MorphOS Developer Guide

Version 0.4 (2013-01-06)

History

Version	Date	Author	Description
0.1	2011-11-16	A. Geisler	First public release
0.2	2011-12-29	A. Geisler, S. Adamczyk	Second public release
0.3	2012-01-06	A. Geisler, S. Adamczyk, M. Wilkens	Third public release
0.4	2013-01-06	A. Geisler	Fourth public release

Overview

This document is about developing software in C and C++ for MorphOS[1]. The first chapter deals with the MorphOS SDK, the MorphOS platform specific features and the provided tools in general. The MorphOS SDK is available for free on the official MorphOS website. Programming native graphical user interfaces for MorphOS using MUI is the topic of the second chapter. Chapter three describes Reggae, the native MorphOS multimedia framework. The last chapter takes a clother look to the MorphOS memory management and how to write your own startup code.

All texts, source codes, examples and pictures within this document are directly taken from tutorials published at Morphzone's library[2] by several authors. Many thanks to all the guys who spent a lot of time on writing these great tutorials. This document doesn't claims itself to be up to date at any time. If you want to ensure reading the latest version of the tutorials please go directly to Morphzone[3].

If you like or dislike this document, if you find some misspelling, broken formatting or other noticable problems don't hesitate to write an email to:

André Geisler, eliot@exception-dev.de

Please do not send me any questions about contents in this document. For technical questions and discussions please join the forum at Morphzone.

Latest version of this document can found at:
http://www.exception-dev.de/projects/mosprogrammingguide/mos_p_guide_de.html

Additional notes to version 0.1: This is first of public release of the MorphOS programming guide. The "Taglists" section is outdated and the chapter about "Excec Lists" is missing at the moment. They will be updated/added in very near future. More chapters will be added in future releases (e.g. Reggae: MorphOS multimedia framework, An Introduction to MorphOS PPC Assembly), so stay tuned!

Addional notes to version 0.2: The second public release provides updated "Taglists" section, additonal section about "ExecLists" and a new chapter about the MorphOS multimedia framework "Raggae".

[1] http://www.morphos.de
[2] http://library.morphzone.org/Main_Page
[3] http://www.morphzone.org

Addional notes to version 0.3: The third release provides one new chapter about the new MorphOS memory system introduced in MorphOS 2.0 and some small changes in layout. The most noticeable change is the new cover which was designed by Matthias “Aramon” Wilkens. At this point many thanks to Matthias for his great work!
Due the fact that there is now at least one chapter which does not concern to MorphOS programming directly, this guide is renamed from **“MorphOS Programming Guide”** to **“MorphOS Developer Guide”**.

Addional notes to version 0.4: After exactly one year a new version of MorphOS Developer Guide is available. There is a new chapter about writing your own startup code and the chapter about the compiler is updated because of some minor changes in the current SDK.

Contents

List of Figures

Chapter 1

First steps in MorphOS programming

Author: Grzegorz Kraszewski
Source: http://library.morphzone.org/First_steps_in_MorphOS_programming

This is a tutorial for people who want to start writing applications for MorphOS. It is assumed that the reader knows C or C++ language, but has no (or very limited) experience in programmimg for MorphOS or Amiga systems.

1.1 Installation of Software Development Kit and its basic usage

1.1.1 Installing the SDK

The official MorphOS SDK provides a complete environment for creating programs for MorphOS. It contains the following components:

- MorphOS includes (for the MorphOS native API).
- Standard C and C++ library includes.
- MorphOS API documentation and example code.
- Two GCC compilers: 2.95.3 and 4.4.5.
- GCC toolchains (one for each compiler), sets of developer utility programs.
- Scribble, a powerful programmer's text editor.
- Perl scripting language (used by some SDK tools).

The first step of installation is to download the SDK archive from the morphos.net[1]. The SDK is delivered as a LHA archive, which must be depacked before proceeding. The easiest way is to open a context menu for the archive (with the right mouse button in an Ambient window) and choose Extract. After depacking a directory named morphossdk is created with an

[1] http://www.morphos.de

Installer application and a big file named sdk.pack inside. Installation is started by running Installer. The only user choice that is needed here is to choose an installation directory. Then there is some time spent watching the progress bar...

After the installation a system reboot may be needed to update system assigns and paths.

1.1.2 Choosing a Compiler

As mentioned above, the SDK delivers two GCC compilers: the old but trusty 2.95.3 and the modern 4.4.5. There is a script named GCCSelect in the SDK, which allows fast switching between compilers. Just type in a shell window

GCCSelect 2.95.3

or

GCCSelect 4.4.5

to change the current compiler. GCCSelect works by making symbolic links to the proper version of GCC and it's tools, so the compiler is always called as **gcc** or **g++**, regardless of the version chosen currently.

Which one to choose? It depends on the code compiled and other constrains. Here is some guidance:

- 2.95.3 version compiles faster and consumes less memory.
- For old code 2.95.3 would be better, as 4.4.5 will produce tons of warnings or even errors on code being flawlessly compiled by the old GCC.
- For new projects, especially written in C++, GCC 4.4.5 is recommended, as the old ones simply do not keep up with modern standards.
- 4.4.5 usually produces faster code (but sometimes also bigger, depending on optimizer options).
- 4.4.5 is a relatively new and complex compiler, may contain more bugs than 2.95.3.

My general advice is to use GCC 4 and only switch to GCC 2 if needed.

One can check which compiler is currently active using the -v compiler option, which displays the compiler version and build options:

```
1 MorphOS:Development> GCCSelect 2.95.3
2 Switching default gcc links to the gcc 2.95.3 package
3 MorphOS:Development> gcc -v
4 Reading specs from /gg/lib/gcc-lib/ppc-morphos/2.95.3/specs
5 gcc version 2.95.3 20050425 (release/emm-zapek-cisc)
6 MorphOS:Development> GCCSelect 4.4.5
7 Switching default gcc links to the gcc 4.4.5 package
8 MorphOS:Development> gcc -v
```

```
Using built-in specs.
Target: ppc-morphos
Configured with: ../configure --target=ppc-morphos --with-ld=/gg/ppc-morphos/bin/ld
--with-nm=/gg/ppc-morphos/bin/nm  --with-as=/gg/ppc-morphos/bin/as
--with-strip=/gg/ppc-morphos/bin/strip
--with-gmp=/Temp/cvs.morphos.net/morphos/morphoswb/development/gcc4/local_libs
--with-mpfr=/Temp/cvs.morphos.net/morphos/morphoswb/development/gcc4/local_libs
--enable-languages=c,c++ --prefix=/gg --with-sysroot=/gg --libexecdir=/gg/lib
--oldincludedir=/gg/include --program-prefix=ppc-morphos- --program-suffix=-4.4.5
--enable-threads=morphos --disable-bootstrap --with-pkgversion=GCC/MorphOS
--with-bugurl=http://www.morphos-team.net --enable-version-specific-runtime-libs
Thread model: morphos
gcc version 4.4.5 (GCC/MorphOS)
```

1.1.3 Standard C and C++ Libraries

These standard libraries are parts of the C and C++ language specifications respectively. They mainly deliver file and console input/output functions, mathematic functions and string operations. The C++ library also provides a set of basic container classes.

- C standard library on Wikipedia [2]
- C++ standard library on Wikipedia [3]

There are two ways to access these libraries on MorphOS. The first (and default) one is by using a MorphOS shared library; ixemul.library. As the name suggests, this library tries to provide some Unix environment emulation on MorphOS, which, other than the standard libraries, includes large part of POSIX [4] standard and some other commonly used functions. ixemul.library is usually used for porting big projects from the Unix/Linux world, for example it is used by GCC itself and many other tools in the SDK.

The second way is to use libnix (as in; lib no ixemul). In contrast to ixemul.library, libnix is statically linked with the application. This is the preferred way for providing standard libraries for MorphOS native applications. It is achieved by passing the -noixemul flag to the compiler and the linker. libnix delivers full C and C++ standard libraries, but its POSIX implementation is less complete.

Another alternative is to not use standard libraries at all. It may sound crazy at first, but the MorphOS native API provides complete file and console I/O as well as some string manipulation functions and many mathematic functions. Using the native API makes applications faster and smaller in size. On the other hand code using the MorphOS API directly is not portable to non-Amiga(like) systems. A -nostdlib compiler option instructs the compiler to not link the code with the standard library. Note that this requires also writing your own custom startup code.

[2] http://en.wikipedia.org/wiki/C_standard_library
[3] http://en.wikipedia.org/wiki/C++_Standard_Library
[4] http://en.wikipedia.org/wiki/Posix

1.2 The First Traditional ”Hello world!”

1.2.1 ”Hello World!” With the Standard C Library

With the standard C library, one can use the ”Hello World!” example exactly as found in a C handbook. It is given below, just for completeness:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Hello World!\n");
6   return 0;
7 }
```

This source may be copied to a text editor and saved as helloworld.c. To compile it, one opens a shell window (from the Ambient menu, or using the rcommand + n key combo) and changes current directory to the one, where the C source is located. The compiler is run as follows:

gcc -o helloworld helloworld.c

The compiler produces a helloworld executable, which is 10340 bytes on my system. Note that MorphOS pays little attention to filename extensions, so ending the executable name with .exe is not needed, however, it can be done. Traditionally MorphOS executables are named without any extensions. The -o compiler option specifies a desired executable name. If this option is not given, the executable will be named a.out (for some historical reasons).

As stated in the SDK section, the standard C library will be provided by ixemul.library. It can be easily confirmed by tracing the disk activity of helloworld using the Snoopium tool. It can

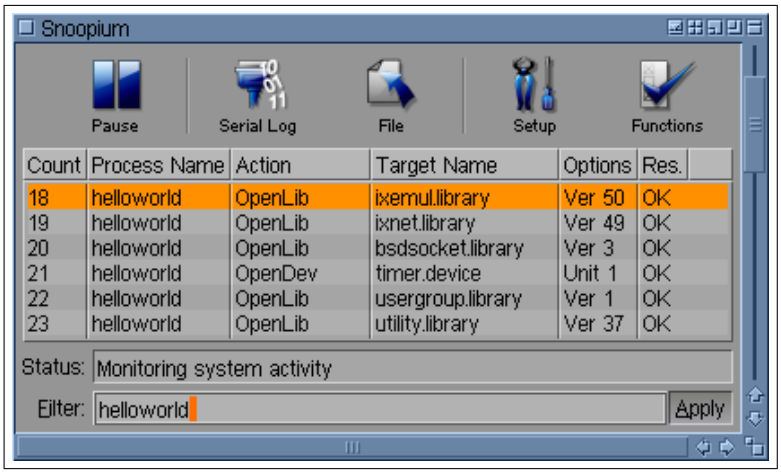

Figure 1.1: Screenshot of Snoppium

also be seen that many other libraries are also opened including ones related to TCP/IP networking. It looks like overkill for such a small program. This happens, because ixemul.library

creates a complete unixlike environment for the application, which is not needed in this simple case. That is why the libnix alternative is recommended for use of the standard library. To use it, a -noixemul option has to be added, so the compiler is called as follows:

gcc -noixemul -o helloworld helloworld.c

The generated executable is much larger (30964 bytes here), which just confirms the fact, that libnix, which is now in use, is a statically linked library. Size of functions used adds to the size of the executable. Every C handbook states, that printf() is the most expensive function of standard I/O, which has just been proven experimentally... On the other hand program activity, as traced with Snoopium, is reduced to three entries. No external resources are opened.

1.2.2 ”Hello World!” With the MorphOS Native API

The MorphOS API (Application Programmer Interface) provides complete file and console input/output. In fact, functions in C and C++ standard libraries are, more or less, complex wrappers around MorphOS native calls. Using the native API has the following advantages:

- Programs are much shorter.
- Programs are faster, thanks to stripping some layers of abstraction.
- Programs are less resource hungry.
- Native API gives full access to MorphOS specific features.

These advantages come at a price:

- Programs using the native API are not portable (except for porting to AmigaOS and AROS to some degree).
- Native printf()-like functions do not support floating point numbers.

The ”Hello World!” example using the native API is as follows:

```
1 #include <proto/dos.h>
2
3 int main(void)
4 {
5   Printf("Hello World!\n");
6   return 0;
7 }
```

The included header includes all things needed to use the dos.library, where the Printf() function is located. The function itself works the same as the standard library printf(), with some minor differences. The code is compiled with this command:

gcc -noixemul -o helloworld helloworld.c

The command is the same as that used for the program using libnix and the standard library printf(), however, the standard C function is not used, so it is not linked. Now the executable size reduces to 13500 bytes.

Why is libnix still needed in spite of the standard library calls not being used? Can't one just compile with -nostdlib? Other than the standard C library, libnix also provides application startup code. A program without this startup code can still work when launched from the shell, but will crash when started from Ambient. The startup code also provides an automatic MorphOS library opening and closing feature. So, excluding libnix completely is possible, but requires writing your own startup code and handling library opening and closing manually.

Note: excluding libnix is usually done for MorphOS components other than applications, like shared libraries or Reggae and MUI public classes. It can also be done for ordinary programs just to make them shorter, especially if a program is small. For bigger projects bytes saved by writing custom startup code are not usually worth the effort.

1.3 Useful Compiler Options

The GCC compiler has hundreds of options. Some of them are irrelevant for typical usage, some of them are irrelevant for PowerPC architecture. This article presents a set of common options used when compiling MorphOS programs. For detailed descriptions of all options see the GCC manual[5].

1.3.1 Compiling and linking

For every project consisting of more than one source file, the process of building the executable program is divided into two stages: compiling and linking. Compilation turns every source code file into an object file. Linking merges all object files (and static libraries) into the final executable. For simple single file projects, these two stages merge into one.

Both the stages have different options. Some options are relevant only for compiling, some only for linking, and some are important for both. Fortunately option names never overlap. Then the safe solution is to pass all the desired options for both stages. Irrelevant ones will be simply ignored.

1.3.2 Options order

The order of passing options to GCC is not important in general. There are some critical exceptions however. The most common one is order of passing static libraries to the linker. Let's assume linking with two static libraries libfoo.a and libbar.a. This requires passing -lfoo -lbar parameters to the linker. However, in the case where libbar uses functions from libfoo, the -lfoo option must be passed after the -lbar option. The linker will be left with unresolved symbols in libbar otherwise.

[5] http://gcc.gnu.org/onlinedocs

1.3.3 Warning options

These options control warnings issued by the compiler on some potentially dangerous language constructs. While some programmers complain about the compiler being too picky, it is recommended to turn most of these options on. It can save hours of time wasted on debugging...

-Wall, turns on warnings for typical potentially dangerous language constructs. Example ones are using a value of assignment as a logical condition, or using arithmetic on void* pointers. While syntactically legal, such constructs may be a result of mistyping, and even when used intentionally, may produce errors that can be very hard to debug. This option is a must for any reasonable programmer.

-Wextra, turns on even more warnings (this option is -W in GCC 2.95.3). There is no serious reason to not use this option together with -Wall.

Note: GCC4 has an irritating feature. String literals are assumed to be arrays of fixed char type. Almost all the MorphOS API functions expect strings to be of type STRPTR which is a typedef of unsigned char*. Passing literals to these functions produces tons of warnings. The clean way to avoid it, is to explicitly cast every literal passed to the MorphOS API as STRPTR. An alternative is to suppress these warnings with **-Wno-pointer-sign**. The disadvantage of this second solution is that it also suppresses pointer signedness warnings for all other integer types, not only for char.

1.3.4 Linker options

-noixemul, instructs the linker to use the static libnix library for standard C/C++ functions and startup code. Without this parameter, the shared library ixemul.library is used.

-s, instructs the linker to strip debug information and symbol tables. This information is not needed in a release executable. Stripping them lowers the executable size significantly.

1.3.5 Optimization options

-On, where n ranges from 0 to 3. The parameter is a global control of execution speed optimizer. Higher numbers make the optimizer more aggressive. **-O2** seems to be the best for everyday use. **-O3** turns on many optimizations which can significantly increase the executable size. Good programmers optimize their algorithms in the first place, compiler optimizations can't fix design errors...

-Os, turns on executable size optimization, at the cost of execution speed. Not very useful for typical applications.

1.4 MorphOS API and Its Organization

An Application Programmer Interface of an operating system usually consists of thousands of functions. MorphOS is no exception here. Its kernel is not monolithic however. The API is functionally (and physically) divided into libraries. Only a few of the largest libraries contain more than 50 functions. A core set of the most important libraries is contained in the system boot image. The rest are placed on the system partition in MOSSYS:Libs (libraries delivered with the system) and SYS:Libs (third party libraries) directories. Disk based libraries are loaded on demand. All these libraries are shared, which means all processes using a library execute the same code loaded to memory once.

1.4.1 Libraries Overview

MorphOS comes with over 100 different libraries. Not all of them are listed below, just the most common ones. Browse the system autodocs in the SDK for more.

- **exec.library**, the master library. This is the system core, responsible for process scheduling, control and creation, communication between processes, memory management, managing other libraries and overall system control. This is the only library which is always open and cannot be closed.

- **dos.library**, responsible for file and console input/output. Provides an interface to advanced filesystem functions (like scanning directories for example). Cooperates with the exec.library in process creation. Delivers basic system time services.

- **graphics.library**, is responsible for low-level graphics functions like drawing pixels and other primitives, copying rectangular blocks of display, scrolling etc. Many programs do not use it directly.

- **intuition.library**, delivers intermediate level graphics interface objects like screens and windows. Interfaces to user input devices (mouse and keyboard to name a few). Provides very basic user controls (gadgets). Provides also BOOPSI (Basic Object Oriented System for Intuition), a language independent object oriented programming framework, used commonly by other components.

- **muimaster.library**, the main interface to MUI (Magic User Interface), which is the MorphOS high level GUI toolkit. Provides a complete, object oriented framework (based on BOOPSI) for GUI driven applications and a rich set of GUI objects.

- **locale.library**, is responsible for system and application internationalization. This simple yet powerful subsystem allows for supporting multiple language versions of a program with a single executable, also provides localization data such as date format, local currency, timezone, number grouping and more.

- **bsdsocket.library** is an interface for TCP/IP networking, compatible with BSD sockets. What is unusual with this library, is that it is neither built into the kernel, nor placed on disk. The TCP/IP stack creates it in memory dynamically.

1.4.2 How to Use a Library in an Application

In typical cases it is pretty automatic. The only thing which has to be done is including the main library header file, which is <proto/[libname].h>, for example <proto/exec.h>, <proto/muimaster.h> and so on. Library opening and closing is handled automatically by the startup code provided by either libnix or ixemul.library. Then one can just use functions from the library. A few big libraries have separate subdirectories in the system include tree. Examples of such libraries are exec.library, dos.library and graphics.library. Header files in these directories contain definitions of constants, data structures, attributes etc. used by the library, divided by functionality. Including of these headers depends on which functions are used in the application, for example to use exec.library memory allocation functions one has to include the <exec/memory.h> header.

Other libraries have a single header file in the libraries directory. Examples are <libraries/locale.h> or <libraries/mui.h> (the latest is some deviation from the naming rule). This single header may be automatically included from the proto file or not.

There are a few cases, where automatic library handling does not work, or cannot be used.

- Third party libraries. Most of them are not included in the autoopen feature.
- Custom startup code (linking with -nostartfiles).
- Opening libraries in a subprocess.
- Dynamic on-demand library opening.
- The library base has been defined in the application code. Autoopening for this library is automatically disabled in this case.

In all of these cases, libraries have to be handled manually.

1.4.3 Manual Library Opening and Closing

While not as convenient as automatic handling, manual opening and closing of libraries is not very complicated. A library base variable has to be defined, then two functions from exec.library: OpenLibrary() and CloseLibrary() have to be used.

The library base is defined in its proto file (the main header) as a global variable. It is a pointer to a Library structure, which should be treated as an opaque pointer. The result returned by OpenLibrary() should be placed in the library base before any function of the library is called. Then, when the library is no longer needed, it should be closed with CloseLibrary() using its base as the argument. The layout for using the hypothetical foobar.library is as follows:

```
1 /* inside <proto/foobar.h> */
2 struct Library *FoobarBase;
3
4 /* inside application */
5 #include <proto/foobar.h>
6 if (FoobarBase = OpenLibrary((STRPTR)"foobar.library", 7)) {
7   /* use library functions here */
8   CloseLibrary(FoobarBase);
9 }
```

The OpenLibrary() call takes two parameters. The first one is just the name of the library to be opened. It is only the name, without path. MorphOS searches a few locations for the library in the following order:

- MOSSYS:Libs/
- LIBS:
- current directory of the application
- PROGDIR:Libs/

In the fourth path PROGDIR: is an automatic assign pointing to the directory containing the application executable.

The second parameter of OpenLibrary() is the minimum required version. Zero here opens any version, any positive number means ”this version or higher”. There is no straightforward way for requesting a particular version of a library. It can be noticed that this approach only works if newer versions of a library are always backward compatible with older ones. On the other hand it avoids having multiple versions of the same library in the system, which is a common problem with Linux shared objects.

The value returned by OpenLibrary() should be always checked against NULL. Even a library built into the boot image may fail to open (because of a memory shortage for example, or having too low a version). Printing some error message in case of a fail is definitely a good idea.

Every successful OpenLibrary() call must be matched with CloseLibrary(). Resource leak is

created otherwise.

There are two special cases for manual library opening and closing: exec.library and dos.library. The first one is always open and cannot be closed, as stated above. The library base for it (named SysBase) is defined and initialized in the startup code. If declared manually in an application's source code, it should be declared as an extern. The dos.library is opened and closed as with any other library, but because the startup code needs it, the DOSBase is already defined and initialized there. As a result the application does not need to open dos.library before using it. If declared in an application, DOSBase should be an extern too.

As with the Amiga historical heritage, some of the most important library bases (SysBase, DOSBase, IntuitionBase, GfxBase and a few more) are not defined as struct Library* but as pointers to library specific structures. Direct poking of these structures was unavoidable in early AmigaOS versions. In MorphOS it is neither needed nor recommended. One can avoid the above definitions (which forces unnecessary typecasting in OpenLibrary() and CloseLibrary()) by #defining __NOLIBBASE__ symbol before including proto files. This disables library bases definitions. All bases can (and must) be then explicitly defined in the code as pointers to struct Library.

Also for traditional reasons, names of some library bases do not follow the [Libname]Base scheme. The most important deviations are: **SysBase** for exec.library, **DOSBase** for dos.library (capitalization), **GfxBase** for graphics.library, **MUIMasterBase** for muimaster.library (capitalization), **CyberGfxBase** for cybergraphics.library. In any case the base name can be checked by looking at the library proto header.

Using the proper base name is very important, as it is used as an implicit argument in all the calls of library functions.

1.5 Common Concepts

This chapter explains some programming techniques and constructs used in MorphOS.

1.5.1 Exec Lists

1.5.1.1 Introduction

A list is the simplest dynamic data structure. An array is even simplier, but it is not dynamic. Adding a single item to an array requires reallocation of memory and copying the whole old contents, similarly for removing. Inserting and removing elements to a list is very fast operation and its computational costs do not increase with number of items. Because of this, list is the basic structure used in exec.library, the MorphOS kernel. One could say the Exec is built on lists. Lists are also used through all the system to manage processes, windows, screens, devices, DOS volumes, fonts, libraries and more. Of course lists are also used by applications to manage dynamic data. Many more sophisticated data structures are built on lists. For all these reasons understanding lists and their MorphOS flavour is essential for every programmer.
Lists may be divided to intrusive and nonintrusive ones. An intrusive list is one requiring that

every list item contains a part called a node. The node is then used for linking items into list. A nonintrusive list creates nodes itself. Both kinds have their advantages. Exec lists are intrusive. Why? For a nonintrusive list adding an item means allocating memory for its node. Exec lists are often used at really low levels of the system, like interrupt handling, process scheduler or input/output hardware devices. Calling memory allocator from there is unacceptable. Also error handling would become a nightmare (every memory allocation can fail...). In some parts of the system lists also have to be extremely fast. Memory allocation is a complex operation and can't be expected to finish in a few processor cycles. Of course on higher levels of the system, intrusiveness has more disadvantages than advantages. For example an item of intrusive list cannot be added to more than one list. Then high level components (for example MUI List class) may be nonintrusive ones.

1.5.1.2 From a Plain List to Exec List

The simplest form of list is a unidirectional one. Node of every item consists only of pointer to the next element. The list is identified by keeping a pointer to its first item. Unidirectional lists however are used only in special cases. While inserting an element at list start is easy, inserting at end requires traversing the whole list. Time of this operation increases linearly with number of items. It is also not possible to traverse the list in the backward direction. Then bidirectional lists are much more common. Exec lists are bidirectional. Every node contains two pointers: one to the next element and one to the previous element. For the first element pointer to the previous one is NULL. For the last one, pointer to the next one is NULL. If a list user keeps pointers to both the first and the last element, she has symmetrical access to the start and the end of the list. While the idea is still simple, we have to complicate

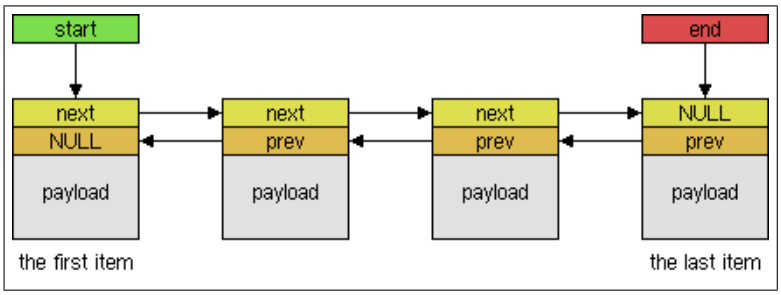

Figure 1.2: A plain, bidirectional list. A node consists of "prev" and "next" pointers.

it a bit. As mentioned before, the list user keeps track of it by maintaining two pointers: to the first and the last item. These pointers will be modified every time an item is added or removed at the start or end of the list respectively. But what if a list has multiple users not knowing about each other? Adding an item at the start or at the end would require pointer update for **all** the users. To solve the problem, two artificial items are introduced: a list head and a list tail. Those two elements do not carry payload, only consist of a node. Their location does not change during the list lifetime so they work as anchors. Inserting an element at list start is in fact inserting it between the list head and the former first element. Similarly element inserted at end is in fact inserted between the former last element and the list tail. As pointers to the head and the tail are constant, they may be shared between multiple users. Exec lists combine the list head and the list tail into one structure, named **list header**. As the Exec was designed in the days when computer memory was counted in kilobytes rather than gigabytes, designers saw a way to save a few bytes. The "previous" field of list head node always contains NULL. Similarly the "next" field of the list tail always

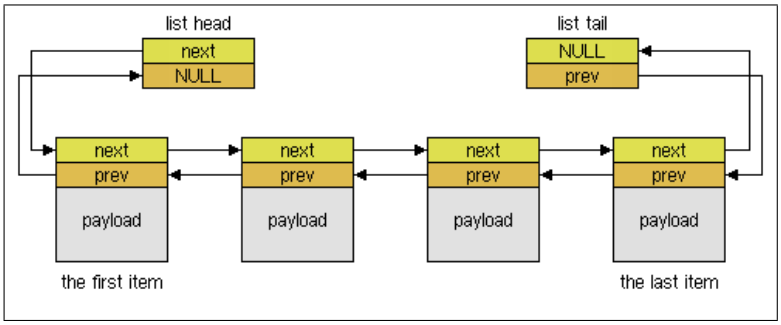

Figure 1.3: Bidirectional list with head and tail pseudoitems.

contains NULL. Then they can be merged into one. That is why the list header contains only three pointers instead of four. For the C language the list header is defined as struct List and lists are commonly referenced by a pointer to it.

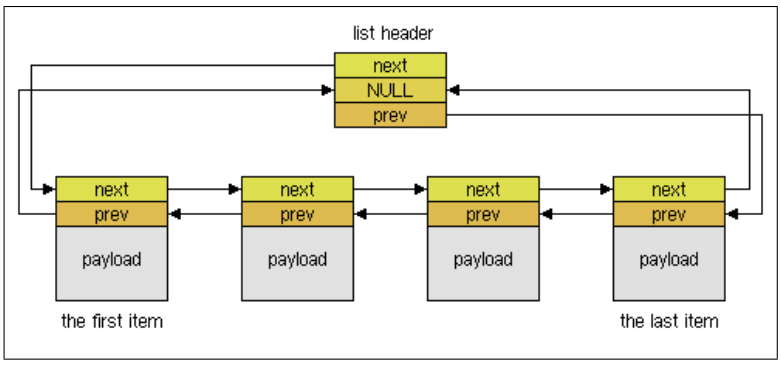

Figure 1.4: The Exec list. Head and tail pseudoelements are merged into the list header.

1.5.1.3 Exec List Elements: Node and Header

Exec defines two kinds of nodes: a full one and a minimal one. The minimal node consists only of pointers to the next and the previous item and is defined in C in <exec/nodes.h> header file as follows:

```
1 struct MinNode
2 {
3   struct MinNode *mln_Succ;     // successor, the next item
4   struct MinNode *mln_Pred;     // predecessor, the previous item
5 };
```

The full node has additional fields used in many system lists. These fields are: item name, item type and item priority.

