MorphOS development Reference manual

Version 1.0

Table of Contents

1 Description of the system

1.1 Introducing MorphOS

MorphOS is the first successful attempt to run Commodore's A1200, A3000 and A4000's operating system on a PPC processor and improved systems free of legacy Commodore hardware. It's based around the Quark microkernel and runs the OS within an emulation box providing a fast 680x0 emulation. A fully PPC native exec replacement provides ways to write native PPC programs running into the emulation box in a mixed mode. For consistency, we'll call the Commodore system 'ABox' from now on.

1.2 Overview of the system

1.2.1 Binaries from the MorphOS distribution

1.2.1.1 The qstartup launcher tool

qstartup is the ABox binary which starts the whole system. See the included documentation on how to use it.

1.2.1.2 The HAL

hal.rom

The HAL (Hardware Abstraction Layer) is the part which makes MorphOS hardware independent. It does the following:

- determines the CPU types, the number of CPUs and clock speed
- scans the ABox Zorro I/O ports for Zorro-II and Zorro-III cards and configures them
- finds local hardware (CVisionPPC, SymbiosPPC, PCI bus)
- creates a resource map for the Quark microkernel
- starts the kernel resource

1.2.1.3 The Quark microkernel

kernel.rom

The Quark microkernel runs the following server threads:

- ExceptionServer
- MasterClanServer
- AddressServer
- ConfigServer: some kind of config filesystem where all the resources the HAL built up are located. It will be used to provide all kinds of internal system preferences
- CPUTimeServer
- SystemInit: this server thread basicaly starts up the system
  - creates the ABox task
  - searches for the *abox.rom*, *module.rom* and kickstart resource
  - maps the initial stack at 0x10000000 with a size of 0x30000 bytes
  - maps the abox.rom at 0x01000000
  - maps the module.rom at 0x10100000
  - grants the new ABox task the needed addressspace. ABox custom chips, memory, CVisionPPC, Symbios SCSI and so on
  - then it starts the new ABox emulation task's userthread0

1.2.1.4 The ABox emulation

abox.rom

The ABox emulation is a Quark task doing the following:

- parses an internal patch tables for kickstarts in the patch database
- creates a task custom exception thread which handles exceptions inside of the emulation process space. Does something like Enforcer at the moment and in the future will provide a little custom chip emulation which is needed for handling the interrupt controller register.
- creates 14 intthreads for all ABox custom chip interrupts. These threads handle the whole exec interrupt system. The stack is somewhere in the 0x10000000 area.
- starts the emulation in the current thread at 0xf80002

1.2.1.5 The modules

module.rom

This contains additional resident modules. They replace the ones in the kickstart file and this list will grow. They are of course fully PPC native.

- **execppc**: API compatible exec replacement allowing a mixed mode for 68k and PPC applications. Among other things it does the following:
  - retrieves the memory map and other configuration informations from the Quark/ConfigServer
  - creates the ExecBase structure and library tables
  - scans the kickstart, modules and extmodules (including OS 3.x RomUpdate files)
  - creates the "ExecStartup" task which calls all coldstart residents
- **expansionppc**: expansion replacement. Asks the Quark/ConfigServer about Zorro-II and Zorro-III resources instead of accessing the hardware directly. It adds them to the expansion.library's database.
- **graphicsppc**: replaces entries which aren't yet covered by CyberGraphX. Will soon be completely merged with CyberGraphX.
- **cybppc**: complete PPC native cybppc.device
- **blizzppc**: complete PPC native blizzppc.device
- **a4091ppc**: complete PPC native A4091 scsi.device
- **a4000tppc**: complete PPC native A4000T scsi.device
- **ppc.library**: complete PPC native ppc.library which emulates the old ppc.library functionality inside the ABox environment. Contains extended ABox elf support which means that elfs sections are now segments in a seglist. That way you can write .elf **ABox** libraries and devices which are loaded by the system
- **ramlib**: PPC native ramlib module with extended .elf ppc native libraries and devices. On ABox, the original rules are:

```
if $libs contains ":" then loadlib($libs)
else
{
    if loadlib("libs:"+$libs) failed
        loadlib($libs)
}
```

  Under MorphOS, the rules change a bit so that potential new .elf libraries (and devices) are loaded before a 68k one. They just need to have the postfix *.elf* added to their filename.

Of course, this postfix is not strictly needed but helps to separate 68k and PPC versions filesystem-wise on a system where booting using a 680x0 processor is possible[1].

