter are heavily based on the JNI manual.

3.2 The two Interfaces

Originally, there was only one interface for writing C modules, the old C interface. Even today one can say that this still is the only interface there is. As mentioned in the introduction the CMOD is no interface of its own, it is rather more of a wrapper around the C interface. To realize that one only needs to take a look at how a *.cmod file is processed when it is compiled. The *.cmod file is precompiled to a *.c file. The precompilation is done by a program called 'precompile.pike'. That generated *.c file contains then a C code translation of the *.cmod file, and is then later the file that is compiled. So out of the source files that so to say use the CMOD interface, source files that only use the C interface are generated, and then these C files are compiled. In the end the C interface is used anyway. CMOD's functionality is that it lets the programmer interact with Pike on a higher level since it

reduces the times that a programmer need to access internal data structures and functions of Pike. The calls to items belonging to the C interface are abstracted through CMOD, which means nothing else than to raise the level where the programmer interacts with Pike.

To visualize that we can take a look at figure 3.1. The figure shows different levels on which a programmer can interact with Pike. The lowest

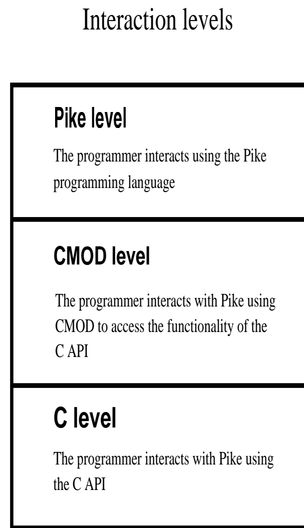

Figure 3.1: Interaction levels in Pike

level mentioned in the picture is to interact with Pike via the C interface. Adding the CMOD layer to the C interface allows the programmer to interact on a higher level. Programming is still done in C, but the programmer can take advantage of the functionality provided by CMOD. The top level is to program in the language Pike. Then the programmer does not use C anymore but Pike.

3.3 CMOD and the Old C Interface

The CMOD interface is an attempt to create an interface that is easier to use than the old C interface. The whole idea, with CMOD is that the programmer should be able to write his module in a more object oriented way. Since Pike is a object oriented language [4] this means that the programmer should be able to realize his module in a way more similar to what the module looks

like on the Pike level. As it says in the tutorial part, a translation from a UML model to a CMOD skeleton can be done in an intuitive way. Therefore CMOD makes it easier to take the step from design to implementation.

One great advantage with CMOD compared to C, is that it makes the code more readable. For a programmer familiar with both interfaces writing a module would be more or less equally easy/hard independent of choice of interface. The difference to work with them is not that big actually, once one has learned the C interface that is.

The CMOD interface has some features that make a big difference in readability of the code. They are:

- All methods and classes are written with CMOD keywords for methods and classes. That way the code gets a structure more similar to the Pike level structure of the module.

- Object variables are passed as variables in their CMOD implementation. This is easier than having to look at the stack.

- Since CMOD provides automatic type checking, no argument checking code needs to be written in CMOD files. Thus the amount of code is reduced

CMOD generates code that fetches arguments from the stack and pushes return values to it. CMOD also generates code that counts the arguments and does type checking. This remarkably reduces the amount of code that a programmer has to write manually in a module. Since the argument checking is the same in every method there is no use writing it over and over again. The argument checking code is not hard to write, but not having it in the actual code of the module of course makes the code shorter and thereby easier to read.

A problem with CMOD is that there are some things that it does not handle that makes the programmer need to know the C interface anyway. For instance does the interface pass the arguments as an svalue pointer when the argument is of type **...|void**. These differing ways of handling arguments can be a bit confusing, because for parameters with a distinct type, the interface passes them as their corresponding C level type.

Another case where the programmer has to alter the stack manually is when a void method is implemented. Normally, when methods with a return value are implemented, the programmer will use a call to **RETURN** in order

to return the value. This call to **RETURN** will then generate code that removes the arguments from the stack and then pushes the return value to the stack. In case of a void method **RETURN** can not be used since it needs a value to return. The programmer then has to pop the arguments from the stack manually instead.

One goal with CMOD should among others be to access the stack transparently. That means that the programmer should never have to deal with stack values and push and pop operations. As it is today the CMOD interface does not do that for the most complicated kind of argument. In order for CMOD to be the high level help that it is intended to be, support for this ought to be added to it.

3.4 The JNI Interface

3.4.1 What is the JNI?

In order to form an opinion of the Pike C module interfaces, it is probably a good idea to compare them to corresponding interfaces in other languages. Due to the limited amount of time, it is only compared to one other interface. The chosen interface was the JNI. JNI stands for Java Native Interface and is Java's interface for writing native methods. A 'native method', in this case, is a method written in another programming language than Java. [2] One reason for writing native methods in Java is to take advantage of platform specific functionality that is not supported by any Java Virtual Machine. In other words, the JNI is useful when writing a Java application that for some reason only can be realized in a platform dependent way. [2]

A second reason for using the JNI is to speed up the execution of time critical parts of programs. This however, is only worth doing for larger pieces of native code since the overhead for starting the JNI is enormous. [3]

3.4.2 How is the JNI Used?

The main program is written in the usual way. The native methods that are to be used are declared using the keyword 'native'. Once the main program is written it can be compiled the normal way. After that the header files for the native methods have to be generated. There is a tool called javah that does that. Then the implementation of the native methods are written.

Finally the native methods have to be compiled to a shared library that can be loaded by the programs that want to use it. In Unix/Linux these files are called *.so files, in Windows *.dll files. Once that library is present the main program that calls the native methods can be run through the Java interpreter. In order to exemplify this one can take a look at figure 3.2. The picture is taken from the JNI tutorial [2] and illustrates how a hello world program is made using a native method that displays the 'hello world' message.:

3.5 Comparative Evaluation

3.5.1 Overview

The following section is a comparative evaluation of the C/CMOD interfaces in Pike and the JNI interface in Java. The following points were chosen as criteria for the evaluation:

1. Functionality - What does the interface do?
2. Performance/efficiency - Does using the interface remarkably affect the execution speed of the program?
3. Readability of code - Is the resulting code readable and thereby easy to debug?
4. Platform dependency - How is it handled?
5. Getting started - How easy/hard is it to get started to program with the interface?
6. Programming with the interface - Is the interface easy to program in?
7. Argument handling - The passing of arguments, how is it handled?

Note that the comparison is not always fair to do since the languages are fundamentally different.

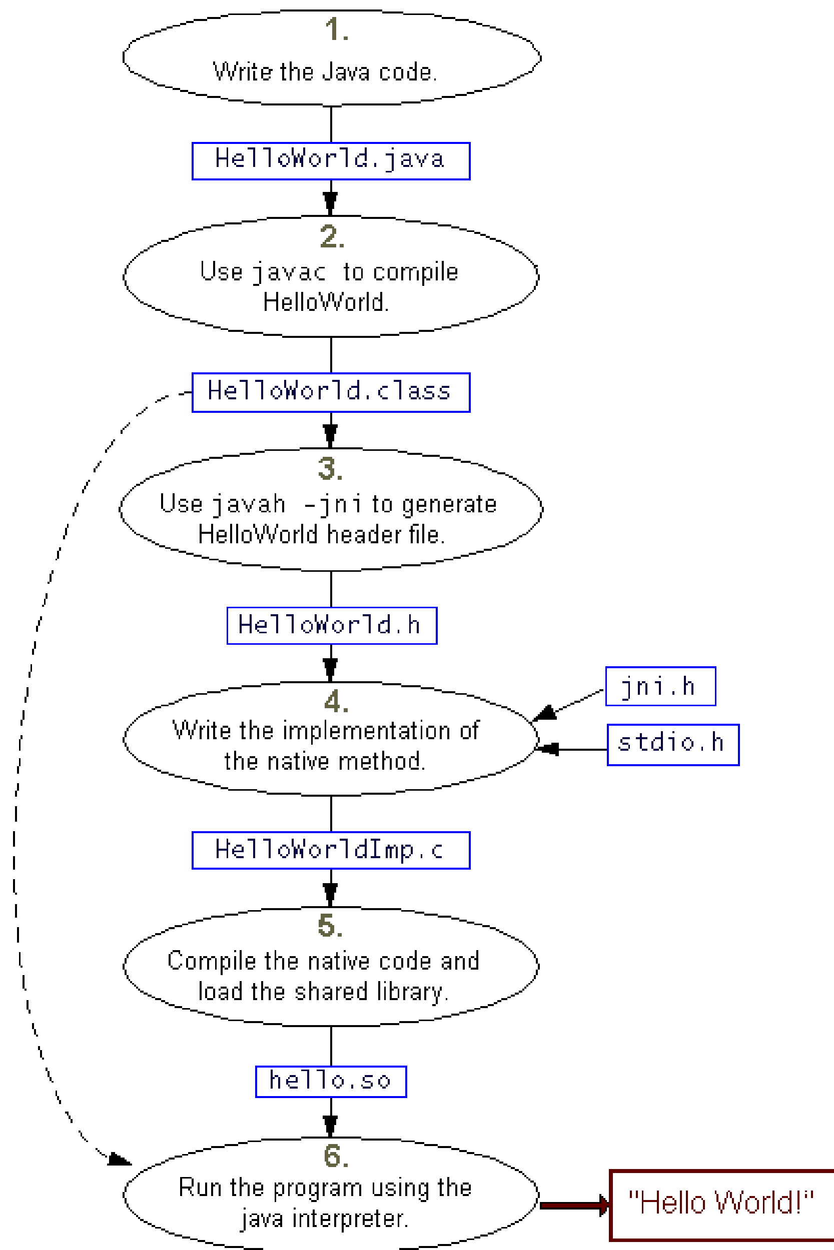

Figure 3.2: Hello world program using JNI

3.5.2 Functionality

Pike

The C/CMOD interface provides a way to extend the Pike system with modules that are implemented in C. That means that if C code in some way is put in a module, a programmer can access and use it through Pike.