```
1 struct Node
2 {
3   struct Node* ln_Succ;     // successor
4   struct Node* ln_Pred;     // predecessor
```

```
  UBYTE        ln_Type;     // item type
  BYTE         ln_Pri;      // item priority
  char*        ln_Name;     // item name
};
```

Additional node fields have meaning only for system lists and may be considered as part of payload. If we ignore C types, a Node is just a MinNode with three fields added at the end. Node priority may be used to implement ordered lists or queues. Exec.library provides functions for ordered item insertion.
As there are two kinds of nodes, there are also two types of list headers. Both are defined in <exec/lists.h>:

```
struct MinList
{
  struct MinNode* mlh_Head;      // pointer to the first real item
  struct MinNode* mlh_Tail;      // merged head "previous" and tail "next"
  struct MinNode* mlh_TailPred;  // pointer to the last real item
};
```

Header for full nodes has additional type field (used for system lists) and one byte padding to make the structure size even.

```
struct List
{
  struct Node* lh_Head;      // pointer to the first real item
  struct Node* lh_Tail;      // merged head "previous" and tail "next"
  struct Node* lh_TailPred;  // pointer to the last real item
  UBYTE        lh_Type;
  UBYTE        lh_pad;
};
```

Again, when one ignores C types, the List structure is MinList with two extra fields.
As Exec lists are intrusive, a custom item structure must contain MinNode (or Node if needed) as the first field. It must be a complete structure, not a pointer. Here is an example:

```
struct MyNode
{
  struct MinNode   Node;        // must be the first
  struct Whatever  Foobar;      // example payload fields
  ULONG            Something;
  char             MoreData[10];
  /* ... */
};
```

There is no limit on custom item size. As items are not copied, big items are manipulated with exactly the same speed as small ones.

1.5.1.4 List Initialization, Empty List Check

The list header, which is either List or MinList structure must be initialized before use. Please note well:

Note: Clearing List to all zeros is NOT a proper Exec list initialization!

Looking at the diagrams above, we know how an empty list should look like:

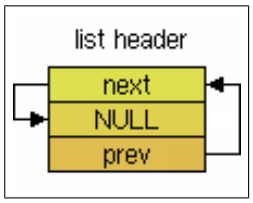

Figure 1.5: An empty list

An empty list contains only its head and tail, both merged into the header. When one splits them back in mind, initialization becomes obvious:

- The ”next” field of the head points to the tail.
- The ”prev” field of the head is NULL.
- The ”next” field of the tail is NULL.
- The ”prev” field of the tail points to the head.

Then the second and the third operation merge into one, as fields are merged. Following code performs proper Exec list initialization. Note the address operators &, forgetting them is a common mistake:

```
1 struct MinList mylist;
2
3 mylist.mlh_Head = (struct MinNode*)&mylist.mlh_Tail;
4 mylist.mlh_Tail = NULL;
5 mylist.mlh_TailPred = (struct MinNode*)&mylist.mlh_Head;
```

In case of full List, initialization is the same, just typecasts have Node instead of MinNode. List initialization is a common operation, so <exec/lists.h> defines a NEWLIST macro for it. The macro takes a pointer to List or MinList, so for the above example it would be called as follows:

```
1 NEWLIST(&mylist);
```

Empty list check is another common operation. It may be derived from the diagram above. One can check if a list is empty using four equivalent conditions:

```
1 /* list is empty */
2 if (mylist.mlh_Head->mln_Succ == NULL)
3 if (mylist.mlh_Head == (struct MinNode*)&mylist.mlh_Tail)
4 if (mylist.mlh_TailPred->mln_Pred == NULL)
5 if (mylist.mlh_TailPred == (struct MinNode*)&mylist.mlh_Head)
```

1.5.1.5 List Iterator

A list iterator is a fragment of code (usually a loop) used for traversing the list and performing operations on its elements. The simplest, browsing iterator is usually implemented as a for loop:

```
struct MyNode *n;
struct MinList *list;    // let's assume it is initialized already

for (n = (struct MyNode*)list->mlh_Head;
     n->Node.mln_Succ;
     n = (struct MyNode*)n->Node.mln_Succ)
{
  /* do something with node 'n' */
}
```

The first part of the for statement initializes the pointer n to the first real item of the list (the successor of the head item). The second part is the loop end condition. Loop ends when successor of the current element is NULL, which happens when the current element is the list tail. Then the loop contents is not executed for the tail, as the tail is not a "real" item, as said above. Finally the third part of for statement moves the pointer to the next list item. A symmetric iterator may be written for browsing a list from the end in backward direction:

```
for (n = (struct MyNode*)list->mlh_TailPred;
     n->Node.mln_Pred;
     n = (struct MyNode*)n->Node.mln_Pred)
{
  /* do something with node 'n' */
}
```

This time the pointer n is initialized to the last real item (predecessor of the list tail), pointer is moved to the previous item in each loop turn. The loop finishes, when the predecessor of the current item is NULL, which is the case for the list head. Again, the loop is not executed for the list head itself.

The <exec/lists.h> header file provides ForEachNode() macro for building a for based iterator. The macro is a bit dangerous however, because it blindly typecasts the node pointer to struct Node* and the list pointer to struct List*. It effectively bypasses the static C language type control and can lead to bugs in code. It is much safer to cast the node pointer to the real type of the list item, as shown in the above example iterators. Then a safer iterator macro should take the type name as one of arguments:

```
#define ForEachNode(n, T, list) for (n = (T)(list)->mlh_TailPred;\
        n->Node.mln_Succ; n = (T)n->Node.mln_Pred)
```

The macro may be used as follows:

```
ForEachNode(n, struct MyNode*, list)
{
  /* do something with node 'n' */
}
```

This macro is not so universal, as it assumes MinList list and also assumes that the MinNode field placed at the start of MyNode structure is named Node. On the other hand it puts static type control in good use. A similar macro may be defined for a backward iterator. In C++ language Exec lists iterators may be defined as templates.

1.5.1.6 Removing List Items From Inside of an Iterator

Some attention has to be put to a case when iterator is used for selective removal of items. Someone may try to do it as follows:

```
/* BAD CODE EXAMPLE */
ForEachNode(n, MyNode, list)
{
  if (/*some condition*/)
  {
    Remove((struct Node*)n);
    FreeVec(n);
  }
}
```

Why the code above is buggy? If the item n is being removed and its memory freed, reference to n in the next loop turn is in fact a reference to free memory. In most cases the memory contents will stay unchanged. That is why such a bug is so popular, it may go unnoticed if the code is not tested enough. Nothing stops the process scheduler to switch our process out, then the memory may be allocated by another process and its contents overwritten. Then, when control will be returned to our process, n points to undefined data and the loop crashes. A solution of this problem is to read the successor from an item before the item is freed. It requires a second pointer to be defined:

```
struct MyNode *n, *n2;

for (n = (struct MyNode*)list->mlh_Head;
     n2 = (struct MyNode*)n->Node.mln_Succ;
     n = n2)
{
  if (/* some condition */)
  {
    Remove((struct Node*)n);
    FreeVec(n);
  }
}
```

Now a pointer to the successor of item n is stored in n2 before the item n is freed. At the next loop turn, the iterator moves to the next item and checks for the list end using valid n2 pointer instead of a reference to free memory.
Things are easier when all items on the list are to be removed unconditionally, so the list is made empty and all the items are disposed. Of course one can still use the safe loop above, but there is a bit faster alternative:

```
while (n = (struct MyNode*)RemHead((struct List*)list)) FreeVec(n);
```

RemTail() function may be used in this loop as well, because when one removes all the items, the order does not matter usually.

1.5.1.7 Adding and Removing Items

Adding an item in arbitrary position of a list requires only updating 4 pointers, regardless of the item size. The diagram below explains the operation.

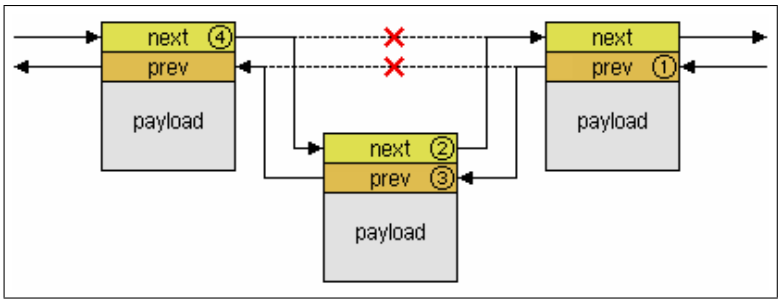

Figure 1.6: Inserting an item into an Exec list.

The diagam corresponds to the following code:

```
1 struct MyNode *n;     /* insert after this item */
2 struct MyNode *a;     /* insert this */
3
4 n->Node.mln_Succ->mln_Pred = &a.Node;    /* 1 */
5 a->Node.mln_Succ = n->Node.mln_Succ;     /* 2 */
6 a->Node.mln_Pred = &n.Node;              /* 3 */
7 n->Node.mln_Succ = &a.Node;              /* 4 */
```

Observe the order of operations. It is important. If one starts from operation 4 for example, he would loose the only link to the rest of the list after the insert position.
Removing link is even faster, as it only requires modifying two pointers: one in the predecessor of removed item and one in its successor. Let's assume element n is to be removed:

```
1 n->Node.mln_Pred->mln_Succ = n->Node.mln_Succ;
2 n->Node.mln_Succ->mln_Pred = n->Node.mln_Pred;
```

Insert and remove operations have been standarizded in MorphOS in two ways: as exec.library API calls and as macros. The library functions are Insert() and Remove(), their macro counterparts are named INSERT() and REMOVE(). Both Insert() function and INSERT() macro take also a pointer to list header. While it is not neccesary in general case, it allows to handle the case when an item is inserted as the first one by passing NULL as the insert position. Inserting as the first element can be also done by passing the list header address as the insert position.

It should be noted, that both Remove() function and REMOVE() macro **do not verify** if the removed node is really in any list. Any attempt to "remove" a node not being in a list may result in random memory trashing and horrible crash.

1.5.1.8 Head and Tail

The most common operations of inserting and removing elements are performed on both the ends of a list. For example common data structures like stack or queue may be implemented using list. For stack, items are added at the list head and also removed from the head. For queue items are added at the tail and removed at the head.

Having a list head and tail pseudoitems gives an advantage, that operations at both list ends are not different than the general case. One just replaces – on the diagram in the previous subchapter – either the element before the insert position with the list head, or the element after the insert position with the list tail. The only difference comes from the fact that head and tail operations usually take just the address of the list header as their argument. The complete set of four operations is provided:

- AddHead() function and ADDHEAD() macro add a node as the first one.
- RemHead() function and REMHEAD() macro remove the first node and return its address.
- AddTail() function and ADDTAIL() macro add a node as the last one.
- RemTail() function and REMTAIL() macro remove the last node and return its address.

All these operations require only an address of the list header. Remove operations return the address of removed node, or NULL if the list is empty.

1.5.1.9 Functions or Macros?

After reading all the sections above it is clear, that most of the list operations is very simple and compiles to a few processor instructions. That is why they are also defined as macros. What to to use then? In general macros are faster, but some of them may be a few bytes longer than calls to library functions (especially INSERT()). On the other hand even a few kilobytes of additional code is usually not a problem nowadays, while gain in speed is often valuable. Then in places where speed is critical, macros are a better choice.

1.5.1.10 Enqueueing

Full list nodes have a priority field, named ln_Pri. The field is often used by the system to maintain prioritized lists or queues. To keep the list ordered by priority, a special insert operation is required. It is provided as an exec.library call named Enqueue(). The function takes a node (it must be full Node, not MinNode), reads its priority and finds a place for insert comparing priorities of nodes. If there are any nodes in the list with the same priority as the inserted one, it is inserted before already existing ones. Of course Enqueue() may be used also for implementing custom queues. One has to take into account however that the priority field is limited to signed 8-bit number and the function does not scale well for very long lists, as the insert position search is linear.

1.5.2 Taglists

A taglist is an array of ”key-value” pairs. The key is always a 32-bit integer number and is called a tag. The value also has a size of 32 bits. It may be an integer, or a pointer to any structure or object. Taglists are commonly used in the MorphOS API for passing a variable number of arguments, usually sets of attributes with their values. A few special key values are used for array termination, concatenation and item skipping. A set of functions in the utility.library may be used for taglist traversing, filtering, searching, copying etc.

Every pair in a taglist is a TagItem structure, defined in <utility/tagitem.h>:

```
1 struct TagItem
2 {
3   ULONG ti_Tag;
4   ULONG ti_Data;
5 };
```

To be self descriptive, every taglist, being just a plain C array, must have some kind of termination. It is very similar to the string null-termination idea. The termination is done with a TagItem having its ti_Tag field set to TAG_END (which happens to be defined as zero). The ti_Data value of the terminating TagItem is ignored, it is usually set to zero too. The illustration below shows a simple taglist:

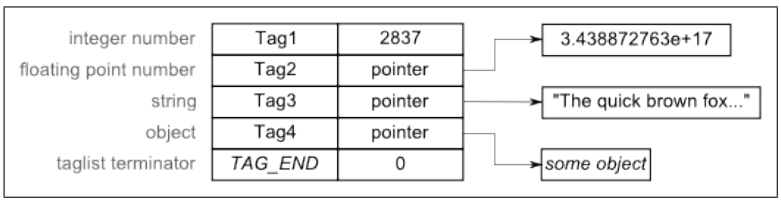

Figure 1.7: A simple taglist

This taglist may be created with the following code:

```
1  double x = 3.438872763e+17;
2  Object *obj = NewObject( /* ... */ );
3
4  struct TagItem mytaglist[] = {
5    { Tag1, 2837 },
6    { Tag2, (ULONG)&x },
7    { Tag3, (ULONG)"The quick brown fox..." },
8    { Tag4, (ULONG)obj },
9    { TAG_END, 0 }
10 };
```

1.5.2.1 Passing Taglists to Functions

Building taglists as global or local variables is not very convenient. That is why almost every MorphOS API function getting a taglist as one of its arguments has two forms. The first one accepts a pointer to the taglist. The second one is a variadic args macro building the taglist on the program stack dynamically. Such function pairs are named according to one of the two conventions:

- SomeFunctionA() takes a pointer to a taglist, SomeFunction() builds the taglist dynamically.
- SomeFunctionTagList() takes a pointer to a taglist, SomeFunctionTags() builds the taglist dynamically.

Continuing the above example, one can pass the example taglist to SomeFunction() in two ways:

```
1 SomeFunctionTagList(mytaglist);
2 SomeFunctionTags(Tag1, 2837, Tag2, (ULONG)&x, Tag3,
3                  (ULONG)"The quick brown fox...", Tag4,
4                  (ULONG)obj, TAG_END);
```

Of course in the second case variable mytaglist need not be defined anywhere. Note that for every taglist-based function the taglist is the last argument. There may be some plain arguments before it. It is a common practice to omit ti_Data for the terminator (it is ignored anyway).

1.5.2.2 Special Tags

Special tags are used to avoid copying large blocks of data when taglists are manipulated. As taglists are plain arrays, operations like removing or inserting elements or merging taglists involve copying their large fragments. Special tags allow for doing these operations by only changing a few tags. Of course this comes at a price. Taglists with special tags (other than TAG_END) may be only manipulated with a set of functions from utility.library. All these functions detect special tags and interpret them properly. It should be noted, that even if taglists are created by hand and do not contain special tags, such tags may appear later, when taglists are passed to system functions. It is therefore recommended to always use utility.library tools to manipulate taglists, especially the taglist iterator named NextTagItem().

TAG_END (TAG_DONE)

I've mentioned this tag already, it works as a taglist terminator. There is no difference between TAG_END and TAG_DONE, both are defined as 0 and may be used alternatively. The first name is used through this article just for being shorter.

TAG_IGNORE

This special tag is used to logically remove a tag item from a taglist without copying large blocks of data. When a tag is to be removed, it is replaced with TAG_IGNORE, tags after it need not to be moved. Any taglist processing function will ignore the tag and advance to the next one immediately.

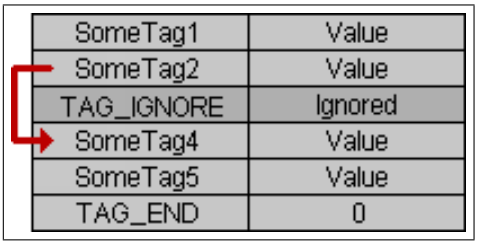

Figure 1.8: Ignored tag entry

TAG_MORE

This tag is used for joining taglists. Joining of plain arrays would require counting their size, allocating memory area and copying all tagitems. It is much simplier with TAG_MORE. If one wants to append taglist B at the end of taglist A, he replaces the terminator of taglist A with TAG_MORE and sets the data field to the address of taglist B. Then, when any taglist manipulation function sees this special tag, it fetches the next tag from the address. In effect taglists are logically joined into one.

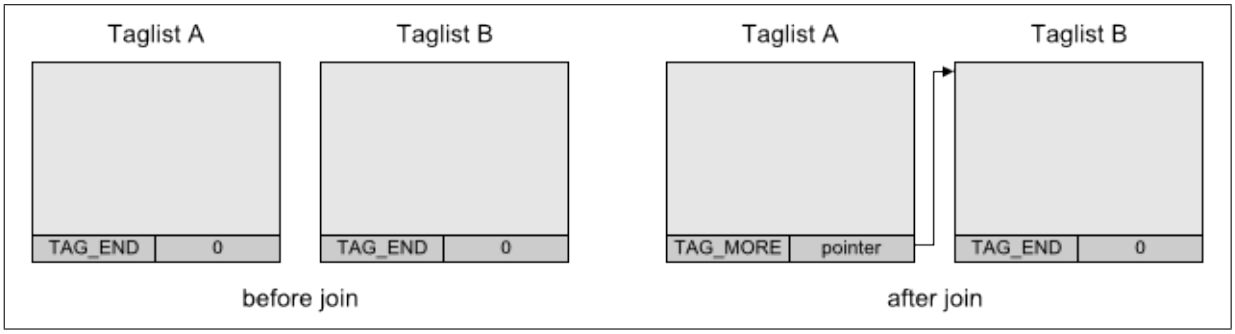

Figure 1.9: Joining two taglists

TAG_SKIP

This rarely used tag is used to logically remove larger fragments from a taglist. When encountered, it directs the taglist iterator to skip a specified number of following tags. The number is specified in the data field of TAG_SKIP. When using TAG_SKIP one should take

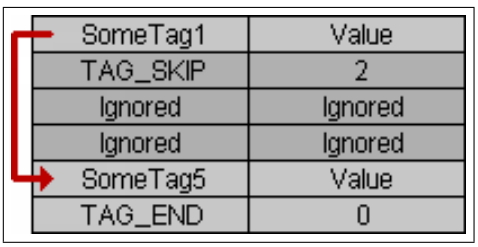

Figure 1.10: Skipping entries of a taglist

care to not skip past the taglist terminator. Taglist processing functions are not protected against such a bug and will try to process random data. Results are unpredictable.

TAG_USER

While TAG_USER is not really a special tag, it is listed here, because it can be found in many source codes. It is defined as $80000000, so it divides tag space into halves. The idea behind it is that system tags (tags being part of the system API) are located in the lower half, while tags defined by applications are always in the upper half. In fact this is only a convention, not enforced and not consistently followed (for example all MUI tags or even Intuition tags are in the user half), as non-special tags are always interpreted in some context. Of course values of special tags must not be used, so values 0 - 3 are reserved (and treating values up to at least 255 as reserved is very good idea too).

1.5.2.3 Traversing Taglists With NextTagItem()

NextTagItem() function from the utlilty.library is the basic taglist iterator. It should be always used to traverse taglists, as it supports all the special tags discussed above. Every call to the function returns an address of the next item in the taglist, of course special items are ”executed” and not returned. The argument of the function is an address of variable being a pointer to the current tag. This pointer is initialized manually to the start of taglist and should not be modified later. A basic loop for taglist browsing is organized as follows:

```
1  struct TagItem *tag, *tagptr;
2
3  // initialization to the start of taglist to be traversed
4  tagptr = my_taglist;
5
6  while (tag = NextTagItem(&tagptr))
7  {
8    /* do something with 'tag' */
9  }
10
```

NextTagItem() returns NULL when it encounters TAG_END terminator.

1.5.2.4 Taglists Processing

Any processing of taglists can be done with just NextTagItem() and manual manipulation of tagitems. On the other hand utility.library provides set of functions for typical operations on taglists. Using them saves time and avoids bugs, which may be possibly made in manually written code. Advanced taglist processing is usually not needed in typical aplications, but comes handy, when taglists are used for data storage.
It should be noted, that this article does not replace autodocs of utility.library functions. These autodocs have been completely rewritten and heavily extended in the MorphOS SDK (compared to Amiga ones). This text is just an overview of possibilities, while autodocs provide detailed descriptions.

1.5.2.5 Finding Tags and Data

Finding particular tag in a taglist is relatively simple task. It can be done with FindTagItem() function. It returns the first occurence of specified tag. If a tag is expected to appear multiple times, the function may be called in a loop, taking its result as an argument for the next call.

```
struct TagItem *tag = my_taglist;

while (tag = FindTagItem(SOME_TAG, tag))
{
  /* do something with 'tag' */

  tag++;  /* start from the next tag after the one just found */
}
```

As all other functions described below, FindTagItem() automatically interprets special tags, so they are never returned to the caller. That is why incrementing the tag pointer in the above loop is always safe.

In many cases (like constructors of MUI classes) we are interested in data of particular tag, and also want a default value if the tag is not present in a taglist. These two operations may be merged into one call to GetTagData() function. It takes a tag, the default value and a taglist as arguments. If the tag is found, its data is returned, if not the result is the default value. Here is an example, the code:

```
if (tag = FindTagItem(SOME_TAG, taglist)) value = tag->ti_Data;
else value = DEF_VALUE;
```

reduces to

```
value = GetTagData(SOME_TAG, DEF_VALUE, taglist);
```

The last function related to finding is TagInArray(). It does not operate on a taglist however. It simply checks if a tag is found in a zero-terminated array of tags. This is not a taglist, just plain array of ULONG-s.

1.5.2.6 Creation and Copying

The simplest way of creating a tagilst is to declare it as an array, be it global or local variable. It has been shown already in this article. If a taglist should be allocated dynamically, one can use any general purpose memory allocation function provided by exec.library. An example with AllocVec():

```
struct TagItem *taglist;    /* taglist with 7 items (including terminator) */

taglist = (struct TagItem*)AllocVec(7 * sizeof(struct TagItem), MEMF_ANY);
```

The utility.library provides also a special function for taglist allocation named AllocateTagItems(), which is in fact nothing more than a wrapper on general purpose memory allocation function. It also clears allocated memory block, so it can be treated as an empty taglist (as TAG_END is 0). Taglist allocated with AllocateTagItems() must be freed by FreeTagItems().

```
struct TagItem *fresh_list;

if (fresh_list = AllocateTagItems(2))
{
  /* Need not to initialize terminator at [1] */
  /* as the taglist is cleared to all zeros. */
  fresh_list[0].ti_Tag = SOME_TAG;
  fresh_list[0].ti_Data = 123456;

  /* more opreations on 'fresh_list' */

  FreeTagItems(fresh_list);
}
```

A copy of an existing taglist can be created with CloneTagItems(). This function is not a simple wrapper, as taglist containing special tags cannot be copied as a memory block. For doing a proper copy, the function traverses the taglist (using NextTagItem()) two times. First it counts plain tags. Then it allocates a memory block for a clone and does second pass over original, copying tags one by one. Order of tags is preserved, so when we iterate both original and clone with NextTagItem(), we get the same results. The difference is that the clone is stripped off of all special tags. All tags ignored and skipped are not cloned, also separate fragments chained with TAG_MORE are merged into one continuous block. The only special tag present in the clone is, of course, TAG_END. As CloneTagItems() allocates memory for the copy, the copy must be freed later with FreeTagItems(). CloneTagItems() is in fact a memory allocation function, so its result must be always checked against NULL.

```
struct TagItem *original, *copy;

if (copy = CloneTagItems(original))
{
  /* do something with 'copy' */

  FreeTagItems(copy);
}
```

There is also a taglist copying function for making copy into a ready buffer. It is a bit misleadingly named RefreshTagItemClones(). In fact the function works the same as CloneTagItems(), but uses a destination buffer passed as an argument instead of allocating a new one. The copy is stripped off of ignored and skipped tags and also defragmented, the same as for CloneTagItems(). It should be noted that RefreshTagItemClones() does not check destination buffer size, so if the original list does not fit into the buffer, memory will be corrupted.

```
struct TagItem copy[7];

/* I'm so sure original list has no more than 6 non-special items */
/* + terminator. If it is longer, total armageddon will be started. */