This is the rule:

```
if $libs contains ":" then
{
    if (loadlib($libs+".elf")) failed
        loadlib($libs)
}
else
{
    if loadlib("libs:"+$libs+".elf") failed
        if loadlib($libs+".elf") failed
            if loadlib("libs:"+$libs) failed
                loadlib($libs)
}
```

- **mathieeeppc**: native IEEE libraries (single, double, ffp)..
- **utilityppc**: native utility.library
- **68040ppc**: native ppc resident, always fails so no diskresident 68040.library is loaded
- **68060ppc**: native ppc resident, always fails so no diskresident 68040.library is loaded
- **dosppc**: fully native dos.library, based on the AROS version
- **cdriveppc**: native CD-ROM filesystem
- **sfsppc**: native SFS Filesystem
- **ramhandlerppc**: native ramdisk
- **diskppc**: native disk.resource
- **bonusppc**: some internal stuff
- **logppc**: some exception logging server which writes such event to a console or file
- **timerppc**: native timer.device
- **keymapppc**: native keymap.library and resource
- **wosemu**: native API-compatible powerpc.library
- **pcippc**: PCI stuff
- **cyberguardppc**: enforcer-like debugging system
- **potgoppc**: native joystick handling
- **consoleppc**: native console.device
- **battclockppc**: native battclock.resource
- **battmemppc**: native battmem.resource
- **cardresppc**: native card.resource
- **ciappc**: native cia.resource
- **miscppc**: native misc.resource
- **filesysresppc**: native filesys.resource
- **inputppc**: native input.device
- **keyboardppc**: native keyboard.device
- **gameportppc**: native gameport.device

And the list is growing..

1.2.2 The 680x0 emulation and native PPC integration

[1] on a CyberStormPPC and BlizzardPPC

1.2.2.1 Different CPUs

Currently MorphOS has 5 emulation systems. One which only works with integers, 3 different kind of FPU emulations and one with dynamic translation (JIT). The working emulation might differ depending on the MorphOS version. All the emulations use some special 0xff?? opcodes for special emulation functions where a normal 680x0 would run into an exception.

1.2.2.2 680x0 EC

This emulation only handles integers thus applications using the FPU won't work. It emulates a 68060 EC (without FPU) with the following differences:

- address translation registers (BAT) are ignored
- 64-bit *div/mul* instructions and *cmp2/chk2* instructions are emulated[2].

1.2.2.3 FPU emulations

The FPU emulations aren't exact nor complete. Their main goal is to get the best possible speed an have them work with SAS/C and GCC FPU programs. They implement all the 6888x opcodes though. The limitations are the following:

- no 96-bit FPU datatype calculations. Those are converted to 64-bit PPC FPU format during load and store operations. This means those datatypes will have less precision but as SAS/C and GCC don't create 96-bit FPU datatypes this is mostly harmless
- no FPU exceptions, they aren't used
- no FPU packed decimals, they aren't generated by any C compiler
- no NAN condition tests, they aren't generated by any C compiler

1.2.2.4 680x0 FPU level 1

This FPU emulation uses the *emulhandle->FPU[]* fields and no directly mapped FPU registers. The format of the FPU[] content is always the same as the native CPU format which means the emulation writes a 64-bit double for a PPC and not a 96-bit extended format. This emulation system is rather slow but doesn't need much memory space.

1.2.2.5 680x0 FPU level 2

This FPU emulation works the same way as the level 1 one but uses directly mapped PPC registers. This needs more space than level 1 but is faster.

1.2.2.6 680x0 FPU level 3

This FPU emulation is similar as level 3 but without any FPU condition code test for each instruction as SAS/C and GCC don't use them. It also uses some kind of "Cyberpatcher" technology where the most used FPU instructions are replaced with emulation opcodes easier to decode. This level is roughly twice as fast as level 2.

1.2.2.7 680x0 JIT

The JIT emulation translates 68k opcodes "on the fly" to equivalent PPC opcodes. When the code is used again it doesn't need to be translated anymore yielding to a significantly faster execution speed. This emulation mode is the most promising one and is currently heavily tested.

[2] in a 68060 they are emulated by the 68060.library or by Cyberpatcher

1.2.2.8 Emulation interface

The 68k emulation uses the following global register layout for integer:

```
<emulinclude/emulregs.h>

register struct EmulHandle *MyEmulHandle __asm("r2");
register ULONG* REG_PC __asm("r13");
register UWORD  REG_SR __asm("r14");
register void (**OPCODETABLE)(void) __asm("r15");

register ULONG REG_D0 __asm("r16");
register ULONG REG_D1 __asm("r17");
register ULONG REG_D2 __asm("r18");
register ULONG REG_D3 __asm("r19");
register ULONG REG_D4 __asm("r20");
register ULONG REG_D5 __asm("r21");
register ULONG REG_D6 __asm("r22");
register ULONG REG_D7 __asm("r23");
register ULONG REG_A0 __asm("r24");
register ULONG REG_A1 __asm("r25");
register ULONG REG_A2 __asm("r26");
register ULONG REG_A3 __asm("r27");
register ULONG REG_A4 __asm("r28");
register ULONG REG_A5 __asm("r29");
register ULONG REG_A6 __asm("r30");
register ULONG REG_A7 __asm("r31");
```

Register *r2*, the EmulHandle pointer must **never** be killed by any application anytime. This conforms to the System V4 ABI which declares *r2* as *reserved for system*. Changing this pointer would mean that following PPC emulation calls wouldn't be possible and that the whole emulation would crash on the next interrupt. This is the same as using an invalid *A7* register in the ABox environment.

The meaning of *REG_PC* and *REG_SR* are obvious.

