Let's talk 64 bit

Let's ARM ourselves for the future!

Gerph, August 2024

0. Introduction

Introduction

How I'll do this talk

I'll be talking about 64 bit issues with RISC OS.

1. What is 64 bit ARM?
2. How is it different to 32 bit ARM ?
3. What problems does 64 bit cause ?
4. How can we address those problems ?
5. A look at some 64 bit code.
6. Conclusion.

Hopefully I won't be *too* technical, but... yeah, it might get complicated.

Introduction

Who am I?

- I'm Charles, but known as Gerph in most things online.
- I worked at RISCOS Ltd, and produced RISC OS Select.
- I ported almost all of RISC OS to be 32 bit.
- I've written the only other implementation of RISC OS - RISC OS Pyromaniac - from scratch.
- I'm a strong advocate of taking RISC OS forward, rather than letting it stay stagnant.

1. What is it?

What is it?

What is 64 bit ARM ?

- 64 bit ARM was introduced in ARMv8.
- It includes a backwards compatible 32 bit *execution state*.
- The '64 bit' refers to the size of memory and the register size.
- Strictly ARMv6 defines AArch64 and AArch32.
- I'll refer to 64 bit ARM and AArch64 which means the same thing.
- And I'll say 32 bit ARM and AArch32 - which again means that same thing.

What is it ?

Why should we care?

- Processors that support 32 bit are not going to be produced forever.
- If you want to run RISC OS on real hardware in the future, you're going to be stuck with the chips that currently exist.
- Times change, you either move with them, or you're left in the background.

2. How is 64 bit different to 32 bit ?

Differences

Instruction set

- AArch32:
  - 32 bit wide instructions for ARM, and 16/32 bit wide for Thumb.
  - Many instructions can be conditional.
  - Load and store multiple (up to 16) registers in a single instruction.
- AArch64:
  - 32 bit wide instructions. No equivalent of Thumb.
  - Instruction set is not compatible with AArch32.
  - General conditional instructions not supported...
  - ... but some instructions allow 'operate register A or B depending on condition'.
  - Pairs of registers can be loaded and stored.

Differences

Memory

- AArch32
  - Memory is limited to 32 bits of logical space.
  - With large physical address extension, physical memory can be 40 bit, but otherwise is limited to 32 bit.
  - The same logical address space is used by all modes.
- AArch64
  - Logical address space is 64 bit.
  - Physical address space is 48 bit.

Differences

Registers (1)

- AArch32
  - Regular registers are 32 bit wide, named with an `r` prefix.
  - SP, LR and PC are regular registers.
- AArch64
  - Regular registers are 64 bit wide, but can be accessed as 32 bit or 64 bit.
  - General register naming is `r`, but when accessed as 64 bit the prefix used is `x`; 32 bit access uses `w` .
  - SP is a special register, and cannot be used for general operations.
  - SP must be aligned to 16 bytes (128 bits; 2 registers).
  - LR is a regular register.
  - PC is a special register, not directly accessible.

Differences

Registers (2)

- AArch32
  - Floating point support is very variable - lots of choices of what is supported.
  - CPSR and SPSR hold the current and saved processor state.
  - SP, LR (and some others) are banked in different modes.
- AArch64
  - Floating point support is much more reliable - it's all there, or it's not.
  - PSTATE holds the current processor state.
  - SPSR exists for each exception level.
  - SP is preserved in a system register on exception.

Differences

Calling standard (1)

- AArch32 uses APCS/ATPCS:
  - `r0-r3` are used to pass parameters, then the stack.
  - `r0-r3` can be used to return values.
  - `r4-r11` are preserved between calls.
  - `r9` might be the static base.
  - `r10` is the stack limit.
  - `r11` is the frame pointer.
  - `r12` is a scratch register.
  - `r13` is the stack pointer.
  - `r14` is the link register, or temporary values.