RefreshTagItemClones(copy, original);
```

The function cannot fail, so it has no result.

1.5.2.7 Filtering and Mapping

Utility.library provides four functions for filtering and mapping tags and data. Two of them, FilterTagItems() and MapTagItems() operate on tags considering only their identifiers. The second two, ApplyTagChanges() and FilterTagChanges() operate on tags also considering their data.

1.5.2.8 Filtering Tags by Identifier

The FilterTagItems() function allows for removing tags from taglist according to specified array of tags. Tag removal is done by changing its identifier to TAG_IGNORE. This function has two modes of operation. The array of tags may work either as array of allowed ones (all others are filtered out), or array of ones to be rejected (all others are kept). It is illustrated on the diagram below. Tag identifiers have been marked with colors. On the left there is original taglist (the same in both diagrams). The center one is an array of filtered tags. It does not contain TagItem structures, but tag identifiers, so it is just an array of ULONGs. The original taglist after performing the operation is shown on the right.

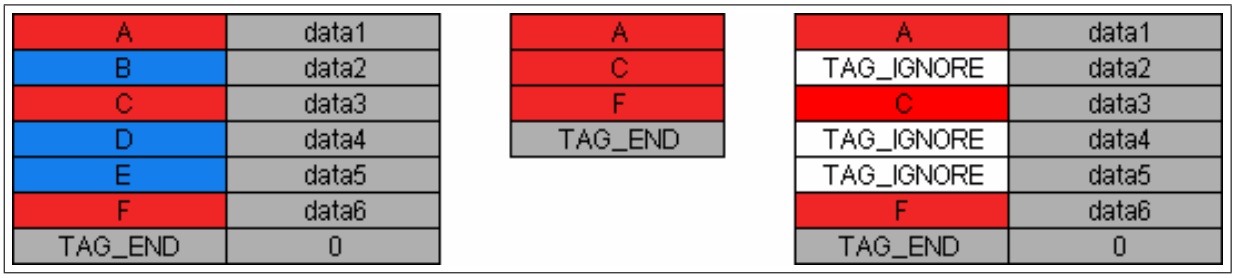

Figure 1.11: Filtering a taglist with TAGFILTER_AND mode. All tags not present in the tag array are rejected.

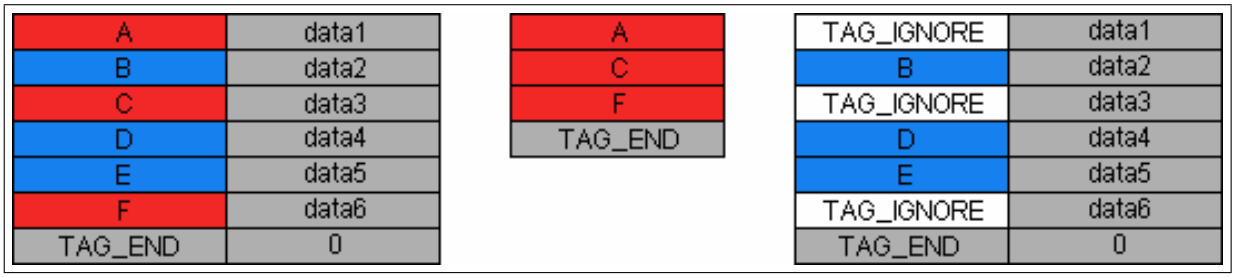

Figure 1.12: Filtering a taglist with TAGFILTER_NOT mode. All tags present in the tag array are rejected.

1.5.2.9 Tag Mapping

In the process of mapping, a set of tag identifiers is being replaced by another set according to a map. Tags not included in the map may be either kept or removed. The map is just another taglist. Tag identifiers of map taglist are source identifiers. Data fields of map taglist

are destination identifiers. Mapping replaces source identifiers with destination ones. The same as for tag filtering, tag mapping does not change data fields. Diagrams below show the process of mapping. A set of ”red tags” is mapped to set of ”blue tags”, while other are either left intact or removed, depending on argument passed to MapTags(). Source taglist before mapping is shown on the left, the map in center and source after mapping on the right. An interesting possibility is mapping ordinary tags to special ones. Special tags

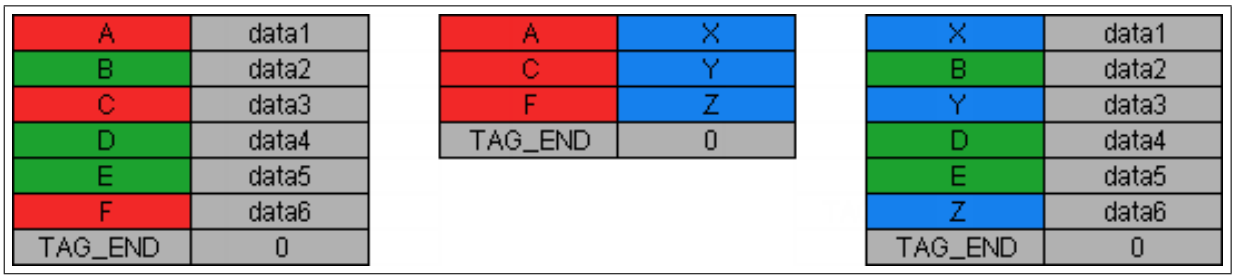

Figure 1.13: Mapping red tags to blue tags with keeping unmapped ones

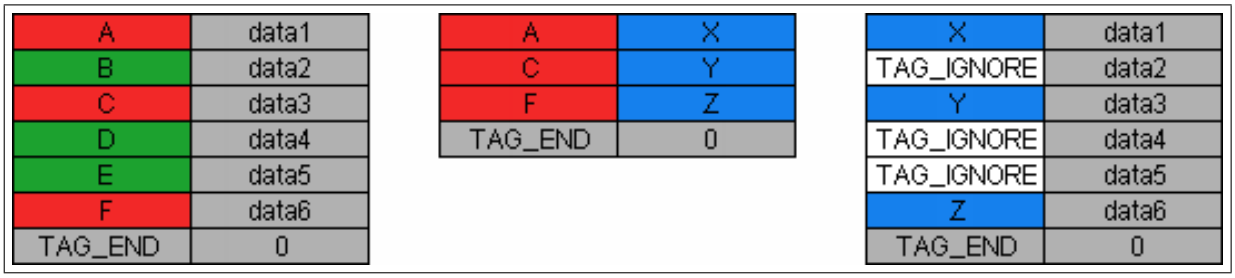

Figure 1.14: Mapping red tags to blue tags with removing unmapped ones

cannot be mapped, because MapTags() reads the source with NextTagItem(), so special tags are interpreted rather than mapped. On the other hand, any ordinary tag may be mapped to a special one. The most useful special map destination is TAG_IGNORE, which makes MapTags() to act the same as FilterTags(), but with the first function one can map some tags and filter other ones in one go. Mapping to TAG_SKIP and TAG_MORE is not that common, but may be useful in special cases (for example replacing a single tag with taglist or switching off tag sequences). MapTags() does not allow for mapping to TAG_END, TAG_IGNORE is used instead.

1.5.2.10 Filtering Tags Data

The ApplyTagChanges() function works on two taglists. The first one is the master, the second is a list of changes. Every tag in master is searched for in the list of changes. If the tag is found, the data of the tag in master list is set to the data in change list. The FilterTagChanges() is more advanced version of the previous function. Applying changes to the master is optional in this case (controlled by a function argument). Additionally the change list is modified. Tags specifying no change, it means found in both the master and the change list and having the same data, are removed from the change list.
The diagram below illustrates principle of working of ApplyTagChanges(). Tags marked green are not subject of changes, and are left intact. Tags in red are possible subject of changes. Data, which are different in the master list and change list are marked blue.

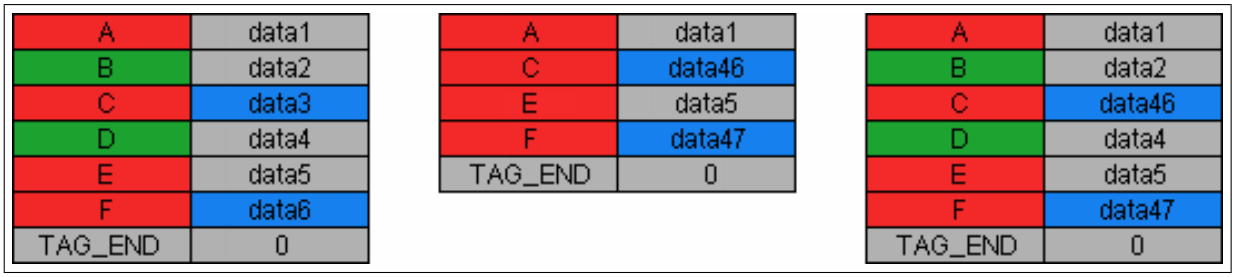

Figure 1.15: Principles of ApplyTagChanges()

Similar diagram explains FilterTagChanges(). It is a bit more complicated, as the function has additional parameter. This boolean parameter controls applying changes to the master taglist. Then one can only learn about changes without applying them.

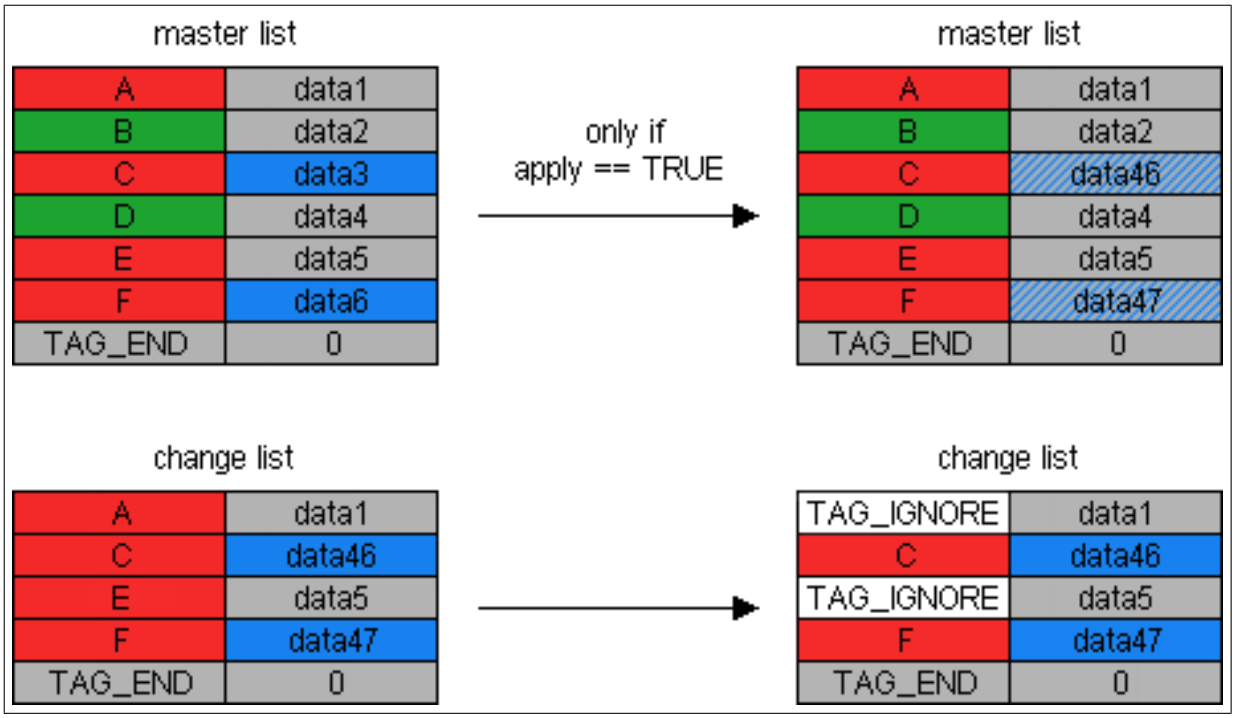

Figure 1.16: Principles of FilterTagChanges()

1.5.2.11 Data Conversion

Data conversion functions automate data exchange between taglists and C-style structures and bitfields. Bitfields are easier, so let's start with them.

1.5.2.12 Bitfields

A bitfield is very effective way of storing boolean (TRUE or FALSE) information. One boolean TagItem occupies 64 bits in memory, while it may be stored in one bit in a bitfield. Also checking a bit value is much faster than checking tag value. The PackBoolTags() function converts a taglist into bitfield. Parameters of this function are a bit similar to MapTags(). There is a source taglist and a map taglist. The map however contains bitmasks for the bitfield instead of tag identifiers. Boolean taglist processing starts from setting the initial bitfield value (taken from the first argument of the function). Then every tag in the map is searched in the source taglist. If found, its boolean value is read and controls how corresponding bitmask is applied to the bitfield. For TRUE, all 1-s from the mask are set in the bitfield. For FALSE all 1-s from the mask are cleared in the bitfield.

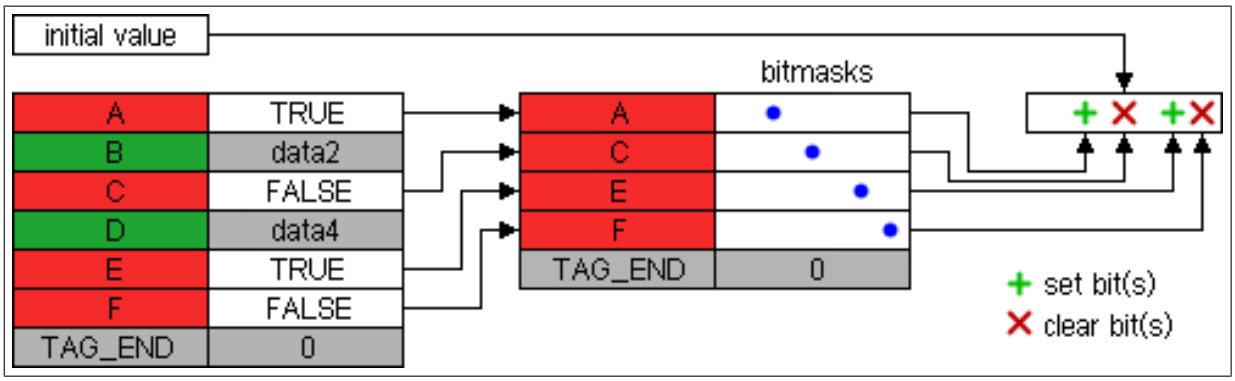

Figure 1.17: Example of a bitfield

In the usual case, the mask of every tag has only one bit set, so tags are just mapped to single bits of the bitfield. It is possible to use multi-bit masks for tags, even overlapping ones. It should be noted however, that overlapped masks make the final result dependent on order of tags in the source taglist.
One may be surprised that utility.library does not provide an inverse function to PackBoolTags(). There are a few reasons for this:

- For the general case of multi-bit and possibly overlapping masks, the inverse function is not possible.
- For the strict case of single bit, not overlapping masks, the inverse is easy to write.
- To some extent, the task of unpacking may be performed with UnpackStructureTags().

1.5.2.13 Structures

A pair of two functions: PackStructureTags() and UnpackStructureTags() can be used to convert a taglist to almost arbitrarily defined data structure and vice versa. Because they were

poorly documented in AmigaOS 3.x autodocs, they were rarely used in the past. MorphOS SDK contains a rewritten from scratch documentation of these functions, based on gathered knowledge and extensive tests of MorphOS and AmigaOS 3.x implementations (which are fully compatible).

The PackStructureTags() function moves data stored in a taglist to a data structure. The process is controlled by pack table. The table contains encoded storing instructions, like offsets and sizes of structure fields as well as source tag identifiers. Supported field widths are single bits as well as 8-, 16- and 32-bit fields, signed or unsigned. The pack table is best constructed using a set of construction macros defined in ¡utility/pack.h¿. These macros automatically generate pack instructions, field offsets etc. taking into account things like field padding inserted by the compiler. If for some reason macros cannot be used, pack table entry layout is described in the autodoc for PackStructureTags().

Tag identifiers for data to be packed are assumed to start from a defined value called tag base. Tag base definition should be the first entry of the pack table and is generated with PACK_STARTTABLE macro. It is also assumed, that offsets between the base and all the tags are no bigger than 1024. Then only offsets are stored in the pack table (using 10 bits out of 32 available), so packing instruction for one tag fits in a single ULONG. If, for some reason, data tags are not in a single offset range, the tag base may be changed with PACK_NEWOFFSET macro. Finally PACK_ENDTABLE macro ends the packing table.

Structure entries, except of single bits, are defined with PACK_ENTRY macro, taking 5 arguments. The first one is the tag base, the second is the identifier of the tag containing data for the field. Tag offset is calculated automatically from them, saving the need of manual bit masking. The next two arguments describe the field, they are structure C type name and field name. Offset of the field is then determined automatically. This takes into account implicit field padding and also works properly for structures declared with __attribute__ ((packed)), commonly used for data loaded from files (like headers or records). The last, fifth argument contains control flags, defining field size.

```
1  struct Foo
2  {
3    ULONG Bar;
4    WORD Blah;
5    BYTE Xyz;
6  };
7
8  #define FOO_TAGBASE  TAG_USER
9  #define FOO_BAR      (FOO_TAGBASE + 0)
10 #define FOO_BLAH     (FOO_TAGBASE + 1)
11 #define FOO_XYZ      (FOO_TAGBASE + 2)
12
13 ULONG PackTable[] = {
14   PACK_STARTTABLE(FOO_TAGBASE),
15   PACK_ENTRY
16       (FOO_TAGBASE, FOO_BAR,  Foo, Bar,  PKCTRL_ULONG | PKCTRL_PACKUNPACK),
17   PACK_ENTRY
18       (FOO_TAGBASE, FOO_BLAH, Foo, Blah, PKCTRL_WORD | PKCTRL_PACKUNPACK),
19   PACK_ENTRY
20       (FOO_TAGBASE, FOO_XYZ,  Foo, Xyz,  PKCTRL_BYTE | PKCTRL_PACKUNPACK),
21   PACK_ENDTABLE
22 }
```

The example above defines some simple structure, a set of tags and a pack table for data exchange between a taglist and the structure. The same table may be used in both directions, as it has been declared as fully symetric (PKCTRL_PACKUNPACK). It is possible to use some tags only while packing taglist to structure (PKCTRL_PACK) or only while unpacking back to taglist (PKCTRL_UNPACK). Desired default values may be stored in the structure before unpacking. If a tag assigned to some field is not found in the source taglist, the field is not changed. If the same table is used for packing and unpacking, it is important to take care of signedness of fields. It does not matter for packing, as ti_Data field is just truncated by discarding higher bytes when stored in 8- or 16-bit field. When unpacking however, fields marked as signed will be properly sign-extended, while unsigned ones will be extended with zeros.

Single bits of fields may be packed and unpacked too. Unlike PackBoolTags(), PackStructureTags() does not allow for multi-bit masks, because pack instruction stores just bit number. On the other hand structure function supports also 8- and 16-bit fields. It is also more flexible in bit handling. Bit may be set according to the value of a boolean tag (with optional negation), it may be also set or cleared based just on the tag presence (tag data is ignored in this case). It is controlled by PACK_BIT / PACK_FLIPBIT and PSTF_EXISTS flags:

- PACK_BIT – sets bit according to tag data,
- PACK_FLIPBIT – sets bit according to negation of tag data,
- PACK_BIT — PSTF_EXISTS – sets bit if tag is present in the taglist,
- PACK_FLIPBIT — PSTF_EXISTS – clears bit if tag is present in the taglist.

Bits are defined using PACK_BYTEBIT, PACK_WORDBIT and PACK_LONGBIT macros depending on the width of bitfield. Here is a simple example:

```
1  struct FooBits
2  {
3    ULONG LongField;    // bitfields should be always
4    UBYTE ShortField;   // declared as unsigned
5  };
6
7  #define FOOBITS_TAGBASE   TAG_USER
8  #define FOO_LongA         (FOOBITS_TAGBASE + 0)
9  #define FOO_LongB         (FOOBITS_TAGBASE + 1)
10 #define FOO_ShortC        (FOOBITS_TAGBASE + 2)
11 #define FOO_ShortD        (FOOBITS_TAGBASE + 3)
12
13 ULONG PackTable[] =
14 {
15   PACK_STARTTABLE(FOOBITS_TAGBASE),
16   PACK_LONGBIT(FOOBITS_BASE, FOO_LongA, FooBits, LongField,
17     PKCTRL_PACKUNPACK | PKCTRL_BIT, 23),
18   PACK_LONGBIT(FOOBITS_BASE, FOO_LongB, FooBits, LongField,
19     PKCTRL_PACKUNPACK | PKCTRL_FLIPBIT, 19),
20   PACK_BYTEBIT(FOOBITS_BASE, FOO_ShortC, FooBits, ShortField,
21     PKCTRL_PACKUNPACK | PKCTRL_BIT | PSTF_EXISTS, 6),
22   PACK_BYTEBIT(FOOBITS_BASE, FOO_ShortD, FooBits, ShortField,
23     PKCTRL_PACKUNPACK | PKCTRL_FLIPBIT | PSTF_EXISTS, 2),
24  PACK_ENDTABLE
25 };
```

The code above assigns tags to bits as follows (note that bit 0 is always the least significant one):

- FOO_LongA is assigned to bit 23 of LongField, tag data is stored in bit,
- FOO_LongB is assigned to bit 19 of LongField, tag data is negated and then stored in bit,
- FOO_ShortC is assigned to bit 6 of ShortField, bit is set if tag is found,
- FOO_ShortD is assigned to bit 2 of ShortField, bit is cleared if tag is found.

Not all of these possibilities are symmetrically available while unpacking to a taglist. UnpackStructureTags() ignores PSTF_EXISTS and PKCTRL_FLIPBIT and simply sets tag data field to FALSE or TRUE depending on the bit.

Chapter 2

Magic User Interface Programming

Author: Grzegorz Kraszewski
Source: http://library.morphzone.org/Magic_User_Interface_Programming

2.1 Introduction

Magic User Interface (MUI for short) is the MorphOS toolkit for creating applications with a graphical user interface. It provides a broad set of gadgets (controls) as well as a complete framework for designing event driven programs. MUI is object oriented, but does not rely on any specific programming language. A BOOPSI (Basic Object Oriented System for Intuition) is used as the foundation of MUI object oriented design.
MUI offers a dynamic layout as its basic mode of operation. The placement of gadgets is determined by grouping them in horizontal, vertical or matrix groups. Pixel coordinates of gadgets adapt dynamically to user preferences like font sizes, spacing between gadgets, objects' frames and backgrounds. The two screenshots below show the same example application. The only difference is that user MUI settings are different. The first settings are

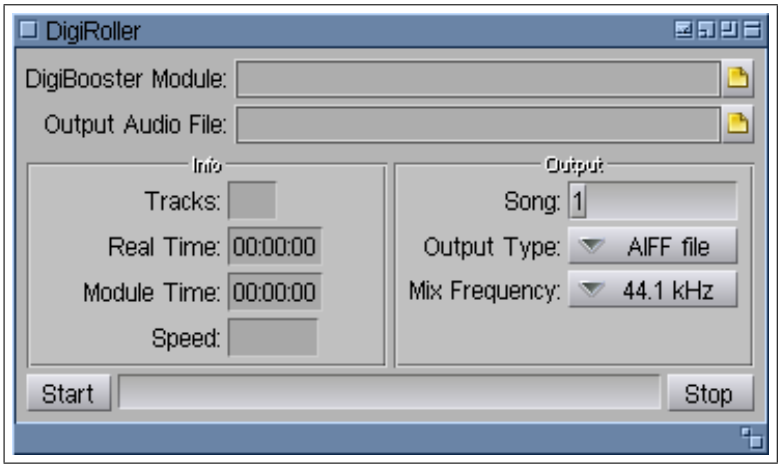

Figure 2.1: MUI application example screenshot

plain and clean. They may even be called a bit oldschool for using simple, vector frames and uniform color backgrounds. A simple window skin, named Mahalaxmi fits this design nicely.

Figure 2.2: MUI application example screenshot

The second example uses some MUI 4 features, like bitmap frames with transparency masks. This dark design is achieved by using the Nox window skin. All gadget position calculations are done automatically, accounting for a larger font and fancy frames. The programmer need not care about user taste and preferences (a good programmer would test the program appearance with a few different settings however).

2.2 The First Steps

2.2.1 Short BOOPSI Overview

2.2.1.1 Object Oriented Programming

Object oriented programming is a technique developed as a response to two trends in the computer market. The first one was increasing complexity of software. Management of a traditionally written codebase becomes harder when the code size increases. The second trend was the increasing popularity of graphical user interfaces, which meant the end of sequential execution of programs. Instead modern programs are event driven, which means the flow of code execution is determined by external events (like user input) and is not known at the time of writing the program. Object oriented programming divides a program into a set of objects interacting with each other using well defined interfaces. Such a modularization simplifies the management of a software project and also fits naturally with the concept of modern graphical user interfaces. User controls (called ”gadgets” in MUI) are just objects in the code and they interact with other objects representing user data.
This short introduction is not intended to be a complete lecture on object oriented programming. On the other hand no knowledge of any particular object oriented programming language is required to get familiar with BOOPSI. Usually the support for OOP techniques comes with a programming language, which is either designed for OOP (like C++, C# or Java) or has OOP support added in a more or less logical way (Objective C, PHP). This is not the case for BOOPSI and MUI however. In this case object oriented programming support comes from the operating system. BOOPSI and MUI can be used with any programming language, including traditional ones, for example C and even assembler.
The BOOPSI module is located in the intuition.library, with some important functions being added from a statically linked libabox. Its primary design goal was to build a framework for

wrapping Intuition GUI elements in an object oriented interface. This approach was unfortunately not flexible enough, so MUI uses only the basic BOOPSI framework. This framework provides the four basic concepts of object oriented programming: classes, objects, methods and attributes. It also supports class inheritance. Because of its simplicity, BOOPSI is easy to understand and use, especially when compared to more sophisticated frameworks, like the one in the C++ programming language.

2.2.1.2 Classes

A class is the basic term of object oriented programming. It is the complete description of its objects, their attributes and methods. In the BOOPSI framework, a class consists of:

- An IClass structure. A pointer to this structure is used as a reference to the class. The IClass structure is defined in the <intuition/classes.h> system header file. There is also the Class type, which is the same as struct IClass.
- A class dispatcher function. When an application calls a method on an object, the object's class dispatcher is called. The dispatcher checks the method's identifier and jumps to this method code. The dispatcher is usually implemented as a big switch statement. For simple classes, which implement only a few short methods, code of these methods is often placed inside case statements. Bigger classes have methods' code separated into functions placed outside of the dispatcher. As every method goes through a dispatcher, all BOOPSI methods are virtual in the C++ meaning. For this reason, calling a method in BOOPSI is usually slower than in C++.

A class defines a set of methods available for its objects (instances) by the set of case statements in the dispatcher. Objects attributes are set using the OM_SET() method and are gotten using OM_GET(). The attributes may also be passed to the object constructor directly. The set of attributes for a class and applicability of the attributes are then defined by the source code of OM_NEW() (the constructor), OM_SET() and OM_GET() methods. There is no formal declaration of class. There is also no division between public and private methods and attributes. Some kind of formal declaration and levels of access may be imposed by putting every class in a separate source code file. An accompanying header file would contain definitions of method identifiers, attribute identifiers and method parameters structures, but only those considered ”public”. Private method identifiers should be defined inside the source code of the class, so they are not visible outside of the class source code.
A BOOPSI class can be shared between applications. All the MUI built-in classes are shared ones. The BOOPSI maintains a system-wide list of public classes (the list can be browsed with the Scout monitoring tool). Shared classes are identified by names. A part of the MUI standard classes is contained inside the main MUI library – muimaster.library. The library adds these classes to the system list when opened for the first time. The rest of the MUI standard classes are stored on the system disk partition in the MOSSYS:Classes/MUI/ directory. Additional third party classes may be placed in the SYS:Classes/MUI/ directory.
Shared classes use the MorphOS shared library framework, in other words a shared BOOPSI class is just a kind of shared library. The class adds itself to the public list of classes, when it is opened from disk. As such, a BOOPSI shared class should be opened with OpenLibrary() before use (see details), especially as BOOPSI classes are usually not included into the list of libraries opened automatically. This is not the case for MUI classes however. MUI shared classes can be used without opening them. It is explained below, in the MUI Extensions to BOOPSI section.

2.2.1.3 Methods

Methods are just actions, which can be performed on an object. A set of available methods is defined by the object's class. Technically speaking, a method is a function called with an object as its parameter in order to change the object's state. In BOOPSI, methods are called using the DoMethod() call from libabox:

```
result = DoMethod(object, method_id, ... /* method parameters */);
result = DoMethodA(object, method_struct);
```

The first, more popular form of the call just builds the method structure on the fly, from arguments passed to it. Any method structure always has the method identifier as the first field. The DoMethodA() call gets a pointer to the method structure, the structure is built by the application. The second form is rarely used. The number and meaning of parameters, as well as the meaning of the result are method specific. Comparison of executing a method with both forms of the call is given below:

```
struct MUIP_SomeMethod
{
  ULONG MethodID;
  LONG ParameterA;
  LONG ParameterB;
};

DoMethod(object, MUIM_SomeMethod, 3, 7);

struct MUIP_SomeMethod mparams = { MUIM_SomeMethod, 3, 7 };
DoMethodA(object, &mparams);
```

The DoMethod() form is more convenient, so it is commonly used. MUI uses specific prefixes for all its structures and constants:

- MUIM_ for method identifiers.
- MUIP_ for method parameter structures.
- MUIA_ for attribute identifiers.
- MUIV_ for special, predefined attribute values.

The C types used in the method structure above may need some explanation. LONG is a 32-bit signed integer, ULONG is an unsigned one. Because the structure is usually built on the processor stack, all parameters are extended and aligned to 32 bits. Then every parameter in the structure must be defined either as a 32-bit integer or a pointer. Any parameter larger than 32 bits must be passed via pointer (for example double precision floats or strings).

2.2.1.4 Setting an attribute

An object's attributes represent its properties. They are written and read using special methods, OM_SET() and OM_GET() respectively. This differs from most object oriented programming languages, where attributes (being implemented as an object's fields) are accessed directly. Manipulating attributes in BOOPSI is slower then, as it implies performing a method. The OM_SET() method does not take a single attribute and its value, but a taglist of them, so one can set multiple attributes at once. The setting of two attributes to an object may be done as follows:

```
1 struct TagItem attributes[] = {
2   { MUIA_SomeAttr1, 756 },
3   { MUIA_SomeAttr2, 926 },
4   { TAG_END, 0 }
5 };
6
7 DoMethod(object, OM_SET, (ULONG)attributes);
```

However, this is cumbersome and the code is not easily readable. The intuition.library makes it easier by providing the SetAttrsA() function, which is a wrapper for the OM_SET() method. Using this function and the array defined above, one can write:

```
1 SetAttrsA(object, attributes);
```

It still requires definition of a temporary taglist, but the function also has a variadic (meaning it can take a variable number of arguments) form SetAttrs(), which allows for building the taglist on-the-fly:

```
1 SetAttrs(object,
2   MUIA_SomeAttr1, 756,
3   MUIA_SomeAttr2, 926,
4 TAG_END);
```

This is not all however. Programmers are lazy and decided that in the common case of setting a single attribute, SetAttrs() is still too much typing. A common practice found in sources using MUI was to define an xset() or set() macro, which is now defined in the system headers, in the <libraries/mui.h> file.