OPCODETABLE is the pointer to the internal 16-bit instruction function table but this is a private register and shouldn't be accessed.

The meaning of *REG_D0* to *REG_A7* should be obvious too.

Let's take a look at the public entries of the **current** EmulHandle structure *r2* **always** points at.

```
<emulinclude/emulinterface.h>

struct EmulHandle {
    ULONG  Dn[8];
    ULONG  An[8];
    ULONG  *PC;                      /* current PC */
    ULONG  SR;                       /* StatusRegister */
    struct SuperHandle *SuperHandle; /* pointer to SuperHandle */
    ULONG  Type;                     /* EmulHandle Type */
    ULONG  Flags;                    /* flags */

    void   (*EmulFunc)(void);
    /* direct emulation jump..you have to setup the regframes */
```

```
    ULONG  (*EmulCallOS)(struct EmulCaos*);
    /* emulation jump for a 68k OSLib function */
    ULONG  (*EmulCall68k)(struct EmulCaos*);
    /* emulation jump for a 68k function */
    ULONG  (*EmulCallQuick68k)(void);
    /* emulation quick jump for a 68k function */
    ULONG  (*EmulCallDirectOS)(LONG lvo);
    /* direct emulation jump for a 68k OSLib function */
    ULONG  (*EmulCallDirect68k)(APTR Function);
    /* direct emulation jump for a 68k function */
    ULONG  Pad1;
    struct Float96 FPU[8];                    /* not used yet... */
    ULONG  FPCR;                              /* not used yet... */
    ULONG  FPSR;                              /* not used yet...*/
    ULONG  FPIAR;                             /* not used yet...*/
    ULONG  Pad3;
    /* private */
};
```

This is the global key to the emulation. The global registers layout defined above is like a cached representation of a real 68k context stored here when necessary.

Now the explanation of the structure:

Dn[]: data registers

An[]: address registers

PC: program counter

SR: status register

SuperHandle: points to the following global structure:

```
struct SuperHandle {
    ULONG  USP;   /* Userstack */
    ULONG  SSP;   /* Supervisor Stack */
    ULONG  VBR;   /* Exception Base Register */
    ULONG  SFC;   /* SFC Register ...not really used */
    ULONG  DFC;   /* DFC Register ...not really used */
    ULONG  CACR;  /* Cache Control Register ...not really used */
    ULONG  TC;
    ULONG  ITT0;
    ULONG  ITT1;
    ULONG  DTT0;
    ULONG  DTT1;
    ULONG  URP;
    ULONG  SRP;
    ULONG  BUSCR;
    ULONG  PCR;
    ULONG  Type;  /* SuperHandle Type..not used yet */
    /* private */
};
```

which handles all the relevant supervisor registers for a 68060EC/68060LC

Type: internal identifier for the types of the emulation. This is private.

Flags: defines in which state the current PPC context is at the moment:

```
#define EMULFLAGSF_PPC       0x1
/* Set when the emulation runs in full native code */

#define EMULFLAGSF_QUICK     0x2
/* Set when the emulation runs quick native code..
 * which is basicaly still the emul register layout
 */

#define EMULFLAGSF_INTERRUPT 0x4
/* Set when the emulation runs in interrupt mode */
```

The meaning of these are explained later when we come to emulation traps and emulhandle hooks.

EmulFunc: function pointer to the emulation start function which was entered. (This is more a private function pointer you shouldn't really care about)

EmulCallOS: function pointer for PPC code to run 68k library functions.

EmulCall68k: function pointer for PPC code to run 68k code directly. Works the same as above...

EmulCallQuick68k: function pointer for PPC code to run 68k code directly. With Quick mode you tell the emulation that you pass the registers directly. Be aware that YOU are responsible for loading/restoring the registers.

EmulCallDirectOS: function pointer for PPC code to run 68k library functions with the 68k registers given by the current EmulHandle.

EmulCallDirect68k: function pointer for PPC code to run 68k code with the 68k registers given by the current EmulHandle.

The FPU fields: obvious

1.2.2.9 Using the emulation in programs

There are new 68k instructions in the range of 0xff00 and 0xffff which are defined in <emul/interface.h>

Example:

```
#define TRAP_LIB 0xff00
```

When the emulation hits this trap it assumes the following memory layout.

```
struct EmulLibEntry {
    UWORD  Trap;
    UWORD  Pad;
    void   (*Func)(void);
};
```

Pad must be zeroed. For future usage :-)

Func is the PPC function

The emulation saves all **integer** registers in the current EmulHandle, sets the EMULFLAGSF_PPC flag in MyEmulHandle->Flags and then calls the PPC function in EmulLibEntry->Func.

The PPC function could look like this.

```
ULONG PPC_Func(void) {
    void  *Arg1 = (void*) REG_A0;
    ULONG Arg2 = REG_D0;
    /* do some stuff */
     return(1);
}
```

REG_#? macros are basicaly *MyEmulHandle->#?[]* accesses.