Java

The JNI gives the possibility to access native methods from the Java Virtual machine. It also has support for embedding the whole Java Virtual Machine in a native application. In other words a C programmer can in his/her programs make function calls that access functionality from the Java Virtual Machine[1]. So the JNI can not only be used as a way to handle platform specific tasks in Java programs, but it can also serve as a way to facilitate making C programs platform independent by taking advantage of the platform independent JVM.

3.5.3 Performance/Efficiency

Pike

To write a C module for Pike actually means nothing more than to extend the virtual machine. In a performance perspective that is of course very good because this means that there is basically no overhead for using a module, a fact that can be taken advantage of. It is for instance possible to speed up the execution of some Pike level application by implementing some of its functionality as a C module in a more optimized way.

Java

Performance is the greatest drawback with the JNI. It takes a lot of overhead to invoke native methods. Little optimization has been done in the code that makes native code run. Roedy Green [3] states that Sun seems to be discouraging the use of JNI at all by making it slow. Therefore JNI can hardly be used as a tool for speeding up programs. [3]

3.5.4 Platform Dependency

Since C is very platform dependent, this is an issue that each JNI user and each developer of modules for Pike has to deal with. Since the JNI and C/CMOD interfaces are a lot different, this is handled in different ways.

Pike

As stated earlier, in Pike the native method is compiled into the virtual machine. That is why issues related to platform dependency is handled in the build process. The programmer that writes a module is also responsible for writing configure files that handle platform specific things to a satisfying extent. What is meant by satisfying extent depends on how the module is supposed to be used. If it is only created for one single purpose it might be sufficient for it to work on as little as one single platform. However, if it is supposed to exist in future releases of Pike, it ought to be as portable as the rest of Pike.

Java

The programmer that writes a native method is fully responsible for handling things related to platform dependency in his/her code. Since one main reason for the JNI is to take advantage of platform dependent things that are not supported by the JVM[2], JNI may be used when writing programs that are not supposed to be platform independent.

3.5.5 Getting Started

Pike

Getting C code to work in Pike is rather complicated since it all has to be built as or within modules. This means that in order to use a native method the module has to be added the normal way. That means that the Pike has to be reconfigured and rebuilt once the native methods, or stubs for the native methods, have been written. This takes time, and it can also cause problems. In theory, it is not that difficult to reconfigure and rebuild, but practice has showed that it can be. There are many things that can go wrong in the Pike build process. The fact that very little documentation of the Pike module system exists as it is, today does of course not make it easier to get started.

Java

When it comes to getting started, the JNI has one huge advantage compared to the Pike C module APIs and that is that it is easier to make native code work in Java using JNI. That is because native methods in Java are loaded in run time using shared libraries. Native methods are written and compiled to shared libraries. After that they can be loaded and thereby used by the main program.

A 300+ pages manual [1] of the JNI including detailed specification of the entire interface exists and can be downloaded for free from Sun's web-site. Some books on this topic have been written too.

3.5.6 Programming with the Interface

Pike

Both the old C and the newer CMOD interface are convenient to use. The most complicated one is the old C interface. It is not trivial to see how some of the key macros in this interface work and what they do. On the other hand, once one has learned this structure of the interface it is just as straight forward to use as CMOD.

The CMOD interface is more convenient since methods and classes are declared the same way they are called on the Pike level. No macros are directly used, which means that they are used, but hidden in the written code. The CMOD keywords are rather few and intuitively chosen.

Java

The JNI is also a little complicated to use. In order for the native methods to work, they need to be named in a certain way. Long, rather complicated looking prototypes are needed here and there. This is not hard to manage, but it takes a little effort to learn. Another problem that a programmer has to deal with when using JNI is that the languages have different data types.

3.5.7 Argument Handling

Pike

Since Pike is entirely based on C, data types are not that big a problem. All Pike types have been implemented using C. Complete data structures to

handle all defined Pike types exist. This means that no conversion of data types is done when arguments are passed between functions. The programmer simply works with the pre-defined data structures for Pike data types. That way data type errors due to bad conversion are avoided, and many other data type errors can be discovered at compile time.

Java

The JNI includes a set of data types that correspond to some Java data types. It contains definitions both for primitive data types, such as int, bool, char and for more complex data types in Java known as reference types. In order to be used in native methods these types have to be converted to some data structure that is valid in C. The JNI has some data structures that are the C level correspondence of some Java data types. JNI also provides functions that the programmer can call in order to convert to and from these types.[1] The point however is that the data types have to be converted and that it is the programmers responsibility to do this correctly. The conversion is a possible source of errors. These errors might slip through compiling phase and crash the JVM at runtime.

3.5.8 Readability of Code

Pike

As mentioned earlier, using CMOD results in very readable code. Classes and methods implemented using CMOD follow a nice structure. Compared to the older C interface CMOD results in less and better structured code. Therefore, it takes less effort to read and understand the code of a module written in CMOD, than it takes to read and understand the code of the same module written in plain C.

Java

When it comes to JNI, my opinion is that using it results in code of which parts are not too readable. Although everything is realized in a well structured straight forward way, there is one thing that affects the overall readability of the code: the long lines of code. These long lines are mainly a consequent of the clumsy way that JNI names components. The names are long and contain a lot of underscores. That of course gives a good detailed

description of the component. The problem is that it with the usual Java style of using lots of modifiers like 'public static native' and packing as much in each line of code as possible, makes the lines long and thereby hard to survey.

3.5.9 Summary of Comparative Evaluation

The comparative evaluation was an attempt to measure how useful the Pike C module APIs are in comparison to its worldwide most known correspondence, namely the JNI. The conclusion one can draw is that for most of the points discussed does Pike perform just as well or even better than Java. As stated before the evaluation the comparison was not 100 percent fair since the concept for how native code is used differ between the two languages. Whereas JNI serves as a glue to compile programs written partially in C and Java the Pike C interface more serve as a way to extend the Pike language. Both approaches have, as it can be seen in the evaluation, their advantages and drawbacks compared to the other one.

The Pike approach makes it possible for a programmer to write C code that when it is run has a short execution time since it then is a compiled part of Pike. The fact that everything is compiled and linked together in Pike also makes it easier to pass arguments, since the internal data structures of Pike can be taken advantage of. On the other hand, the Java approach allows usage of native code without having to rebuild the whole system. To sum up, the Pike approach is more convenient for programming, but it is a bit more of a clumsy solution than the Java approach.

Concerning the code the usage of the different interfaces result in, my opinion is that CMOD results in more readable code than both the old C interface and the JNI. It is the only one of these three that provides automatic type checking, which of course reduces the amount of code. It also has a good structure and the naming is concise. That makes the code relatively easy to survey and to debug.

Concerning user friendliness it is hard to reach some final conclusion. For a novice, the JNI is probably easier to get started with. Since there is not as much compiling to do compared to Pike, unsuccessful tries will not take as much time. On the other hand, once the programmer has got started, the usage of the Pike interfaces is just as easy as, the usage of the JNI and very convenient. Fact remains that the whole build process of Pike is its greatest drawback in the field of user friendliness. Once one has got started

programming is rather easy, but the initial hurdle that a Pike novice has to tackle is “too big”.

Chapter 4

Results and Future Work

4.1 Results

As already stated several times before, this project consisted of several quite separate parts, and therefore also do not form one distinct conclusion. A case study for developing a C based module in Pike was done and resulted in the following:

Two implementations of a module that handles Bzip2 were done. The one that uses CMOD is a part of Pike 7.5 and can be found in the Pike CVS.

The project resulted in three tutorials for Pike 7.5. Because these tutorials are heavily based on the development of the Bz2 module, they are not all covering. Still they contain a lot of information on how to write C modules for Pike. A Pike developer can get guidance on how to write a C module by reading them.

The evaluation part found that the CMOD wrapper is a nice feature that facilitate the writing and also the maintenance of modules. A great advantage with CMOD is that it makes the code more readable since it both reduces the amount of code and gives the code a better structure.

Further on was a comparison done between the Pike C module interface and the Java Native Interface. That evaluation found that the Pike CMOD/C interface in many aspects performs just as well as, or even better than the JNI, when serving as a way to use C in a high level language. Pike's main advantages are speed and that it is convenient to program in. The advantage of JNI in comparison to Pike is its less clumsy design that makes it a lot easier to get started programming.

4.2 Future Work

The Bz2 module is working but can of course be improved. One future improvement is to look over the buffer allocation, which is now rather primitive.