```
1 #define set(object, attribute, value)\
2         SetAttrs(object, attribute, value, TAG_END)
```

Then, setting a single attribute can be coded as follows:

```
1 set(object, MUIA_SomeAttr1, 756);
```

The OM_SET() method returns the number of attributes applied to the object. If some attributes are not known to the object's class (and superclasses), they are not counted. This return value is usually ignored, it may be used for testing an attribute applicability.
MUI provides a few additional methods for setting attributes, namely MUIM_Set(), MUIM_NoNotifySet() and MUIM_MultiSet(). They are mainly used in notifications.

2.2.1.5 Getting an attribute

The OM_GET() method gets a single attribute from an object. There is no multiple attributes getting method. Its first, obvious parameter is the attribute identifier. The attribute value is not returned as the result of the method however. Instead the second parameter is a pointer to a memory area, where the value is to be stored. This allows for passing attributes larger than 32 bits, they are just copied to the pointed memory area. This only works for fixed size attributes. Text strings cannot be passed this way, so they are passed as pointers (a pointer to the string is stored at a place in memory pointed to by the second parameter of OM_GET()). The three examples below demonstrate all three cases:

```
1 ONG value1;
2 QUAD value2;    /* 64-bit signed integer */
3 STRPTR *value3;
4 /* integer attr */
5 DoMethod(object, OM_GET, MUIA_Attribute1, (ULONG)&value1);
6 /* fixed size big attr */
7 DoMethod(object, OM_GET, MUIA_Attribute2, (ULONG)&value2);
8  /* string attr */
9 DoMethod(object, OM_GET, MUIA_Attribute3, (ULONG)&value3);
```

In cases when an attribute is returned by pointer, the data pointed to should be treated as read-only unless documented otherwise.
Similarly as for OM_SET(), there is a wrapper function for OM_GET() in the intuition.library, named GetAttr(). This function unexpectedly changes the order of arguments: attribute identifier is the first, object pointer is the second. The three examples above may be written with GetAttr() as follows:

```
1 GetAttr(MUIA_Attribute1, object, &value1);
2 GetAttr(MUIA_Attribute2, object, (ULONG*)&value2);
3 GetAttr(MUIA_Attribute3, object, (ULONG*)&value3);
```

The third parameter, a storage pointer is prototyped as pointer to ULONG, so in the first example type casting is not needed.
The <libraries/mui.h> system header file defines a macro get(), which reverses the order of the two first arguments of GetAttr() and adds the typecasting to ULONG*. The order of arguments of get() is the same as for set(), which helps to avoid mistakes. The third line of the above example may be rewritten with get() this way:

```
1 get(object, MUIA_Attribute3, &value3);
```

The most often used attributes are integers (32-bit or shorter) and strings. Both of them fit into a 32-bit variable, as strings have to be passed via pointers. Taking this into account, MUI programmers invented a function (sometimes defined as a macro), which just returns the attribute value instead of storing it at a specified address. The function is named xget() and works as shown below:

```
value1 = xget(object, MUIA_Attribute1);
/* MUIA_Attribute2 can't be retrieved with xget() */
value3 = (STRPTR)xget(object, MUIA_Attribute3);
```

The xget() function may be defined in the following way:

```
inline ULONG xget(Object *obj, ULONG attribute)
{
  ULONG value;

  GetAttr(attribute, object, &value);
  return value;
}
```

The function is very simple and is compiled to a few processor instructions. That is why it is declared as inline, which causes the compiler to insert the function's code in-place instead of generating a jump. This makes the code faster, albeit a bit bigger. Except for working only with 32-bit attributes, xget() also has the disadvantage of loosing the OM_GET() return value. The value is boolean and is TRUE if the object's class (or any of its superclasses) recognizes the attribute, FALSE otherwise. This value is usually ignored, but may be useful for scanning objects for supported attributes.

The xget() function is not defined in the system headers. It has been described here because of its common use in MUI applications. Its counterparts for bigger sized arguments may be defined if needed.

2.2.1.6 Object construction

Having a class, the programmer can create an unlimited number of objects (instances) of this class. Every object has its own instance data area, which is allocated and cleared automatically by the BOOPSI. Of course only object data are allocated for each instance. The code is not duplicated, so it must be reentrant (static variables and code self-modification must not be used).

Objects are created and disposed with two special methods: the constructor, OM_NEW() and the destructor, OM_DISPOSE(). Of course the constructor method cannot be called on an object, because it creates a new one. It needs a pointer to the object's class instead, so it cannot be invoked with DoMethod(). The intuition.library provides NewObjectA() and NewObject() functions for calling the constructor. The difference between them is that NewObjectA() takes a pointer to a taglist specifying initial values for objects. NewObject() allows the programmer to build this taglist from a variable number of function arguments.

NewObject[A]() has two alternative ways of specifying the created object's class. Private classes are specified by pointers of Class type. Shared classes are specified by name, which is a null-terminated string. If the pointer is used for class specification, the name should be NULL, if a name is used, the pointer should be NULL. Four examples below show creating instances of private and public class with both NewObjectA() and NewObject():

```
Object *obj;
Class *private;

struct TagItem initial = {
  { MUIA_Attribute1, 4 },
  { MUIA_Attribute2, 46 },
  { TAG_END, 0 }
};
```

A private class, NewObjectA():

```
obj = NewObjectA(private, NULL, initial);
```

A private class, NewObject():

```
obj = NewObject(private, NULL,
  MUIA_Attribute1, 4,
  MUIA_Attribute2, 46,
TAG_END);
```

A public class, NewObjectA():

```
obj = NewObjectA(NULL, "some.class", initial);
```

A public class, NewObject():

```
obj = NewObject(NULL, "some.class",
  MUIA_Attribute1, 4,
  MUIA_Attribute2, 46,
TAG_END);
```

NewObject[A]() returns NULL in case of an object creation failure. Usual reasons are: wrong class pointer/name, lack of free memory, wrong/missing initial values of attributes. The return value of NewObject[A]() should always be checked in the code.

2.2.1.7 Object destruction

The OM_DISPOSE() method is used to destroy an object. Unlike OM_NEW() the destructor may be invoked with DoMethod():

```
DoMethod(object, OM_DISPOSE);
```

The intuition.library has a wrapper for this however, named DisposeObject():

```
DisposeObject(object);
```

2.2.1.8 MUI Extensions to BOOPSI

The Magic User Interface not only builds on BOOPSI but also extends it. Other than providing a broad set of classes, MUI also modifies the BOOPSI mode of operation a bit. Two modifications are discussed in this chapter: extension of the IClass structure and MUI's own functions for object construction and destruction.

MUI uses the MUI_CustomClass structure for its internal class representation. This structure contains the standard Class structure inside. It is important when creating objects from MUI private classes with NewObject(), that the Class structure must be extracted from the MUI_CustomClass structure:

```
1 struct MUI_CustomClass *priv_class;
2 Object *obj;
3
4 obj = NewObject(priv_class->mcc_Class, NULL, /* ... */ TAG_END);
```

MUI's second modification of BOOPSI is using its own functions for object construction and destruction, MUI_NewObject[A]() and MUI_DisposeObject() respectively. These two functions are used only for objects of MUI shared (public) classes. Objects of private classes are created with NewObject() as shown above. The main advantage of MUI_NewObject() is automatic opening and closing of disk based shared classes. Here is an example:

```
1 Object *text;
2
3 text = MUI_NewObject(MUIC_Text, MUIA_Text_Contents, "foobar", TAG_END);
```

MUIC_Text is a macro defined in <libraries/mui.h> and it expands to "Text.mui" string. All MUI public classes should be referenced by their MUIC_ macros rather than by direct string literals. It helps to detect mistyped class names, as a typo in a macro will be detected during compilation. The MUI checks if a class named Text.mui has been added to the public list of classes. If not, the class is found on disk, opened and used for creating the requested object. Closing the class when no longer in use is handled automatically too. All MUI objects should be disposed using MUI_DisposeObject(), which takes the object to be disposed as its only argument, the same as DisposeObject().

```
1 MUI_DisposeObject(text);
```

2.2.2 Event Driven Programming, Notifications

2.2.2.1 Event Driven Programming

Event driven programming is the natural consequence of the invention and development of graphical user interfaces. Most traditional, command line programs work like a pipe: data are loaded, processed and saved. There is no, or limited user interaction (like adjusting

processing parameters, or choosing an alternative path). A GUI changes all that. A GUI based program initializes itself, opens a window with some icons and gadgets, then waits for user actions. Fragments of the program are executed in response to user input, after an action is finished, the program goes back to waiting. This way the program flow is determined not by the code, but rather by input events sent to the program by the user via the operating system. This is the basic paradigm of event driven programming.

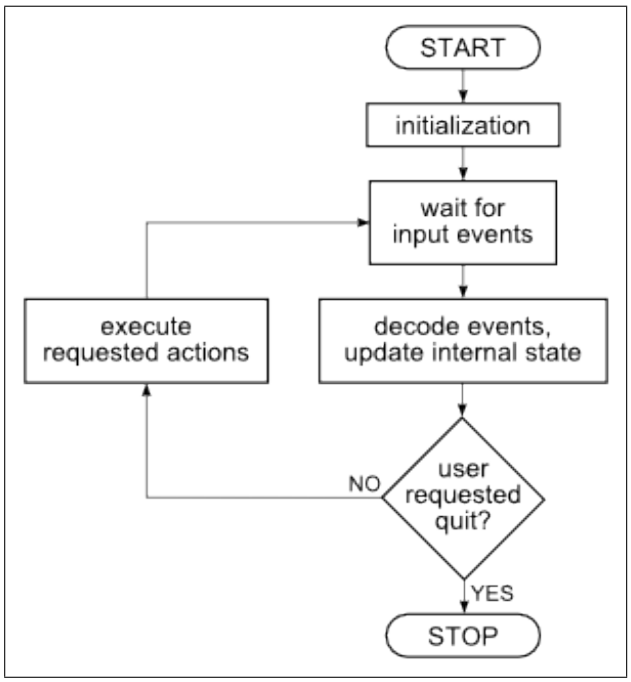

Figure 2.3: Execution flow of an event driven program

2.2.2.2 Notifications in MUI

There are two approaches to input event decoding in an event driven program: centralized and decentralized. Centralized decoding is programmed as a big conditional statement (a switch statement usually, or a long cascade of if statements) inside the main loop of the fig. 1. flowchart. Depending on the decoded event, subroutines performing requested actions are called. Decentralized input event decoding is a more modern idea. In this case, the GUI toolkit receives and handles incoming input events internally and maps them to attribute changes of GUI objects (for example, clicking with the mouse on a button changes its attribute to ”pressed”). Then an application programmer can assign actions to attribute changes of chosen objects. This is done by creating notifications on objects.

MUI uses decentralized input event decoding. All input events are mapped to attribute changes of different objects. These are visible GUI gadgets (controls) usually, but some events may be mapped to attributes of a window object, or an application object (the last one has no visible representation). After creating the complete object tree, but before entering the main loop, the program sets up notifications, assigning actions to attribute changes. Notifications can also be created and deleted dynamically at any time.

A notification connects two objects together. The source object triggers the action after one of its attributes changes. The assigned action (method) is then performed on the target

object. The notification is set up by calling the MUIM_Notify() method on the source object. Arguments of the method can be divided into the source part and the target part. The general form of the MUIM_Notify() call is shown below:

```
DoMethod(source,
         MUIM_Notify,
         attribute,
         value,
         target,
         param_count,
         action,
         /* parameters */);
```

The first four arguments form the source part, the rest is the target part. The complete call can be ”translated” to a human language in the following way:

When the source object changes its attribute to the value,
perform action method on the target object with parameters.

There is one argument not explained with the above sentence, namely param_count. This is just the number of parameters following this argument. The minimum number of parameters is 1 (the action method identifier), there is no upper limit other than using common sense.

A notification is triggered when an attribute is set to a specified value. It is often useful to have a notification on any attribute change. A special value MUIV_EveryTime should be used as a triggering value in this case.

The target action of a notification can be any method. There are a few methods designed specifically to be used in notifications:

MUIM_Set() is another method for setting an attribute. It is used when a notification has to set an attribute on the target object. OM_SET() is not suitable for using in notifications because it takes a taglist containing attributes to be set. This taglist cannot be built from arguments and must be defined separately. MUIM_Set() sets a single attribute to a specified value. They are passed to the method directly as two separate arguments. The example below opens a window when a button is pressed:

```
DoMethod(button,
         MUIM_Notify,
         MUIA_Pressed,
         FALSE,
         window,
         3,
         MUIM_Set,
         MUIA_Window_Open,
         TRUE);
```

Those not familiar with MUI may ask why the triggering value is set as FALSE. It is related to the default behaviour of button gadgets. The gadget triggers when the left mouse button is released, so at the end of a click. The MUIA_Pressed attribute is set to TRUE when the

mouse button is pushed down, and set to FALSE when the mouse button is released. Now it should be obvious why the notification is set to trigger at a MUIA_Pressed change to FALSE.

MUIM_NoNotifySet() works the same as MUIM_Set() with one important exception. It does not trigger any notifications set on the target object when the attribute has changed. It is often used to avoid notification loops, not only in notification, but also standalone in the code.

MUIM_MultiSet() allows for setting the same attribute to the same value for multiple objects. Objects are specified as this method's arguments and the final argument should be NULL. Here is an example, disabling three buttons after a checkmark is deselected:

```
1  DoMethod(checkmark,
2           MUIM_Notify,
3           MUIA_Selected,
4           FALSE,
5           application,
6           7,
7           MUIM_MultiSet,
8           MUIA_Disabled,
9           TRUE,
10          button1,
11          button2,
12          button3,
13          NULL);
```

What is interesting is that while the target notification object is completely irrelevant here, it must still be a valid object pointer. The application object is usually used for this purpose, or the notification source object.

MUIM_CallHook() calls an external callback function called a hook. It is often abused by programmers being reluctant to perform subclassing of standard classes, instead implementing program functionality as new methods. Calling a method from a notification is usually faster and easier (however, a hook needs some additional structures to be defined).

MUIM_Application_ReturnID() returns a 32-bit integer number to the main loop of a MUI program. With this method MUI's decentralized handling of input events can be turned into a centralized one. MUI programming beginners tend to abuse this method and redirect all the event handling back to the main loop, putting a big switch statement there. While rather simple, this programming technique should be avoided in favour of implementing program functionality in methods. Adding code inside the main loop degrades the GUI responsiveness. The only legitimate use of MUIM_Application_ReturnID() is to return a special value MUIV_Application_ReturnID_Quit used for ending the program.

2.2.2.3 Reusing Triggering Value

When the action of a notification is to set an attribute in the target object, it is often desired to forward the triggering value to the target object. It is very easy, when the notification is set to occur on a particular value. Things change however, when the notification is set to occur on any value with MUIV_EveryTime. A special value MUIV_TriggerValue may be used for this. It is replaced with the actual value of the triggering attribute at every trigger. Another

special value, MUIV_NotTriggerValue is used for boolean attributes and is replaced by a logical negation of the current value of the triggering attribute.

The first example uses MUIV_Trigger_Value, to display the value of a slider in a string gadget:

```
DoMethod(slider,
         MUIM_Notify,
         MUIA_Numeric_Value,
         MUIV_EveryTime,
         string,
         3,
         MUIM_Set,
         MUIA_String_Integer,
         MUIV_TriggerValue);
```

The second example connects a checkmark with a button. When the checkmark is selected, the button is enabled. Deselecting the checkmark disables the button:

```
DoMethod(checkmark,
         MUIM_Notify,
         MUIA_Selected,
         MUIV_EveryTime,
         button, 3,
         MUIM_Set,
         MUIA_Disabled,
         MUIV_NotTriggerValue);
```

MUIV_EveryTime, MUIV_TriggerValue and MUIV_NotTriggerValue are defined as particular values in the 32-bit range. Because of this, it is impossible to set a notification on the value 1 233 727 793 (which is MUIV_EveryTime). It is also impossible to set the value to a fixed number 1 233 727 793 (MUIV_TriggerValue) or 1 233 727 795 (MUIV_NotTriggerValue) using MUIM_Set() in a notification.

2.2.2.4 Notification loops

There may be hundreds of notifications defined in a complex program. Changing an attribute may trigger a notification cascade. It is possible that the cascade contains loops. The simplest example of a notification loop is a pair of objects having notifications on each other. Let's assume there are two sliders which should be coupled together. It means moving one slider should move the other one as well. A set of two notifications can ensure this behaviour.

```
DoMethod(slider1,
         MUIM_Notify,
         MUIA_Numeric_Value,
         MUIV_EveryTime,
         slider2,
         3,
         MUIM_Set,
         MUIA_Numeric_Value,
```

```
9           MUIV_Trigger_Value);
10 DoMethod(slider2,
11          MUIM_Notify,
12          MUIA_Numeric_Value,
13          MUIV_EveryTime,
14          slider1,
15          3,
16          MUIM_Set,
17          MUIA_Numeric_Value,
18          MUIV_Trigger_Value);
```

When the user moves slider1 to value 13, the first notification triggers and sets slider2 value to 13. This triggers the second notification. Its action is to set slider1 value to 13, which in turn triggers the first notification again. Then the loop triggers itself endlessly... Or rather it would, if MUI had no anti-looping measures. The solution is very simple: if an attribute is set for an object to the same value as the current one, any notifications on this attribute in this object are ignored. In our example the loop will be broken after the second notification sets the value of slider1.

2.2.2.5 The ideal MUI main loop

The ideal main loop of a MUI program should contain almost no code inside. All actions should be handled with notifications. Here is the code of the loop:

```
1  ULONG signals = 0;
2
3  while (DoMethod(application,
4                  MUIM_Application_NewInput,
5                  (ULONG)&signals)
6         != (ULONG)MUIV_Application_ReturnID_Quit)
7  {
8    signals = Wait(signals | SIGBREAKF_CTRL_C);
9    if (signals & SIGBREAKF_CTRL_C) break;
10 }
```

The variable signals contains a bit-mask of input event signals sent to the process by the operating system. Its initial value 0 just means ”no signals”. When the MUIM_Application_NewInput() method is performed on the application object, MUI sets signal bits for input events it expects in the signals variable. These signals are usually signals of application windows’ message ports, where Intuition sends input events. Then the application calls the exec.library function Wait(). Inside this function, the execution of the process is stopped. The process scheduler will not give any processor time to the process until one of the signals in the mask arrives. Other than input event signals set by MUI, only the system CTRL-C signal is added to the mask. Every well written MorphOS application should be breakable by sending this signal. It can be sent via the console by pressing the CTRL + C keys, or from tools like the TaskManager. When one of the signals in the mask arrives at the process, the program returns from Wait(). If the CTRL-C signal is detected, the main loop ends. If not, MUI decodes the received input events based on its received signal mask, translates events to changes of attributes of relevant objects and performs the triggered notifications. All this happens inside the MUIM_Application_NewInput() method. Finally the signal mask is updated. If any

notification calls the MUIM_Application_ReturnID() method, he identifier passed is returned as the result of MUIM_Application_NewInput(). In the event of receiving the special value MUIV_Application_ReturnID_Quit the loop ends.

Any additional code inserted into the loop will introduce delay in GUI handling and redrawing. If a program does some processor intensive calculations, the best way to deal with them is to delegate them into a subprocess. Loading the main process with computational tasks may result in the program being perceived as slow and unresponsive to user actions.

2.2.3 ”Hello World!” in MUI

This is the window of the example MUI application. It is very simple, it contains only one static text object and no gadgets, except for window border gadgets. Note that the number of gadgets on the right side of the window depends on user MUI settings. The source code of the application is very simple too and fits into 60 lines, including proper vertical spacing. It is cut in pieces in the article for better reading. A complete, ready to compile version is available too.

Figure 2.4: Screenshot of HelloWorld in MUI

```
1 #include <proto/muimaster.h>
2 #include <proto/intuition.h>
```

Header files for muimaster.library and intuition.library are included. Note that these libraries will be opened and closed automatically.

```
1 Object *App, *Win;
```

Global pointers for the application object and the window object. While using many global variables is considered inelegant, having globals for the most important objects is handy, especially in such a small project.

```
1 Object* build_gui(void)
2 {
3   App = MUI_NewObject(MUIC_Application,
4     MUIA_Application_Author, (ULONG)"Grzegorz Kraszewski",
5     MUIA_Application_Base, (ULONG)"HELLOWORLD",
6     MUIA_Application_Copyright, (ULONG)"© 2010 Grzegorz Kraszewski",
```

```
7      MUIA_Application_Description, (ULONG)"Hello World in MUI.",
8      MUIA_Application_Title, (ULONG)"Hello World",
9      MUIA_Application_Version, (ULONG)"$VER: HelloWorld 1.0 (16.11.2010)",
10     MUIA_Application_Window, (ULONG)(Win = MUI_NewObject(MUIC_Window,
11       MUIA_Window_Title, (ULONG)"Hello World",
12       MUIA_Window_RootObject, MUI_NewObject(MUIC_Group,
13         MUIA_Group_Child, MUI_NewObject(MUIC_Text,
14           MUIA_Text_Contents, (ULONG)"Hello world!",
15         TAG_END),
16       TAG_END),
17     TAG_END)),
18   TAG_END);
19 }
```

The above function creates the complete object tree for HelloWorld. The master object is the Application class instance. It has a Window object. The window root is a Group object containing one Text object. The application object has 6 attributes working as descriptors used in different places in the system. They are not required for running the program, but help integrating it with the system. The meaning of these attributes is explained in the autodoc of Application class in the SDK. The rest of the attributes are self-explaining, please refer to the autodocs of respective MUI classes for details.

The function illustrates a typical way of creating a MUI interface. The complete object tree is created in one big MUI_NewObject() call containing nested sub-calls. The order of code execution is different to the order of reading. The most nested objects are created first and passed to constructors of their parents. The application object is created last. This way of creating the application also ensures automatic error handling. If any of constructors fails, it passes NULL to a parent constructor, making it fail too and then dispose all successfully constructed child objects. Finally the application constructor fails and returns NULL. Then only two states of the application are possible: either the application is fully constructed, or it is not constructed at all. This behaviour greatly simplifies error handling.

```
1 void notifications(void)
2 {
3   DoMethod(Win, MUIM_Notify, MUIA_Window_CloseRequest, TRUE, App, 2,
4    MUIM_Application_ReturnID, MUIV_Application_ReturnID_Quit);
5 }
```

The next step is to make notifications. This simple program contains only one notification, which terminates the program after the window close gadget is clicked. MUI maps a left mouse button click on the gadget to a change of MUIA_Window_CloseRequest attribute. The notification target is the application, the MUIM_Application_ReturnID() method then passes the quit action identifier to the main loop.

```
1 void main_loop(void)
2 {
3   ULONG signals = 0;
4
5   set(Win, MUIA_Window_Open, TRUE);
6
7   while (DoMethod(App, MUIM_Application_NewInput, &signals)
```

```
8           != MUIV_Application_ReturnID_Quit)
9   {
10    signals = Wait(signals | SIGBREAKF_CTRL_C);
11    if (signals & SIGBREAKF_CTRL_C) break;
12  }
13
14  set(Win, MUIA_Window_Open, FALSE);
15 }
```

The standard main loop has been discussed in the previous chapter. The only addition is opening the window before the loop and closing it after.

```
1  int main(void)
2  {
3    App = build_gui();
4
5    if (App)
6    {
7      notifications();
8      main_loop();
9      MUI_DisposeObject(App);
10   }
11
12   return 0;
13 }
```

Finally, the main function of the program. It builds the object tree first and checks if it succeeded. In case of a fail, the program ends immediately. After all the objects are created, notifications are added and the program enters the main loop. When the loop finishes, the application object is disposed. It also disposes all its sub-objects.

2.3 Subclassing

2.3.1 General Rules and Purpose of Subclassing

2.3.1.1 Introduction

Subclassing is one of the essential object oriented programming techniques. In MUI subclassing is used for the following purposes:

- Implementing program functionality as a set of methods. The Application class is usually used for this purpose.
- Customizing classes by writing methods intentionally left unimplemented (or having some default implementations) in standard MUI classes. The most common example is the List class, but also the Numeric one and others.
- Writing custom drawn gadgets or areas. The Area class is subclassed in this case.

Regardless of the reason for subclassing, it is always done in the same way. A programmer must write new methods or override existing methods, create a dispatcher function, define an instance data structure (an empty one in some cases), then create the class. It is worth noting, subclassing MUI classes is done the same as subclassing BOOPSI ones. The only difference is that MUI provides its own functions for class creation and disposition.

2.3.1.2 Object Data

Object data are stored in a memory area automatically allocated for every object created. The object data area is used for storing attribute values and for internal variables and buffers. This area is usually defined as a structure. Size of the area is passed to the MUI_CreateCustomClass() function. In a class hierarchy, every class may add its own contribution to the object data area. Unlike in C++, a class has no direct access to anything except its own data area. It can't access data defined in the superclass (In C++ it is possible that a field is declared as protected or public). Object data defined in any of the superclasses may only be accessed using methods or attributes provided by these superclasses.
Because of internal BOOPSI design, the size of the data instance area is limited to 64 kB. Large buffers should be allocated dynamically in OM_NEW() and freed in OM_DISPOSE(). The area data is always cleared to all zeros at object creation. If a class does not need any object instance data it can pass 0 as the area size to MUI_CreateCustomClass().

2.3.1.3 Writing Methods

A MUI method is just a plain C function, but with a partially fixed prototype.

```
1 IPTR MethodName(Class *cl, Object *obj, MessageType *msg);
```

The method return value may be either integer or pointer to anything. That is why it uses the IPTR type which means "integer big enough to hold a pointer". In the current MorphOS it is just a 32-bit integer (the same as LONG). If a method has no meaningful value to return, it can just return 0. Two first, fixed arguments are: pointer to the class and pointer to the object. The last one is a method message. When a method is being overridden, the type of message is determined by the superclass. For a new method, message type is defined by the programmer. Some methods may have empty messages (containing only a method identifier), in this case the third argument may be omitted. Most methods need access to the object instance data. To get a pointer to the data area, one uses the INST_DATA macro, defined in <intuition/classes.h>. An example below shows the macro usage:

```
1 struct ObjData
2 {
3   LONG SomeVal;
4   /* ... */
5 };
6
7 IPTR SomeMethod(Class *cl, Object *obj)
8 {
9   struct ObjData *d = (struct ObjData*)INST_DATA(cl, obj);
```

```
10
11   d->SomeVal = 14;
12   /* ... */
13   return 0;
14 }
```

If a method is an overridden method from a superclass, it may want to perform the superclass method. There are no implicit super method calls in MUI. The superclass method must always be called explicitly with the DoSuperMethodA() call:

```
1 result = DoSuperMethodA(cl, obj, msg);
2 result = DoSuperMethod(cl, obj, MethodID, ...);
```

The second form rebuilds the method message from variable arguments, and is used when the message is modified before calling the superclass method. The super method may be called in any place of the method, or may not be called at all. For MUI standard classes and methods, rules of calling super methods are described in the documentation and will be discussed later in this tutorial. For custom methods the question of calling a super method is up to the application programmer.

2.3.1.4 The Dispatcher

A dispatcher function is a kind of jump table for methods. When any method is called on an object (with DoMethod()), BOOPSI finds the dispatcher of the object's class and calls it. The dispatcher checks a method identifier, which is always the first field of any method message. Based on the identifier, a method is called. If a method is unknown to the class, the dispatcher should pass it to the superclass with the DoSuperMethod() call.

The dispatcher is a kind of hook function. It makes its calling convention independent of programming language. A disadvantage of this is that the dispatcher's arguments are passed in virtual M68k processor registers. This inconvenience allows support for legacy M68k software and also allows for native PowerPC classes to be used by M68k applications and old M68k classes to be used by native applications. Being a hook, a dispatcher needs an EmulLibEntry structure to be created and filled first. The structure is defined in <emul/emulinterface.h> and acts as a data gate between PowerPC native code and the M68k emulator.

```
1 const struct EmulLibEntry ClassGate =
2       {TRAP_LIB, 0, (void(*)(void))ClassDispatcher};
```

Then the dispatcher is defined as follows:

```
1 IPTR ClassDispatcher(void)
2 {
3   Class *cl = (Class*)REG_A0;
4   Object *obj = (Object*)REG_A2;
5   Msg msg = (Msg)REG_A1;
6
7   /* ... */
8  }
```

Arguments of the dispatcher are the same as arguments of a method. They are passed in virtual M680x0 processor address registers A0, A1 and A2 instead of being just arguments. The dispatcher's data gate is passed as an argument to MUI_CreateCustomClass(). The data gate is used even when a native application calls a native dispatcher. It introduces some overhead, but it's negligible. Many programmers prefer to hide these details behind a set of preprocessor macros, such macros have not been used here however, for better understanding.

The Msg type is a root type for all method messages. It defines a structure containing only the method identifier field (defined as ULONG). All following parameters have to keep the CPU stack aligned, as DoMethod() builds the message on the stack. It requires that every parameter is defined either as an IPTR or as a pointer.

After receiving arguments the dispatcher checks the method identifier from the message and jumps to the respective method. It is usually implemented as a switch statement. If only a few methods are implemented, it may also be an if/if else cascade. Here is a typical example:

```
1  switch (msg->MethodID)
2  {
3    case OM_NEW:
4        return MyClassNew(cl, obj, (struct opSet*)msg);
5    case OM_DISPOSE:
6        return MyClassDispose(cl, obj, msg);
7    case OM_SET:
8        return MyClassSet(cl, obj, (struct opSet*)msg);
9    case OM_GET:
10       return MyClassGet(cl, obj, (struct opGet*)msg);
11   case MUIM_Draw:
12       return MyClassDraw(cl, obj, (struct MUIP_Draw*)msg);
13   case MUIM_AskMinMax:
14       return MyClassAskMinMax(cl, obj, (struct MUIP_AskMinMax*)msg);
15   /* ... */
16   default:
17       return DoSuperMethodA(cl, obj, msg);
18 }
```

For every method a message pointer is typecast to a message structure of this particular method. Some programmers place the method's code inside the switch statement directly, especially if methods are short and only a few. In the example above, some methods of Area class are overridden. The naming scheme used for the method functions is just an example, there are no constraints on this. Although prefixing method function names with a class name has an advantage of avoiding name conflicts between custom classes if method functions are not declared as static.

2.3.1.5 Class Creation

Having all components done (methods, dispatcher, gate, object data structure) one can create a MUI class.

```
1 struct MUI_CustomClass *MyClass;
2
3 MyClass = MUI_CreateCustomClass(NULL, MUIC_Area,
4           NULL, sizeof(struct MyClassData), (APTR)&MyClassGate);
```

The first argument is a library base if the created class is public. Writing MUI public classes will be covered later. Let's say for now, that public classes are implemented as shared libraries, so such a public class has a library base. For private classes this argument should always be NULL.

The next two arguments are used alternatively and specify the superclass. The superclass may be either private (referenced by pointer) or public (referenced by name). Public classes are usually subclassed, so the pointer is set to NULL as in the example. More complex projects may use multilevel subclassing and subclass their own private classes. In this case, a pointer to a private class is passed as the first argument and the second one is NULL.

The fourth argument defines the size of the object data area in bytes. In most cases object data area is defined as a structure, so using the sizeof() operator is the obvious way of determining the size. If the class does not need any per-object data, zero may be passed here.

The last argument is an address of the data gate (EmulLibEntry structure). Programmers experienced on M68k programming may notice that there is a difference - in M68k code only the dispatcher function address is used here. As mentioned above, seamless M68k code support requires that program execution passes through the data gate when going from system code to the dispatcher. That is why the data gate address is placed as this argument, then the data gate contains a real dispatcher address.

2.3.1.6 Class Disposition

A MUI class is disposed with a call to MUI_DeleteCustomClass().