The PPC function's result is always in *r3* and is then moved to *REG_D0* by the emulation. The emulation then does an internal RTS as such LibEntry are assumed to be called as subroutines by JSR or BSR.

```
#define TRAP_LIBNR 0xff05
```

This is the same as *TRAP_LIB* but *REG_DO* is not changed. This is useful to replace functions without result.

```
#define TRAP_LIB_QUICK 0xff01
```

Using this trap prevents the emulation to save registers. You have to know about the register layout. This is useful for small PPC functions which are only really called by 68k code, not by PPC code. Beware that if the PPC function itself calls subfunctions, the code generated by gcc will begin to look ugly because of the lack of free registers. Killing *A7* and/or *SR* will lead to a crash, as will calling non quickmode code which might kill the register layout temporarily.

In quickmode, the EMULFLAGSF_QUICK flag is set in EmulHandle->Flags.

Using quickmode in C requires to tell the C compiler about the global register layout by using the EMUL_QUICKMODE define before inclusion of <*emul/emulregs.h*>

```
#define EMUL_QUICKMODE
#include <emul/emulregs.h>
```

But avoid using quickmode for functions which can be called by PPC code as it will be inefficient in that case.

Important note:
PPC code runs completely transparent in the same exec task context it was called. The PPC exec does a complete PPC context save/restore on each task switch. As a consequence, the emulation doesn't stop multitasking when it runs into PPC code. It also doesn't need to send messages to special PPC code tasks outside of its context like its needed in the old PowerUP dual cpu system (ppc.library).

Let's see how it works from the PPC side:

The key to the emulation from the PPC side is the EmulCaos structure which is a stripped down version of the PowerUP EmulCaos structure, without the unnecessary cachefields.

```
struct EmulCaos
{
    union
    {
        int Offset;
        APTR Function;
    } caos_Un;
    ULONG reg_d0;
    ULONG reg_d1;
    ULONG reg_d2;
    ULONG reg_d3;
    ULONG reg_d4;
    ULONG reg_d5;
    ULONG reg_d6;
    ULONG reg_d7;
    ULONG reg_a0;
    ULONG reg_a1;
    ULONG reg_a2;
    ULONG reg_a3;
    ULONG reg_a4;
    ULONG reg_a5;
    /*
     * here you have to put the library base pointer if you want
     * to call a Library.
     */
    ULONG reg_a6;
};
```

You simply fill out the structure above and then

```
Result = (*MyEmulHandle->EmulCallOS)(&MyCaos);
```

All ABox library functions are available as easy inline macros in <emulinclude/includegcc/ppcinline> and there's a script/makefile to create macros from 3rd party libraries in <emulinclude/makefile> and <emultools/fd2inline>

Some examples:

PPC Hooks:

```
static void DispatcherFunc(void);
struct EmulLibEntry GATEDispatcherFunc=
{
    TRAP_LIB, 0, (void (*)(void)) DispatcherFunc
};

ULONG DispatcherFunc(void)
{
    struct IClass *cl=(struct IClass*) REG_A0;
    Msg msg=(Msg) REG_A1;
    Object *obj=(Object*) REG_A2;
```

```
    /* put your code here */
    return (0);
}

DispatcherHook.h_Entry = (void*) &GATEDispatcherFunc;
```

PPC Interrupts:

```
#include <emul/emulinterface.h>
struct IntData
{
    struct EmulLibEntry InterruptFunc;
    struct Interrupt Interrupt;
    struct ExecBase *SysBase;
};

BOOL MyInterrupt(void)
{
    struct IntDataData *MyIntData=(struct IntData*) REG_A1;
    struct ExecBase *SysBase=MyIntData->SysBase;
    u_int8_t *PPCRegs;

    /* put your code here */
    return (0);
}

MyIntData.SysBase = SysBase;
MyIntData.InterruptFunc.Trap = TRAP_LIB;
MyIntData.InterruptFunc.Extension = 0;
MyIntData.InterruptFunc.Func =(void (*)(void))MyInterrupt;
MyIntData.Interrupt.is_Node.ln_Type = NT_INTERRUPT;
MyIntData.Interrupt.is_Node.ln_Pri = 0;
MyIntData.Interrupt.is_Node.ln_Name = "My int";
MyIntData.Interrupt.is_Data = &MyIntData;
MyIntData.Interrupt.is_Code = (void (*)(void))&MyIntData.InterruptFunc;

AddIntServer(INTB_VERTB, &MyIntData.Interrupt);
```

PPC SetFunctions:

```
#include <exec/libraries.h>
struct TagItem MyTags[]=
{
    SETFUNCTAG_MACHINE, MACHINE_PPC,
    SETFUNCTAG_TYPE, SETFUNCTYPE_NORMAL,
    TAG_END,0
};

OldNewLoadSeg = NewSetFunction((struct Library*) DOSBase,
    (ULONG (*)(void))NEW_NewLoadSeg,
    LVO_NewLoadSeg,
    &MyTags);
```