As it says in the results the tutorials are just 'a good start'. A lot of work can be done here to extend and improve the content of these. The tutorials on the C interface and the CMOD interface can be extended to contain more functions and macros and so on. The already existing example and their explanations can of course if needed be improved and made more detailed. Fields that are hardly covered and of great importance for Pike developers is how to handle platform specific problems. In other words, the whole setup of a module, how it is to be written and how it is to be used need to be explained more detailed. This would involve digging more in autoconf and try to visualize that in a good way. Further it would involve digging more in the platforms themselves and find out more on their specialties, and what problems a programmer might encounter as a consequence of that.

Modules that provide Pike with new functionality can be written. Hopefully this report will serve (one of) its purpose(s) and guide new Pike developers on how to write modules.

This report states that CMOD is an improvement compared to the old C interface. What can be improved is the support for the ...|void arguments. That goes also for the return of void methods. CMOD needs to support RETURN(void) or RETURN(). Until this has been done, CMOD is incomplete, which is a pity since the working parts are that nice.

4.3 Acknowledgements

There are some people who have helped me out during this project that I would like to thank.

Special thanks to:

Dr. Uwe Aßmann For giving me the opportunity to do this master thesis project. Thank you also for all constructive criticism.

Martin Nilsson For being my tutor, i.e. helping out, reading my papers and giving me feedback and so on....

Also thanks to:

Johan Sundström For being Martin's stand-in the first two weeks when he was away. Thanks also for being the one that kept me company at the 'office' :-)

Henrik Grubbström For patiently answering my 'stupid' questions although it is not your job. Also thanks for the evaluation of the Bz2 module.

Marcus Agehall For reading and giving feedback on my tutorials.

Leif Stensson For reading my report and giving more feedback when I thought I was done.

Glossary

Some words and their meaning in THIS report

API	Application Programming Interface
Bz2	Short for Bzip2
Bzip2	A program for data compression based on the Burrows-Wheeler algorithm
C library	A collection of C functions and data structures that can be imported to a C program
C module	A module written in the language C
CVS	Concurrent Version System, A software system to handle concurrent development of source code
(to) deflate	In terms of data compression it means: to compress; to pack
(to) inflate	In terms of data compression it means: to decompress; to unpack
JIT	Just In Time, in terms of compilation this means to compile components when they are needed
JNI	Java Native Interface
JVM	Java Virtual Machine
libbzip2	A C programming library for data compression based on the Burrows-Wheeler algorithm
method	function in an object oriented high level language
Pike	A modern programming language
Pike module	A module written in the language Pike

Bibliography

[1] Sheng Liang(1999): *The Java Native Interface, Programmer's Guide and Specification*, Addison-Wesley Pub Co, ISBN 0-201-32577-2

[2] Beth Stearns: *The Java Tutorial, trail: Java Native Interface* http://java.sun.com/docs/books/tutorial/native1.1/ (2003-02-02 20:00)

[3] Roedy Green: *JNI, The Java Native Interface*, http://mindprod.com/jni.html (2003-02-02 20:00)

[4] Fredrik Hubinette & Thomas Padron-McCarthy:*The pike beginner's tutorial*, http://pike.ida.liu.se/docs/tutorial/ (2003-02-02 18:00)

[5] Pike development team: *Reference Manual for Pike v7.5 release 1*, http://pike.ida.liu.se/generated/manual/ref/ (2003-02-02 20:00)

[6] Pike development team: *About pages on the pike web site*, http://pike.ida.liu.se/about/ (2003-02-11 17:00)

[7] Mark Nelson: *Data Compression with the Burrows-Wheeler Transform*,Dr. Dobb's Journal (September 1996)

[8] Arturo San Emeterio Campos: *The introduction to Data Compression* ,http://www.arturocampos.com/cp_ch1.html (2003-02-02 18:00)

[9] Julian Seward: *bzip2 and libbzip2, a program and library for data compression*,ftp://sources.redhat.com/pub/bzip2/docs/manual.pdf (2003-02-02 19:00)

[10] Jean-loup Gailly & Mark Adler: *zlib 1.1.4 Manual*,http://www.gzip.org/zlib/manual.html 2003-02-16 15:00)

[11] Doug Bagley: *The Great Computer Language Shootout*, http://www.bagley.org/ doug/shootout/ (2003-02-03 12:00)

Appendix A - Tutorials

Overview

This appendix is a collection of tutorials. The tutorials are written in a way so that each tutorial can be read as an independent document. That is why some things that have already been mentioned earlier in the report might be repeated here. The language used in these tutorials may also differ a bit from the language used in the rest of the report. The reason for that is that the tutorials are written more like instructions. The intended reader of the tutorials is a programmer who wants to learn how to write C modules for Pike. This appendix consists of three major parts. First comes a tutorial that introduces the reader to the C API of Pike. After that follows a tutorial on the CMOD API. This appendix then finishes off with a tutorial on how to write a C module for Pike.

The C API

Overview of This Tutorial

This tutorial contains explanations of the most common functions, macros, data structures and variables that a programmer needs to know about in order to do the C code implementation part of a C module. Note that this is no reference manual, nor is it a manual on how to write C modules. Words printed in bold in the text are of programming specific meaning. It can be a variable, function, data structure, data type or a macro. Using bold is an attempt to clarify their meaning in the running text. The text that is written in courier font is example code. It exists in order to show definitions of functions, macros and variables, and to illustrate how they are used.

The Absolute Minimum

In order to be recognized as a module by Pike, the least a module must define are the two functions:

```
void pike_module_init()
void pike_module_exit()
```

As one can tell by the name, **pike_module_init()** is the function run when the module is being initialized. In **pike_module_init()** the whole module is so to say defined. By using (for instance) the functions and macros listed below, every component of the module (class, method, variable) can be defined within the Pike language and mapped to its corresponding C level implementation.

The **pike_module_exit()** function is as the name indicates run when the module exits. Therefore it can contain clean up that is to be done upon exit.

set_init_callback()

```
/*Prototype:*/
void set_init_callback(void (*init_callback)(struct object *));
```

set_init_callback() sets the initialization callback of a class. The example line of code sets the function **init_my_module()** as the initialization callback of the corresponding class.

```
/*Example of usage:*/
   set_init_callback(init_my_module);
```

set_exit_callback()

```
/*Prototype:*/
void set_exit_callback(void (*exit_callback)(struct object *));
```

set_exit_callback() sets the exit callback of a class. The example line of code sets the function **exit_my_module()** as the exit callback of the corresponding class.

```
/*Example of usage:*/
   set_exit_callback(exit_my_module);
```

start_new_program()

The **start_new_program()** is a kind of starting mark that is set first of all when defining a class in **pike_module_init()**. After this come all definitions, settings and so on. It is called without any arguments. Even though it is not entirely correct, Java and C++ programmers can think of **start_new_program()** as a 'class {' statement.

```
/*Example of usage:*/
   start_new_program();
```

end_class()

```
/*Macro:*/
   end_class(NAME,FLAGS)
```

end_class() is a terminating mark for **start_new_program()**. It takes two arguments. The first one is the name by which the class defined between the **start_new_program()** and the **end_class()**, will be called. The second argument is a flag. The following example line of code results in a public class called 'MyClass'.

```
/*Example of usage:*/
   end_class("MyClass",0);
```

The fact that **start_new_program()** is terminated by **end_class()** is a bit confusing. It is this way because the words 'program' and 'class' are used synonymously in Pike and for some reason the two macros happened to be named this way.

ADD_STORAGE()

```
/*Macro:*/
   ADD_STORAGE(X)
```

ADD_STORAGE() is used to allocate a space in memory. It takes only one argument namely the name of the data type for which storage is to be provided. The macro is used to implement things like object variables. The following example line of code placed in a class makes sure a storage for the struct **number_and_letter** is created upon each instantiation of the class.

```
/*Example of usage:*/
   struct number_and_letter{
     int number;
     char letter;
   };
   ADD_STORAGE(number_and_letter);
```

The first **ADD_STORAGE()** that is placed in a class will yield a handle that can be accessed for instance when writing a function for that class. This handle is kept track of so that the storage can be accessed as long as the corresponding object exists. When writing the implementation of a class this handle is then the Pike_fp->current_storage variable.

If a second storage is added to a class, it will be found in an offset from the first storage. Same goes for a third storage if it is added. The programmer will have to keep track of the offsets manually.

Pike_fp->current_storage

```
/*Variable:*/
   Pike_fp->current_storage
```

This variable contains a pointer to the current storage. Current storage means the storage belonging to the object that is currently worked with.

The pointer is used to access storage when writing the implementations of class methods and such. The following example shows how a 'THIS' pointer is defined and used in order to access the object variable **value**.

```
/*Example of usage:*/
   struct data{
     int value;
     int label;
   };
   #define THIS (struct data *)Pike_fp->current_storage
   THIS->value = 14;
```

ADD_FUNCTION()

```
/*Macro:*/
   ADD_FUNCTION(NAME, FUNC, TYPE, FLAGS)
```

ADD_FUNCTION() is a macro to define a Pike level method and map it to a C level function in the module. It takes four arguments. The first argument is the Pike level name of the function (or method as it is called on the Pike level). This is the name a Pike programmer will call the method by. Note that this argument is a string (array of characters). The second argument is the name of the corresponding C level function. That means that when the Pike method with the name equal to the first argument is invoked, Pike will look for the C function with a name equal to the second argument and run that function. The third argument specifies what type of method it is and what arguments the method takes. As it is seen in the example the macro **tFunc()** is used in order to specify the third argument. The fourth argument is a flag that specifies the modifier of the method.