Differences

Calling standard (1)

- AArch64 uses AAPCS:
  - `x0-x7` are used to pass parameters, then the stack.
  - `x0-x7` can be used to return values.
  - `x8-x15` can be corrupted between calls.
  - `x16` and `x17` are used for inter-procedure workspace, but can be corrupted otherwise.
  - `x18` is reserved for platform use.
  - `x19-x28` are preserved between calls.
  - `x29` is the frame pointer.
  - `x30` is the link register.
  - `x31` is either the stack pointer or a zero register depending on use.
  - Flags are corruptible across calls.

Differences

Privilege model

- AArch32:
  - Different modes and privilege levels - `USR`, `SVC`, `IRQ`, `FIQ`, `ABT`, `UND`, `MON`, `SYS`, `HYP`.
  - Page tables have permissions per privilege level.
  - Exceptions such as aborts, SWIs or interrupts go through exception vectors.
- AArch64:
  - Different exception levels - `EL0`, `EL1`, `EL2`, `EL3`.
  - `EL0` is similar to `USR`; `EL1` is similar to SVC.
  - Different page tables might exist per exception level (`EL2` and `EL3` have their own; `EL0` and `EL1` share page tables).
  - The exception vector used depends on the privilege level being entered.
  - The reason for changing privilege raise is reported through the 'Exception Syndrome'.

Differences

Example AArch64 code

```
os_inkey:                               // int os_inkey(delay)
    STP     x29, x30, [sp, #-16]!
    AND     x1, x0, #255
    LSR     x2, x0, #8
    MOV     x0, #0x81                   // INKEY
    MOV     x10, #0x6                   // OS_Byte
    SVC     #0                          // Returns R1 = character read
                                       //         R2 = 0 =>char was read, 27=>escape,
                                       //              255=>nothing read
// We are going to return -1 for nothing read, and -2 for escape
    CMP     x2, #27
    BEQ     __os_inkey_escape
    CMP     x2, #255
    CSINV   x0, x1, xzr, NE             // if x2!=255 x0=x1 else x0=-1
os_inkey_exit:
    LDP     x29, x30, [sp], #16
    RET
```

4. What problems does this cause?

Problems

Instruction set (1)

- None of the existing software will work straight off.
- C code will need to be recompiled - and probably modified.
- Assembler code will need re-writing - and there's a lot of it.
- BASIC... would need a re-written BASIC in order to work.
- Norcroft compiler won't work - it only builds for AArch32.
- Some instructions that seem familiar don't work the same way (eg `ORR` with a constant).
- Instructions like `ADRP` to get the address of a page base are used a lot - assuming that the loaded code starts at a known page boundary offset.

Problems

Instruction set (2)

- `SWI` numbers in RISC OS are 24 bits, but the SVC instruction only allows for 16 bits.

Problems

Memory

- Much larger memory means that pointers are larger.
- Existing APIs which use descriptors in memory can't use the whole of memory without being changed - that means registrations that use pointers, and descriptor blocks. If you know the Wimp well, then think of the indirected text pointers, which would need to be 8, not 4 bytes.

Problems

Registers

- Greater numbers of registers, and different restrictions means changes to some interfaces - environmental particularly.
- The stack being 16 byte aligned means that in hand-written code you need to apply more care.

Problems

Calling standard

- The calling standard doesn't provide a static base at all (no equivalent to `R9` in APCS re-entrant code) - more difficulty in compiling for different bases. If you're used to the assembler form of module code, this affects the module workspace registers.
- The calling standard doesn't encourage the stacking of input parameters, so backtraces don't include as much information.
- Related to the calling standard, the compilers don't write function signature strings before functions.
- There is no stack limit register - software stack limit checks are not possible with the regular compilers.
- Without a static base in the C code, having modules that can be instantiated is harder.

Problems

Privilege model

- Some operations that you might assume aren't possible (checking the exception level doesn't work at `EL0` - no equivalent of a `USR` mode check).