```
1 if (MyClass) MUI_DeleteCustomClass(MyClass);
```

Some conditions must be fulfilled before calling this function.

- Call it only if the class was successfully created. Calling it with a NULL class pointer is deadly (hence the checking for NULL in the example).
- Do not delete a class if it has any remaining subclasses or objects. The best practice is to create all classes before creating the application GUI and to dispose them after the final MUI_DisposeObject() of the main Application object. Classes should be deleted in the reversed order of creation. MUI_DeleteCustomClass() returns a boolean value. It is FALSE when a class cannot be deleted because of potentially orphaned subclasses or objects.

2.3.2 Overriding Constructors

An object constructor (OM_NEW() method), takes the same message structure opSet as the OM_SET() method. The message contains the ops_AttrList field, being a pointer to a taglist containing the initial object's attributes. Implementation of a constructor for an object without child objects is simple. The superclass constructor is called first, then, if it succeeds, the constructor initializes object instance data, allocates resources needed and sets initial values of attributes from tags passed via ops_AttrList.
A rule of thumb when overriding constructors is to never leave a half-constructed object. The constructor should either return a fully constructed object, or fail completely, freeing all successfully obtained resources. This is important if the object obtains more than one resource and any of the resource allocation has failed (for example allocating a big chunk of memory or opening a file). An example implementation below obtains three resources: A, B and C:

```
1  IPTR MyClassNew(Class *cl, Object *obj, struct opSet *msg)
2  {
3    if (obj = DoSuperMethodA(cl, obj, (Msg)msg))
4    {
5      struct MyClassData *d = (struct MyClassData*)INST_DATA(cl, obj);
6
7      if ((d->ResourceA = ObtainResourceA()
8       && (d->ResourceB = ObtainResourceB()
9       && (d->ResourceC = ObtainResourceC())
10     {
11       return (IPTR)obj;    /* success */
12     }
13     else CoerceMethod(cl, obj, OM_DISPOSE);
14   }
15   return NULL;
16 }
```

If the object destructor frees resources A, B and C (which would be logical considering the constructor allocates them), the cleanup job may be delegated to the destructor. It requires however, that the destructor must be prepared for destruction of a not fully constructed object. It can't assume all three resources have been allocated, so it should check every resource pointer against NULL before calling a freeing function. The destructor also takes care of calling a superclass destructor when resources are freed. See Overriding Destructors for some destructor example code and explanation.
The only question remaining is what CoerceMethod() does and why it is used instead of a plain DoMethod()? The CoerceMethod() call works exactly the same as DoMethod(), but performs method coercion by a forced call to the dispatcher of the class specified as the first argument instead of the dispatcher of the object's true class. It makes a difference, when the class in question is later subclassed. The flowchart below explains the problem:

The class B on the diagram is a subclass of the class A and similarly, the class C is a subclass of B. Let's assume an object of the class C is being constructed. As every constructor calls the superclass first, the call goes up to rootclass (the root of all BOOPSI classes) first. Then going down the class tree, every class constructor allocates its resources. Unfortunately the constructor of class A has been unable to allocate one of its resources and decided to fail. If it had just called DoMethod(obj, OM_DISPOSE) it will unnecessarily execute destructors in classes B and C, while constructors in these classes have been not yet

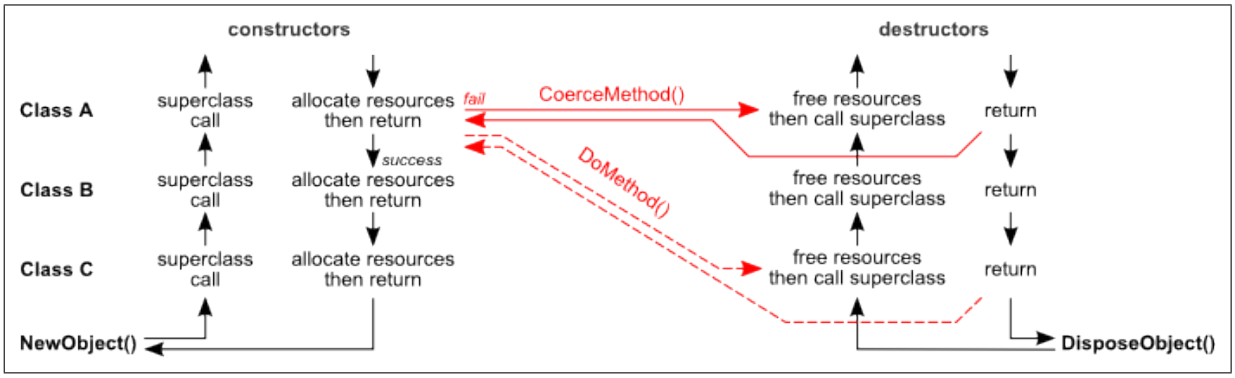

Figure 2.5: Flowchart of CoerceMethod()

fully executed. Even if these destructors can cope with this, calling them is superfluous. With the CoerceMethod() the destructor in the class A is called directly. Then class A constructor returns NULL, which causes constructors in classes B and C to fail immediately without resource allocation attempts.

2.3.2.1 Objects with child objects

While retaining the same principles, the constructor of an object with subobjects is designed a bit differently. The most commonly subclassed classes able to have child objects are Application, and Group. The Window class is also often subclassed in a similar way. While a Window object can have only one child, specified by MUIA_Window_RootObject, this child often has multiple subobjects. The constructor should create its child objects first, then insert them into the ops_AttrList taglist and call the superclass constructor. If it succeeds, then resources may be allocated if needed. As any of the three constructor stages may fail, proper handling of errors becomes complicated. Also inserting objects created into the taglist as values of child tags (like MUIA_Group_Child) is cumbersome. Fortunately one can use the DoSuperNew() function, which merges the creation of subobjects and the calling of the superclass into one operation. It also provides automatic handling of failed child object construction. An example below is a constructor for a Group subclass putting two Text objects in the group.

```
1  IPTR MyClassNew(Class *cl, Object *obj, struct opSet *msg)
2  {
3    if (obj = DoSuperNew(cl, obj,
4      MUIA_Group_Child, MUI_NewObject(MUIC_Text,
5        /* attributes for the first subobject */
6      TAG_END),
7      MUIA_Group_Child, MUI_NewObject(MUIC_Text,
8        /* attributes for the second subobject */
9      TAG_END),
10   TAG_MORE, msg->ops_AttrList))
11   {
12     struct MyClassData *d = (struct MyClassData*)INST_DATA(cl, obj);
13
14     if ((d->ResourceA = ObtainResourceA()
15      && (d->ResourceB = ObtainResourceB()
16      && (d->ResourceC = ObtainResourceC())
17     {
```

```
18        return (IPTR)obj;    /* success */
19      }
20      else CoerceMethod(cl, obj, OM_DISPOSE);
21    }
22    return NULL;
23  }
```

An important thing to observe is the fact, that DoSuperNew() merges the taglist passed to the constructor via the message ops_AttrList field and the one specified in the function arguments list. It is done with a special TAG_MORE tag, which directs a taglist iterator (like NextTagItem() function) to jump to another taglist pointed by the value of this tag. Taglist merging allows for modifying the object being constructed with tags passed to NewObject(), for example adding a frame or background to the group in the above example.
The automatic handling of failed child objects works in the following way: when a subobject fails, its constructor returns NULL. This NULL value is then inserted as the value of a ”child” tag (MUIA_Group_Child) in the example. All MUI classes able to have child objects are designed in a way that:

- the constructor fails if any ”child” tag has a NULL value,
- the constructor disposes any successfully constructed child objects before exiting.

Finally DoSuperNew() returns NULL as well. This design ensures that in case of any fail while building the application, all objects created are disposed and there are no orphaned ones.

2.3.3 Overriding Destructors

The only task of a destructor is freeing resources allocated by the constructor and other methods (some resources may be allocated on-demand only). In any case the destructor must leave the object in the same state as right after DoSuperMethod()/DoSuperNew() in the constructor. After that the destructor calls a super class destructor. The destructor receives an empty message.

```
1 IPTR MyClassDispose(Class *cl, Object *obj, Msg msg)
2 {
3   struct MyClassData *d = (struct MyClassData*)INST_DATA(cl, obj);
4
5   if (d->ResourceA) FreeResourceA();
6   if (d->ResourceB) FreeResourceB();
7   if (d->ResourceC) FreeResourceC();
8   return DoSuperMethodA(cl, obj, msg);
9 }
```

The example destructor follows the example of the constructor in the Overriding Constructors article. Three resources obtained in the constructor are freed here. The destructor is also prepared for a partially constructed object, every resource is checked against NULL before freeing. If for some type of resource NULL is a valid handle, an additional flag may be added to the object instance data area.

2.3.4 Overriding OM_SET()

The OM_SET() method receives an opSet structure as its message. The structure is defined in the <intuition/classusr.h> header.

```
struct opSet
{
  ULONG              MethodID;      /* always OM_SET (0x103) */
  struct TagItem    *ops_AttrList;
  struct GadgetInfo *ops_GInfo;
};
```

The most important field is ops_AttrList. It is a pointer to a taglist containing attributes and values to be set. The ops_GInfo field is an obsolete legacy thing and is not used by modern components like MUI or Reggae. The method implementation should iterate the taglist and set all attributes recognized. The operation of setting an attribute may be just setting some field in an object instance data, it may also trigger some actions (like for example object redrawing). It is recommended however that complex actions are implemented as methods rather than attribute changes. A reference implementation of OM_SET() may look like this:

```
IPTR MyClassSet(Class *cl, Object *obj, struct opSet *msg)
{
  struct TagItem *tag, *tagptr;
  IPTR tagcount = 0;

  tagptr = msg->ops_AttrList;

  while ((tag = NextTagItem(&tagptr)) != NULL)
  {
    switch (tag->ti_Tag)
    {
      case SOME_TAG:
        /* attribute setting actions for SOME_TAG */
        tagcount++;
      break;

      /* more tags here */
    }
  }

  tagcount += DoSuperMethodA(cl, obj, (Msg)msg);
  return tagcount;
}
```

The taglist iteration is done with the NextTagItem() function from the utility.library. The function returns a pointer to the next tag each time it is called and keeps the current position in tagptr. The advantage of this function is automatic handling of special tag values (TAG_MORE, TAG_IGNORE, TAG_SKIP), they are not returned, but their actions are performed instead.

The OM_SET() function returns the total number of recognized tags. It is implemented with

tagcounter. It gets incremented on every tag recognized and finally the number of tags recognized by superclass(es) is added.

Common bugs in OM_SET() implementation are:

- ignoring tag counting
- calling the super method in the default case of a switch statement. It causes the supermethod to be called multiple times, once for every tag not handled by the subclass.

2.3.5 Overriding OM_GET()

The OM_GET() method, used for getting an attribute from an object, receives an opGet structure as its message. The structure is defined in the <intuition/classusr.h> header file:

```
1 struct opGet
2 {
3   ULONG  MethodID;     /* always OM_GET (0x104) */
4   ULONG  opg_AttrID;
5   ULONG *opg_Storage;
6 };
```

Unlike OM_SET(), this method handles only one attribute at a time. The attribute is placed in the opg_AttrID field. The field opg_Storage is a pointer to a place where the attribute value should be stored. It is defined as a pointer to ULONG, but it may point to anything (for example to some larger structure). It allows for passing attributes not fitting in a 32-bit variable. Because OM_GET() does not have a taglist iteration loop, its implementation is simple:

```
1  IPTR MyClassGet(Class *cl, Object *obj, struct opGet *msg)
2  {
3    switch (msg->opg_AttrID)
4    {
5      case Some_Integer_Tag:
6        *msg->opg_Storage = /* value of the tag */;
7      return TRUE;
8
9      case Some_String_Tag:
10       *(char**)msg->opg_Storage = "a fixed string value";
11     return TRUE;
12   }
13
14   return DoSuperMethodA(cl, obj, (Msg)msg);
15 }
```

The implementation consists of a switch statement with cases for all recognized attributes. If an attribute is recognized, the method should return TRUE. It is very important, as MUI notifications rely on OM_GET() and will not work on the attribute if TRUE is not returned. Unknown attributes are passed to the superclass. The DoSuperMethodA() call may be alternatively placed as the default clause of the switch statement. It is important that msg-> opg_Storage is dereferenced when storing the attribute value. If the value type is not integer, a typecast is needed. For a value type T, the dereferencing combined with typecast is denoted as *(T*).

2.3.6 Subclassing Application Class

Every MUI application is (or at least should be) an event driven one. This means the application provides a set of actions, which may be triggered with user activity (like using the mouse and keyboard). The straightforward way of implementing this set of actions is to implement it as a set of methods added to some MUI object. For simple programs, the best candidate for adding such methods is the master Application object. More complex programs (for example ones using a multi-document interface) may add actions to other classes, for example the Window class.

Why methods? Implementing actions as methods has many advantages:

- Methods may be used directly as notification actions. It saves a programmer from using hook tricks or cluttering the main loop with lots of ReturnID values.
- Methods may be coupled directly with scripting language interface (formerly known as ARexx interface) commands.
- Methods used in notifications are executed immediately in response to user actions. No delay is introduced by the main loop (especially if it is not empty).
- A notification triggering attribute value may be passed directly to a method as its parameter.
- Using methods improves code modularity and object encapsulation. Functionality meant to be handled in the scope of an object is handled in its method, without the code spreading across the project.

In a well designed MUI program, all program actions and functionality are implemented as methods and the program internal state is stored as a set of attributes and internal application instance data fields. An example of such a program is discussed throughly in the MUI Subclassing Tutorial: SciMark2 Port article.

2.3.7 MUI Subclassing Tutorial: SciMark2 Port

2.3.7.1 The application

Many programming tutorials tend to bore readers with some useless examples. In this one a ”real world” application will be ported to MorphOS and ”MUI-fied”. The application is SciMark 2[1]. SciMark is yet another CPU/memory benchmark. It performs some typical scientific calculations like Fast Fourier Transform, matrix LU decomposition, sparse matrix multiplication and so on. The benchmark measures mainly CPU speed at floating point calculations, cache efficiency and memory speed. Being written in Java initially, it has been rewritten in C (and in fact in many other languages). The C source is available on the project homepage[2].

The source uses only the pure ANSI C standard, so it is easily compilable on MorphOS using the provided Makefile. One has just to replace the $CC = cclineto$CC = gcc, to match the name of the MorphOS compiler. As a result, a typical shell-based application is obtained. Here are example results for a Pegasos 2 machine with G4 processor:

[1] http://math.nist.gov/scimark2/index.html
[2] http://math.nist.gov/scimark2/download_c.html

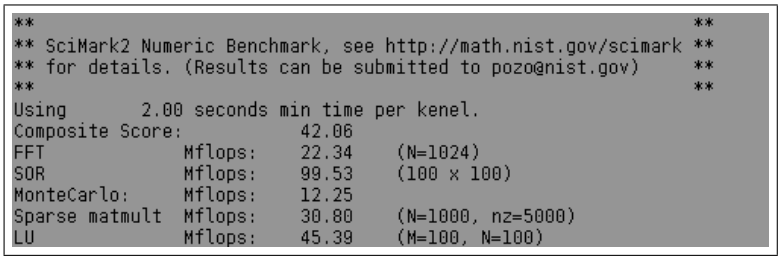

Figure 2.6: Screenshot of Scimark with no options

Not very impressive in fact. This is because no optimizaton flags are passed to the compiler in the makefile. They can be added by inserting the line $CFLAGS = -O3 below the$CC = gcc line. Let's also link with libnix (a statically linked unix environment emulation, see Standard C and C++ Libraries) by adding -noixemul to CFLAGS and LDFLAGS. After rebuilding the program and running it again the results are significantly improved (the program has been compiled with GCC 4.4.4 from the official SDK).

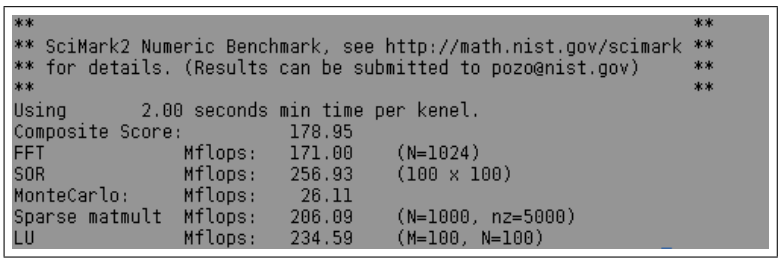

Figure 2.7: Screenshot of Scimark with options

This shows how important optimization of the code is, especially for computationally intensive programs. Optimized code is more than 4 times faster!

2.3.7.2 Code inspection

The original source code is well modularized. Five files: FFT.c, LU.c, MonteCarlo.c, SOR.c and SparseCompRow.c implement the five single benchmarks. Files array.c and Random.c contain auxiliary functions used in the benchmarks. The file Stopwatch.c implements time measurement. An important file kernel.c gathers all the above and provides timing for the five functions performing all the benchmarks. Finally scimark2.c contains the main() function and implements the shell interface.
The planned MUI interface should allow the user to run every benchmark separately or run all of them. There is also a -large option, which increases memory sizes for calculated problems, so they do not fit into the processor cache. A general rule of porting is that as few files as possible should be modified. The rule makes it easier to upgrade the port when a new version of the original program is released. In the case of SciMark, only one file, scimark2.c has to be replaced. An advanced port may also replace Stopwatch.c with code using timer.device directly for improved accuracy of time measurements however, this is out of scope of this tutorial.
A closer look at "scimark2.c" reveals that there is a Random object (a structure defined in "Random.h"), which is required for all the benchmarks. In the original code it is created with new_Random_seed() at the program start and disposed with delete_Random() at exit. The

best place for it in the MUI-ified version is the instance data area of the subclassed Application class. Then it can be created in OM_NEW() of the class and deleted in OM_DISPOSE(). These two methods should then be overridden.

2.3.7.3 GUI design

Of course there is no one and only proper GUI design for SciMark. A simple design, using a limited set of MUI classes is shown on the left. There are five buttons for individual benchmarks and one for running all of them. All these buttons are instances of the Text class. On the right there are gadgets for displaying benchmark results. These gadgets also belong to the Text class, just having different attributes. The ”Large Data” button, of the Text class of course, is a toggle button. Surprisingly the status bar (displaying ”Ready.”) is not an instance of the Text class, but instead the Gauge class. Then it will be able to display a progress bar when running all five tests. Spacing horizontal bars above the ”All Benchmarks” button are instances of the Rectangle class. There are also three invisible objects of the Group class. The first is a vertical, main group, being the root object of the window. It contains two subgroups. The upper one is the table group with two columns and contains all the benchmark buttons and result display gadgets. The lower group contains the ”Large Data” toggle button and the status bar.

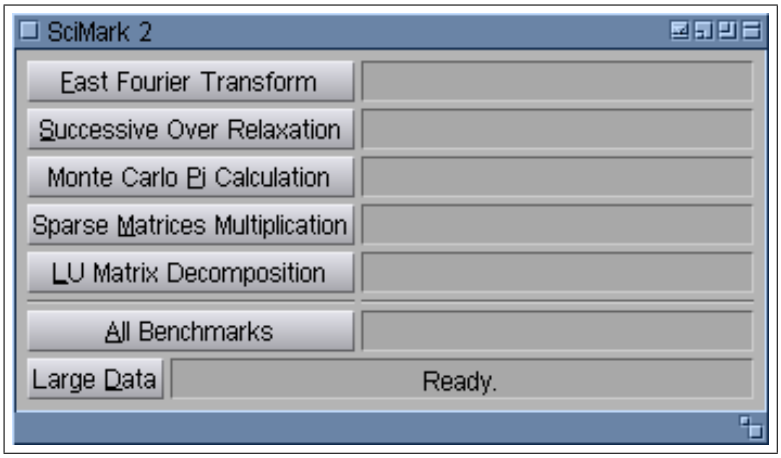

Figure 2.8: Screenshot of Scimark 2 GUI

The simplest way to start with GUI design is just to copy the ”Hello World” example. Then MUI objects may be added to the build_gui() function. The modified example is ready to compile and run. It is not a complete program of course, just a GUI model without any functionality added.
A quick view into the build_gui() function reveals that it does not contain all the GUI code. Code for some subobjects is placed in functions called from the main MUI_NewObject(). Splitting the GUI building function into many subfunctions has a few important advantages:

- Improved code readability and easier modifications. A single MUI_NewObject() call gets longer and longer quickly as the project evolves. Editing a large function spanning over a few screens is uncomfortable. Adding and removing GUI objects in such a function becomes a nightmare even with indentation used consequently. On the other hand the function can have 10 or more indentation levels, which makes it hard to read as well.

- Code size reduction. Instead of writing very similar code multiple times, for example buttons with different labels, a subroutine may be called with a label as an argument.
- Debugging. It happens sometimes that MUI refuses to create the application object because of some buggy tags or values passed to it. If the main MUI_NewObject() call is split into subfunctions, it is easy to isolate the buggy object by inserting some Printf()-s in subfunctions.

2.3.7.4 Methods and attributes

The SciMark GUI just designed, defines six actions for the application. There are five actions for running individual benchmarks and the sixth one for running all the tests and calculating the global result. Actions will be directly mapped to Application subclass methods. There is also one attribute connected with the ”Large Data” button, it determines the sizes of problems solved by benchmarks. Methods do not need any parameters, so there is no need to define method messages. An attribute may be applicable at initialisation time (in the object constructor), may be also settable (needs OM_SET() method overriding) and gettable (needs OM_GET() method overriding). Our new attribute, named APPA_LargeData in the code only needs to be settable. In the constructor it can be implicitly set to FALSE, as the button is switched off initially. GET-ability is not needed, because this attribute will be used only inside the Application subclass.
It is recommended that every subclass in the application is placed in a separate source file. It helps to keep code modularity and also allows for hiding class private data. This requires writing a makefile, but one is needed anyway, as the original SciMark code consists of multiple files. Implementing the design directions discussed above a class header file and class code can be written. The class still does nothing, just implements six empty methods and overrides OM_SET(), OM_NEW() and OM_DISPOSE(). In fact it is a boring template example and as such it has been generated with the ChocolateCastle template generator. Unfortunately ChocolateCastle is still beta, so files had to be tweaked manually after generation.
The next step in the application design is to connect methods and attributes with GUI elements using notifications. Notifications must of course be created after both source and target object are created. In the SciMark code they are just set up after executing build_gui(). All the six action buttons have very similar notifications, so only one is shown here:

```
1 DoMethod(findobj(OBJ_BUTTON_FFT, App), MUIM_Notify, MUIA_Pressed, FALSE,
2          App, 1, APPM_FastFourierTransform);
```

The ”Large Data” button has a notification setting the corresponding attribute:

```
1 DoMethod(findobj(OBJ_BUTTON_LDATA, App), MUIM_Notify, MUIA_Selected,
2     MUIV_EveryTime, App, 3, MUIM_Set, APPA_LargeData,
3     MUIV_TriggerValue);
```

Notified objects are accessed with dynamic search (the findobj() macro), which saves the programmer from defining global variables for all of them.

2.3.7.5 Implementing functionality

The five methods implementing single SciMark benchmarks are very similar, so only one, running the Fast Fourier Transform has been shown:

```
1  IPTR ApplicationFastFourierTransform(Class *cl, Object *obj)
2  {
3    struct ApplicationData *d = INST_DATA(cl, obj);
4    double result;
5    LONG fft_size;
6
7    if (d->LargeData) fft_size = LG_FFT_SIZE;
8    else fft_size = FFT_SIZE;
9
10   SetAttrs(findobj(OBJ_STATUS_BAR, obj),
11     MUIA_Gauge_InfoText,
12     (LONG)"Performing Fast Fourier Transform test...",
13     MUIA_Gauge_Current, 0,
14   TAG_END);
15
16   set(findobj(OBJ_RESULT_FFT, obj), MUIA_Text_Contents, "");
17   set(obj, MUIA_Application_Sleep, TRUE);
18   result = kernel_measureFFT(fft_size, RESOLUTION_DEFAULT, d->R);
19   NewRawDoFmt("%.2f MFlops (N = %ld)",
20   RAWFMTFUNC_STRING, d->Buf, result, fft_size);
21   set(findobj(OBJ_RESULT_FFT, obj), MUIA_Text_Contents, d->Buf);
22   set(obj, MUIA_Application_Sleep, FALSE);
23   set(findobj(OBJ_STATUS_BAR, obj), MUIA_Gauge_InfoText, "Ready.");
24   return 0;
25 }
```

The code uses dynamic object tree search for accessing MUI objects.
The method sets the benchmark data size first, based on the d->LargeData switch variable. This variable is set with the APPA_LargeData attribute, which in turn is bound to the ”Large Data” button via a notification. Then the status bar progress is cleared and some text is set to inform the user what is being done. The result textfield for the benchmark is cleared as well.
The next step is to put the application in the ”busy” state. It should be always done, when the application may not be responding to user input for anything longer than, let's say half a second. Setting MUIA_Application_Sleep to TRUE locks the GUI and displays the busy mouse pointer when the application window is active. Of course offloading processor intensive tasks to a subprocess is a better solution in general cases, but for a benchmark it makes little sense. A user has to wait for the benchmark result anyway before doing anything else, like starting another benchmark. The only usability problem is that a benchmark can't be stopped before it finishes. Let's leave it as is for now, for a benchmark, where the computer is expected to use all its computing power for benchmarking, a few seconds of GUI being unresponsive is not such a big problem.
The next line of code runs the benchmark, by calling kernel_measureFFT() function from the original SciMark code. After the benchmark is done, the result is formatted and displayed in the result field using NewRawDoFmt(), which is a low-level string formatting function from exec.library and with the RAWFMTFUNC_STRING constant, it works just like sprintf(). It uses a fixed buffer of 128 characters (which is much more than needed, but adds a safety margin) located in the object instance data. Unsleeping the application and setting the status bar text to ”Ready.” ends the method.

The APPM_AllBenchmarks() method code is longer so it is not repeated here. The method is very similar to the single benchmark method anyway. The difference is it runs all 5 tests accumulating their results in a table. It also updates the progress bar after every benchmark. Finally it calculates a mean score and displays it.

2.3.7.6 Final port

The complete source of SciMark2 MUI port[3]. The program may be built by running make in the source directory.

2.4 Useful Techniques

2.4.1 Locating Objects in the Object Tree

After the complete object tree is created, there is no direct access to any object except the main Application object. A way to access other objects is needed. There are a few ways to do this:

- Storing pointers to objects in global variables. This is the simplest way and may work well in simple projects. The disadvantage is it breaks object oriented design principles (like data encapsulation) and creates a mess when the number of global variables reaches 50, 100 or more.
- Store pointers in fields of some subclass instance data (for example the Application instance). A good idea, but a bit tedious to implement. An object's instance data area does not exist until the object is created (to be precise - until rootclass constructor is executed) and the Application object is created as the last one. Then pointers to subobjects have to be stored in some temporary variables. This technique also requires that a parent object of the cached one is an instance of a custom (subclassed) class and the parent creates its subobjects inside the constructor, which is not always true.
- Use the MUIA_UserData attribute and the MUIM_FindUData() method to find objects dynamically. This is the best solution when objects are accessed rarely (for example once, just to set notifications). For frequently accessed objects (let's say several times a second) it may be combined with caching objects' pointers in an instance data of some subclassed object.

The last approach works as follows: every object to be searched has the MUIA_UserData attribute set to some predefined unique value. Then at any time the object may be found by this value using the MUIM_FindUData() method on a direct or indirect parent object, for example on the master Application object.