The following example of usage adds a method that has one string as argument and an integer as second optional argument. The return value of the method is of type array.

```
/*Example of usage:*/
   ADD_FUNCTION("my_method", my_method_C_impl,
                tFunc(tString tOr(tInt,tVoid),tArray), 0);
```

A Pike declaration of the method would look like this:

```
   array my_method(string str, int|void number)
```

tFunc

```
/*Macro:*/
   tFunc(arg1 arg2 arg3 ... argn, ret)
```

This macro specifies a function. The arguments of the macro specify the type of the specified Pike method's arguments and return value respectively. Note here that all arguments of the specified function are listed on a row separated only by a space. After the comma comes the type of the return value of the specified Pike method. The example below specifies a method that takes an **int** as first argument, a **string** as second argument, and has a third optional argument of type **int**. The return type is **int**. Note that the Pike types are meant here.

```
/*Example of usage:*/
   tFunc(tInt tString tOr(tInt,tVoid), tInt)
```

The Stack

All arguments to, and return values from, underlying C functions are passed through the stack. The elements on the stack are svalue structs so that all kinds of data structures can be placed "on" it.

Pike_sp

Pike_sp is the global stack pointer. Pike_sp[0] always points to the top of the stack. Pike_sp[-1] is the top element on the stack, Pike_sp[-2] the element below, and so on. The example shows a typical fetching of an integer from the top element of the stack.

```
/*Example of usage:*/
   int number = Pike_sp[-1].u.integer;
```

When a C level function is called and the arguments are passed to it by the stack, it can find its first argument in Pike_sp[-args], its second in Pike_sp[1-args] and its n'th argument in Pike_sp[n-1-args]. This means that argument i is found in Pike_sp[i-1-args] for every 1<=i<=args. The elements of **Pike_sp** are of type **struct svalue**.

svalue

```
/*Definition of data type svalue:*/
   union anything
   {
     INT_TYPE integer; /* Union initializations
                          assume this first. */
     struct callable *efun;
     struct array *array;
     struct mapping *mapping;
     struct multiset *multiset;
     struct object *object;
     struct program *program;
     struct pike_string *string;
     struct pike_type *type;
     INT32 *refs;
     struct ref_dummy *dummy;
     FLOAT_TYPE float_number;
     struct svalue *lval;        /* only used on stack */
     union anything *short_lval; /* only used on stack */
     void *ptr;
   };

   struct svalue
   {
     unsigned INT16 type;
     unsigned INT16 subtype;
     union anything u;
   };
```

The **svalue** is a data structure to store references to any data type that a programmer might need to pass through the stack.

Interacting with the Stack

args

Usually the C level function that correspond to a Pike level method are invoked with the argument **args**. **args** is an integer that specifies how many

'arguments' the C level function should take. Note here that when speaking of 'arguments' in this context we do not mean the actual arguments of the C function, but the arguments of the Pike method that it implements. Each underlying C function has only one actual argument and that is the **args** parameter. The **args** top items on the stack then is the data that the C function has to fetch and use. In other words, what the args parameter actually does specify is with how many arguments the overlying Pike method was invoked. For the module programmer **args** is then number of items that his underlying C function has to remove from the stack.

Pop Functions

```
/*Prototype:*/
   void pop_n_elems(INT32 elems);
```

pop_n_elems() is a function that pops n elements from the stack. Normally this function is called at the end of a C level function that has 'arguments' on the stack. The most usual call is:

```
/*Example of usage:*/
   pop_n_elems(args);
```

This pops the top **args** elements from the stack. If the stack has not been altered since the function was called then that is the same as removing the 'arguments' from the stack. There is also a function called **pop_stack()**.

```
/*Example of usage:*/
   pop_stack();
```

This function can be called instead of **pop_n_elems()** if top element on the stack is to be removed.

Push Functions

Return values are pushed to the stack. Since Pike has a lot of different data types there are a lot of push functions. They all work more or less the same. What they do is that they take the parameter and push it to the stack. The call to push a variable called **arg** of type **foo** would be **push_foo(arg)**. Two examples of push functions are listed below:

```
/*Example of usage:*/
   push_int(1378);
```

push_int() pushes an integer on the stack. The example pushes 1378 on the top of the stack.

```
/*Example of usage:*/
   struct pike_string *str = make_shared_binary_string("",0);
   push_string(str);
```

push_string() pushes a **pike_string** to the stack. The example shows how the empty string is pushed to the stack.

Pike_error()

This function is used to throw exceptions when errors occur. A call to **Pike_error()**, like the one in the example, raises an exception. The argument is a message which is printed to standard error if the exception is not caught.

```
/*Example of usage:*/
    Pike_error("Too many arguments in call to method");
```

The CMOD API

Overview of This Tutorial

This tutorial describes the CMOD interface in Pike. It contains descriptions of the most common CMOD keywords and lists some examples that illustrates how these keywords are used. The words printed in bold in this tutorial have some programming specific meaning. It can be a function, method, class, variable, macro or CMOD keyword.

INIT - keyword

INIT is used to set an initialization callback. Each class ought to have an **INIT**. What stands in between the curly brackets of **INIT** is what will be done upon initialization of an object of that class. In the example below two object variables called **number_of_iterations** and **current_string** are assigned their initial values.

```
/*Example of usage:*/
   INIT{
     THIS->number_of_iterations = 0;
     THIS->current_string = "";
   }
```

EXIT - keyword

EXIT is used to set an exit callback. What stands in between the curly brackets of **EXIT** is what will be done just before an object of that class is being garbage collected. In the example memory allocated and held as an object variable is freed before the object is garbage collected.

```
/*Example of usage:*/
   EXIT{
     free(THIS->buffer);
   }
```

PIKECLASS - keyword

This keyword is used to add a Pike class. It is followed by the name of the class that is to be added. In the example, a class called **MyClass** is added.

```
/*Example of usage:*/
   PIKECLASS MyClass{}
```

PIKEFUN - keyword

PIKEFUN adds a method to the class. It is followed by the corresponding Pike prototype of the method that is to be added. The whole implementation of the method is written in between the curly brackets. The example shows how a method named **my_method()** is added.

```
/*Example of usage:*/
   PIKEFUN array my_method(string arg1, int|void arg2){}
```

my_method takes a string as first argument, has an optional second integer argument, and returns an array. Note here that all data types are Pike types. That is, the type of the method and the types of the arguments are all specified using the Pike data type keywords.

CVAR - keyword

CVAR is used to declare object variables for internal use. **CVAR** is followed by the C declaration of the variable. **CVAR** provides storage for the variable whose declaration follows right after. The example statement placed in a class would result in an object global integer variable named **number**. **number** would then be accessed using the **THIS** pointer.

```
/*Example of usage:*/
   CVAR int number;
```

THIS - keyword

THIS is a pointer to the current object. It is used mainly to access items within the object such as object variables. In the example an object variable **number** is incremented by five.

```
/*Example of usage:*/
   THIS->number += 5;
```

RETURN - keyword

RETURN returns its argument from the method in whose implementation it is placed to the caller. What this actually means is to first remove **args** elements from the stack and then push its argument to the stack. The example is a return of 0.

```
/*Example of usage:*/
   RETURN(0);
```

Writing a C Module

Overview of This Tutorial

This tutorial contains some information on what a programmer needs to keep in mind when writing a module for Pike using the C or CMOD API. The aim is to describe how C level implementation is to be done, not how Pike level design is done. It is however an attempt to describe the writing of a module both using CMOD and not using CMOD. The descriptions are first of general kind. Later on in this tutorial these descriptions are exemplified by a very small module called 'Rev'.

The document contains some terms that are C or CMOD API specific. Therefore it is recommended to read the documentation of the C API and the documentation of the CMOD API before or in parallel with this tutorial.

The Files of a Module

A C/CMOD module normally consists of the following files

1. The C/CMOD code files
2. A configure.in file
3. A Makefile.in file
4. A testsuite.in file

This is the minimum set of files needed in the source of a module. The C/CMOD code files can be one or more files. They contain the whole C/CMOD implementation of the module. The configure.in file is a file for the configure script. The configure script is responsible for generating the global Makefile. In configure.in conditions and tests are written for when and if the module is to be build.

The Makefile.in is a file needed to make additions to the global Makefile so that the module is built when the Pike is built. Each Makefile.in contains a list of the targets needed in order to make that specific module.

Getting Started

In order to write a new C module for Pike the first thing we need to do is to add a new module to our Pike source tree. To do that we need to create a new directory for the module in our module tree. In that directory we need a skeleton for our new module. A skeleton in this context means a working set of the four files mentioned in the previous section. A skeleton that contains the files [1] necessary and some guiding instructions [2] for getting started can be found on the Pike web site. There is also a module with a hello world application[3], that compiles out of the box and can serve as a guiding example.

Once the skeleton files are added and adjusted so that the new module ought to exist the Pike has to be reconfigured and recompiled. After that, if our module exists we can start to write C/CMOD code.