Problems

RISC OS APIs (1)

- Some operations are just bad design...
  - Bouncing in and out of privileged modes (`OS_EnterOS`) being one.
  - Directly controlling IRQs (`OS_IntOff`) being another.
  - Allowing the user mode application to handle operations in privileged modes as part of the environment, taking control away from the OS, is another.
- Some APIs use flags for input and output which are hard to work with when writing in C or other languages.
  - `OS_ReadEscapeState`, `OS_ReadC` or `OS_ReadLine` return the escape flag.
  - `OS_BGet` and `OS_BPut` reporting failure with the `C` flag.
  - Many of the older vectors use the flags in unfriendly ways.
- Returning with the `V` flag set to signal errors is pretty unfriendly, so maybe we limit it?

Problems

RISC OS APIs (2)

- Executables need to be runnable.
  - Modules (system extensions) need to be recognised, and safe to run on older systems.
  - Absolutes (user executables) need to be recognised, and safe to run on older systems.
  - Utilities (user tools) need the same.
- Some APIs need to be updated to reflect the new architecture.

Problems

Others...

- Most applications won't exist for RISC OS.
- The user base is small.
- The developer base is smaller.
- Developing for AArch64 may take away from regular RISC OS development.

5. How can we address those problems ?

Decisions

How do we decide what matters?

- RISC OS is complicated.
- We have to make decisions on what is important now, and in the future.
- Doing something is better than talking about the problem for another 20 years.
- With the small number of developers, it makes sense to approach things in a piecemeal way, addressing small parts of the problem.
- Making pragmatic steps to move in the right direction, even if it's not for AArch64 directly, will make the process simpler.
- Recognise that the process will take a long time.
- Recognise also that there might not be end-user visible effects in much of the changes - replacing existing modules with better written versions shouldn't even be noticeable to end-users, but it is still important work.

Decisions

What decisions do I feel make sense?

- I've talked about many different aspects of the problems.
- There will be many things that I and other people will have considered apart from these.
- I've got experience writing a separate version of RISC OS, so I have a unique outlook on the problem.

Let's go through a few things from the areas that we talked about and how I feel they should be handled.

Decisions

Instruction set (1)

- The instruction set is only actually a problem if you're writing lots of code in assembler.
- Stop writing code in assembler...
- ... about 20 years ago.
- All new code should be in C (or other high level language), with minimal assembler, out of line of the primary source.
- Use interface libraries that you can easily replace if the interfaces change, rather than writing direct calls to RISC OS interfaces in business logic.
- Assume that a C compiler (or similar) will be used for AArch64, rather than hand-written assembler.

Decisions

Instruction set (2)

- Since the SWI number cannot fit into the SVC instruction, use a register.
- `R10` is used by the `OS_CallASWI` interface, so let's use that.
- Define that all RISC OS `svc` calls are to `SVC #0`, passing the SWI number in `x10`.

Decisions

Memory (1)

- Many of the interfaces in RISC OS would need to change if you wanted to use the whole 64 bit memory space.
- That would mean that (for example) control blocks would have different forms in 64 bit and 32 bit systems to allow full access to memory.
- Or newly created APIs would be needed, thus reducing the likelihood that people would use them.
- So - initially - don't support the full address space.
- Just support 32 bit logical address space.
- All the APIs have a 1:1 mapping in general (not withstanding some simplifications for less common operations).

Decisions

Memory (2)

- Use the instruction's SVC number (`#0` being for RISC OS SWIs) for future expansion.
    - `#0` means 'SWI with an interface compatible with 32 bit memory systems'.
    - `#1` means 'SWI with an interface compatible with 64 bit memory systems'.
- That gives forward compatibility when we want to increase the capabilities in the future.

Decisions

Registers

- `x18` is defined for operating system use.
- It isn't used in the same way in each system.
- This could be the application context, or workspace for parts of the system.
- The `EL1` value for `x18` could be different to that for the application in `EL0`.

Decisions

Calling standard (1)

- Adopt AAPCS as the only register mapping on the system.
- Since we're saying that all code should be written in C anyhow, this naturally happens.
- We can define SWI calls to pass in their registers in `x0` - `x9`, with the SWI number in `x10`...
- ... and to return like that too, because this is compatible with AAPCS (although it does allow a couple more registers to be passed through than it expects.
- Assume that FP will always be used - if FP is always used, we can always trace the execution.