```
1 #define OBJ_SOME_BUTTON 36
2
```

[3] http://krashan.ppa.pl/morphzone_tutorials/scimark2_mui.lha

```
/* Somewhere in the initial tags for the button */
MUIA_UserData, OBJ_SOME_BUTTON,
/* ... */

/* Let's get the pointer to the button now */

Object *some_button;

some_button = (Object*)DoMethod(App, MUIM_FindUData, OBJ_SOME_BUTTON);
```

This operation is so common that it is usually encapsulated in a macro:

```
#define findobj(id, parent)\
        (Object*)DoMethod(parent, MUIM_FindUData, id)

some_button = findobj(OBJ_SOME_BUTTON, App);
```

The macro may of course be used directly in other functions, like in this example changing the button label:

```
set(findobj(OBJ_SOME_BUTTON, App), MUIA_Text_Contents, "Press Me");
```

Note that the findobj() macro is not defined in the system MUI headers, so it should be defined in the application code.

2.4.2 Text Class: Buttons, Textfields, Labels

2.4.2.1 Introduction

The Text class is the most commonly used one for creating gadgets. This is because it not only creates plain labels (also called ”static text” in other GUI engines), but also framed read-only text gadgets and textual buttons. In fact MUI has no special class for buttons, a button is just a Text object with a proper frame and background and user input activated. This versatility can have the disadvantage of allowing for creation of style guide nonconforming interfaces.

The Text class uses the MUI Text Rendering Engine for text output. It allows for multiline text, using styles (bold, italic), colors and inlined images. These features should be used sparingly to keep user interfaces consistent and comfortable to use. The rendering engine is controlled by inserting escape sequences in the text. Another engine feature is aligning the text inside the object rectangle (left, right, centered).

2.4.2.2 Common attributes

- MUIA_Background and MUIA_Frame are attributes inherited from the Area class. They specify a background and frame used for an object.
- MUIA_Text_Contents specifies the text. It may contain formatting engine escape sequences. The text is buffered internally, so the value of this attribute may point to a local variable or a dynamically allocated buffer.
- MUIA_Text_PreParse may be considered a fixed prefix, which is always added at the start of text before rendering. It is usually used for applying constant styles and formatting, for example setting it to ”
  33c” will always center the text. This attribute simplifies text handling when the text displayed is not constant (for example is loaded from a locale catalog).
- MUIA_Font is another attribute inherited from the Area class and specifies a font to be used for text rendering. In most cases it is one of the fonts predefined by the user in MUI settings.

For more attributes and detailed descriptions refer to the Text class autodoc in the SDK.

2.4.2.3 Labels

Labels are the simplest form of Text instances. They have no frame and inherit their background from the parent object (neither MUIA_Frame nor MUIA_Background is specified). They use the standard MUI font in most cases, so MUIA_Font does not need to be specified either. An important (and often forgotten) detail is a proper text baseline align when a label is used with some framed gadget also containing text (a String gadget for example). The default MUI behaviour for vertical text positioning is to center it. If the framed gadget uses uneven vertical padding, baselines of the label and the gadget may be unaligned. A special attribute MUIA_FramePhantomHoriz solves this problem. It is specified for a label with a TRUE value. The label also has MUIA_Frame specified with the same frame type as the gadget. Then the label gets an invisible frame of this type (frame horizontal parts to be exact, hence the attribute name) and the text is laid out accordingly. As a result the label and the gadget text are always aligned vertically, assuming they use the same font.

Figure 2.9: Screenshot of label alignment

The magnified screenshot above illustrates the label align problem. The string gadget uses uneven padding (top padding is 2 pixels, bottom padding is 0 pixels), which causes the text baseline to misalign by 1 pixel shown on the left. The label on the right has been defined with two additional tags:

```
1 MUIA_FramePhantomHoriz,   TRUE,
2 MUIA_Frame,               MUIV_Frame_String,
```

A label can have one character underlined with the MUIA_Text_HiChar attribute. It is used to create a visual hint for a hotkey of a labeled gadget. See Buttons section below for details.

2.4.2.4 Textfields

A textfield is a read-only framed area showing some (usually changing at runtime) text. The difference between a label and a textfield is that the latter has a frame and a background specified:

```
1 MUIA_Frame,              MUIV_Frame_Text,
2 MUIA_Background,         MUII_TextBack,
```

2.4.2.5 Buttons

A text button is an instance of the Text class too. It has more attributes than a plain label however, because it handles user input. MUI has a predefined background and frame for buttons:

```
1 MUIA_Frame,              MUIV_Frame_Button,
2 MUIA_Background,         MUII_ButtonBack,
```

These attributes also specify a frame and background for the ”pressed” state. MUI also has separate font settings for buttons. Forgetting the MUIA_Font attribute for buttons is one of most common errors in MUI design.

```
1 MUIA_Font,               MUIV_Font_Button,
```

Many users (and programmers) just have the default font defined for buttons, so the bug is not visible. It is recommended to always test a GUI with some unusual font settings for buttons, so the problem is easily visible. Button text is usually centered, which may be done either by inserting the ”
33c” sequence at the start of the button label, or using MUIA_Text_PreParse.

```
1 MUIA_Text_PreParse,      "\33c",
```

After definition of the button appearance it is time to handle user input. The button behaviour is defined by the MUIA_InputMode attribute with three values:

- MUIV_InputMode_None - The default value, button does not react on anything.
- MUIV_InputMode_RelVerify - A simple pushbutton activated by a left mouse button click.
- MUIV_InputMode_Toggle - A two-state button, one click switches it on, another one switches it off.

Another common bug with MUI buttons is to omit keyboard handling. The mouse is not everything. The first, obligatory step is to enter the button into the window's TAB key cycle chain:

```
1 MUIA_CycleChain,         TRUE,
```

Any gadget entered into the chain may be selected by pressing the TAB key (for default MUI keyboard settings). The selected object has a special frame drawn around it. Then it may be activated by some key set in MUI preferences. For buttons the default "pressing" key is the return key. A rule of thumb for cycle chaining:

Every gadget accepting user input must be added to the TAB cycle chain.

Another keyboard handling feature provided by MUI is hotkeys. A hotkey just activates a button associated to it. Hotkeys have the following features:

- MUI provides a visual hint of a button hotkey by underlining the hotkey letter in the button label. It implies that the hotkey letter must exist in the label.
- There is visual feedback for using a hotkey, the button is pressed as if it has been clicked with the mouse.

Not every button in a GUI has to have a hotkey. The best practice is to assign hotkeys only for the most used buttons, especially if there are many buttons in a window. A hotkey is defined with two attributes:

- MUIA_Text_HiChar - this attribute specifies a letter to be underlined in the label. It is case insensitive.
- MUIA_ControlChar - this attribute specifies a hotkey. Of course it should be the same as the above one. It should be a lowercase letter, as uppercase forces a user to press SHIFT, making the hotkey less comfortable to use. here is also no visual hint for SHIFT requirement. Digits may be also used as hotkeys if the label contains them. Using punctuation and other characters should be avoided. An example of use:

```
1 MUIA_Text_Contents,      "Destroy All",
2 MUIA_Text_HiChar,        'a',
3 MUIA_Text_ControlChar,   'a'
```

Note that these attributes take a single char, not a string.

Chapter 3

Reggae: MorphOS multimedia framework

Author: Grzegorz Kraszewski
Source: http://library.morphzone.org/Reggae:_MorphOS_multimedia_framework

3.1 Introduction

Reggae is the name for modular MorphOS subsystem for handling media files (currently pictures and sounds, video and other contents will come in the future). Reggae is implemented as a large set of MorphOS shared libraries stored in MOSSYS:Classes/Multimedia/ directory. Third party Reggae classes may be copied to SYS:Classes/Multimedia/. Using Reggae an application developer can easily perform following tasks related to media processing:

- Recognizing media type and format.
- Streaming media via different transports.
- Demultiplexing compound media streams.
- Decoding media to plain, uncompressed format.
- Processing by applying filters.
- Presenting media to the user.
- Encoding and multiplexing.

Reggae is an object oriented framework. Every media processing task creates a pipe (or tree) of Reggae objects connected to each other. Media data travel along this structure in relatively small chunks. This pipelined processing allows for handling very big data, much bigger than available system memory.

3.2 Overview

This section contains general Reggae information, its design principles, usage patterns and rules.

3.2.1 Kinds of Reggae classes

3.2.1.1 Multimedia.class

This is the master class of Reggae. It estabilishes basic methods and attributes. It also performs Reggae initialization when opened the first time after boot. All other classes are subclasses of multimedia.class. This class is also responsible for data formats detection and automatic building of decoding tree. It also provides secondary functionality like event logging, metadata support, AltiVec friendly memory allocations and more. Because of all these features multimedia.class is the one and only Reggae class having shared library API except of BOOPSI (object) one.

3.2.1.2 Streams

Streams form input data abstraction layer of Reggae. A stream is always the first object in any Reggae processing structure. It has one output port. All streams have common set of attributes and methods for data fetching and control. Currently available streams are:

- *memory.stream* for accessing data in memory: buffered, generated or embedded in application code.
- *file.stream* for reading files via *dos.library*.
- *http.stream* is easy to use yet powerful implementation of HTTP/1.1 protocol client. It can fetch any network resource available via HTTP GET request.

While *memory.stream* and *file.stream* are relatively simple wrappers, *http.stream* is a component useful stadalone as well as a Reggae data source. Any application can just use it for easy data downloading via HTTP, without even touching sockets API and dealing with HTTP internals.

3.2.1.3 Demuxers

Every media format recognized by Reggae has its own demuxer. Demuxer class is responsible for format recognition, header decoding, metadata extraction and demultiplexing (hence the name) media streams to separate output ports. While there is no real demultiplexing in simple audio or image formats, splitting functionality between demuxer and decoder allows for code reusing, as multiple demuxers may use the same decoder class (for example all demuxers for audio formats using uncompressed PCM, use *audiopcm.decoder*).
There are also a few ”general” demuxers not associated with particular data format. They either handle some common metadata (like *id3tag.demuxer*), or common compression schemes at datastream level (*xpk.demuxer*). Such demuxers are usually the first stage of demuxer cascade.

3.2.1.4 Decoders

A decoder takes a single, demuxed media stream and converts it to one of 3.2.2 Reggae common formats. This conversion usually means decompression and decoding of stream. Some decoders are dedicated to one particular media format, some are more general and used with many demuxers.

3.2.1.5 Filters

A Reggae filter accepts data in one of 3.2.2 Reggae common formats, performs some transformation on data and deliver them in the same or other Reggae common format. Most filters have the same format on inputs and outputs, but it is not the rule of thumb. Imagine a filter generating visualisations for audio player, it will accept audio and generate video.

3.2.1.6 Encoders

(Dec.2011) TBD

3.2.1.7 Muxers

(Dec.2011) TBD

3.2.1.8 Outputs

Outputs form output abstraction layer of Reggae. They are not symmetrical to 3.2.1.2 streams however. Outputs can be divided into two groups:

- **user presentation outputs**, which direct data stream to some output device of a computer (*audio.output*, *picture.output*),
- **storage outputs**, which store data stream somewhere (*file.output*).

Constructing a chain of connected Reggae objects does not start data processing automatically. Reggae is a pull-driven system, so something must pull data at the end of chain. It may be the application, who actively call *MMM_Pull()* method on output port(s) at the end of chain. Then it just gets data in some memory buffer and can do whatever it wants with them. Alternatively the last chain object may be an instance of some Reggae output class. Then the whole processing is done by Reggae.
Every Reggae output object creates a subprocess which pulls data from the chain of objects and either stores them or presents to the user. It means that **all Reggae data processing is automatically offloaded from application to the subprocess**. Main application process is free to handle GUI, display processing progress and control the processing by performing methods on the output object. Subprocessing Reggae chain makes creating GUI-based applications easier, and allows for example background processing, just by setting subprocess priority below 0. It may be useful for storage outputs. User presentation outputs have to work in real time, so their default priorities are above 0.

3.2.1.9 Internal classes

There are some helper classes, which are used by Reggae internally, but may be interesting for advanced Reggae programmers. Here is a brief description of them:

- *processblock.class* is used for grouping sets of connected objects into groups seen as a single object from outside. For example *MediaNewObject()* call returns a ”single” object, which is in fact a complete tree consisting of at least four objects (a stream, a multiread buffer, a demuxer and a decoder).
- *multiread.buffer* implements a FIFO buffer with data peeking feature. It is used in format recognition process to present the same stream header to multiple recognition routines, even if the source stream is not seekable.

See autodocs of these classes for more details.

3.2.2 Reggae common formats

Reggae **common format** or basic media format is the final result of media decoding, also intermediate format for processing (with “filters”) and input format for encoding and multiplexing. These formats are just raw PCM samples for audio and raw ARGB pixels for video. Stream description, like number of audio channels or video dimensions, is provided via attributes.

3.2.2.1 Audio common formats

All audio common formats are streams of linear PCM samples. Multichannel streams are interleaved. Stereo streams are intereleaved in [left, right] order. Interleaving for more channels is not yet defined. There are three sample formats defined:

- **MMFC_AUDIO_INT16** - the most common format, samples are 16-bit signed integers. Best for realtime processing targetted at *audio.output*.
- **MMFC_AUDIO_INT32** - very high quality, but slow processing. 32-bit signed integers. May be useful for non-realtime processing. Note that some filters may not support it.
- **MMFC_AUDIO_FLOAT32** - some compromise between the two above. Samples are single precision IEEE 754 floats with range normalized to <-1.0, +1.0>. Using floats makes avoiding internal overflow easier, but rounding errors may be dangerous.

3.2.2.2 Video common formats

Video common format is just an rectangular array of pixels. Row scan order is top-to-bottom, line scan order is left-to-right. Lines are not padded in any case. There is only one pixelformat defined currently:

- **MMFC_VIDEO_ARGB32** - every pixel takes 4 bytes, 8 bits per component, 8 bits for not premultiplied alpha channel. Pixel components order is [A, R, G, B].

3.3 Tutorials

This section contains Reggae programming tutorials with example code.

3.3.1 General

3.3.1.1 Accessing Reggae in applications

Reggae is operated from application with two APIs. One of them is generic object oriented BOOPSI API with its *DoMethod()*, *GetAttr()*, *SetAttrs()* etc. The second API is just a shared MorphOS library one, provided by *multimedia.class* and is, of course, Reggae specific. To use Reggae one must open *multimedia.class* as every shared library. For BOOPSI API *intuition.library* must be opened, which almost all programs do anyway.

Opening and closing multimedia.class
The first step is to add needed includes:

```
1 #include <proto/multimedia.h>
2 #include <proto/exec.h>
3 #include <proto/intuition.h>
4 #include <clib/alib_protos.h>
```

The first file contains definition of multimedia.class library API.
It also includes $<$*classes/multimedia/multimedia.h* $>$, containing Reggae structures, constants, tags and macros. The rest of includes are not Reggae specific, in fact most projects include them anyway, as they define basic system services. There are also additional Reggae headers, their including depends on application and will be covered in further tutorials.
Now we are ready for Reggae initialization. All what has to be done is opening *multimedia.class* just like an ordinary MorphOS shared library. The only noticeable difference is that library name contains path part, as Reggae classes are not directly on library search path:

```
1 struct Library* MultimediaBase;
2
3 if (MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52))
4 {
5   /* Now Reggae is ready to use until the library is closed. */
6   CloseLibrary(MultimediaBase);
7 }
```

As with every library, the name of the base is important, as it is implicitly used in all calls to the library API as a hidden parameter. Note also the usual error handling, *multimedia.class* is disk-based and also performs some disk activity at startup, then checking the library base against NULL is recommended. 52 is the current *multimedia.class* version. Applications should request this version, as previous ones have some important features missing. Reggae cleanup is done with classic *CloseLibrary()* call.

Note: Some programmers use automatic library opening and closing by linking with libauto. multimedia.class can be added to the list of automatically opened libraries, but it is not added by default.

A complete example code shows Reggae initialization. The example opens *multimedia.class* and, if it succeeded, prints the version and revision of *multimedia.class* to the console:

```
/*====================================================================*/
/* Reggae example: opens Reggae and prints its version to the console. */
/* by Grzegorz Kraszewski 2009                                        */
/* compile with: gcc -noixemul -o tutorial_basic tutorial_basic.c     */
/*====================================================================*/
#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/multimedia.h>

struct Library *MultimediaBase;

int main(void) {
  if (MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)) {
    Printf("Reggae opened, version %ld.%ld.\n",
        MultimediaBase->lib_Version, %% MultimediaBase->lib_Revision);
    CloseLibrary(MultimediaBase);
  } else
    PutStr("Reggae failed to open.\n");

  return 0;
}
```

Opening and closing individual classes

Typical Reggae usage is just calling *MediaNewObject()* to get a data stream recognized, demultiplexed and decoded down to common formats automatically. In this case Reggae opens (and later closes) all needed classes automatically. Things change, when application builds processing chain by hand, or just uses single components. Then every used class must be opened and closed explicitly. It is done in the same way as with *multimedia.class*. Let's assume our application uses just *http.stream* object to download some data from the net. Before object creation class must be opened. It is typically done in application setup code:

```
struct Library *HttpStreamBase;

HttpStreamBase = OpenLibrary("multimedia/http.stream", 51);
```

As for every disk based library, checking the base against NULL is highly recommended. The class must be unloaded after use, for example in application cleanup:

```
CloseLibrary(HttpStreamBase);
```

Reggae classes are handled just like ordinary shared libraries. Note that "multimedia/" path is added to the class name, reason for this has been explained above. When a class is not used constantly in application, it may be a good idea to open it just before it is needed and close right after it is not needed anymore. For example if *http.stream* is used only for automatic application updates, there is no need to keep it opened after update checking/downloading. The last remark concerns names of library bases for classes. No Reggae

class **except of *multimedia.class*** has library-style API. Because of this library base name is not used for anything else than opening and closing the clas. That is why, the name is not important and can be anything. In these tutorials, bases names derived from classes names are used, just for improved code readability.

I don't like C/C++, what now?

Reggae can be used from any programming language. It has to fulfill some minimal set of features however. The first one is ability to call functions from standard MorphOS shared libraries, *intuition.library* and *exec.library* at least. Then ability to call functions from *multimedia.class*. Many programming languages used on MorphOS have tools for creating bindings (or stubs, or whatever it is called) to shared libraries, based on C headers and *.fd files. Creating bindings for *multimedia.class* should not be difficult then. Then the language must have *DoMethod()* call. For C/C++ this call is provided by *libabox*, a static library. Another important feature is ability to handle tag-based functions in some sane way. Taglist can be always built by hand, but passing tags as arguments to variable args functions is convenient.

3.3.1.2 Downloading web resources with http.stream - basics

Introduction

The *http.stream* class is one of Reggae stream classes, in other words data sources. In a chain of Reggae objects, a *http.stream* instance will be always the first object, having only one, output port. A *http.stream* object may be also used standalone, not connected to anything, just to retrieve any data resource reachable via HTTP protocol and particularly its GET and POST requests. From this point of view, *http.stream* is just embeddable HTTP/1.1 client with simple yet powerful API. A brief list of its features is given below:

- Socket API encapsulation. *http.stream* completely isolates application (and its programmer) from *bsdsocket.library* and TCP/IP stack. Only very basic knowledge of TCP/IP is needed to use *http.stream* with success.
- Unlike *bsdsocket.library* base instances, *http.stream* objects may be shared between processes (with the only exception that object must be disposed by proces which created it).
- The class has builtin parser of HTTP response headers.
- The class has also an easy to use HTTP request header builder, so custom fields may be added to the header.
- HTTP proxies are supported.
- The class supports chunked transfer and media streaming over HTTP.
- Optional user agent spoofing is possible.
- When connecting, HTTP redirections may be followed automatically.
- The class is able to handle streams longer than 4 GB.
- Easy protocol debugging via MediaLogger.

The class has some disadvantages however. Some of them may be removed in future versions:

- No support for persistent connections.
- No support for HTTP compression.
- Making connection, sending request and receiving response header is done in the constructor, so it is synchronous to the application. Any network delay in constructor blocks the application until timeout or other error is reached. It can be worked around by putting all the network operation on a subprocess.

Minimal example

When we skip any error handling, the whole process of downloading data via HTTP protocol reduces to three lines of code:

```
#define DATA_LENGTH 7465 /* just example value */

UBYTE buffer[DATA_LENGTH]; /* place for data */
Object *http;

http = NewObject(NULL, "http.stream", MMA_StreamName,
    "www.morphzone.org", TAG_END);
DoMethod(http, MMM_Pull, 0, buffer, DATA_LENGTH);
DisposeObject( http);
```

We assume here, *http.stream* class has been loaded previously with *OpenLibrary()* (see "Opening and closing individual classes"). The code will download first 7465 bytes of MorphZone main page (HTML code), assuming there will be no error. This assumption is rather risky, because a network operation can fail for numerous reasons. Then we will be calling method on the NULL pointer and disposing NULL later, which can even lead to application crash. For this reason *http.stream* offers a few ways for handling errors. They will be discussed later, for now a minimal error handling is checking *NewObject()* result against NULL. This is used in a simple example downloading the first 1000 bytes of a resource specified in the commandline and dumping them into the console. Note that using this program for binary resources (like images) may result in rather weird output... I recommend running this example along with MediaLogger, to learn *http.stream* protocol debugging features. Example:

```
/*------------------------------------------------------------------------*/
/* Basic example of http.stream Reggae class usage.                       */
/* It downloads the first 1000 bytes of web resource given in commandline */
/* and dumps them into the console.                                       */
/* Written by Grzegorz Kraszewski in 2010. Public domain.                 */
/*------------------------------------------------------------------------*/
#define __NOLIBBASE__
#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/intuition.h>
#include <proto/multimedia.h>

```

```
13 UBYTE Buffer[1000]; /* place for data */
14
15 STRPTR ArgTemplate = "URL/A"; /* for ReadArgs() */
16
17 extern struct Library *SysBase, *DOSBase;
18 struct Library *IntuitionBase, *MultimediaBase, *HttpStreamBase;
19
20 LONG Download(STRPTR url) {
21   Object * http;
22   LONG result = RETURN_OK;
23
24   if (http = NewObject(NULL, "http.stream", MMA_StreamName, (LONG) url,
25       TAG_END)) {
26     LONG data_len;
27
28     data_len = DoMethod(http, MMM_Pull, 0, (LONG) Buffer, 1000);
29
30     if (data_len) {
31       Write(Output(), Buffer, data_len);
32       PutStr("\n\n");
33     } else
34       PutStr("No data received.\n");
35
36     DisposeObject(http);
37   } else {
38     PutStr("Some network (?) error occured.\n");
39     result = RETURN_ERROR;
40   }
41   return result;
42 }
43
44 BOOL AppSetup(void) {
45   if (!(IntuitionBase = OpenLibrary("intuition.library", 50)))
46     return FALSE;
47   if (!(MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)))
48     return FALSE;
49   if (!(HttpStreamBase = OpenLibrary("multimedia/http.stream", 51)))
50     return FALSE;
51   return TRUE;
52 }
53
54 void AppCleanup(void) {
55   if (HttpStreamBase)
56     CloseLibrary(HttpStreamBase);
57   if (MultimediaBase)
58     CloseLibrary(MultimediaBase);
59   if (IntuitionBase)
60     CloseLibrary(IntuitionBase);
61 }
62
63 int main(void) {
64   int result = RETURN_OK;
65   struct RDArgs *args;
66   LONG params[1];
67
68   if (AppSetup()) {
69     if (args = ReadArgs(ArgTemplate, params, NULL)) {
70       STRPTR url = (STRPTR) params[0];
71
72       if (strncmp(url, "http://", 7) == 0)
73         url += 7;
```

```
74
75        result = Download(url);
76      } else
77        result = RETURN_ERROR;
78    } else
79      result = RETURN_FAIL;
80
81   AppCleanup();
82   return result;
83 }
```

Length of data

Usefulness of the above example is limited. It downloads only predefined amount of data (or less, if the resource turns out to be shorter). Usually we want to download all the data and this implies getting the length of it somehow. A few scenarios are possible:

The length of data is known before downloading

This is the easiest, but the most rare case. It can be handled exactly as in the example from the previous section – a statically sized buffer and single *MMM_Pull()* call.

The server sends a static file

Then it knows the size and passes it in the response header (*Content-Length* field). The *http.stream* object extracts it automatically. Then data length may be obtained by getting *MMA_StreamLength* attribute. It means that the length is known before data downloading, so a buffer may be allocated dynamically. The attribute is 64-bit, so it should be get as follows:

```
1 QUAD length;
2 length = MediaGetPort64(http, 0, MMA_StreamLength);
```

This example shows *MMA_StreamLength* usage. It creates the object, asks of the data length, allocates a buffer, downloads data to the buffer and finally stores the buffer in a file:

```
1  /*------------------------------------------------------------------------*/
2  /* An example of http.stream Reggae class usage.                          */
3  /* It downloads the whole HTTP resource, getting its length by            */
4  /* MMA_StreamLength attribute. It fails in case of 0 length (which means  */
5  /* chunked transfer usually). Downloaded resource is stored in a file     */
6  /* given as the second argument.                                          */
7  /*                                                                        */
8  /* Written by Grzegorz Kraszewski in 2010. Public domain.                 */
9  /*------------------------------------------------------------------------*/
10 #define __NOLIBBASE__
11 #include <proto/exec.h>
12 #include <proto/dos.h>
13 #include <proto/intuition.h>
14 #include <proto/multimedia.h>
15
16 STRPTR ArgTemplate = "URL/A,FILE/A"; /* for ReadArgs() */
17
```

```
18 extern struct Library *SysBase, *DOSBase;
19 struct Library *IntuitionBase, *MultimediaBase, *HttpStreamBase;
20
21 /* Indexes of arguments in the table filled by ReadArgs(). */
22
23 #define ARG_URL   0
24 #define ARG_FILE  1
25
26 LONG Download(STRPTR url, STRPTR file) {
27   Object *http;
28   LONG result = RETURN_OK;
29
30   if (http = NewObject(NULL, "http.stream", MMA_StreamName, (LONG) url,
31       TAG_END)) {
32     QUAD total_bytes;
33     LONG downloaded_bytes;
34     BPTR destination;
35
36     if ((total_bytes = MediaGetPort64(http, 0, MMA_StreamLength)) > 0) {
37       UBYTE *buffer;
38       /*-----------------------------------------------------------------*/
39       /* Note the way of passing 64-bit numbers to Printf(). They have  */
40       /* to be splitted into two 32-bit parts.                          */
41       /*-----------------------------------------------------------------*/
42
43       Printf("Downloading %Ld bytes...\n", (LONG)(total_bytes >> 32),
44           (LONG)(total_bytes & 0xFFFFFFFF));
45
46       /*-----------------------------------------------------------------*/
47       /* Now the buffer is being allocated dynamically. Because data    */
48       /* length is 64-bit, we have to be sure it is not higher than     */
49       /* 2^31-1 before passing it to AllocMem() and later MMM_Pull().   */
50       /*-----------------------------------------------------------------*/
51
52       if ((total_bytes < 2147483648LL)
53           && (buffer = AllocVec((LONG) total_bytes, MEMF_ANY))) {
54         downloaded_bytes = DoMethod(http, MMM_Pull, 0, buffer,
55             (LONG) total_bytes);
56         Printf("Finished. %ld bytes downloaded, saving...\n",
57             downloaded_bytes);
58
59         if (destination = Open(file, MODE_NEWFILE)) {
60           if (FWrite(destination, buffer, downloaded_bytes, 1) == 1) {
61             Printf("%ld bytes saved to \"%s\".\n", downloaded_bytes,
62                 file);
63           } else {
64             PrintFault(IoErr(), file);
65             result = RETURN_ERROR;
66           }
67           Close(destination);
68         } else {
69           PrintFault(IoErr(), file);
70           result = RETURN_ERROR;
71         }
72         FreeVec(buffer);
73       } else {
74         PutStr("No memory for data.\n");
75         result = RETURN_FAIL;
76       }
77     } else {
78 PutStr    ("HTTP data size is unknown.
```

```
79          The server uses chunked transfer probably.\n");
80          result = RETURN_ERROR;
81        }
82      DisposeObject(http);
83    } else {
84      PutStr("Some network (?) error occured.\n");
85      result = RETURN_ERROR;
86    }
87    return result;
88  }
89
90  BOOL AppSetup(void) {
91    if (!(IntuitionBase = OpenLibrary("intuition.library", 50)))
92      return FALSE;
93    if (!(MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)))
94      return FALSE;
95    if (!(HttpStreamBase = OpenLibrary("multimedia/http.stream", 51)))
96      return FALSE;
97    return TRUE;
98  }
99
100 void AppCleanup(void) {
101   if (HttpStreamBase)
102     CloseLibrary(HttpStreamBase);
103   if (MultimediaBase)
104     CloseLibrary(MultimediaBase);
105   if (IntuitionBase)
106     CloseLibrary(IntuitionBase);
107 }
108
109 int main(void) {
110   int result = RETURN_OK;
111   struct RDArgs *args;
112   LONG params[2];
113
114   if (AppSetup()) {
115     if (args = ReadArgs(ArgTemplate, params, NULL)) {
116       STRPTR url = (STRPTR) params[ARG_URL];
117       STRPTR file = (STRPTR) params[ARG_FILE];
118
119       if (strncmp(url, "http://", 7) == 0)
120         url += 7;
121
122       result = Download(url, file);
123     } else
124       result = RETURN_ERROR;
125   } else
126     result = RETURN_FAIL;
127
128   AppCleanup();
129   return result;
130 }
```

The server sends dynamically generated data

Data are usually generated by some server-side script, written in PHP or other language. In this case server does not know the length a *priori* so it switches to HTTP chunked transfer mode. The *http.stream* object handles it automatically, and reports 0 as *MMA_StreamLength*, which means that the length is unknown. The only way to process such data is downloading

it in blocks in a loop until the object reports *MMERR_END_OF_DATA* error code. The loop code may look like this:

```
LONG chunk, error = 0;

while (!error)
{
  chunk = DoMethod(http, MMM_Pull, 0, buffer, BUFFER_SIZE);

  /* Do something with 'chunk' bytes of data in 'buffer'. */

  if (chunk < BUFFER_SIZE)
  {
    if (MediaGetPort(http, 0, MMA_ErrorCode) == MMERR_END_OF_DATA))
    {
      break; /* downloading finished */
    }
    else
    {
      error = 1; /* downloading failed */
    }
  }
}
```

The same loop, just enhanced with progress and error reporting is used in the complete example being just a Reggae based, very simple, universal HTTP downloader application. What may be interesting, it deals properly with data longer than 4 GB, assuming the fiilesystem of destination file is 64-bit.