Pike Level and C Level

One way to think of a module, is to divide it into Pike level and C level. The items on the Pike level are abstract entities through which the Pike programmer is interfaced. The C level on the other hand contains the implementation of all Pike level items. That means that all Pike methods that belong to a C module in reality are C functions invoked through a Pike level interface. Therefore writing the C code of a module can be divided into two major steps.

First, the Pike level interface needs to be defined. The module programmer has to define the components of the module. In other words, he has to define the classes, methods, constants, and variables that his module is going to contain. Where needed he also has to map the components to the C level item (in most cases a function) that will represent the implementation of that component.

When the Pike interface has been defined, what we have is stubs for the whole module. This is where the second major step starts: to turn these stubs into a working module. In plain English: the code for the C functions that are the implementation of the defined Pike methods has to be written.

[1] http://pike.ida.liu.se/projects/docs/cmods/skeleton.tgz
[2] http://pike.ida.liu.se/projects/docs/cmods/steps.xml
[3] http://pike.ida.liu.se/projects/docs/cmods/hello.tgz

The following two sections describe more in detail how these two steps are conducted. There are major differences depending on if we use CMOD or not, that is why both sections are divided into two parts.

Define the Pike Level Interface

Without CMOD

Let us assume we want to write a new C module and that a Pike level design of that module already has been done. Let us also assume we want to do it the old fashion way and not use CMOD. As mentioned in the section above we need to define our Pike level interface. The question is how is that done? The keyword here, if we are not using CMOD, is the **pike_module_init()** function. Even though it might not be entirely correct from a technical point of view, we can say that the whole module with all classes and methods is defined in this function. The sequence chart in figure 5.1 illustrates what happens when a module is initialized. As it can be seen in the chart the initialization is so to speak the definition of the Pike level interface since each component in the module is added here.

To add a class **start_new_program()** and **end_class()** calls are placed in the **pike_module_init()** function. These two calls when used together add an empty class to the module.

If we want to add a method to that empty class we use the macro **ADD_FUNCTION()**. We place the call to **ADD_FUNCTION()** somewhere in between the calls to **start_new_program()** and **end_class()** of the class that we want the method to belong to.

To add private object variables the macro **ADD_STORAGE()** is useful. Variables that need to be kept in a storage, are those that need to be 'remembered' over several function calls. In other words, variables that in some way represent the inner state of an object need to be kept in storage. To add storage to a class we need to place a call to **ADD_STORAGE()** somewhere in between the call to **start_new_program()** and the call to **end_class()** of the class that we want the storage to belong to. The first call to **ADD_STORAGE()** after a call to **start_new_program()** will yield a handle. Storages added by other calls will lie within an offset of this storage. The module programmer will have to keep track of that offset manually if he chooses to add more than one storage. Therefore, an easy way to handle multiple object variables without adding more than one storage is to pack

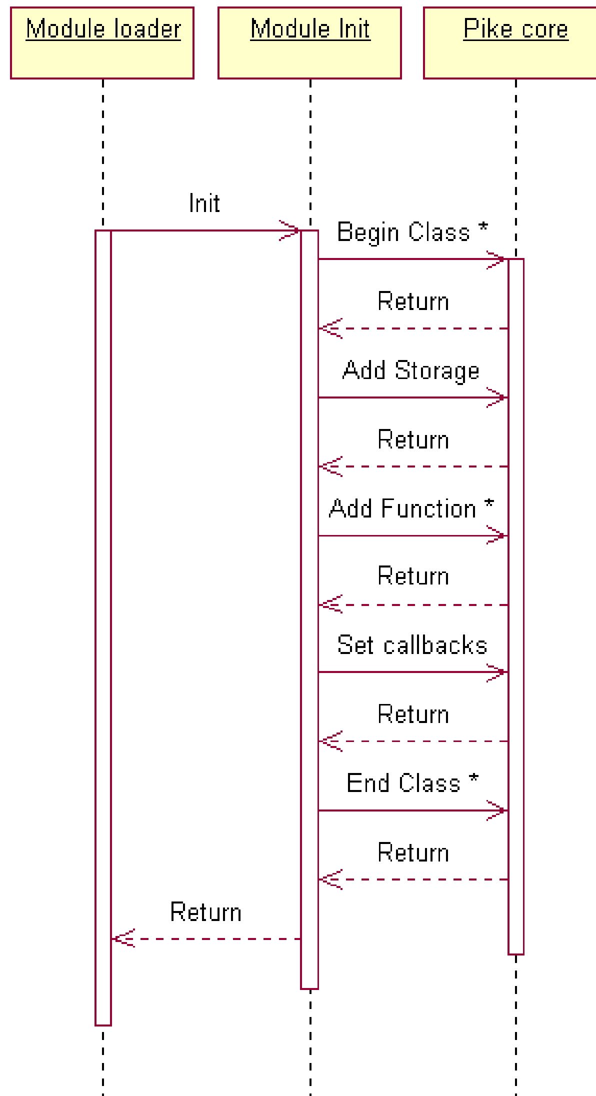

Figure 5.1: Sequence chart, initialization of a module

them all into one struct on the C level. Then this struct is made the first and only added storage of the class. Every object variable can then be accessed using the **Pike_fp->current_storage** pointer. So in order to add all private object variables, we first create a struct that contains all the variables needed. Then place a call to **ADD_STORAGE()** with that struct as only argument, somewhere in between the corresponding **start_new_program()** and **end_class()** .

Last in each definition of a class we need to set the initialization and exit callbacks. The init callback is set by adding a call to **set_init_callback()**. The exit callback is set by adding a call to **set_exit_callback()**.

In order for the now defined Pike level interface to compile, stubs for the functions that will be written later on, have to be added. All C functions that represent the implementation of a Pike method need to look like this:

```
static void function_name(INT32 args).
```

That is just the way the C API works. If we have defined a method (using **ADD_FUNCTION()**) that is specified to be implemented by the C function **function_name()**, a function like the one just mentioned will be requested.

Further all functions that have been set as init- or exit callback also need to exist to make the Pike level interface compile.

With CMOD

Using CMOD when writing the module makes it a little easier to define the Pike level interface. What needs to be done is in abstract:

- Add classes using a **PIKECLASS ......** statement. Everything that belongs to that class is then placed in between its curly brackets.

- Use **PIKEFUN ...(...)...** to add a method to a class. Place it somewhere in between the curly brackets of the **PIKECLASS** statement of the class that the method is supposed to belong to.

- Use **CVAR ...** to add a private object variable. Place the variable declaration in the class that it is supposed to belong to.

Note that even though we use CMOD, the functions **pike_module_init()** and **pike_module_exit()** need to exist. They can be kept empty though.

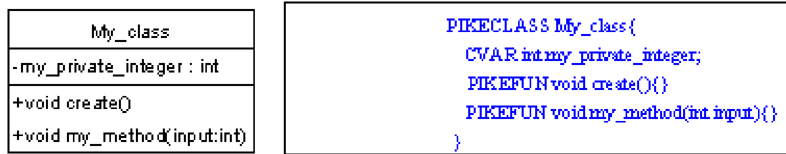

Figure 5.2: A class modeled with UML and its CMOD correspondence

As we can see CMOD is very convenient since the design can easily be translated to an implementation. If we for instance have a UML design model the mapping from the most basic UML components to CMOD is straight forward. As it can be seen in figure 5.2, UML components in this particular case have intuitive CMOD equivalents, so the translation is easy.

Write the Functions

Once the classes and methods have been defined and mapped one still needs to do the 'real' implementation of the module. Since one common reason for writing a C module is to take advantage of already existing C libraries, much of the implementation has, in many cases, already been done. However, what always needs to be done is to fit the interface of the C library to the Pike interface of the module in a good way. Again, there are some differences what needs to be done, depending on if we are using CMOD or not.

Without CMOD

We have stubs for the functions, so then we can go on writing the code. What needs to be done in each function that implements a method is in abstract:

1. Read the parameters from the stack. The argument args specifies how many parameters that should be read. Pointers to the parameters can be found in **Pike_sp[-args]**, **Pike_sp[1-args]**, **Pike_sp[2-args]** and so on..

2. Make function calls, assignments, and so on until we have the desired result, i.e. until the return value is right and all variables in the storage have their desired value. In other words, write the actual program.

3. Pop the parameters from the stack and push the return value from 2) to the stack.

As we can see all input to and output from the functions is passed through the stack.

With CMOD

If we are using CMOD, then we have our skeleton to start writing the code. That is because CMOD generates some code that accesses the stack for us, so we can simply go ahead and write the implementation of the methods in plain C. In the end of each method (or should we call it function, since that is what it actually is) a **RETURN** statement should be placed. That **RETURN** removes the arguments from the stack and pushes the return value to it.

There are however two special cases where some extra code has to be written. The first one is when we have a method with optional arguments. Then CMOD does not manage the stack for us, instead we will have to check the args parameter and if the arguments exist, fetch them from the stack, just the same way as it is done when writing modules without CMOD.

The other special case is when we implement a method of type void. At the end of that method we can not use **RETURN** since there is nothing to return. Instead a **pop_n_elems(args)** needs to be done in order to keep the stack up to date.

Error Handling

One important thing the programmer that implements a module has to take care of is error handling. As we said earlier the module implementor's main task is to make the pieces fit each other. In order for that to work the input (and output) parameters must be valid. Since Pike does not provide adequate type checking, each module has to do that itself. The absolute minimum of error handling is to check the arguments in the implementation of each method.