Decisions

Calling standard (2)

- SWI calls in assembler then look like this:

```
    MOV     x10, #4                     // OS_ReadC
    SVC     #0
```

- If you were to define that in a function, it would look like this:

```
os_readc:
    STP     x29, x30, [sp, #-16]!
    MOV     x10, #4                     // OS_ReadC
    SVC     #0
    LDP     x29, x30, [sp], #16
    RET
```

Decisions

Calling standard (3)

- Because there's no static base register, handling things like multiple instantiation of modules is harder.
  - So, let's not even try.
  - Multiple instantiation is almost never used.
  - And when it is, it's never used well.
  - Preferred instances get really confusing when you're trying to use SWIs or * commands.
  - It's a poor man's substitute for designing your module in a way that can handle entrancy in different contexts.

Decisions

Privilege model

- We've mentioned things like the fact that the user mode application should remain user mode only.
- It would be very useful to define privileges for modules as well, so that they might be able to execute more safely.
- I'm not sure that this is something to attack early on, though.

Decisions

RISC OS external APIs (1)

- Ensure that tests exists for RISC OS APIs, so that you can define what the API does, and compare to what the new implementations will be.
- Being able to characterise the behaviour of a given interface means you can replicate the test, and the implementation.
- That's better than writing the code and declaring it done, without proof or validation.
- As interfaces are implemented, the tests can warn you what the deviance from the expectation is.

Decisions

RISC OS external APIs (2)

- Redesign the APIs that use flags to avoid them:
  - `OS_ReadC` should return the escape state in `x1` rather than the `C` flag, for example.
  - Vector `CnpV` should be shot and replaced with a less ugly clone (it takes 2 flags on input, whose combination means different things).
  - (many others)
- Replace the environment handlers with `EL0`-only handler state.
  - Rather than being entered from other modes, these should only be entered in `EL0`.
  - The application only ever executes in `EL0`.
  - Maybe create a process model using `OS_TaskControl` eventually.

Decisions

RISC OS internal APIs (3)

- If modules are written in C, why not make them easier to work with?
  - Replace the module interface to match that used by CMHG.
  - For example, SWI calls could pass a register block in, together with the SWI numbers in `x0` and `x1` , rather than the registers being in `x0` - `x9` and the register number in `x10`.
  - Rather than returning with `V` set and a pointer to an error in `R0`, just return the pointer to an error or 0 in `x0`, just like CMHG expects.
  - This simplifies the implementation of the module veneers to almost nothing.
  - Without module instantiation, the need for `r12` private workspace pointer almost entirely goes away.
- The result would be that actually your SWI entry point could be *just* a C function, with no veneer at all.

Decisions

RISC OS executables (AIF - 1)

- Absolute files already have a way to recognise the module is not compatible with a system by using the header flags.
- Recognising absolutes that are unsuitable is easily handled through the `Service_UKCompression` calls, if necessary.
- Updating the header to define it as incompatible with 32 bit systems if it is run is easy enough.
- Let's define the initial section of the AIF to be 32 bit ARM with headers set appropriately.

Decisions

RISC OS executables (AIF - 2)

```
_start
        NOP                             ; was decompression
        NOP                             ; was reloc
        NOP                             ; was zero init
        BL      entry                   ; was image entry
        SWI     OS_Exit                 ; OS_Exit
        DCD     <variable>              ; read only size
        DCD     <variable>              ; read write size
        DCD     0                       ; debug size
        DCD     <variable>              ; zero init size
        DCD     0                       ; debug type
        DCD     0x8000                  ; linked base address
        DCD     0                       ; workspace size (obsolete)
        DCD     64                      ; address mode
        DCD     0                       ; data base address
        DCD     0                       ; reserved
        DCD     0                       ; reserved
entry
        ADR     r0, error_block         ; was decompression
        SWI     OS_GenerateError        ; was reloc
error_block
        DCD     0
        = "AArch64 binaries cannot be run on 32 bit RISC OS" ,0
        ALIGN
        DCD     0
```