PPC Tasks

```
struct TaskInitExtension TaskInitExt;
struct TagItem MyTags[4];

MyTags[0].ti_Tag = TASKTAG_CODETYPE;
MyTags[0].ti_Data = CODETYPE_PPC;
MyTags[1].ti_Tag = TASKTAG_PC;
MyTags[1].ti_Data = (ULONG) &Dev_DeviceTask;
MyTags[2].ti_Tag = TASKTAG_STACKSIZE;
MyTags[2].ti_Data =(ULONG) 4096;
MyTags[3].ti_Tag = TAG_END;

TaskInitExt.Trap = TRAP_PPCTASK;
TaskInitExt.Extension = 0;
TaskInitExt.Tags = &MyTags[0];

AddTask(&MyDevice->Process.pr_Task,
    (void (*)(void))&TaskInitExt,
    NULL);
```

PPC Processes

Here the situation looks a bit different...in the future we plan a rewrite of DOS' CreateNewProc() to extend the tags for the new ppc features. You can't use the TaskInitExtension above because the PC ptr isn't directly passed to the AddTask().

Solutions are these:

```
struct EmulFunc MyEmulFunc;

MyEmulFunc.Trap = TRAP_FUNC;
MyEmulFunc.Address =(ULONG) ProcessFunc;
MyEmulFunc.StackSize = 8192;
MyEmulFunc.Extension = 0;
MyEmulFunc.Arg1 =(ULONG) SysBase;
MyEmulFunc.Arg2 = 0;
MyEmulFunc.Arg3 = 0;
MyEmulFunc.Arg4 = 0;
MyEmulFunc.Arg5 = 0;
MyEmulFunc.Arg6 = 0;
MyEmulFunc.Arg7 = 0;
MyEmulFunc.Arg8 = 0;

MyTags[0].ti_Tag = NP_Entry;
MyTags[0].ti_Data =(ULONG) &MyEmulFunc.Trap;
MyTags[1].ti_Tag = NP_Name;
MyTags[1].ti_Data =(ULONG) "My Task";
MyTags[2].ti_Tag = TAG_END;
MyProcess = CreateNewProc(&MyTags[0]);
```

You can also use the following way but it doesn't provide a way to set the stack:

```
struct EmulLibEntry GATE_ProcessFunc=
{
    TRAP_LIB,
    0,
```

```
    (void (*)(void)) &ProcessFunc
};

MyTags[0].ti_Tag = NP_Entry;
MyTags[0].ti_Data =(ULONG) &GATE_ProcessFunc;
MyTags[1].ti_Tag = NP_Name;
MyTags[1].ti_Data =(ULONG) "My Task";
MyTags[2].ti_Tag = TAG_END;

MyProcess = CreateNewProc(&MyTags[0]);
```

Be wary to not put the trap structure on the stack because of the asynchronous nature of CreateNewProc().

1.2.2.10 Limitations

The emulation system has certain optimisations because the speed is more important than exact emulation in certain cases:

- the emulation doesn't check for odd PCs for speed issues (maybe this will be added later only for rte, rts, rtd, branch, jxx).
- The interrupt system works completely different than in an ABox machine. The interrupt vectors aren't called at all, so software which patches itself into the interrupt vectors of the 68k isn't compatible anymore. But legal ABox software shouldn't do that.

2 Programming for MorphOS

2.1 Porting code

2.1.1 MorphOS executables

MorphOS executables are in ELF (Enhanced Link Format). They are relocatable. They work in a slightly different way than PowerUP executables so you have to put an

```
ULONG __abox__ = 1;
```

in your code so that it's properly recognized as a MorphOS executable and not mistaken as a PowerUP one. You don't need to do that if you use libnixppc's startup code.

2.1.2 Structure alignments

There's a difference between PPC alignment and 68k alignment when compiling with gcc and other compilers. For datatypes smaller than 4 bytes, gcc will possibly pad the structure in order for a following 4-byte datatype to be long word aligned. Consider this structure:

```
struct LibraryVector
{
    UWORD Command;
    APTR Location;
};
```

gcc will pad it that way:

```
struct LibraryVector
{
    UWORD Command;
    UWORD pad;
    APTR Location;
};
```

so that the *Location* field is long word aligned.

This works as long as your structures are local to your program. As soon as the structure is an external one (like an OS structure) or you share it with a 68k program (like a preference file) you are into trouble. There is a way to instruct gcc to use "68k" alignment using the *#pragma pack (n)* directive. *n* is the number of bytes you want to align to. If you don't supply *n*, the default alignment is used. So to use the normal 2-bytes alignment of ABox, you can do the following:

```
#pragma pack(2)
struct LibraryVector
{
    UWORD Command;
    APTR Location;
};
#pragma pack()
```

2.1.3 Variable arguments (varargs)

2.1.3.1 680x0 stack layout

The 680x0 processors have a pretty simple stack layout. It just grows backward in memory and is linear. Consider the following example:

```
move.l a6,-(sp)
; do some stuff...
move.l (sp)+,a6
```

With the first move instruction, we decrease the stack pointer by 4 bytes then we store the value of the **a6** register in the stack. That value is saved and we can then change **a6** within our code. The last instruction takes care of restoring the original value of **a6**.
Now let's store more data on the stack:

```
move.l #0,-(sp)
move.l #1,-(sp)
move.l #2,-(sp)
move.l #3,-(sp)
```

Let's say **sp** is located at $5000 after the last instruction. The stack would look like the following:

```
$5000 : 00000003
$5004 : 00000002
$5008 : 00000001
$500C : 00000000
```

Now if we want to put the value at $5004 saved on the stack into **d1**, we can do:

```
move.l 4(sp),d1
```

As you can see, handling the stack with a 680x0 is easy.