Argument Checking without CMOD

As stated above, arguments must be checked in each method. What needs to be checked is that the number of arguments that the method is invoked with is valid, and that all the arguments that the method is invoked with have

the right type. If anyone of these checks give a negative result an exception should be thrown.

Argument Checking with CMOD

CMOD provides automatic type checking and therefore reduces the amount error checking code a programmer must write manually. CMOD generates code for argument checking in the implementation of the methods. The types of the arguments and the number of arguments with which the method is invoked is checked, and if something is wrong, the Pike exception handler is invoked.

The special case is if we have methods with optional argument(s). Then the arguments need to be checked manually in the same way as when CMOD is not used.

Throwing Exceptions

If an error of exceptional kind occurs, then an exception should be thrown. But before we can invoke the exception handler, we need to be certain that we do not hold any memory that will not be released once the exception handler is called. If we do not release all such memory before we call the exception handler there is a great risk that the method leaks memory. If the exceptional event in any way broke the inner state of an object, then it is probably a good idea to restore it before invoking the exception handler. When we have assured our selves that it is safe to call the exception handler, it can be called using **Pike_error()**.

Init and Exit Routines

To assure that the initial state of an object always is the desired, each class need to have a proper initialization routine. It is not safe to only set initial values in the **create()** method of a class since this method is not certain to be run. This is the case when a class is being inherited. Therefore, initialization of classes is required to be done in the initialization callback. So we need to add an initialization for each class in the module.

If needed, proper exit routines should also be written. Their task is to do cleanup before the object is garbage collected.

An Example - The Rev Module

This is a very simple C module. This text contains extracts of its code. The whole source code file can be found at the end of this appendix. A CMOD implementation of this module does not exist but some extracts of the code has been translated to CMOD.

The module has one class called ReversedString. This class has two methods: the mandatory **create()** method, and a method **write()**. The ReversedString class takes the 49 first characters of a string, reverses the string, and stores it as an object variable. A call to the method **write()** writes the content of the reversed string **nr_of_times** times to the screen. As we can see the module is totally useless, it is just an example to illustrate how a module is written.

Define the Pike Level Interface

Without CMOD

As stated earlier Pike needs the function **pike_module_init()** and the function **pike_module_exit()** to recognize the module as a module.

Since the ReversedString module is that simple the **pike_module_init()** function was just the following few lines:

```
struct data{
   char buf[50];
}

void pike_module_init(void)
{
   start_new_program();
   ADD_STORAGE(struct data);
   ADD_FUNCTION("create", reversed_string_create,
                 tFunc(tString, tVoid), 0);
   ADD_FUNCTION("write", reversed_string_write,
                 tFunc(tInt, tVoid),0);
   set_init_callback(init_reversed_string);
   set_exit_callback(exit_reversed_string);
   end_class("ReversedString",0);
}
```

The effect of each line of code is described below.

```
start_new_program();
```

Tells the system that this is the beginning of a new class.

```
ADD_STORAGE (struct data);
```

Creates storage for a struct data. This is the first **ADD_STORAGE()** in the class. As a consequence of this the struct data is accessed through the global variable **Pike_fp->current_storage** which points directly to the struct.

```
ADD_FUNCTION("create", reversed_string_create,
              tFunc(tString, tVoid),0);
```

Adds the Pike level **create()** method, or in other words the constructor, of the class and maps it to the **reversed_string_create()** C level function. The first parameter of **ADD_FUNCTION()** is the name of the Pike level method, the second parameter is the underlying C level function to which it is mapped, that means the low level function that is called when the high level method is invoked. The third parameter specifies what data types the Pike level method's arguments and return values are. The fourth argument specifies whether the Pike level method is static or public or...

```
ADD_FUNCTION("write", reversed_string_write,
               tFunc(tInt, tVoid), 0);
```

This line adds the **write()** method. It works in the same way as the addition of the create method.

```
set_init_callback(init_reversed_string);
```

This line sets the init callback. Each time a ReversedString object is initialized **init_reversed_string()** will be run.

```
set_exit_callback(exit_reversed_string);
```

This line sets the exit callback. The **exit_reversed_string()** function will be run each time a ReversedString object is about to be garbage collected.

```
end_class("ReversedString",0);
```

This function call is the terminating mark for the class that started at the **start_new_program()** call ends here, and that the name of that class is 'ReversedString'.

So that was the whole **pike_module_init()**. The module also needs a **pike_module_exit()** function. In this case there is nothing to be done in that function. Still it needs to exist, and that is why there is such an empty function.

With CMOD

The ReversedString realized with CMOD instead looks as follows:

```
PIKECLASS ReversedString{
   CVAR char reversed_string[50];
   PIKEFUN void create(string input){
       /*write the implementation of
         create here */
   }
   PIKEFUN void write(int nr_of_times){
       /*write the implementation of
         write here */
   }
   INIT{
      /*write the init callback here*/
   }
   EXIT{
      /*write the exit callback here*/
   }
}

void pike_module_init(){}
void pike_module_exit(){}
```

As we can see the declaration is straight forward and do not need that much explanation. The **CVAR** call results in that storage is added for a fifty character long **char** array. **PIKEFUN** adds the method that follows right after the keyword. **INIT** and **EXIT** add the init- and exit callback respectively. Finally, **pike_module_init()** and **pike_module_exit** need to exist in order for the module to be recognized as a module, they can be kept empty though.

Now, having handled the 'mandatory' parts, let us take a look at the functions that implements the functionality of the module.

Write the Functions

This section contains some description of how the Rev module works.

Without CMOD

`reversed_string_create()`

This is the function that will be run upon creation of a ReversedString object, because we set that earlier in **pike_module_init()**. It does the following:

1. Checks whether the constructor was called with one argument, i.e. if args == 1, throws an exception if not
2. Checks whether the only argument is a **pike_string** , throws an exception if not.
3. Takes the 49 first characters, using the **Pike_fp->current_storage** pointer, reverses the order of them, and stores them in storage.
4. Removes one argument from the stack, i.e. does a **pop_n_elems(args)**

`reversed_string_write()`

This is just a function to generate some output in order to test if everything in the create function and the storage works. It does the following:

1. Checks the argument, i.e. check that args == 1 and that the argument is an **int**.
2. If the argument check succeeded, then it fetches the argument from the stack, and reads the content of the storage.
3. Prints the content of the storage on the screen

The callback functions

The init and exit callback functions exist in this module, but since nothing needs to be done in them they are left empty. These functions need to take a **struct object*** as argument, or else warnings will be generated when the module is compiled.

With CMOD

```
PIKEFUN void create(string input){}
```

This is where the implementation of the **create()** method would be written. What it would do is:

1. Read the 49 first characters from **input**, reverse the order of them and copy them to **reversed_string**.
2. Pop the parameters from the stack,

```
PIKEFUN void write(int nr_of_times){}
```

This is then where the implementation of **write()** would be written. What it would do is:

1. Print the content of **reversed_string nr_of_times** times
2. Pop **args** elements from the stack

INIT and EXIT

In this case neither INIT nor EXIT need to contain anything, since nothing needs to be done upon initialization or exit of a **ReversedString** object. Therefore they are kept empty.

```
/**
 *File: reversed.c
 *Description: The code of the Rev module */

#include "global.h"

#include "interpret.h"
#include "svalue.h"
#include "stralloc.h"
#include "array.h"
#include "pike_macros.h"
#include "program.h"
#include "stralloc.h"
#include "object.h"
#include "pike_types.h"
#include "threads.h"
#include "dynamic_buffer.h"


/* This must be included last! */
#include "module_magic.h"

/*PLEASE NOTE THAT this code is not that good from a programming point
  of view, it is only intended to serve as an example of a simple
  C module*/

struct data{
  char buf[50];
};

static void reversed_string_create(INT32 args){
  struct pike_string *input;
  char *s;
  char tmpbuf[50];
  char retbuf[50];
  int i;

  int j = 0;

  if (args != 1){
    Pike_error("Method ReversedString->create() takes ONE argument");
  }


  /* a little bit messy, assigns s a pointer to the storage of the reversed string*/
  s = (char *)&(((struct data *)Pike_fp->current_storage)->buf);
  input = Pike_sp[-args].u.string;

  strncpy((char*)&tmpbuf, (char *)(&input->str),49); /* copy the input string to a buffer*/
  tmpbuf[49]='\0';
  /* find the terminating null-character*/
  for(i = 0; i < 50; i++){
    if (tmpbuf[i]=='\0'){
      break;
    }
  }
  /*reverse the string, result in retbuf[]*/
```

```
  while (i >= 0 && j < 50){
    i--;
    retbuf[j] = tmpbuf[i];
    j++;
  }
  retbuf[j] = '\0';
  /* store the reversed string, and pop arguments from stack*/
  strcpy(s, (char *)&retbuf);
  pop_n_elems(args);

}

static void reversed_string_write(INT32 args){
  int i = 0;
  int nr_of_times;

  if (args != 1){
    Pike_error("Method ReversedString->create() takes ONE argument");
  }
  nr_of_times = Pike_sp[-args].u.integer;
  for(i = 0; i < nr_of_times; i++){
    printf("%s",(char *)&(((struct data *)Pike_fp->current_storage)->buf));
  }
  pop_n_elems(args);
}

static void init_reversed_string(struct object *o){}

static void exit_reversed_string(struct object *o){}

void pike_module_exit(void) {}

void pike_module_init(void)
{
  start_new_program(); /*begin of class*/

  ADD_STORAGE(struct data);
  /*add the methods create and write*/
  ADD_FUNCTION("create", reversed_string_create, tFunc(tString, tVoid), 0);
  ADD_FUNCTION("write", reversed_string_write, tFunc(tInt, tVoid), 0);
  set_init_callback(init_reversed_string);
  set_exit_callback(exit_reversed_string);
  end_class("ReversedString",0);
}
```

Appendix B - Bz2 Module Requirements Specification

Introduction

Purpose

The product whose software requirements are specified in this document is a in C written interface for Pike to access the library libbzip2. The final product will be a module to handle Bzip2 compression and decompression in Pike. However, since a corresponding C library already exists, the scope of the product will be to implement a glue between Pike and libbzip2.