Decisions

RISC OS executables (AIF - 3)

- Then the 64 bit code starts at `&8100` (file offset `&100`), and has a similar format to the original AIF header.

```
_aif64_entry:
        NOP                     // Relocation code (not currently used)
        BL      _zeroinit       // Zero initialisation
        BL      _start
        MOV     x10, #0x11      // OS_Exit
        SVC     #0
```

- Why no decompression entry?
  - Memory is cheap.
  - Discs are cheap.
  - Distribution formats like Zip (using deflate) or other more modern formats are more efficient.
  - Makes patching harder.
  - Why bother?

Decisions

RISC OS executables (AIF - 4)

- Future developments would almost certainly integrate ELF loading for application space.
- ELF linking removes a lot of the need to define a specific header format, etc.

Decisions

RISC OS executables (Modules - 1)

- Modules have had a standard 'feature flags' to differentiate them since the early 32 bit system was created.
- Pyromaniac indicates the architecture of the module with one of 16 values in the feature flags - bits 4-7 contain the architecture:
  - `0` => AArch32
  - `1` => AArch64
  - `2` => x86-64
  - `15` => Python (for PyModules in RISC OS Pyromaniac)
- https://pyromaniac.riscos.online/pyromaniac/prm/kernel/modules/modules-supplement.html

Decisions

RISC OS executables (Modules - 2)

- Just setting the features flag won't work on earlier systems that don't know about it.
  - Also set bit 30 of the module initialisation offset to indicate that the architecture is present (and not ARM).
  - This prevents earlier systems from loading the module, as it is an invalid offset, so makes them safe.
  - Could be recognised by `Service_ModulePreInit` on earlier systems.
- We can also include an indication of the zero-initialisation space needed, so that the OS can reserve enough room after the module.
  - Indicate that this information is present with a bit in the features flag.

Decisions

RISC OS executables (Modules - 3)

```
.section .init.rmf, "a"

.word     0                // offset to code    start code
.word     init + (1<<30)   // offset to code    initialisation code
.word     final            // offset to code    finalisation code
.word     service          // offset to code    service call handler
.word     title            // offset to string  title string
.word     help             // offset to string  help string
.word     command_table    // offset to table   help and command keyword table
.word     swi_base         // number            SWI chunk base number
.word     swi_handler      // offset to code    SWI handler code
.word     swi_names        // offset to table   SWI decoding table
.word     0                // offset to code    SWI decoding code
.word     messages_file    // offset to code    Messages filename
.word     modflags         // offset to table   Module features

modflags:
.word   0 + (1<<2) + (1<<4) // 32 bit NOT supported + zero-init present + AArch64
.word   _zinit_size         // Size of our Zero initialised area

title:
.asciz "ModuleWithInit"
help:
.ascii "ModuleWithInit"
.byte 9
.asciz "1.00 (11 Aug 2024) A test of module init/final code"
```

Decisions

RISC OS executables (Modules - 4)

- Compiled C code commonly uses the `ADRP` instruction to get the relative address of the start of a page.
- Example of reading a 32 bit value from workspace.

```
00008580 : 90000000 : .... : ADRP    x0, &00008000
00008584 : b94ce000 : ..L. : LDR     w0, [x0, #&ce0]
```

- This means that we assume that the binary is loaded at a known position relative to a page boundary.
- That's true for Absolutes at `&8000`, but modules have traditionally been loaded at an arbitrary position (usually with the last nibble ending in `&4`), so this requirement doesn't hold.

