Three More Cortex-M Inferno Ports

David Boddie
david@boddie.org.uk

ABSTRACT

Starting from a port of Inferno to a single microcontroller, work is ongoing to extend support to a wider group of microcontrollers and small systems based on ARM Cortex-M4 and Cortex-M7 cores.

Introduction

A port of Inferno to the STM32F405 [1] microcontroller has been available for over a year. Based on an ARM Cortex-M4 core, this microcontroller provides a basic level of features that makes code written for it applicable to other Cortex-M4 microcontrollers, and also to more powerful systems based on the Cortex-M7 core, without too much adjustment.

It is unclear how much code could be reused to support other related cores, such as the Cortex-M0+, which is used by the RP2040 microcontroller. However, since that microcontroller is already supported by the Raspberry Pi Pico port [2], no effort has been made to investigate this.

The descriptions that follow briefly cover some implementation issues that affected ports to systems based on Cortex-M4 and Cortex-M7 cores made during the last year. It may be useful to consult the Wikipedia article [3] about ARM Cortex-M cores to learn more about the similarities and differences between them.

Toolchain

The existing ports of Inferno to ARM systems relied on the availability of a regular 32-bit instruction set. In contrast, Cortex-M4 cores only provide support for Thumb and Thumb-2 instructions, making the use of a different toolchain necessary: a combination of `tc`, `5a` and `tl`.

Of these tools, `tc` and `tl` were modified to add support for the floating point instruction set used by Cortex-M4 cores with the appropriate extension. Additionally, `tl` was adjusted to re-enable existing support for compiling string constants into the text section of a binary. Depending on the configuration and tools used to build an Inferno system, this can lead to a substantial reduction in RAM usage at run-time.

Task switching

Cortex-M4 cores differ from traditional ARM cores in both the instruction sets used and in the available processor modes. This means that some techniques used by many existing ports of Inferno to ARM systems are not applicable to these cores. The way that task switching is performed is one area that needed revising to fit a different processing model.

Existing ARM ports rely on the ability of the processor to switch between the processor modes during exception handling, using this ability to implement a mechanism for task switching that ensures that `setlabel` and `gotolabel` calls are always called in the same processor mode.

On Cortex-M4 cores, exceptions cause the processor to enter a Handler mode that it can only leave by returning from the exception back to the interrupted code. While it is possible to perform task switching while running in this mode, the task switching mechanism also needs to be able to run in the normal Thread mode. As a result, a different mechanism is used to ensure that Thread mode is always used when performing task switching.

Floating point support

Cortex-M4 cores tend to provide some level of hardware floating point arithmetic support, and this is the case with the STM32F405 microcontroller. The existing code in ports such as the Raspberry Pi port [4] is useful in terms of providing a framework for supporting floating point instructions, but required adjustment to work with the floating point instructions supported by the newer cores.

Use of a floating point emulator is necessary for Cortex-M4 cores because they only tend to support single precision arithmetic, making it necessary to trap instructions that operate on double precision values. Many existing ports tend to rely completely on an emulator to handle all floating point instructions when an undefined instruction exception occurs. This maintains a set of emulated floating point registers associated with each process. However, since we want to take advantage of the hardware support for floating point, an approach that maintains the actual registers is used. This relies on the modification of stacked floating point registers during exceptions.

One simplification made to the floating point emulator is to only use it to handle one undefined instruction at a time instead of trying to handle as many consecutive instructions as possible. Since supported and unsupported instructions may be mixed together, the additional work of opportunistically checking for consecutive instructions is time that could be spent letting the hardware either execute them or raise another exception.

One disadvantage to this kind of partial emulation solution is the additional overhead it places on the stack for the kernel and each process, as space for a full set of floating point registers is reserved on the stack when an exception occurs. A pure software solution might prove to be sufficient if use of hardware floating point support doesn't provide substantially faster floating point arithmetic performance.

It should be noted that Cortex-M7 cores can be provided with double precision floating point units. For these, either minimal or no emulation is needed.

Memory savings

As described in the previous work [5], some effort was needed to reduce the amount of RAM used by Limbo modules on a running system. This was achieved by expanding the code sections of Dis files during compilation of an Inferno system and running them directly from flash memory.

It was observed that the freezing process had the effect of moving much of the root file system from the data section to the text section of the binary. This has the effect of reducing the initial RAM overhead of the system. To take advantage of this feature for general use, the `data2texts` utility was created to replace the existing `data2s` utility which normally translates binary data into assembly language `DATA` structures for compilation into the system. This variant simply uses `TEXT` directives instead, enabling the data to be stored alongside kernel code.