2.1.3.2 Handling varargs in C

A vararg is a function which has a variable number of arguments passed on the stack.

```
void foobar(int foo, ...)
```

If we are on a 680x0 processor[1], we can access the arguments using something like:

```
int arg1, arg2;

arg1 = (int *)(&foo + 1);
arg2 = (int *)(&foo + 2);
/* and so on */
```

But if you try the previous code with a PPC processor, it won't work. Unlike the 68k, the PPC has no linear stack but uses both the memory and registers yielding to a slightly more complicated, and different code. Unfortunately, the previous example is often seen in 68k programs but there's a way that doesn't require knowing the layout of the stack of every target machine your C program will compile for. The *stdarg.h* header was created to provide portability for functions with a variable number of arguments.

There are 4 important elements in this header:

[1] or any processor with a linear stack

va_start() macro
va_arg() macro
va_end() macro
va_list datatype

The *va_start()* macro is used to initialize a *va_list* by supplying the last non-vararg argument of a function. Access to the variable arguments is done by repeatedly calling the *va_arg()* macro. Finally, *va_end()* is called when access to the variable arguments is no longer needed.
Let's say you have a vararg function like:

```
void foo(int bar, ...)
```

With two variables like:

```
int arg1;
char *arg2;
```

That you will initialize from values taken from your vararg. You first need a *va_list* variable.

```
va_list va;
```

Then you call *va_start()* to initialize it. You supply the va_list and the argument just before the vararg "..." (in that case, that is **bar**).

```
va_start(va, bar);
```

va now contains the first argument in the vararg. To access those arguments, you use the *va_arg()* macro. Note that the first call to *va_arg()* returns the first argument of the vararg.

```
arg1 = va_arg(va, int);
arg2 = va_arg(va, char *);
```

Finally you can call *va_end()* when you're done.

```
va_end(va);
```

Beware, some compilers don't support passing structures by values in a varargs function[2]. There's also a limit in the minimal datatype size supported.
The following program shows how to use varargs macros properly.

[2] No self-respecting programmer uses that anyway.

```
/*
 * Simple vararg example.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

#ifdef __SASC
#define STDARGS __stdargs
#else
#define STDARGS
#endif

/*
 * This routine works on 68k and processors
 * where the stack is linear.
 */
void STDARGS outstuff_68k(FILE *f, ...)
{
    int *p;

    p = (int *)(&f + 1);

    while (*p)
    {
        fprintf(f, "%ld\n", *p++);
    }
}


/*
 * This routine works everywhere. PPC included.
 */
void STDARGS outstuff_vararg(FILE *f, ...)
{
    va_list va;
    int i;

    va_start(va, f);

    while ((i = va_arg(va, int))
    {
        fprintf(f, "%ld\n", i);
    }
    va_end(va);
}


int main(int argc, char *argv[])
{
    printf("output in flat 68k mode:\n");
    outstuff_68k(stdout, 1, 2, 3, 4, 5, NULL);

    printf("output using varargs:\n");
    outstuff_vararg(stdout, 1, 2, 3, 4, 5, NULL);
```

```
    return (0);
}
```

2.1.3.3 ABox varargs

ABox vararg functions use a 68k-like vararg scheme. The fd2inline converter will usually take care of it and generate proper inlines so you usually don't have to worry. But there are cases like when writing a wrapper around such a function or using a function in libamiga.a where you have to know how to handle them.

Let's say you want to create a vararg function which launches an executable, there's how you could do it in ABox 68k (using gcc):

```
ULONG mySystemTags(STRPTR cmd, ...)
{
    /* do some stuff */
    return(SystemTagList(cmd, (struct TagItem *)(&cmd + 1));
}
```

To do the same on a PPC you can use a gcc extension added by Emmanuel Lesueur which allows to have 68k-like varargs. It's an attribute which has to be specified in the function's **prototype**. A common way is to put it in the header where your function is defined. There's the previous example reworked:

```
ULONG mySystemTags(STRPTR cmd, ...) __attribute__((varargs68k));

ULONG mySystemTags(STRPTR cmd, ...)
{
    va_list va;
    va_start(va, cmd);
    /* do some stuff */
    return(SystemTagList(cmd, (struct TagItem *)va->overflow_arg_area));
}
```

Be wary that *__attribute__((varargs68k))* only works in the function's prototype and you must do one.