Decisions

RISC OS executables (Modules - 5)

- In 64 bit modules will be in a managed area which also loads them at a page boundary + `&4`.
  - This allows the blocks to be allocated in a special form of `OS_Heap`.
- Traditionally the modules were loaded into the 'RMA' - 'Relocatable Module Area'.
- With different requirements for the code and data, the modules can instead be placed in an explicit 'Module Area', which has page aligned allocations.
- Rename the 'RMA' to 'Random Memory Area'.
  - That better describes what it is used for!
  - The new dynamic area is number 9 - the area which was reserved in RISC OS 4 for the 'read-only module area'.
- With a bit more thought, this will allow module code regions to be protected from being overwritten.

Decisions

RISC OS executables (Utilities - 1)

- Utilities have had a standard header defined since RISC OS Select.
  - https://riscos.com/support/developers/riscos6/programmer/codeformats.html
- We can reuse this to indicate the form of the executable, in a similar way to the AIF files.
- Instead of saying that the utility is 32 bit or 26 bit, we can say it's built for AArch64.

Decisions

RISC OS executables (Utilities - 2)

```
        AREA |Asm$$code|, CODE, READONLY

OS_GenerateError * 0x2b
OS_Exit          * 0x11

        ENTRY
_start
        B       arm32_entry
        DCD     &79766748                 ; magic 1
        DCD     &216C6776                 ; magic 2
        DCD     <variable>                ; Read only size
        DCD     <variable>                ; Read/write size
        DCD     64                        ; built for 64 bit
        DCD     <variable>                ; AArch64 entry point offset
arm32_entry
        ADR     r0, error_block           ; was decompression
        CMP     r0, #1<<31                ; set V flag
        MOV     pc, lr
error_block
        DCD     0
        = "AArch64 binaries cannot be run on 32 bit RISC OS", 0
        ALIGN
```

Decisions

RISC OS architecture APIs (1)

- The decision to use only 32 bits of logical address space for the operating system means that we don't have to redesign existing APIs.
  - All pointers will be truncated down to 32 bits.
  - This means that applications and modules just have to be recompiled with suitable types to make them work in the new system.
  - The bulk of the APIs won't have changed (except for replacing some ugliness in flags), so the system can be made to work more quickly.
  - Later work may enable larger memory regions.
  - By doing this we reduce the problem to one of language - just the instruction set needs to be addressed, not the mechanism by which the OS functions.

Decisions

RISC OS architecture APIs (2)

- The Debugger module able to disassemble instructions used by the current system.
  - `Debugger_Disassemble` will need to disassemble AArch64.
  - RISC OS Pyromaniac provides `Debugger_DisassembleArch` which allows other architectures to be disassembled.
  - This means that existing systems can disassemble AArch64 code, and future AArch64 OSs can still disassemble 32 bit code.
  - The architecture number matches that module architecture numbers.
  - See https://pyromaniac.riscos.online/pyromaniac/prm/programmers/debugger-supplement.html for documentation.
  - Try it on shell.riscos.online - `*MemoryI 6 <address>` disassembles AArch64.
  - My work on a replacement Debugger will address some of this.

Decisions

RISC OS architecture APIs (3)

- To make it easier to understand the environment we run in, we need to be able to ask what the system is.
  - `OS_PlatformFeatures 64` is implemented in RISC OS Pyromaniac to describe the CPU register layout used by the exception block.
  - This describes the current architecture (using the same architecture number), and the registers present in the block.
  - Debuggers (like the Debugger module and others) can use this information to report the system type.
  - RISC OS Pyromaniac reports the buffer size as 33 * 64 bit registers + 1 * 32 bit register - this doesn't account for VFP registers, which is problematic.
  - See: https://pyromaniac.riscos.online/pyromaniac/prm/kernel/progenv-supplement.html

Decisions

RISC OS architecture APIs (4)

- Other informational interfaces might need to change:
  - `OS_Byte &81` returns ARM based systems as `&Ax` - what about AArch64? `&40` maybe? (that's free)
  - Most of `OS_PlatformFeatures` may not make sense for AArch64.
  - `OS_ReadSysInfo 8` for platform class probably should report differently.
- In most cases, user-facing APIs should be pretty much the same.