```
/*-------------------------------------------------------------------------*/
/* An example of http.stream Reggae class usage.                           */
/*                                                                         */
/* It demonstrates dynamic downloading using fixed size buffer and         */
/* downloading loop. This code handles HTTP chunked transfer it may be     */
/* also used for streamed media.                                           */
/*                                                                         */
/* This program will handle files bigger than 4 GB assuming the target     */
/* filesystem is 64-bit.                                                   */
/*                                                                         */
/* This code shows also how to deal with 64-bit numbers in dos.library     */
/* Printf() function and its derivatives like FPrintf().                   */
/*                                                                         */
/* Usage of the program: http_dynamic [URL] [FILE]                         */
/*                                                                         */
/* Written by Grzegorz Kraszewski in 2010. Public domain.                  */
/*-------------------------------------------------------------------------*/
#define __NOLIBBASE__
#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/intuition.h>
#include <proto/multimedia.h>

STRPTR ArgTemplate = "URL/A,FILE/A"; /* for ReadArgs() */

extern struct Library *SysBase, *DOSBase;
struct Library *IntuitionBase, *MultimediaBase, *HttpStreamBase;

```

```
#define BUFFER_SIZE  8192
UBYTE *Buffer;

/* Indexes of arguments in the table filled by ReadArgs(). */

#define ARG_URL   0
#define ARG_FILE  1

LONG DownloadLoop(Object *http, BPTR destination) {
  QUAD total, done = 0;
  LONG chunk;
  LONG error = 0;

  total = MediaGetPort64(http, 0, MMA_StreamLength);

  while (!error) {
    chunk = DoMethod(http, MMM_Pull, 0, (LONG) Buffer, BUFFER_SIZE);

    if (FWrite(destination, Buffer, chunk, 1) == 1) {
      done += chunk;

      if (chunk < BUFFER_SIZE) {
        LONG reggae_error = MediaGetPort(http, 0, MMA_ErrorCode);

        if (reggae_error == MMERR_END_OF_DATA) {
          Printf("Done %Ld bytes.", (LONG)(done >> 32),
              (LONG)(done & 0xFFFFFFFF));
          PutStr("                    \n");
          break;
        } else {
          Printf("Failed at byte %Ld.",
              (LONG)(done >> 32),
              (LONG)(done &
                  % 0xFFFFFFFF));
          PutStr("                    \n");
          error = RETURN_ERROR;
        }
      } else {
        Printf("Downloaded %Ld ", (LONG)(done >> 32),
            (LONG)(done & 0xFFFFFFFF));
        if (total)
          Printf("of %Ld ", (LONG)(total >> 32),
              (LONG)(total & 0xFFFFFFFF));
        Printf("bytes.                   \r");
      }
    } else {
      PrintFault(IoErr(), "Writing error");
      error = RETURN_ERROR;
    }
  }

  return error;
}

LONG Download(STRPTR url, STRPTR file) {
  Object *http;
  LONG result = RETURN_OK;
  BPTR destination;

  if (http = NewObject(NULL, "http.stream", MMA_StreamName, (LONG) url,
      TAG_END)) {
```

```
90      if (destination = Open(file, MODE_NEWFILE)) {
91        result = DownloadLoop(http, destination);
92        Close(destination);
93      } else {
94        PrintFault(IoErr(), file);
95        result = RETURN_ERROR;
96      }
97
98      DisposeObject(http);
99    } else {
100     PutStr("Some network (?) error occured.\n");
101     result = RETURN_ERROR;
102   }
103
104   return result;
105 }
106
107 BOOL AppSetup(void) {
108   if (!(IntuitionBase = OpenLibrary("intuition.library", 50)))
109     return FALSE;
110   if (!(MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)))
111     return FALSE;
112   if (!(HttpStreamBase = OpenLibrary("multimedia/http.stream", 51)))
113     return FALSE;
114
115  /*-------------------------------------------------------------------*/
116  /* Memory allocated with AllocTaskPooled() is freed automatically when  */
117  /* the process exits.                                                */
118  /*-------------------------------------------------------------------*/
119
120   if (!(Buffer = AllocTaskPooled(BUFFER_SIZE)))
121     return FALSE;
122   return TRUE;
123 }
124
125 void AppCleanup(void) {
126   if (HttpStreamBase)
127     CloseLibrary(HttpStreamBase);
128   if (MultimediaBase)
129     CloseLibrary(MultimediaBase);
130   if (IntuitionBase)
131     CloseLibrary(IntuitionBase);
132 }
133
134 int main(void) {
135   int result = RETURN_OK;
136   struct RDArgs *args;
137   LONG params[2];
138
139   if (AppSetup()) {
140     if (args = ReadArgs(ArgTemplate, params, NULL)) {
141       STRPTR url = (STRPTR) params[ARG_URL];
142       STRPTR file = (STRPTR) params[ARG_FILE];
143
144       if (strncmp(url, "http://", 7) == 0)
145         url += 7;
146
147       result = Download(url, file);
148     } else
149       result = RETURN_ERROR;
150   } else
```

```
151     result = RETURN_FAIL;
152
153   AppCleanup();
154   return result;
155 }
```

It is important that RFC 2616, the HTTP specification, does not specify, that static files **must** be served without chunked transfer. On the other hand the server is not forced to use chunked transfer for dynamically generated contents. Assumption that server will not use chunks for a file just because the file is static one, may fail. Then, the safe way is to use download loop always and treat *MMA_StreamLength* as a hint only.

3.3.1.3 Writing Reggae classes

Reggae functionality can be extended by adding classes to it. New classes may be either public or private.

Private Reggae classes

Private classes are simple. They are statically linked with an application and may be only used in this single application. A private Reggae class is just a BOOPSI class derived from multimedia.class and implementing the standard set of Reggae methods. The class is created with *MakeClass()* and later referenced by pointer.

Public Reggae classes

A public Reggae class is more complicated. It is a separate disk based component, which may be loaded by any application. Public classes use shared library framework, so they are just a special case of MorphOS shared libraries. The differencies are as follows:

- Public Reggae class has no shared library API, except of standard open, close and expunge vectors.
- The class creates a named BOOPSI class at first open with *MakeClass()* and adds it to the system list with *AddClass()*. Then the class can be used by multiple applications (the code is shared) and is referenced by name, similarly to an ordinary shared library.

The following example shows the difference in creating objects of public and private class:

```
1 Object *pub, *priv;
2
3 priv = NewObject(PrivClassPointer, NULL, /* tags */, TAG_END);
4 pub = NewObject(NULL, "public.class", /* tags */, TAG_END);
```

Of course a public class has to be opened with *OpenLibrary()* before use (see 3.3.1.1 opening and closing individual classes) and closed when no longer used.

3.3.2 Audio

3.3.2.1 Playing a sound from file

Playing a sound file from disk is one of most common media related tasks. Reggae can perform it with a few lines of code. Using Reggae for audio playback has several advantages:

- Wide range of supported audio formats. A codec is selected and loaded by Reggae automatically.
- Playback is asynchronous. Reggae offloads decoding and playback to a dedicated process. The main application may perform other tasks during playback. It gets informed when the playback ends.
- Reggae streams audio from disk, so it does not load the whole file to memory. Doublebuffering is fully automatic.
- Audio is played through selected unit of AHI. Multiple sounds (up to 32, depending on user settings of AHI) may be played simultaneously.

Playing audio directly from disk is best suited for long sounds without low latency requirements. A typical example is music player or playing background music in the game.
From Reggae point of view, the task of playing audio from disk can be divided in two major parts. The first one is to get raw audio samples out of encoded file. The second task is to feed audio data to the output.

Opening a sound file

This part of the job is highly automated. Reggae recognizes the file format and builds complete decoding pipeline for the file with a single function call. The result is returned to the application as one, opaque object (it may contain many objects inside, but it is irrelevant for application programmer).

```
1 Object* media;
2
3 media = MediaNewObject(
4   MMA_StreamType, (ULONG)"file.stream",
5   MMA_StreamName, (ULONG)"RAM:sound",
6   MMA_MediaType, MMT_SOUND,
7 TAG_END);
```

This single call creates a complete decoding infrastructure for a specified file. Data source is specified by two tags, *MMA_StreamName* and *MMA_StreamType*. The first one is the name of the source. In case of files it is just path to the file, which may be absolute (as in the example), relative to the current directory, or relative to program executable location (using *PROGDIR*: assign). *MMA_StreamType* is used to specify which Reggae stream class (or ”transport”) should be used. Of course *file.stream* is for disk based files (and other things recognized by DOS as filesystems).
The last tag is a kind of filter. If Reggae recognizes the file, but it is not sound, the file will be rejected, and the function will return NULL. Of course if file is not recognized at all, NULL

will be returned as well. Checking the result of *MediaNewObject()* against NULL is a very good idea.
In case of success *media* contains a pointer to Reggae object, having at least one output data port, port 0.

Creating output

The second step is to add audio output object to the Reggae processing pipeline. Then one can ”run” the pipeline, which results in playing the file. The output object belongs to *audio.output* class. Before an object can be created, the class must be loaded from disk. It is done by opening the class with *OpenLibrary()*.

```
1 struct Library* AudioOutputBase;
2
3 AudioOutputBase = OpenLibrary("multimedia/audio.output", 51);
```

It is worth noting that *audio.output* has no specific functions in its shared library API (it is true for all Reggae classes except of the main *multimedia.class*). Then, the name of variable holding the library base is completely irrelevant (as the name is never used implicitly), and may be anything, ”hja76_d62eg” for example. The name used in the example is a bit more readable however.
After class opening, an instance of the class may be created:

```
1 Object* play;
2
3 play = NewObject(NULL, "audio.output", TAG_END);
```

The instance is created with generic *NewObject()* call. There are no tags for attributes. The output object will read all sound properties from media object when they are connected together. I remind again that checking return value here may be a good idea. If objects are ready, let's connect them:

```
1 MediaConnectTagList(media, 0, play, 0, NULL);
```

Output port 0 of *media* object is connected with input port 0 of play object. Both the objects form a complete Reggae processing pipeline. Now we are ready to play sound. The whole playback control is done by talking to output object.

Making noise

Playback is controlled with three methods: *MMM_Play()*, *MMM_Stop()* and *MMM_Pause()* performed on the audio.output instance.

- ***MMM_Play()*** starts playback if object is stopped, is ignored when object is playing.
- ***MMM_Stop()*** stops playback and rewinds the audio stream to the start (if possible).
- ***MMM_Pause()*** (available since version 51.14 of audio.output) stops playback, but does not rewind audio stream. Following *MMM_Play()* will continue from paused position.

All the methods are performed immediately, so just:

```
1 DoMethod(play, MMM_Play);
```

starts the playback and:

```
1 DoMethod(play, MMM_Stop);
```

stops it at any time. All methods are asynchronous to the caller and return immediately. Even if *MMM_Play()* setup time is long (because of prebuffering for example), calling process is not stopped because setup is done by *audio.output* process.

Waiting for end of sound

Because *audio.output* plays the sound asynchronously, there must be a way to inform the main process about sound end. By ”sound end” I mean either actual audio stream end, or calling *MMM_Stop()*. Then the application programmer need not to write separate code for handling natural and forced playback stop.
The class offers two methods for signalling sound end event, namely audio process can send a **signal** or can reply a **message**. Application specifies method choosen and its parameters by performing one of the two methods described below on *audio.output* object. Methods are usually called before the playback is started, but may be also called when object is already playing. The later solution is tricky however, as the sound may be very short, so a method may be called **after** the sound end. In this case signalling requests will be never triggered.

MMM_SignalAtEnd() method should be used, when we want to receive a signal to be Wait()-ed. It has two parameters, pointer to process to be signalled and signal number (not mask!) to be sent. We usually want to be signalled ourselves, but it is not a requirement, so process A can start playback, but signal may be received by process B. A typical usage may look like this:

```
1 DoMethod(play, MMM_SignalAtEnd, FindTask(NULL), SIGBREAKB_CTRL_C);
2 DoMethod(play, MMM_Play);
3 Wait(SIGBREAKF_CTRL_C);
```

In this code we send a request to be signalled themselves with system CTRL-C signal. It can be of course allocated private signal. Note that *MMM_SignalAtEnd()* method expect **signal number** while following Wait() needs a signal mask.

```
1 /*-------------------------------------------------------------------------*/
2 /* Reggae example: playing audio file from disk, end of sound is signalled */
3 /* with a system signal. File name is read from the commandline.           */
4 /*                                                                         */
5 /* This example has only very basic error handling without user feedback.  */
6 /* This is done for code simplicity. Reggae problems may be traced with    */
7 /* MediaLogger tool.                                                       */
8 /* ------------------------------------------------------------------------*/
9 #define __NOLIBBASE__
```

```
10 #define USE_INLINE_STDARG
11
12 #include <proto/exec.h>
13 #include <proto/dos.h>
14 #include <proto/intuition.h>
15 #include <proto/multimedia.h>
16 #include <classes/multimedia/sound.h>
17
18 extern struct Library *SysBase, *DOSBase;
19 struct Library *IntuitionBase, *MultimediaBase, *AudioOutputBase;
20 CONST_STRPTR Template = "FILE/A";
21
22 BOOL init_resources( VOID) {
23   if (!(IntuitionBase = OpenLibrary("intuition.library", 50)))
24     return FALSE;
25   if (!(MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)))
26     return FALSE;
27   if (!(AudioOutputBase = OpenLibrary("multimedia/audio.output", 51)))
28     return FALSE;
29   return TRUE;
30 }
31
32 VOID free_resources( VOID) {
33   if (AudioOutputBase)
34     CloseLibrary(AudioOutputBase);
35   if (MultimediaBase)
36     CloseLibrary(MultimediaBase);
37   if (IntuitionBase)
38     CloseLibrary(IntuitionBase);
39 }
40
41 STRPTR read_args( VOID) {
42   struct RDArgs *args;
43   LONG params[1];
44   STRPTR filename = NULL;
45
46   if (args = ReadArgs(Template, params, NULL)) {
47     if (filename = AllocVecTaskPooled(strlen((STRPTR) params[0]) + 1)) {
48       strcpy(filename, (STRPTR) params[0]);
49     }
50
51     FreeArgs(args);
52   }
53
54   return filename;
55 }
56
57 VOID play_sound(STRPTR filename) {
58   Object *source, *player;
59
60   if (source = MediaNewObjectTags(MMA_StreamName, (LONG) filename,
61       MMA_StreamType, (LONG) "file.stream", MMA_MediaType, MMT_SOUND,
62       TAG_END)) {
63     if (player = NewObject(NULL, "audio.output", TAG_END)) {
64       if (MediaConnectTagList(source, 0, player, 0, NULL)) {
65 /*-------------------------------------------------------------*/
66 /* System CTRL-C signal is used. This way the same code will */
67 /* handle both sound stream end, and user break by pressing  */
68 /* CTRL+C in the console window (or sending this signal from */
69 /* TaskManager).                                             */
70 /*-------------------------------------------------------------*/
```

```
71
72          DoMethod(player, MMM_Sound_SignalAtEnd, (LONG) FindTask(NULL),
73              SIGBREAKB_CTRL_C);
74          DoMethod(player, MMM_Play);
75          Wait(SIGBREAKF_CTRL_C);
76        }
77
78        DisposeObject(player);
79      }
80
81      DisposeObject(source);
82    } else
83      Printf("Reggae was unable to decode \"%s\".\n", (LONG) filename);
84  }
85
86  int main(void) {
87    int result = RETURN_OK;
88    STRPTR filename;
89
90    if (init_resources()) {
91      if (filename = read_args()) {
92        play_sound(filename);
93        FreeVecTaskPooled(filename);
94      } else
95        result = RETURN_ERROR;
96    } else
97      result = RETURN_FAIL;
98
99    free_resources();
100
101   return result;
102 }
```

MMM_ReplyMsgAtEnd() signals the end of sound by sending a system message prepared by application to some message port set up by application as well. This method is useful especially when an application uses multiple sounds at once. Number of signals available to a process is very limited. Number of created messages is limited only by available memory. The method is also useful if application creates message port for other purposes. Then audio end messages can be directed to this port and distinguished by message contents. Typical usage looks as follows:

```
1  struct MsgPort *port; /* created elsewhere */
2  struct Message *msg; /* allocated elsewhere */
3
4  msg->mn_Node.ln_Type = NT_MESSAGE;
5  msg->mn_Length = sizeof(struct Message);
6  msg->mn_ReplyPort = port;
7
8  DoMethod(play, MMM_Sound_ReplyMsgAtEnd, msg);
9  DoMethod(play, MMM_Play);
10 WaitPort(port);
11 GetMsg(port);
```

The main difference between these two methods is that message signalling is ”one-shot”. After the message is sent to application’s port, it must be got from the port and reinitialized before it can be reused again. Signal method may be used repeatedly, which is comfortable when a short sound is triggered multiple times.

```
/*-------------------------------------------------------------------------*/
/* Reggae example: playing audio file from disk, end of sound is signalled */
/* with a message reply. File name is read from the commandline.           */
/*                                                                         */
/* This example has only very basic error handling without user feedback.  */
/* This is done for code simplicity. Reggae problems may be traced with    */
/* MediaLogger tool.                                                       */
/* ------------------------------------------------------------------------*/
#define __NOLIBBASE__
#define USE_INLINE_STDARG

#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/intuition.h>
#include <proto/multimedia.h>
#include <exec/memory.h>
#include <classes/multimedia/sound.h>

extern struct Library *SysBase, *DOSBase;
struct Library *IntuitionBase, *MultimediaBase, *AudioOutputBase;
CONST_STRPTR Template = "FILE/A";

BOOL init_resources( VOID) {
  if (!(IntuitionBase = OpenLibrary("intuition.library", 50)))
    return FALSE;
  if (!(MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)))
    return FALSE;
  if (!(AudioOutputBase = OpenLibrary("multimedia/audio.output", 51)))
    return FALSE;
  return TRUE;
}

VOID free_resources( VOID) {
  if (AudioOutputBase)
    CloseLibrary(AudioOutputBase);
  if (MultimediaBase)
    CloseLibrary(MultimediaBase);
  if (IntuitionBase)
    CloseLibrary(IntuitionBase);
}

STRPTR read_args( VOID) {
  struct RDArgs *args;
  LONG params[1];
  STRPTR filename = NULL;

  if (args = ReadArgs(Template, params, NULL)) {
    if (filename = AllocVecTaskPooled(strlen((STRPTR) params[0]) + 1)) {
      strcpy(filename, (STRPTR) params[0]);
    }
    FreeArgs(args);
  }
  return filename;
}

VOID play_sound(STRPTR filename) {
  Object *source, *player;
  struct MsgPort *port;
  struct Message *msg;

```

```
61   if (port = CreateMsgPort()) {
62     if (msg = AllocVec(sizeof(struct Message),
63         MEMF_PUBLIC | MEMF_CLEAR)) {
64       msg->mn_Node.ln_Type = NT_MESSAGE;
65       msg->mn_ReplyPort = port;
66       msg->mn_Length = sizeof(struct Message);
67
68       if (source = MediaNewObjectTags(MMA_StreamName, (LONG) filename,
69           MMA_StreamType, (LONG) "file.stream", MMA_MediaType,
70           MMT_SOUND, TAG_END)) {
71         if (player = NewObject(NULL, "audio.output", TAG_END)) {
72           if (MediaConnectTagList(source, 0, player, 0, NULL)) {
73             ULONG signals, sigmask = 1 << port->mp_SigBit;
74
75 /*-------------------------------------------------------------*/
76 /* I want to wait both for message reply (end of sound) and  */
77 /* CTRL-C signal (user break). That is why I calculate       */
78 /* signal mask and use Wait() instead of WaitPort().         */
79 /*-------------------------------------------------------------*/
80
81             DoMethod(player, MMM_Sound_ReplyMsgAtEnd, (LONG) msg);
82             DoMethod(player, MMM_Play);
83             signals = Wait(SIGBREAKF_CTRL_C | sigmask);
84
85             if (signals & SIGBREAKF_CTRL_C)
86               PutStr("User break!\n");
87
88             if (signals & sigmask) {
89               GetMsg(port); // empty port queue
90               PutStr("End of sound.\n");
91             }
92           }
93           DisposeObject(player);
94         }
95         DisposeObject(source);
96       } else
97         Printf("Reggae was unable to decode \"%s\".\n",
98             (LONG) filename);
99       FreeVec(msg);
100    }
101    DeleteMsgPort(port);
102  }
103 }
104
105 int main(void) {
106   int result = RETURN_OK;
107   STRPTR filename;
108
109   if (init_resources()) {
110     if (filename = read_args()) {
111       play_sound(filename);
112       FreeVecTaskPooled(filename);
113     } else
114       result = RETURN_ERROR;
115   } else
116     result = RETURN_FAIL;
117   free_resources();
118   return result;
119 }
```

3.3.2.2 Playing a sound from memory

While 3.3.2.1playing a sound from file is the most common way, there are applications where it has several disadvantages. When a sound is short, played many times and low latency is required, playing this sound from memory will be better option. Seeking in the sound, or restarting it will be substantially faster then, as it does not involve any disk activity.
On the other hand playing from memory should be used with care. Audio data are very space consuming usually. Five seconds audio effect stored as PCM in audio CD quality takes 861 kB of memory. A solution for this problem is to use compressed formats and let Reggae decompress it on the fly.
There are two ways of placing audio file in memory. Firstly, it can be loaded from disk. Secondly the audio file contents may be embedded into the executable file.

Buffering sound file from disk

In this method sounds are stored as files separated from application executable. Memory buffers for sounds are allocated dynamically. This approach allows for easy changing of sounds or – for example – delivering low quality versions for machines having less memory, or even running without sound, when memory is low. The code and error handling is a bit more complicated however. The process of sound buffering does not involve Reggae at all. File is opened and sized, then buffer is allocated, file is read into it and closes. These tasks may be performed by native *dos.library* calls (*Open()*, *Read()*, *Close()*), or C standard library calls (*fopen()*, *fread()*, *fclose()*). As the later ones are just wrappers on native calls, using native ones is recommended, unless code portability is important. The following code buffers a file in memory with minimal error checking:

```
1  BPTR handle;
2  LONG size;
3  APTR buffer;
4  struct FileInfoBlock *fib;
5
6  if (fib = AllocDosObject(DOS_FIB, NULL))
7  {
8    if (handle = Open("PROGDIR:sounds/anysound.wav", MODE_OLDFILE))
9    {
10     if (ExamineFH(handle, fib))
11     {
12       size = fib->fib_Size;
13
14       if (buffer = AllocVecTaskPooled(size))
15       {
16         if (Read(handle, buffer, size) == size)
17         {
18           /* use buffer as memory.stream data here */
19         }
20         FreeVecTaskPooled(buffer);
21       }
22     }
23     Close(handle);
24   }
25   FreeDosObject(DOS_FIB, fib);
26 }
```

Programmers new to MorphOS may notice some new things in the code above. While not related to Reggae, they are worth some explanation. The first one is *PROGDIR*: assign

(or link in Unix nomenclature). It just means the directory, where running executable file is located. It is then an easy way to refer to application data with relative paths. When user moves the application directory somewhere, paths using *PROGDIR*: are still valid.
The second thing is *AllocVecTaskPooled()* call. Every MorphOS process has an automatically assigned memory pool, which is disposed when process ends. Using this pool for allocations, an application need not track them, as all memory not freed explicitly with *FreeVecTaskPooled()* will be freed when memory pool is disposed.

Embedding sound in application executable

Embedding sound in the code has the advantage of simplicity. The application is more self-contained. There are no disk operations involved, so there is no need for error handling code. On the other hand it can make executable very big, and there is no chance for user to change sounds. Audio file of any fomat can be converted to C code (as a large table) with BinToC tool. Generated source is added to the project and compiled. Then address of the table (denoted in C just as the table name) and its length in bytes, are passed as parameters to *memory.stream* object (see code fragment below).

Memory stream as data source

Reggae uses the *memory.stream* class to access data located in system memory. Its usage is similar to *file.stream*, there are some differencies however. The first one is stream name. For *memory.stream* it is a string containing stream address as a hexadecimal number, like for example ”2749FA0C”. *MMA_StreamName* attribute is not used often however. One usually has the address just as number, not as text. Converting it to text just to make Reggae to convert it back to number makes not much sense. Then *MMA_StreamHandle* attribute comes with help. Its value is just the address of stream, passed as number. Another very important attribute is *MMA_StreamLength*. Memory based streams have no ”natural” end. When one is reading a file, DOS just reports EOF (end of file) condition, when the file ends. In memory one can read endlessly, until he hits the end of physical memory space. That is why *MMA_StreamLenght* is a **required** attribute for memory streams. Reggae will refuse to create a stream object, if the attribute is not specified. Note also that the attribute in general is **64-bit** one, and takes a **pointer to 64-bit number**. Passing just a 32-bit number as the value is a common mistake here. Code snippet below shows typical creation of memory stream object from a sound embedded in executable file:

```
1  /* The length is just example. */
2  CONST UBYTE SoundData [12837] = { /* audio data here */ };
3  QUAD length = 12837;
4  Object *sound;
5
6  sound = MediaNewObject(
7    MMA_StreamType, "memory.stream",
8    MMA_StreamHandle, SoundData,
9    MMA_StreamLength, &length,
10 TAG_END);
```

When sound is buffered from file, one has to check the file size first, then allocate a buffer and load the file into it using usual *dos.library* calls, or C standard library calls. The process is shown in the complete example source code. After the buffer is loaded, *memory.stream* object is created the same way as above.

```
/*-------------------------------------------------------------------------*/
/* Reggae example: loads a sound to memory buffer, then plays it at every  */
/* keypress. Demonstrates usage of memory.stream for low playback latency. */
/* To have real fun load some short sample like gun shot.                  */
/*                                                                         */
/* Every [ENTER] stops the sample (to rewind it to the start) and plays it */
/* again. Even if sample ends before keypress, stopping it is harmless.    */
/*                                                                         */
/* This example has only very basic error handling without user feedback.  */
/* This is done for code simplicity. Reggae problems may be traced with    */
/* MediaLogger tool.                                                       */
/* ------------------------------------------------------------------------*/
#define __NOLIBBASE__
#define USE_INLINE_STDARG

#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/intuition.h>
#include <proto/multimedia.h>
#include <classes/multimedia/sound.h>

extern struct Library *SysBase, *DOSBase;
struct Library *IntuitionBase, *MultimediaBase, *AudioOutputBase;
CONST_STRPTR Template = "FILE/A";

struct BufInfo {
  APTR buffer;
  LONG size;
};

BOOL init_resources( VOID) {
  if (!(IntuitionBase = OpenLibrary("intuition.library", 50)))
    return FALSE;
  if (!(MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)))
    return FALSE;
  if (!(AudioOutputBase = OpenLibrary("multimedia/audio.output", 51)))
    return FALSE;
  return TRUE;
}

VOID free_resources( VOID) {
  if (AudioOutputBase)
    CloseLibrary(AudioOutputBase);
  if (MultimediaBase)
    CloseLibrary(MultimediaBase);
  if (IntuitionBase)
    CloseLibrary(IntuitionBase);
}

STRPTR read_args( VOID) {
  struct RDArgs *args;
  LONG params[1];
  STRPTR filename = NULL;

  if (args = ReadArgs(Template, params, NULL)) {
    if (filename = AllocVecTaskPooled(strlen((STRPTR) params[0]) + 1)) {
      strcpy(filename, (STRPTR) params[0]);
    }

    FreeArgs(args);
```

```
61    }
62
63    return filename;
64  }
65
66  BOOL load_sound(STRPTR filename, struct BufInfo *binf) {
67    BPTR file;
68    LONG size;
69    struct FileInfoBlock *fib;
70    BOOL result = FALSE;
71
72    SetIoErr(0);
73    binf->buffer = NULL;
74
75    if (fib = AllocDosObject(DOS_FIB, NULL)) {
76      if (file = Open(filename, MODE_OLDFILE)) {
77        if (ExamineFH(file, fib)) {
78          binf->size = fib->fib_Size;
79
80          if (binf->buffer = AllocTaskPooled(binf->size)) {
81            if (Read(file, binf->buffer, binf->size) == binf->size)
82              result = TRUE;
83            else {
84              FreeTaskPooled(binf->buffer, binf->size);
85              result = FALSE;
86              PrintFault(IoErr(), "Loading failed");
87            }
88          }
89        }
90
91        Close(file);
92      } else
93        PrintFault(IoErr(), "Can't open file");
94
95      FreeDosObject(DOS_FIB, fib);
96    }
97
98    return result;
99  }
100
101 VOID main_loop(struct BufInfo *binf) {
102   Object *source, *player;
103   QUAD stream_length = (QUAD) binf->size;
104
105   if (source = MediaNewObjectTags(MMA_StreamHandle, (LONG) binf->buffer,
106       MMA_StreamType, (LONG) "memory.stream", MMA_StreamLength,
107       (LONG) & stream_length, MMA_MediaType, MMT_SOUND, TAG_END)) {
108     if (player = NewObject(NULL, "audio.output", TAG_END)) {
109       if (MediaConnectTagList(source, 0, player, 0, NULL)) {
110         LONG c;
111
112         for (;;) {
113           c = FGetC(Input());
114           if (c == 'q')
115             break;
116           else {
117             DoMethod(player, MMM_Stop);
118             DoMethod(player, MMM_Play);
119           }
120         }
121       }
```

```
122
123       DisposeObject(player);
124     }
125
126     DisposeObject(source);
127   } else
128     PutStr("Reggae was unable to decode buffer.\n");
129 }
130
131 int main(void) {
132   int result = RETURN_OK;
133   STRPTR filename;
134   struct BufInfo binf;
135
136   if (init_resources()) {
137     if (filename = read_args()) {
138       if (load_sound(filename, &binf)) {
139         PutStr("Press [ENTER] to (re)trigger sound."
140             "Press [Q] then [ENTER] to quit.\n");
141         main_loop(&binf);
142         FreeTaskPooled(binf.buffer, binf.size);
143       } else
144         result = RETURN_ERROR;
145
146       FreeVecTaskPooled(filename);
147     } else
148       result = RETURN_ERROR;
149   } else
150     result = RETURN_FAIL;
151
152   free_resources();
153
154   return result;
155 }
```

Playback and playback control

To play the sound, one connects created media object with an *audio.output* object, exactly the same as for playing from disk. There is also no difference in controlling the playback, or waiting for sound end. Thanks to stream abstraction Reggae ”does not care” what the stream is. Just seek and retrigger operations are much faster. It is important for short sound effects retriggered many times (think of a shot sound in a game). The example code linked above allows user for retriggering the sound pressing ENTER key. It can be done very fast without delays, assuming some simple audio compression is used (just press and hold ENTER, retrigger rate will be as fast as key repetition rate set in system preferences).
The example shows also ”launch and forget” strategy of playing sounds with Reggae. There is no check for sound end. *MMM_Stop()* just stops and does seek to the start. Then *MMM_Play()* starts playback. It does not matter if retrigger happens while previous sound is still playing or not. There is also no sound end check when user stops the program. Disposing an active (playing) *audio.output* object is perfectly safe.

3.3.2.3 Playing a continuous, synthesized wave

This tutorial shows how raw sound data may be played with Reggae. The example code synthesizes a 1 kHz sine wave and plays it continuously. The wave is synthesized into a

table in memory. Then *memory.stream* is used to access it. The next object, an instance of *rawaudio.filter*, attaches audio parameters to the raw data. Finally *audio.output* plays the wave in a endless loop, using its looping feature.