Intended Audience

The type of reader that this document is intended for is mainly a Pike developer. The document could also be of interest for a Pike programmer.

Overall Description

Product Functions

The system is to implement classes and methods for handling Bzip2 compression in Pike. The functionality that those shall provide the programmer with is in abstract:

1. read data from a Bzip2 file
2. write data to a Bzip2 file
3. other normal file operations on Bzip2 files such as open, close and end of file.
4. compress a data stream using the Burrows-Wheeler algorithm implementation of libbzip2
5. decompress a data stream compressed with the Burrows-Wheeler algorithm implementation of libbzip2

User Classes and Characteristics

Basically there is only one user class that will anticipate this product and that is the Pike programmer. Since he/she knows how to use Pike, he/she should have no problem reading the manual and understand how to use the module.

Operating Environment

The software will be a part of the Pike system. It must therefore peacefully coexist with the rest of the Pike system. The software ought to be as platform independent as the rest of Pike, i.e. run on all hardware and software platforms supported by Pike.

User Documentation

The documents beside this that will be produced for the module are:

1. A Software Design Specification (SDS)
2. A Test Report

Assumptions and Dependencies

The only thing that could affect the requirements in the document is changes in the two things the glue depends on, namely Pike and libbzip2. However changes in Pike and/or libbzip2 that makes the glue incompatible are rather unlikely.

External Interface Requirements

User Interfaces

The user interface will be an Application Programming Interface. This means that the interface simply is a set of Pike classes and their methods. The class hierarchy is described in detail in the Software Design Specification.

System Features

Read Data from File

The user should be able to read uncompressed data from a Bzip2 file as if it was a normal file. That means that the reading method should follow the 'standard' for file reading in Pike.

Write Data to File

The user should be able to write uncompressed data to a Bzip2 file as if it was a normal file. The result should be a new Bzip2 file. The writing method must of course also follow the 'standard' for file writing in Pike.

Other Common File Operations

These are the other methods for Bzip2 file operations such as open and close, end of file, file length and so on. The methods should be as similar as possible to the common file object operations in Pike, in order to make the module as usable as possible for a Pike programmer.

Compress a Data Stream

The module is to support arbitrarily large/small packages of uncompressed data being fed to a stream for compression. The user then has the power to decide when the stream should be initialized, fed with data, flushed, and terminated.

Decompress a Data Stream

The module is to support arbitrarily large/small packages of compressed data being fed to a stream for decompression. When fed with a piece of compressed data, the module should decompress it as far as possible. This will generate a fragment. The resulting fragments concatenated should then be the same as the data that was once compressed.

Memory Management

The module is to provide a high level interface so that the Pike programmer using the module does not need to care about allocating and deallocting memory for decompression and compression. The interface should automatically manage all memory.

Other Nonfunctional Requirements

Software Quality Attributes

As stated in the environments section the module must not render any platform dependency what so ever. After having added the module, building Pike must work just as fine as it did before the module was added.

Appendix C - Bz2 Module Design Specification

Introduction

The purpose of this document is to specify the design of a libbzip2 interface for Pike. The document is to serve as the guideline when implementing the interface. It is written in such a way that a person not directly involved in the project, but with some knowledge about Pike, should be able to read it and then understand how the interface is implemented. The scope of the document is to describe the methods and classes that are implemented in order to provide Pike with the functionality described in the Software Requirements Specification. The intended audience is a Pike developer, or a Pike user interested in how this specific module, or C modules in general, are interfaced.

Design Considerations

The implementation will be done using the raw C module API in Pike. Using the CMOD API in Pike would perhaps have been an easier way. But I thought that doing the raw C implementation will make it easier for me to understand how CMOD works. In order to analyze and document both the CMOD and C API, I kneed to know how both work.

Since the API of libbzip2 is almost identical to the API of zlib, and since there is already an interface for the zlib (the Gz module), effort has been put in making the Gz module and the new Bz2 module similar from a user perspective. That is methods and classes that exist in the Gz module have corresponding methods and classes in the Bz2 module. Some more methods have also been added in order to make the Bz2 interface more user friendly.

Development Method

The method for developing this design has more or less been reading. Studying the zlib and especially the libbzip2 documentation carefully rendered some good ideas for what tasks the future Pike module should be able to

perform. Then these ideas were matched against the libbzip2 API in order to see how this was to be realized. After that a skeleton for the module was built in order to establish the final design.

Detailed System Design

The following section is a listing of the components that the interface consists of and their composition and responsibilities. In other words, this section describes in detail the Pike level interface to be implemented. Note that this is a description of the Pike level design. It says nothing about how this should be implemented on the C level, because that is rather a matter of implementation than of design.

Classification

A libbzip2 - Pike interface

Resources

The libbzip2 library. Everything worth knowing about libbzip2, such as for instance documentation and source code can be found on the libbzip2 website. The URL is: http://sources.redhat.com/bzip2

Responsibility

The interface provides the Pike language with the functionality of the libbzip2 library. It makes the functionality of libbzip2 accessible in Pike through a high level interface. High level in this context means that it takes care of some low level problems (mainly memory management) that a C programmer using libbzip2 has to deal with.

Composition

The module consists of three components namely

1. The Inflate class
2. The Deflate class

3. The File class

Component 1

Classification
The Inflate class

The Inflate Class - Responsibility
The responsibility of the implementation of this class is to provide methods for inflating (unpacking) data. The data fed to an Inflate object is a stream of Bz2 compressed data. The object is able to handle data being fed to it in arbitrarily large/small chunks.

The Inflate Class - Composition
The composition of the Inflate class is the following:
Class name: Bz2.Inflate
Class method 1: void create()
Class method 2: string inflate(string data)

The Inflate Class - Responsibility of the Class Methods
Class method 1:
This is the constructor of the Inflate class. It creates an Inflate object. That means that it takes care of all necessary initialization so that the inflating of a data stream can be performed.

Class method 2:
This is the method that is used to do the inflating. The argument data is a string of compressed data from a data stream. The method decompresses (inflates) this string as much as possible and returns the unpacked data in a string. If no data was decompressed, the empty string "" is returned, and if something goes really wrong, the method returns 0. Data that could not be decompressed because the end of the compression block was still missing, is stored internally and a new attempt to decompress it is done the next time this method is called and fed with more data.

Component 2

Classification
The Deflate class

The Deflate Class - Responsibility
The responsibility of the implementation of this class is to implement methods for deflating (packing) data. It provides methods for streaming Bz2 packing. The data can as usual be fed in arbitrarily large/small strings.

The Deflate Class - Composition
The composition of the Deflate class is the following:
Class name: Bz2.Deflate
Class method 1: void create(int(1..9)|void block_size, int(1..250)|void work_factor)
Class method 2: void feed(string data)
Class method 3: string read(string data)
Class method 4: string finish(string data)
Class method 5: string deflate(string data,int(0..2)|void mode)
mode is element of BZ_RUN, BZ_FLUSH, BZ_FINISH

The Deflate Class -Responsibility of the Class Methods
Class method 1:
This is the constructor of the deflate class. It creates a Deflate object. That means that it takes care of all necessary initialization so that the deflating (compressing) of a data stream can be performed. The first optional argument for the method sets what block size (the larger the blocks, the higher the compression) the deflate methods will use when compressing. The block size actually used is 100000*blocksize bytes. The second optional argument specifies how quickly the compression library will turn to a fallback algorithm when running into difficulties during compression.

Class method 2:
This method feeds the string data to the internal buffer that is to be

deflated. Errors are rather unlikely and will be handled by throwing exceptions. The most likely error to occur is a memory error, which means that it runs out of memory.

Class method 3:
This method feeds the string data to the internal buffer that is to be deflated. Then it compresses the whole buffer, returns the compressed data as a string, and empties the buffer. What it does not do is to terminate the stream, it just does some partial packing. It returns the compressed chunk of data in a string. If there is no data to return it returns the empty string "".

Class method 4:
This method feeds the string data to the buffer and compresses the whole buffer, just like read. The difference between read and finish is that finish also adds an end of data stream marker to the compressed data and resets the Deflate object. It terminates the old data stream and initializes a new one so that more deflate calls to this object can follow. The return value is then as usual the packed data as a string. The method returns 0 if there is no data to return.