Decisions

RISC OS Libraries

- What about the more general libraries?
- The C library is obviously vital if we're writing everything in C (of course, you could use another language, but you can't reuse as much code).
  - Many C libraries exist, which can be used as the basis for a C library, without the legacy of the Codemist library.
  - Some can be written from scratch - it's not hard.
- RISC_OSLib is frustrating, but it can be converted, and then some things may just start working.
- OSLib has its `defmod` tool to write assembler files, which could be updated.
  - My own OSLib parser is relatively easy to re-target to generate AArch64 code.

Decisions

C modules and applications

- Convert them to modern C.
- Recompile to AArch64.
- Use tests for the original code (or write them) to compare the behaviour.
- I've spoken about testing modules in detail, which you can find on my site.
- You can find other examples of tests in the `riscos-tests` repository on GitHub.
- I've converted one a few things now, and in general it's not that hard - even if I have had to update the C library as I go.

Decisions

Assembler-only modules

- Re-write modules in C.
- Start from the documented API.
- Don't try to convert the assembler implementation unless it's actually required - just do what the API requires.
- I know this method works, because I've done it for a good chunk of RISC OS.
- Don't perpetuate the insanity by attempting to create a new AArch64 assembler module.
- Write tests as you go and compare the assembler and C implementation.
- RISC OS Pyromaniac has many tests that can be reused.
- Release versions of the module as you go for people to test.
- Eventually, commit to only the maintainable C version.
- Julie Stamp can probably give good advice on how to do this.

5. A look at 64 bit RISC OS code

64 bit RISC OS code

Utility code

```
        STP     x29, x30, [sp, #-16]!
        MOV     x3, x1

        MOV     x10, #OS_WriteS
        SVC     #0
.asciz "Hello "
.balign 4

        LDRB    w1, [x3]
        ADR     x2, message_world
        CMP     w1, #0
        CSEL    x0, x2, x3, eq

        MOV     x10, #OS_Write0
        SVC     #0

        MOV     x10, #OS_NewLine
        SVC     #0

        MOV     x0, #0      // no error return
        LDP     x29, x30, [sp], #16
        RET

message_world:
.asciz "world"
```

Executing the code

We can run the code with:

```
pyrodev --command hello_world
```

... and we can debug it with:

```
pyrodev --command hello_world --boot-debug trace
```

Hello world utility demo

64 bit RISC OS code

C code (1)

- That utility was largely just to show how you can write utilities and what the code looks like.
- We should be writing C code, though, so let's look at a C hello world program.
- This program does a little more than utility.

64 bit RISC OS code

C code (2)

```
/*******************************************************************
 * File:        hello_world_printf
 * Purpose:     Showing that Printf and the C library work
 * Author:      Gerph
 * Date:        10 Aug 2024
 *******************************************************************/

#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    if (argc > 1)
    {
        printf("Args:\n");
        for (int arg=1; arg<argc; arg++)
        {
            printf("  %s\n", argv[arg]);
        }
    }
    printf("C library:\n");
    printf("%s", _clib_version());
    return 0;
}
```

Hello world C demo

64 bit RISC OS code

C code (3)

```
Hello world!
Args:
  whee
C library:
C library/64
GerphCore vsn 0.01 (10 Aug 2024)
```

- That shows that we're able to run the programs in a sensible way.
- It might not be too interesting, but I've ensured that the C library reports itself with the defined format for version extraction.
- The qualifier for the library is given as `/64`, since I wanted to differentate it from the others - this also means that the `jmpbuf` is a different layout.
- This follows the pattern established in: https://www.riscos.com/support/developers/riscos6/programmer/extendedc.html

64 bit RISC OS code

More example programs

- Hello world was the first RISC OS 64 program.
  - It proved that the system worked.
  - Validation of the way that RISC OS could run things in AArch64.
- The second program was an animation using the Draw module.
  - Which was a nice proof that the interfaces were usable in the new environment.
- The third program was a little more challenging...