Additional savings to the initial RAM footprint of the system are obtained by passing the `-t` option with the `tl` loader. This has the effect of placing string constants in the text section of the binary, again reducing the amount of data copied into RAM.

The following table shows the effect of using `data2texts` instead of `data2s` to encode data, and the larger effect of including string constants in the text section of the binary for a microcontroller with 384 kilobytes of RAM. The upper part of the table shows the effect on a system with non-frozen Limbo modules; the lower part shows the effect with frozen Limbo modules.

		Data strings		*Text strings*	
		data2s	**data2texts**	**data2s**	**data2texts**
	kernel	477272	477272	477272	477272
Non-frozen Limbo modules	**maxsize**	165120	324608	339712	339968
	hw	132832	132832	132832	132832
	kernel	614584	614608	614584	614608
Frozen Limbo modules	**maxsize**	309504	324608	339712	339968
	hw	94048	94048	94048	94048

The **kernel** rows show the size of the system binary in bytes, containing the kernel and root file system. With non-frozen modules, this value is not affected by the changes, and there is only a small effect on the system with frozen modules. What is noticeable is the increase in size of the binary when modules are frozen.

The **maxsize** rows show the number of bytes of RAM available to the system. Using the `data2texts` utility provides a memory saving that is largely overshadowed by the effect of compiling string constants into the kernel text. Since there is an overlap in these approaches, the combination of them yields only a marginal additional memory saving. Freezing Limbo modules also provides a similar benefit, but only when these other approaches are not used.

Where freezing modules provides a distinct benefit is in the run-time use of RAM by modules. The **hw** rows show the corresponding values in bytes from the summary provided by reading the `/dev/memory` file, indicating the allocation high water mark after running the `ls` utility. Note that this is only affected by the use of frozen modules, with the other approaches having no effect on the run-time RAM usage. This is where we see clear RAM savings by eliminating the need to allocate memory for parts of the module loading process at run-time.

New platforms

The initial choice of the STM32F405 microcontroller as a target for Inferno was determined by its low RAM provision, but was also a pragmatic decision based on the perceived effort needed to port Inferno to the ATSAMD51J20 microcontroller. Returning to this microcontroller later, the port was not as difficult as expected.

Similar microcontrollers with more RAM were obtained to determine how much of Inferno's functionality could be used on these small devices. The first of these was the SparkFun MicroMod Artemis processor, which uses an Ambiq Apollo3 core. In an attempt to expand coverage to related processors, a development board with an i.MX RT1062 processor was also obtained due to its improved features and its use of a Cortex-M7 core.

Generally, the limiting factors to a viable Inferno port appear to be the amount of flash memory and RAM provided by a target. Informal tests have indicated that about 256KB is the minimum amount of flash memory that a system image needs for Thumb-2 code, and probably at least 512KB would be a comfortable minimum. Although some activities are possible with less than 256KB of RAM, there is more room for experimentation with 384KB or greater.

Since the ports created after the first used the initial port as a base, the result was four ports with a fair amount of duplicated code. To make this easier to work with, the shared code was consolidated in a single `cortexm` directory in much the same way that the `sa1110` directory contains common code for a number of older related ports.

Conclusion and future work

Porting Inferno to microcontrollers based on ARM Cortex-M cores continues to be an interesting activity, with plenty of scope for improvements and optimisations. It is hoped that future work will help to reduce Inferno's memory footprint even further, making it possible to run it on a wider range of microcontrollers.

The source code associated with this work can be found in the following repository:
https://github.com/dboddie/inferno-os/tree/cortexm

References

1. David Boddie, "STM32F405 Port of Inferno",
   https://github.com/dboddie/inferno-os/tree/stm32f405
2. Caerwyn Jones, "Raspberry Pi Pico Port of Inferno",
   https://github.com/caerwynj/inferno-os/tree/pico
3. Wikipedia, "ARM Cortex-M", *https://en.wikipedia.org/wiki/ARM_Cortex-M*
4. Lynxline, "Porting Inferno OS to Raspberry Pi",
   http://lynxline.com/porting-inferno-os-to-raspberry-pi/
5. David Boddie, "Hell Freezes Over: Freezing Limbo modules to reduce Inferno's memory footprint",
   9th International Workshop on Plan 9, Waterloo, Canada, 2023
   https://9e.iwp9.org/