```
/*-------------------------------------------------------------------------*/
/* Reggae example: playing a synthesized, continuous 1 kHz sine wave.      */
/* Sampling frequency is 44100 Hz. This example shows usage of             */
/* memory.stream and rawaudio.filter to use raw, synthesized audio data    */
/* in Reggae.                                                              */
/*                                                                         */
/* This example has only very basic error handling without user feedback.  */
/* This is done for code simplicity. Reggae problems may be traced with    */
/* MediaLogger tool.                                                       */
/* ------------------------------------------------------------------------*/
#define __NOLIBBASE__
#define USE_INLINE_STDARG

#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/intuition.h>
#include <proto/multimedia.h>
#include <exec/memory.h>
#include <classes/multimedia/sound.h>
#include <math.h>

extern struct Library *SysBase, *DOSBase;

struct Library *IntuitionBase, *MultimediaBase, *AudioOutputBase,
    *MemoryStreamBase, *RawAudioFilterBase;

struct ReggaePipeline {
  Object *stream;
  Object *rawaudio;
  Object *player;
};

/*-------------------------------------------------------------------------*/
/* In theory a single looped period of sine is enough to play a continuous */
/* wave. Unfortunately one period of 1 kHz sine sampled at 44100 Hz      */
/* occupies  44.1 sampling periods, which is not integer. Using 10 periods */
/* (441 samples) would be enough, but I've used 100 periods, to avoid tight*/
/* looping, which is CPU consuming. Sample format in the table is 16-bit.  */
/*-------------------------------------------------------------------------*/

WORD SineWave[4410];

VOID wave_generate( VOID) {
  LONG i;

  for (i = 0; i < 4410; i++)
    SineWave[i] = sin(100.0 * M_PI * (double) i / 2205.0) * 32767.0;
}

BOOL init_resources( VOID) {
  if (!(IntuitionBase = OpenLibrary("intuition.library", 50)))
    return FALSE;
  if (!(MultimediaBase = OpenLibrary("multimedia/multimedia.class", 52)))
    return FALSE;
  if (!(AudioOutputBase = OpenLibrary("multimedia/audio.output", 51)))
    return FALSE;
```

```
57   if (!(MemoryStreamBase = OpenLibrary("multimedia/memory.stream", 51)))
58     return FALSE;
59   if (!(RawAudioFilterBase = OpenLibrary("multimedia/rawaudio.filter",
60       51)))
61     return FALSE;
62   return TRUE;
63 }
64
65 VOID free_resources( VOID) {
66   if (RawAudioFilterBase)
67     CloseLibrary(RawAudioFilterBase);
68   if (MemoryStreamBase)
69     CloseLibrary(MemoryStreamBase);
70   if (AudioOutputBase)
71     CloseLibrary(AudioOutputBase);
72   if (MultimediaBase)
73     CloseLibrary(MultimediaBase);
74   if (IntuitionBase)
75     CloseLibrary(IntuitionBase);
76 }
77
78 BOOL build_pipeline(struct ReggaePipeline *rpi) {
79   QUAD stream_length = 8820;
80   // size of the sine table in *bytes*
81
82   rpi->stream = NewObject(NULL, "memory.stream", MMA_StreamHandle,
83       (LONG) SineWave, MMA_StreamLength, (LONG) & stream_length,
84       // pointer to 64-bit value!
85       TAG_END);
86
87   rpi->rawaudio = NewObject(NULL, "rawaudio.filter", MMA_Sound_Channels,
88       1, MMA_Sound_SampleRate, 44100, MMA_Sound_Volume, 65536,
89       // full volume, this is the default, but let's set it anyway
90       TAG_END);
91
92   rpi->player = NewObject(NULL, "audio.output", TAG_END);
93
94   if (rpi->stream && rpi->rawaudio && rpi->player) {
95     MediaSetPort(rpi->rawaudio, 1, MMA_Port_Format, MMFC_AUDIO_INT16);
96     // default, but let's set it for clarity
97     MediaConnectTagList(rpi->stream, 0, rpi->rawaudio, 0, NULL);
98     MediaConnectTagList(rpi->rawaudio, 1, rpi->player, 0, NULL);
99     return TRUE;
100  }
101
102  return FALSE;
103 }
104
105 VOID play_sound(struct ReggaePipeline *rpi) {
106 /*-------------------------------------------------------------------------*/
107 /* The sound will be looped, so it never ends. The only way to break it   */
108 /* will be to press CTRL+C in the console or to send CTRL-C signal other  */
109 /* way. I'm only waiting for this signal then.                             */
110 /*-------------------------------------------------------------------------*/
111
112  // turn looping on
113  MediaSetPort(rpi->player, 0, MMA_Sound_LoopedPlay, TRUE);
114  DoMethod(rpi->player, MMM_Play);
115  Wait(SIGBREAKF_CTRL_C);
116 }
117
```

```
118 VOID destroy_pipeline(struct ReggaePipeline *rpi) {
119   DisposeObject(rpi->player);
120   DisposeObject(rpi->rawaudio);
121   DisposeObject(rpi->stream);
122 }
123
124 int main(void) {
125   int result = RETURN_OK;
126
127   if (init_resources()) {
128     struct ReggaePipeline rpi;
129
130     wave_generate();
131     if (build_pipeline(&rpi))
132       play_sound(&rpi);
133     else
134       result = RETURN_FAIL;
135
136     destroy_pipeline(&rpi);
137   } else
138     result = RETURN_FAIL;
139
140   free_resources();
141
142   return result;
143 }
```

Raw sine wave synthesis

In theory a single period of sine wave is enough to play it continuously. There are two reasons for using more periods however. The first is number of samples in one period. We want to play a 1 kHz wave sampled at 44.1 kHz. Then one period would contain 44.1 samples, which is not integer obviously. The second reason is processing overhead of tight loop. Every loop turn takes additional processing time, so it is better, when the loop is longer.
Taking both these reasons into account, 100 periods of the sine are generated into the table containing 4410 samples, which makes 0.1 second of sound. The generation is then straightforward and uses *sin()* function from the standard C math library. As the argument of *sin()* is in radians (so one period is 2π), it goes from 0 to 200π in 4410 steps (without the last value). The amplitude of the sine wave (which is normally 1.0) is scaled to 16-bit signed range by multiplying by 32767.

Using memory.stream for synthesis buffer

Basic usage of memory.stream has been discussed in 3.3.2.2“Playing a Sound From Memory” tutorial. *MMA_StreamHandle* attribute is used to pass the memory table address, *MMA_StreamLength* one takes the table size **in bytes** (note, it is not the same as the table size in the declaration, as every element of the table ocuppies two bytes). Note also, *MMA_StreamLength* is 64-bit attribute, and as such is passed **as a pointer** to a QUAD variable containing the value.

Applying audio parameters with rawaudio.filter

While the sine wave generation is done, the *memory.stream* output cannot be connected directly to *audio.output* object. This is because stream classes deliver just plain stream of bytes with no meaning assigned. In the previous tutorial Reggae was able to play this

stream, because it has been self-descripting (contained a header, for example AIFF or WAVE one). Raw data are not self-descripting, so we have to describe it to Reggae ourselves. A ”conversion” of stream into audio signal is done with *rawaudio.filter* object. Usage of quotes is intentional. In fact this class does not convert the data, it just attaches audio format and attributes to the data stream. Then the stream is recognized by Reggae as audio and may be further processed, which of course includes playing it with *audio.output*.
Applying audio information consists of two parts. Audio attributes (number of channels, sample rate and volume) are set during *rawaudio.filter* object creation. The samples format is set by setting it on the output port of created object:

```
1 Object *rawaudio;
2
3 rawaudio = NewObject(NULL, "rawaudio.filter",
4   MMA_Sound_Channels, 1,
5   MMA_Sound_SampleRate, 44100,
6   MMA_Sound_Volume, 65536,
7 TAG_END);
8
9 MediaSetPort(rawaudio, 1, MMA_Port_Format, MMFC_AUDIO_INT16);
```

Most of the code above is redundant, as it happens that values set match default values for *rawaudio.filter*. Namely default number of channels is 1, default volume is 65536 ($10000 = full volume) and default sample format is 16-bit integer. Default sampling rate differs however, as it is 8000 Hz. Anyway all attributes have been set in the example code just for completness.
One can use any of the three Reggae audio formats for raw data. Except of 16-bit integers, Reggae handles 32-bit signed integers (MMFC_AUDIO_INT32) and 32-bit signle precision floats
(MMFC_AUDIO_FLOAT32). Using these for playback makes not much sense however, as data will be converted to 16 bits anyway.

Rawaudio.filter allows also for using all formats supported by *audiopcm.decoder* and *laws.decoder*. It includes PCM 8/16/24/32 bits in both endians (8 bit data may be either signed or unsigned), 32-bit floats in both endians and 8-bit nonlinearly quantized data according to A-law or π-law. In this case, proper decoder object must be created and connected to the filter output. This feature may be useful for loading raw data from files created by external applications. For internally generated data using of Reggae standard formats is recommended (it avoids additional conversions).

Looping the sound to make it continuous

This is the easy part. Looping feature is built into *audio.output* class. This feature is controlled with *MMA_Sound_LoopedPlay* attribute. When it is set to TRUE, *audio.output*, after encounering sound end, seeks to the start of stream and continues playback. Thanks to doublebuffering, there is no gap in playback, assuming seek is done fast enough (which is true for memory streams and file streams on local storage media). Then, in case of our sine wave, looping is seamless. As mentioned above, the loop should not be too short. For memory streams 0.1 second is enough, for disk based streams 0.5 second would be safe.
The *MMA_Sound_LoopedPlay* attribute may be either passed to *NewObject()* or set later with *SetAttrs()*, or *MediaSetPort()* on *audio.output* instance input port. The example code uses the second approach.

Chapter 4

Additional

4.1 In-depth: The New MorphOS Memory System

Author: Harry Sintonen
Source: http://library.morphzone.org/Reggae:_MorphOS_multimedia_framework
Originally published at: http://morphos-team.net/tlsf.html

4.1.1 Foreword

Many parts of AmigaOS were designed 25 years ago. At the time system resources were scarce and processors were relatively slow.
For Exec a simple and in many cases fast memory management algorithm was chosen: First Fit algorithm. It gives a very fast allocation speed in non-fragmented scenarios. Memory deallocation isn't very efficient however. The major problem is memory fragmentation: First Fit routine will get exponentially slow when fragmentation grows. Over time the fragmentation can get so severe that the system is visibly laggy. Finally, First Fit routine itself causes yet more fragmentation by not returning the best but first fit.
MorphOS inherited the memory management algorithm from AmigaOS. It worked quite well, but it soon became clear that it had some serious problems. Especially longer sessions (several days) of using complex applications (for example IBrowse) would easily slow the system down a lot. So First Fit had to go and something better had to replace it.
I had three design goals in mind when developing the new memory system:

1. It must be compatible with the old one, as much as possible. Legacy applications should continue to work as before.
2. It should reduce the effects of memory fragmentation.
3. It should reduce memory fragmentation. An as large as possible contiguous memory block should be available.

4.1.2 Compatibility

The first requirement was clearly the most critical one as MorphOS traditionally wants to maintain as much compatibility as possible, while introducing new advanced things, too.

It also proved to be the most challenging requirement of all to fulfill. AmigaOS-compatible memory API is cluttered with obscurities such as AllocAbs(), AvailMem() and AddMemList(), freeing partial memory blocks, the MEMF_REVERSE flag and documented memory alignment characteristics.
The challenge was to introduce a faster memory system without breaking these functions. For example there is a common code sequence to obtain aligned memory blocks:

```
1  void *allocaligned(ULONG size, ULONG flags, ULONG align)
2  {
3    void *ptr = AllocMem(size + align - 1, flags & ~MEMF_CLEAR);
4    if (ptr)
5    {
6      ULONG alignptr = ((ULONG) ptr + align - 1) & (ULONG) -align;
7      Forbid();
8      FreeMem(ptr, size + align - 1);
9      ptr = AllocAbs(size, (APTR) alignptr);
10     Permit();
11     if (ptr && (flags & MEMF_CLEAR))
12       memset(ptr, 0, size);
13   }
14   return ptr;
15 }
```

For this to continue to work properly the new system must implement AllocAbs() properly and adhere to the original alignment specifications.

Another example is a routine which allocates the largest memory block:

```
1  void *alloclargest(ULONG flags)
2  {
3    ULONG largest;
4    void *ptr;
5    Forbid();
6    largest = AvailMem(flags | MEMF_LARGEST);
7    ptr = AllocMem(largest, flags);
8    Permit();
9    return ptr;
10 }
```

For this to work the largest memory block returned by AvailMem() must always be available for allocation as well.
AddMemList() is a somewhat uncommon feature of the memory system. This function can be used to add memory to the system runtime.
All these are implemented and are functioning correctly in the new memory system.
The new system is even compatible with some rather nasty tricks, such as assuming that AllocVec() keeps the allocation size at ptr-4. This is often used to implement realloc() like functionality for AllocVec()ed memory. Supporting this was no problem since it would not affect any other things adversely.
We clearly understand that code using such tricks is seriously broken, but we also understand that often there is no way to fix these applications, either. So for the end user it is better to support such ”hacks”, rather than to fail miserably. Developers can use tools such

as MungWall to detect these hacks, but the end users should not need to worry about such things!
In the end only two things had to go:
First, FreeMem() can no longer be used to free partial memory blocks. This feature was a side effect of FreeMem() internally calling Deallocate(), which in turn is guaranteed to support partial free. With the new memory system Deallocate() is no longer called. Luckily even the AmigaOS SDKs have since a long time discouraged partial block FreeMem, so this is no big problem I believe.
Second, MEMF_REVERSE no longer has any effect on the allocation. The flag is quietly ignored. Frankly I believe that adding such a flag to the memory system was a mistake to begin with. In MorphOS context it has no application anyway, so I don't believe the removal has any adverse effects.

4.1.3 Reducing Effects of Fragmentation

Fragmentation happens, there is no way around it. Over time the memory will fragment. The trick is to reduce the adverse effect it has on system performance.
The choice of algorithm was critical here. The Two Level Segregated Fit (http://rtportal.upv.es/rtmalloc/
algorithm provides a guaranteed constant O(1) allocation/deallocation cost regardless of the memory fragmentation.
This means that regardless of the memory fragmentation allocations and deallocation will run at the same speed. Even after say 5 years uptime.

4.1.4 Reducing Memory Fragmentation

he choice of algorithm is critical here, as well. For example the original First Fit algorithm generates a lot of fragmentation over time. To give fast allocation speeds it returns the first memory block fitting the requirements.
The new algorithm should instead return the best possible memory chunk, making sure that fragmentation only occurs when absolutely necessary.
The Two Level Segregated Fit algorithm excels here too, giving average fragmentation lower than 15%.

4.1.5 The Implementation

MorphOS 2.0 and later has a pluggable memory interface. Basically different allocator schemes can be selected at boot time. For now the possible options are First Fit (same as in AmigaOS and MorphOS 1.x) and Two Level Segregated Fit (default). The user can configure the desired memory system by adjusting the MorphOS boot command in Open-Firmware.

The TLSF algorithm is remarkably simple:
The algorithm uses a segregated fit mechanism to implement a good fit policy. The segregated fit mechanism uses an array of free lists, with each array holding free blocks within a size class. The classes are powers of two: 16, 32, 64 and so on. For performance and fragmentation reduction reasons the array has two levels. In the MorphOS implementation

each list is divided to 32 sub-divisions. This combined with other implementation specifics allows for a maximum allocation size of 2GB. Each array of lists has a bitmap to mark which lists are occupied with empty blocks and which are empty (all memory allocated). The free blocks themselves hold information about the block, similar to the original First Fit routine. For additional block coalesce performance during the memory deallocation a back pointer to previous free block is maintained as well.
In order to allocate a memory block the allocation size is split between the two indexes. These indexes are used to find a free block, which now is an O(1) operation. The found block is removed from the free list and marked as used. If the free block found is larger than the allocation size, the block is split. The size of the remaining block is again converted to two indexes. The new free block is inserted to the correct position indicated by the indexes.
To free memory the block size is obtained from the block header. The block is merged with the previous and next block, if possible. If merging is possible the other block is marked as used and removed. The size of the final resulting block is again converted to two indexes. The new free block is merged as free and inserted into the correct position indicated by the indexes.
The actual implementation adds a couple of things the original algorithm didn't have: maintaining the remaining free size (for AvailMem), possibility to allocate the largest possible free block and allocating a memory block at an absolute address.

In addition the characteristics of the TLSF allocator give a possibility to automatically detect wrong deallocation sizes passed to the memory freeing routines. This gives extra safety in form of rejecting obviously illegal calls. For application developers it gives extra debug so that they can identify and fix the bug early.

The TLSF memory system fills all the design goals: It is very compatible with even the oldest applications. It maintains good performance regardless of the memory fragmentation. It reduces memory fragmentation.

4.2 Writing Custom Startup Code

Author: Grzegorz Kraszewski
Source: http://library.morphzone.org/Writing_Custom_Startup_Code

4.2.1 Forword

One can find in every C programming handboook, that program execution starts from main() function. In fact we have such an impression, when we write a program. There is no evidence to disprove it. In fact however, this is not true. There are at least a few, and sometimes even a few teens of kilobytes of code between the start of program execution and the first line of main(). Most of this code is not really needed in most cases.
What is being done by this code? Let's say for the start, it is perfectly possible to have a program without any startup code at all. The system can jump into main() right away. Unfortunately such a program would run only from commandline window. It would crash, when started from Ambient. It is because Ambient sends a special message to a message port of a freshly created process. This message serves two purposes. Firstly, it contains Ambient launch parameters, namely program icon descriptor and optionally descriptors of

other icons, which have been shift-clicked, or dropped onto a panel. Secondly, a reply for the message is a signal for Ambient that the program finished its execution. Sending a reply is obligatory. This is the minimal set of things to be done by startup code. In practice it also should open required shared libraries. When one wants to use the C standard library, either with libnix or ixemul.library, the startup code also creates a ”standard environment” for the C standard library and POSIX functions (more on this). Because of this, startup code linked when using one of these libraries is quite complex, so also long.

4.2.2 Reasons for Writing Own Startup

The main advantage of an own startup is its shortness. Reducing program startup time is negligible. Very short startup is good for very short programs (for example shell commands), a few kB in size. In this case the standard startup may be easily longer than the program code itself. One can also use own startup just for satisfaction of making the program shorter by those few kilobytes. Custom startup code cannot be used, when the program uses ixemul.library. When the program is linked with libnix, the possibility of using own startup depends on the standard C library functions used. Most of them do not need any preparations and will work with any startup. Some more complex functions however require constructors to be executed in startup. If we use such functions, we will get linker errors of unresolved symbols. In such a case there is a simple choice – one either must replace these functions with something else, or just use the standard startup code. Own startup is then useful mostly when standard C library is not used at all (in favour of the native MorphOS API), or only simple functions from it are used.
If we are still determined to use own startup, it is the time to tell the compiler about it. Skipping standard startup is done with **-nostartfiles** argument. Then when we try to use our startup with libnix, we use **-nostartfiles** together with **-noixemul**. Programmers wanting to go the pure MorphOS API way (without the C library), should use **-nostdlib** option, which also implies **-nostartfiles**.

4.2.3 Let’s Write It

Before we start to write the code, note that except things executed before calling the main() function, some code must be also called after it returns. Then we also have ”cleanup code”. As this code is usually placed in the same function (the one that calls main()), both the parts are commonly called just startup code.
As mentioned before, program execution does not really start from the main() function. Where does it start then? When an ELF executable is loaded from disk, a section named ”.text” is found and operating system jumps to the start of its contents. When a program is written in C, it means start of the first function in code, as in C there is no way to write code outside of a function. It must be noted, that C compiler may reorder functions in a single object file. The GCC 2.95.3 compiler never does it, but aggressive optimizer of GCC 4 can change order of functions. Fortunately it is done only inside a single source file. To make sure that our startup function will be the first, it must be placed in a separate file. Then resulting object file must be linked as the first one, as linking order is always preserved. After this important note it is time for the code:

```
#include <proto/exec.h>
#include <proto/dos.h>
#include <dos/dos.h>
#include <workbench/startup.h>
```

The thing starts with including needed header files. We will need two basic system libraries: exec.library and dos.library. It explains why standard startup code, be it libnix or ixemul.library, opens these two libraries – it simply needs them for itself.

```
struct Library *SysBase;
struct Library *DOSBase;
```

As our code will use these two libaries, we need to define their bases.

```
extern ULONG Main(struct WBStartup *wbmessage);
```

This is a declaration of the main function of our program. As the object file containing startup code should contain only one function (the entry one), the rest of code has to be moved to other object files, for reasons explained above. That is why the main function has to be declared here, as we call it from the startup code. Alternatively its declaration may be placed in some header file and included here. The name Main() is arbitrary, it can be anything. I've just called it typically, capitalizing the first letter to avoid possible name confilct with the standard library. The argument of Main() is startup message (mentioned above) being sent by Ambient. If we do not plan to use it inside Main(), we can just declare it this way:

```
extern ULONG Main(void);
```

The next important thing is to define a mysterious global symbol __abox__.

```
ULONG __abox__ = 1;
```

While not needed in the code, this symbol is used by the system executable loader to differentiate between MorphOS ELF executables and other possible PowerPC ELF binaries. If there is no __abox__ defined, the executable will be recognized as PowerUP one and executed through ppc.library, with unpredictable results.

```
ULONG Start(void)
{
  struct Process *myproc = 0;
  struct Message *wbmessage = 0;
  BOOL have_shell = FALSE;
  ULONG return_code = RETURN_OK;
```

Start() is the code entry point. Again, name of this function is not important, it may be anything. It just has to be the first function in the linked executable. Some local variables are declared here, which will be needed later myproc will contain a pointer to our process, wbmessage will hold the Ambient startup message pointer. Variable have_shell will be used to detect if the program has been started from shell console or from Ambient. Finally return_code is just the return code of the program, it will be returned to the system. The return value is usually 0 when the program executed succesfully and RETURN_OK constant is just 0.

```
1 SysBase = *(struct Library**)4L;
```

Time for initialization of the SysBase, the base of exec.library. The library is always open. For historical and backward compatibility reasons the base pointer is always placed by the system at address $00000004, so we just take it from there. Having exec.library available, our code can check whether it has been started from shell or from Ambient:

```
1 myproc = (struct Process*)FindTask(0);
2 if (myproc->pr_CLI) have\_shell = TRUE;
```

This information is taken from the Process structure being just system process descriptor. The exec.library call FindTask() returns the calling task's own descriptor if 0 is passed as its argument. In case we are started from Ambient, receiving its message is compulsory:

```
1 if (!have_shell)
2   {
3     WaitPort(&myproc->pr_MsgPort);
4     wbmessage = GetMsg(&myproc->pr_MsgPort);
5   }
```

The startup message is being sent to process system port, so we receive it there. The message may be then passed to our Main() function, if we plan to make some use of it, like handling additional icon arguments.

```
1 if (DOSBase = OpenLibrary((STRPTR)"dos.library", 0))
2   {
```

The next step is opening dos.library, this library is opened in a pretty standard way. In fact this minimal startup code does not need it. There are two reasons to open it anyway. First, it is hard to imagine a program, which does not need dos.library – even "Hello world!" needs it. Secondly, all standard startup codes open it, so usually main code takes it for granted. Then my startup behaves conventionally and opens dos.library as well.

```
1 return_code = Main((struct WBStartup*)wbmessage);
```

Yes, after these few lines we are ready to call the main code. As stated above, passing the startup message from Ambient is optional. On the other hand, receiving the result and passing it back to the system later is obligatory.

```
1 CloseLibrary(DOSBase);
2   }
3   else return_code = RETURN_FAIL;
```

From this point the startup code becomes cleanup one. Note also that proper error handling must be done. dos.library is being closed, but if its opening failed before, the result of execution is changed to RETURN_FAIL. This is the hardest fail and means total inability to execute. In practice MorphOS can't boot if dos.library is not present in the system. But OpenLibrary() may fail for other reasons, for example simple lack of free memory. Then the startup code has to handle it in some reasonable way.

```
1 if (wbmessage)
2 {
3     Forbid();
4     ReplyMsg(wbmessage);
5 }
```

This snippet of code handles the Ambient startup message. Even if we make no use of it, it must be replied at exit. But what does Forbid() do here? This function halts system multitasking, specifically it prevents the system process scheduler to switch our process away. Usually it may be done for a very short period of time only and followed by a matching Permit(). At the first glance this code makes no sense then, a process stops process switching and ... exits. We have to know one important thing however: process switching is automatically reenabled when the process which called Forbid() ends. Then here is what happens:

- Our task calls Forbid(), so no other process can interrupt it.
- It replies the Ambient startup message. As multitasking is stopped, Ambient is unable to receive yet. The message just waits at its message port.
- Our task exits. Then the system restores multitasking.
- Ambient gets CPU time and receives the message. Note that at this point it is absolutely certain, that our task does not exist anymore. Possibility of a race condition is eliminated. Without Forbid() it could be possible that our process is removed from the system while it still executes.

Of course multitasking halt period is extremely short, because our cleanup code ends immediately after replying to Ambient:

```
1  return return_code;
2 }
```

4.2.4 $VER: - program identification string

This topic is not strictly related to startup code, but the version string is usually placed in it, so I've decided to write a few words about it. The version string is a short text in some defined format. This string contains the program name, version and revision number, compilation date and optionally copyrigth or author info. The version string is decoded by many applications including Ambient, the system command version, the Installer program and more. The text starts with **$VER:**, so it can be easily found in the program executable. As version tools search for the version string from the start of the executable file, it is best if version string is placed as close to the beginning of the file as possible. If the version string is declared as a simple string constant, it is unfortunately placed in one of the ELF data sections. These sections are placed after the code section by the linker. However we can force the version string to be placed in the code section:

```
1 __attribute__ ((section(".text"))) UBYTE VString[] =
2   "$VER: program 1.0 (21.6.2011) © 2011 morphos.pl\r\n";
```

Using a GCC specific extension __attribute__ we can push the string into the ELF section named .text, which is the code section. As the startup code object is linked as the first object, the version string will appear at the beginning of the executable, just after the code of the Start() function. Why after? It is simple, if we place it before the real code, the operating system will jump "into" the string, trying to execute it, and then of course it will crash.

4.2.5 A Complete Example

A complete "Hello world!" example[1] with custom startup code shows the described ideas at work. It only uses the MorphOS API, so is compiled with **-nostdlib** option. Executable size is 1592 bytes. For comparision, libnix startup and printf() gives 30 964 bytes, when one replaces printf() with MorphOS Printf() from dos.library it is still 13500 bytes.
As the project consists of two *.c files, a simple makefile is added to it. Example may be compiled just by entering make in a console.

[1] LHA archive is available here: http://krashan.ppa.pl/mph/files/helloworld.lha