RISC OS 64 bit programs 2 and 3 demo

64 bit RISC OS code

When things go wrong (1)

- RISC OS Pyromaniac can report similar information for AArch64 to that for 32 bit.
- Exceptions aren't properly handled just yet, though, so it's not quite as useful as you might like.
- However it's still enough to get a useful trace when something breaks.
- My `crash` program, tries to print a string at &9876543, which causes an abort.

64 bit RISC OS code

When things go wrong (2)

```
==== Begin exception report ====
Exception triggered: Exception 'Data Abort'
Fault address: &00000000
Fault status:  Read, Fault status &0
  x0  = &00000000, x1  = &09876543, x2  = &ffffffd8, x3  = &00000001
  x4  = &00000000, x5  = &00000001, x6  = &00000000, x7  = &00000000
  x8  = &00000000, x9  = &00000000, x10 = &00000003, x11 = &00000000
  x12 = &00000000, x13 = &00000000, x14 = &00000000, x15 = &00000000
  x16 = &00000000, x17 = &00000000, x18 = &00000000, x19 = &00000001
  x20 = &0410944c, x21 = &00000000, x22 = &00000000, x23 = &00000000
  x24 = &00000000, x25 = &00000000, x26 = &00000000, x27 = &00000000
  x28 = &00000000, x29 = &00107f20, lr  = &ffff0090, xzr = &00000000
  sp  = &00107f20, pc  = &00008214
  CPSR= &00000010 : USR-32 ARM fi ae qvczn
Locations:
  x20 -> "crash,ff8 " in DA 'System heap'
  x29 -> [&00107fa0, &00000000, &000083d0, &00000000] in DA 'Application Space'
  lr is DA 'Exception vectors'
  pc is DA 'Application Space': Function myprint+&8
C backtrace:
      8214 function print_many
      83d0 function main
      8660 function __main
```

64 bit RISC OS code

"Show me the source code!"

- All these example programs are built using GCC 11 for AArch64.
- They're cross compiled inside a docker container.
- There is a little bit of magic to make things work in the way that RISC OS expects - headers and static linkages.
- They're all publicly available on GitHub:
  - https://github.com/gerph/riscos64-simple-binaries/ (includes C library)
  - https://github.com/gerph/riscos64-mfoot-chuckieegg/
- They have their own C library.
  - It's pretty simplistic, but you can see it is sufficient to run these demos.

64 bit RISC OS code

And there's more...

- The first few examples were custom, but ChuckieEgg was written by someone else and converted to 64 bit.
- It shows that an application that you might know can be made to work - and it didn't take that long.
- Chuckie egg's port took 7 hours.
- Most of that time was writing the C library.
- An hour of it was realising that I'd implemented `_kernel_swi` wrongly.

Are those applications enough to show that 64 bit RISC OS isn't so hard to get working if you have the inclination ?

64 bit RISC OS code

And there's more...

- Yes, those of you who have seen my presentations before will know what's coming.

64 bit RISC OS code

And there's more...

- Yes, those of you who have seen my presentations before will know what's coming.
- This entire presentation was running on RISC OS Pyromaniac in my presentation tool compiled for AArch64.

64 bit RISC OS code

And there's more...

- Yes, those of you who have seen my presentations before will know what's coming.
- This entire presentation was running on RISC OS Pyromaniac in my presentation tool compiled for AArch64.
- The ImageFileRender module was also ported to AArch64 and is providing the images for the presentation.

64 bit RISC OS code

And there's more...

- Yes, those of you who have seen my presentations before will know what's coming.
- This entire presentation was running on RISC OS Pyromaniac in my presentation tool compiled for AArch64.
- The ImageFileRender module was also ported to AArch64 and is providing the images for the presentation.

Is that enough to show that 64 bit RISC OS code can be produced from original applications?

Final 64 bit demo

6. Conclusion

Conclusion

We've talked a little while about 64 bit RISC OS.

- What it is
- What's different.
- What problems we have.
- How we can deal with some of those problems.
- Some examples of 64 bit RISC OS code running.

Questions

I'll take any questions that people have.

Slides and Info: **http://presentation.riscos.online/64 bit/**