2.1.4 FPU alignment exceptions

2.1.4.1 FPU instructions

The PPC uses the following FPU instructions for fetching float (32-bit) and double (64-bit) data:

lfs (Load Floating-Point Single)
lfsx (Load Floating-Point Single Indexed)
lfsu (Load Floating-Point Single with Update)
lfsux (Load Floating-Point Single with Update Indexed)
lfd (Load Floating-Point Double)
lfdx (Load Floating-Point Double Indexed)
lfdu (Load Floating-Point Double with Update)
lfdux (Load Floating-Point Double with Update Indexed)

and for storing them it uses:

stfs (Store Floating-Point Single)
stfsx (Store Floating-Point Single Indexed)
stfsu (Store Floating-Point Single with Update)
stfsux (Store Floating-Point Single with Update Indexed)
stfd (Store Floating-Point Double)
stfdx (Store Floating-Point Double Indexed)
stfdu (Store Floating-Point Double with Update)
stfdux (Store Floating-Point Double with Update Indexed)
stfiwx (Store Floating-Point as Integer Word Indexed)

If those instructions aren't used on longword (4 bytes) aligned data, an exception happens.

Some systems implement an exception handler which emulates unaligned accesses. Nevertheless, these emulated accesses have a very significant performance hit on the program's speed. MorphOS' exception handler will output a debug log and halt the program (this used to be the case but is no longer true, emulation is implemented now (beta release 3)). In any case, the best solution is to get rid of unaligned accesses.

2.1.4.2 pointer to double/float

When there are pointers to double/float, gcc might generate a load/store of unaligned data because it assumes its default alignment of 4 bytes is enough[1]. To solve it, we have to force gcc to read the double/float as an integer by the use of the following macro:

```
#define TO_DOUBLE(x) ({ double d; memcpy(&d, x, sizeof(d)); d; })
```

The following is an example of a program which was written to test the performance hit of alignment exceptions.

```
/*
 * Alignement test. Largely inspired by some Apple code
 * (http://developer.apple.com/dev/techsupport/develop/issue28/looney.html)
 */

#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define NUMACCESSESPERCYCLE 200
#define NUMCYCLES 20000
#define TABLESIZE 8 * NUMACCESSESPERCYCLE

enum {
    AL_LONG,
    AL_FLOAT,
    AL_DOUBLE
};

/*
 * Prevents gcc to optimize floats away
```

[1] unless overriden by the #pragma pack directive

```
 */
float get_some_float(void)
{
    return (0);
}

double get_some_double(void)
{
    return (0);
}

double alignloop(int offset, int type)
{
    struct timeval tvs, tve;
    char bt[TABLESIZE];
    long i, j;

    gettimeofday(&tvs, 0);

    switch (type)
    {
        case AL_LONG:
        {
            long *p = (long *)&bt[offset];
            for (i = 0; i < NUMCYCLES; i++)
            {
                for (j = 0; j < NUMACCESSESPERCYCLE; j++)
                {
                    p[j] = 1;
                }
            }
        }
        break;

        case AL_FLOAT:
        {
            float *p = (float *)&bt[offset];
            float x = get_some_float();
            for (i = 0; i < NUMCYCLES; i++)
            {
                for (j = 0; j < NUMACCESSESPERCYCLE; j++)
                {
                    p[j] = x;
                }
            }
        }
        break;

        case AL_DOUBLE:
        {
            double *p = (double *)&bt[offset];
            double x = get_some_double();
            for (i = 0; i < NUMCYCLES; i++)
            {
                for (j = 0; j < NUMACCESSESPERCYCLE; j++)
                {
```

```
                    p[j] = x;
                }
            }
        }
        break;
    }

    gettimeofday(&tve, 0);

    return (tve.tv_sec * 1000000.0 + tve.tv_usec) -
            (tvs.tv_sec * 1000000.0 + tvs.tv_usec);


void main(void)
{
    printf("long aligned: %g usec\n", alignloop(0, AL_LONG));
    printf("long unaligned: %g usec\n", alignloop(1, AL_LONG));
    printf("float aligned: %g usec\n", alignloop(0, AL_FLOAT));
    printf("float unaligned: %g usec\n", alignloop(1, AL_FLOAT));
    printf("double aligned: %g usec\n", alignloop(0, AL_DOUBLE));
    printf("double unaligned: %g usec\n", alignloop(1, AL_DOUBLE));
}
```

2.1.4.3 structure copy

The PPC backend of gcc uses the FPU instructions **lfd** and **stfd** for copying structures during structure assignements as this yields a significant speed gain. Although it is hardly covered by many C books, it is possible to copy a whole structure in C without referencing each member[2] or using functions like memcpy(). Consider the following example:

```
#include <stdio.h>

struct point {
    int x;
    int y;
};

struct target {
    struct point tg_coord;
    int tg_flag;
};

struct launch {
    struct point ln_start;
    struct target ln_end;
};

int main(void)
{
    struct launch *p;
    char buffer[sizeof(struct launch) + 1];

    p = (struct launch *)(buffer + 1); /* set p to be unaligned */
```

[2] Some compilers like SAS/C even allow full structure comparisons using a similar technique.

```
    p->ln_start.x = 1;
    p->ln_start.y = 2;
    p->ln_end.tg_coord = p->ln_start;

    printf("ln_end.tg_coord.x: %ld, ln_end.tg_coord.y: %ld\n",
            p->ln_end.tg_coord.x,
            p->ln_end.tg_coord.y);

    return (0);
}
```

As you can see, both values of *p->ln_start* are copied correctly. The problem is that gcc assumes all the structures are aligned and this obviously isn't the case. There will be 4 alignment emulations needed to copy this structure.
To solve that you have to copy all members manually:

```
p->ln_end.tg_coord.x = p->ln_start.x;
p->ln_end.tg_coord.y = p->ln_start.y;
```

Or use memcpy():

```
memcpy(p->ln_end.tg_coord, p->ln_start, sizeof(struct point));
```

Which is what one would do when copying only some fields between 2 static structures anyway.