Class method 5:
This method is nothing more than method 2, 3, and 4 in one. The first argument is the string that is to be compressed, the second argument is used to specify which one of feed, read, and finish that will be run. The second argument is optional and if it is omitted finish will be run by default. The constants that can be used to specify the mode are taken directly from the libbzip2 library. They are defined on the top level of the Bz2 module. The method returns the resulting string. If the method is run in run mode, it does a feed and returns the empty string. If it is run in flush or finish mode the returned string will be the same as it would have been if read() or finish() would have been run instead.

Component 3

Classification
The File class

The File Class - Responsibility
The responsibility of the implementation of this class is to provide methods for handling Bzip2(*.bz2) files. It provides methods to read from and write to Bzip2 files as if they were normal binary files.

The File Class - Composition
The composition of the Deflate class is the following:
Class name: Bz2.File
Class method 1: void create(void)
Class method 2: int(0..1) read_open(string file)
Class method 3: int(0..1) write_open(string file)
Class method 4: int(0..1) open(string file, void|string mode)
Class method 5: int(0..1) close()
Class method 6: int|string read(int len)
Class method 7: int write(string data)
Class method 8: int(0..1) eof()

The File Class - Responsibility of the Class Methods

Class method 1:
This method is the constructor of the class. It takes no arguments and does only the basic initialization.

Class method 2:
This method opens a Bzip2 file for reading and associates it with the object. The argument is a string containing the name of the file that should be opened. The return value is one if the open was successful and 0 otherwise.

Class method 3:
This method opens a Bzip2 file for writing and associates it with the object. The argument is a string containing the name of the file that should be opened. The return value is one if the open was successful, 0 otherwise.

Class method 4:
This method opens a Bzip2 file for either writing or reading and associates it with the object. The first argument is a string specifying the name of the file that should be opened. The second optional argument should either be the string "r" or "w" depending on if the file should be opened for reading or writing. The mode argument will by default be set to "r" if it is omitted. Due to the methods read_open and write_open this method is redundant, the only reason for keeping it is that this is the way files are opened in the Gz module. The method returns 1 upon success otherwise 0.

Class method 5:
This method closes the file associated with the object. It returns 1 if the close was successful, 0 otherwise.

Class method 6:
This method reads uncompressed data from a Bzip2 file. The argument len specifies how many bytes of uncompressed data that is to be read. The read data is returned as a string. If the read is unsuccessful the method returns 0.

Class method 7:
This method writes the string data to the file associated with the object. The method returns the number of compressed bytes written. It returns 0 if the write is unsuccessful (or if the method was invoked with len=0).

Class method 8:
This method returns 1 if end of file is reached, 0 otherwise.

Note That

All three components are implemented using the functions and data structures of the libbzip2. For more details on the classes and there implementation see the source code file 'libbzip2mod.c'.

Appendix D - Bz2 Module Test Report

Introduction

This paper is a documentation of the tests that were performed on the Bz2 module during its implementation. Due to lack of time and the limited scope of the module less effort was put in the test process. That was because the only thing more rigorous testing might had contributed to the project, if anything at all, is higher quality of the module.

Test Items

The items to be tested were the components of the Bz2 module as they are described in the design specification. Every subcomponent became so to say one test item. Anyway, the items are:

Test item 1: The 'Inflate' class
Item 1.1: The 'create' method
Item 1.2: The 'inflate' method

Test item 2: The 'Deflate' class
Item 2.1: The 'create' method
Item 2.2: The 'feed' method
Item 2.3: The 'read' method
Item 2,4: The 'finish' method
Item 2.5: The 'deflate' method

Test item 3: The 'File' class
Item 3.1: The 'create' method
Item 3,2: The 'read_open' method
Item 3.3: The 'write_open' method
Item 3.4: The 'open' method
Item 3.5: The 'close' method
Item 3,6: The 'read' method

Item 3.7: The 'write' method
Item 3.8: The 'eof' method

Test Process

Test Strategy

There is one keyword that kind of summarizes the whole test process namely 'continuously'. While the module evolved, subcomponent by subcomponent, it was also tested subcomponent by subcomponent. That means that when a method was written it was also immediately tested as far as possible. Once that method fulfilled its test cases a new subcomponent was made an tested. With time the subcomponents formed component, and so entire components were tested and finally the entire module. So the test strategy was bottom up beginning with the smallest subcomponents and working the way up.

Tools

One tool that was for test was 'Hilfe'. 'Hilfe' stands for 'Hubbes incremental LPC front end' and is a Pike front end. The advantage with using Hilfe is that one can run the current Pike without making an install. It takes a lot of time to install a Pike and therefore lots of it can be saved by running Hilfe instead.

Another tool that was used was the Bzip2 program. It is used to compress data to *.bz2 files and to extract data from *.bz2 files.

Test Cases

Test Case 1

Tests items: 1.1, 2.1, 3.1
Execution:
- Create one of each object. If successful then the test succeeded.

Test Case 2

Tests items: 1.1, 1.2, 2.1, 2.4
Execution:
- Generate a long string of data
- Create one 'Inflate' and one 'Deflate' object
- Compress the whole string using Deflate->finish()
- Divide the compressed data into small chunks
- Decompress the compressed data chunk by chunk using Inflate->inflate().
- Compare the result to the original data. If it is consistent then the test succeeded

Test Case 3

Tests items: 1.1, 1.2, 2.1, 2.2, 2.3, 2.4, 2.5
Execution:
- Generate a long string of data
- Create one 'Inflate' and one 'Deflate' object
- Divide the string into small chunks
- Compress the in data chunk by chunk using alternating calls to Deflate()->feed() and Deflate()->read(). Finish off with calling Deflate()->finish()
- Divide the compressed data into small chunks.
- Decompress the compressed data chunk by chunk using Inflate->inflate().
- Compare the result to the original data. If it is consistent then the test succeeded
- In order to test item 2.5 replace the calls to feed(), read(), and finish() with deflate(...,Bz2.BZ_RUN), deflate(...,Bz2.BZ_FLUSH), and deflate(...,Bz2.BZ_RUN) respectively.

Test Case 4

Tests items: 3.1, 3.2, 3.5, 3.6

Execution:
- Create a *.bz2 file out of a text file with known content using Bzip2
- Create a 'File' object
- Open the created *.bz2 file for reading using File()->read_open().

- Read the entire file into a string using File()->read()
- Close the open *.bz2 file
- Compare the content of the string to the content of the original file. If it is the same the test succeeded.

Test Case 5

Tests items: 3.1, 3.3, 3.5, 3.7

Execution:
- Create a 'File' object
- Open a *.bz2 file for writing using File()->write_open()
- Create a long string of data
- Write the entire created string to the open file using File()->write()
- Close the open *.bz2 file
- Decompress the created *.bz2 file using bunzip2. Compare the result to the created long string of data. If it is the same the test succeeded.

Test Case 6

Tests items: 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8

Execution:
- Create a 'File' object
- Open a *.bz2 file for writing using File()->write_open.
- Read the content of an input file into a string
- Divide the string into small chunks
- Write the string chunk by chunk to the open file using File()->write()
- Close the open file
- Open the created *.bz2 file for reading using File()->read_open()
- Read a fixed amount of bytes from the open file using File()->read(). After each read append the read data to an output file. Do this over and over until the end of file is reached
- Close the open file
- Compare the input file to the output file. If they do not differ the test succeeded

Quality Testing

Test Robustness

This is the most challenging part of the testing. The whole idea behind testing robustness is to run the program doing everything in one's power to make it crash, and see how the program reacts on that. In the case of the Bz2 module the following was done in order to achieve that:
- Methods were invoked with bad number of arguments
- Methods were invoked with argument of bad type or containing bad data
- Streams were broken calling feed(), read(), finish(), and create respectively inflate() and create() at unexpected times
- Attempts to open already open File objects in new modes were done
- Attempts to write to File objects opened for reading were done
- Attempts to read from File objects opened for writing were done
- Pike programs were exited without closing the open files
- The internal buffers used in the glue were made as small as one byte in order to let it run out of memory.

Look for Memory Leaks

In order to find all memory leaks Pike can be run in dmalloc mode. Dmalloc stands for debug malloc and is a feature that keeps track of all memory allocations and deallocation that a Pike program does when it is run. That way Pike can tell upon termination what memory has been returned and what memory not. By running test programs in dmalloc mode memory leaks that occur during normal program execution will reveal.

It is a little bit more tricky when exceptions are thrown, because then all of a sudden the exception handler is invoked and the execution takes another way. Therefore it is of great importance to release all memory that is held, before throwing an exception in order to avoid memory leaks.

Results

Faults Discovered on the Way

The most common errors that were found were memory leaks and broken pointers. There were also some minor faults with the usage if libbzip2. All found faults were corrected.
The tests described in section 3 and 4 above were all executed, in dmalloc mode, with satisfying results. Test programs that realize these tests can be found in the source directory of the Bz2 module. Since test case 3 covers test case 2 and 1, and since test case 6 covers both test case 4 and 5, the final test programs that were put in the source do tests similar to the test cases 3 and 6. Test cases 1, 2 , 4, and 5 were .used to test subcomponents during development while test case 3 and 6 test entire components.

All in all, both functional testing and quality testing was successfully performed

Current Status of the Bz2 Module

All faults and weaknesses found during testing were corrected. Everything seems to be working fine and the module is now ready for the CVS.