2.1.5 Hooks

2.1.5.1 Writing hooks

First of all, hooks should be avoided as much as possible and only be used in cases where you have no other choice[3]. In the cases where hooks are necessary, we will attempt to cover most questions regarding them and provide ideas to ease their writing for code designed for both ABox 68k and MorphOS PPC native builds.

2.1.5.2 Registerized arguments

Let's examine the Hook structure:

```
struct Hook
{
    struct MinNode h_MinNode;
    ULONG      (*h_Entry)();    /* assembler entry point */
    ULONG      (*h_SubEntry)(); /* often HLL entry point */
    APTR       h_Data;          /* owner specific  */
};
```

The purpose of the *h_SubEntry* field is to support compilers where you can't specify registers for parameters. We won't talk about them here as we will use *h_Entry* directly. If you need more information, you can refer to <*utility/hooks.h*> which contains an example on how to use hooks with those compilers.

The function in *hook.h_Entry* uses the following parameters:

- **a0** - a pointer to the hook structure

[3] This is especially true for MUI applications where code abusing *MUIM_CallHook* will quickly become an unmaintainable mess.

- **a1** - a pointer to a parameter structure (message)
- **a2** - a pointer to specific data (object)

Basically **a1** and **a2** are the parameters. **a0** can be used to get *hook.h_Data*. As you can see, a function in *hook.h_Entry* needs the arguments in registers. A function prototype could look like that:

```
ULONG func(struct Hook *hook, APTR object, APTR message);
```

With a compiler like SAS/C, one could write it the following way[4]:

```
ULONG __asm __saveds func(register __a0 struct Hook *hook,
                          register __a2 APTR object,
                          register __a1 APTR message);
```

Using a hook within a 68k program is easy. Declare a *Hook* structure, fill-in the *h_Entry* field to point to *func()* which has its arguments specified in registers and it's done. Using them within PPC code requires a different strategy.

2.1.5.3 Using hooks in PPC code

When we are within PPC code and we want to write a PPC hook function, it has to use the same ABI[5], which means it expects its arguments from those 68k registers. Since our hook function is PPC code, we have to write a gate that will fetch the values of the 68k registers.

First we must redo the prototype of our function:

```
ULONG func(void);
```

And write a gate:

```
struct EmulLibEntry GATEfunc = {
    TRAP_LIB, 0, (void (*)(void))func
};
```

Then modify the function to fetch the value of the 68k registers:

```
ULONG func(void)
{
    struct Hook *hook = (struct Hook *)REG_A0;
    APTR object = (APTR)REG_A2;
    APTR message = (APTR)REG_A1;

    /* actual code goes here.. */
```

And finally fill-in your hook.h_Entry correctly

```
h.h_Entry = (APTR)&GATEfunc;
```

That's it. A working PPC hook.

2.1.5.4 Hooks macros

The use of the following macros is pretty convenient:

[4] The order for the arguments is kept here for portability reasons, even if it's beyond our scope. See <*utility/hooks.h*> for more information.

[5] Application Binary Interface

```
/*
 * Hook, called with the following:
 * n - name of the hook (_hook appened at the end)
 * y - a2
 * z - a1
 *
 * return type is LONG
 */
#ifdef __MORPHOS__
#define M_HOOK(n, y, z) \
    static LONG n##_GATE(void); \
    static LONG n##_GATE2(struct Hook *h, y, z); \
    struct EmulLibEntry n = { \
    TRAP_LIB, 0, (void (*)(void))n##_GATE }; \
    static LONG n##_GATE(void) { \
    return (n##_GATE2((void *)REG_A0, (void *)REG_A2, (void *)REG_A1)); } \
    static struct Hook n##_hook = { 0, 0, (void *)&n }; \
    static LONG n##_GATE2(struct Hook *h, y, z)
#else

#define M_HOOK(n, y, z) \
    static LONG ASM SAVEDS n##_func(__reg(a0, struct Hook *h), \
    __reg(a2, y), __reg(a1, z));
    static struct Hook n##_hook = { 0, 0, (HOOKFUNC)n##_func }; \
    static LONG ASM SAVEDS n##_func(__reg(a0, struct Hook *h), \
    __reg(a2, y), __reg(a1, z))

#endif /* !__MORPHOS__ */
```

Back to our previous example, declaring a similar hook would be done with

```
M_HOOK(foo, APTR object, APTR message)
{
    /* actual code goes here.. */
```

Then your hook structure is *foo_hook* and can be used as eg, MUIA_List_DisplayHook, &foo_hook.