Google

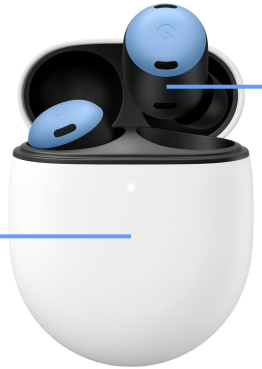
Through the Compiler's Keyhole:
Migrating to Clang Without Seeing the Source

LLVM Dev Mtg '25

Petr Hosek (phosek@google.com)

October 2025

2 years ago we migrated Pixel Buds Pro...

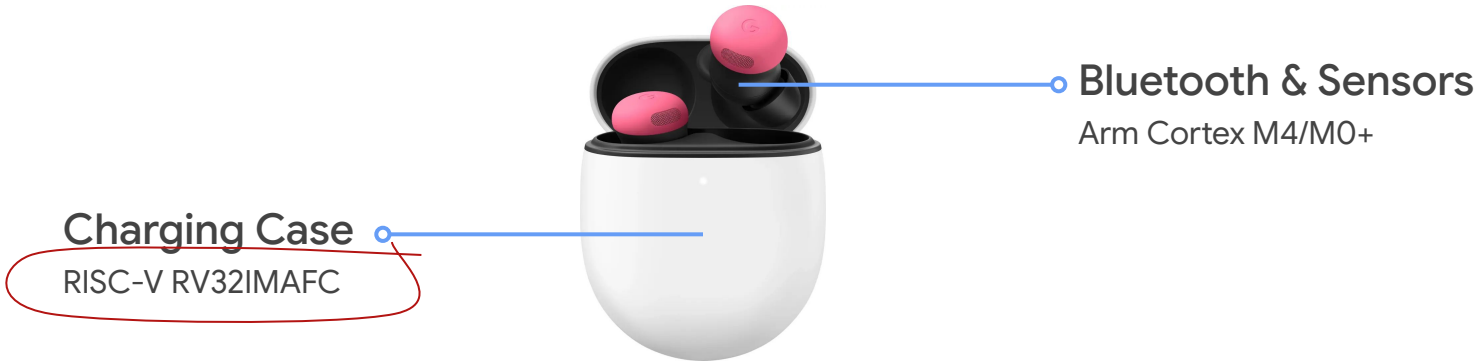


Bluetooth & Sensors
Arm Cortex M4/M0+

Charging Case
Arm Cortex M4

[LLVM Toolchain for Embedded Systems](#) Prabhu Karthikeyan Rajasekaran

Last year we ***partially*** migrated Pixel Buds Pro 2...



[Modern Embedded Development with LLVM](#) Petr Hosek

```
$ clang --target=riscv32-unknown-elf -march=rv32imafc -mabi=ilp32f … -lsdk -o app.elf
ld.lld: error: libsdk.a(a.o):(function f: .text+0x0): unknown relocation (245) against symbol
…
```

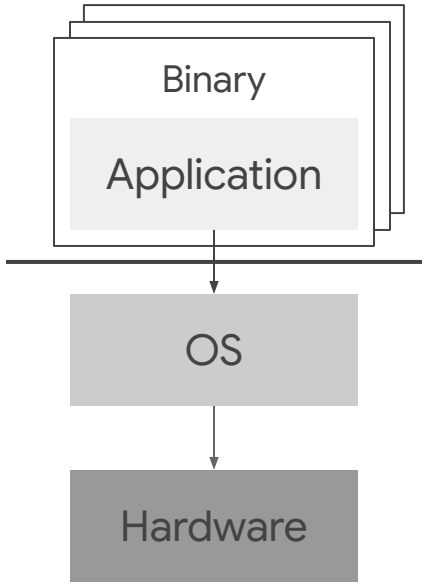
What's the issue?

RISC-V RV32 core has vendor extensions which include custom branch instructions that require custom relocations.

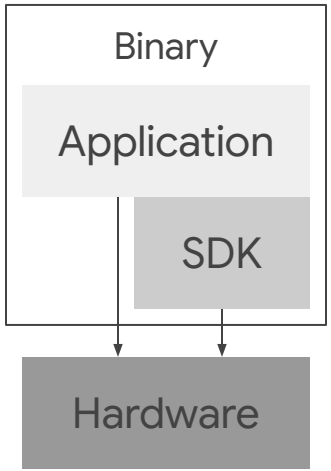
- The vendor toolchain unconditionally emits custom instructions.
- There is no support for handling custom relocations in LLVM (yet).

We cannot link the vendor SDK with objects built with our toolchain.

[lld] Add infrastructure for handling RISCV vendor-specific relocations. #159987



MPU



MCU

What were our options?

Continue using the vendor toolchain

We can't use a single toolchain for the entire product.

Rewrite the vendor SDK to be compatible

No readily available tooling, but we could build on top of LLVM.

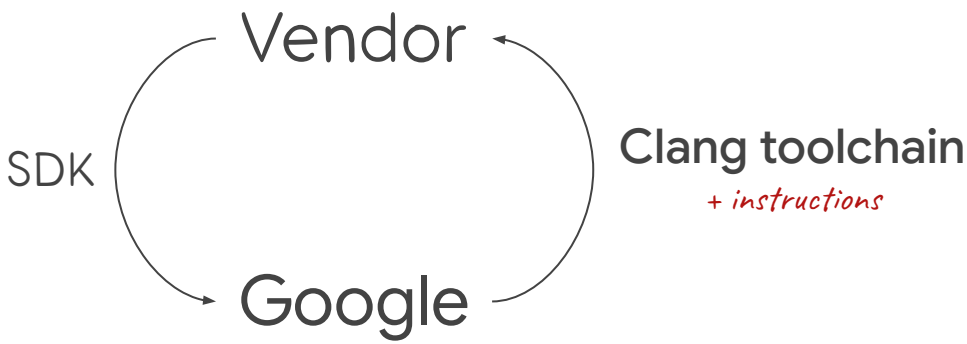
Rebuild the vendor SDK with our toolchain

Requires support from the vendor.

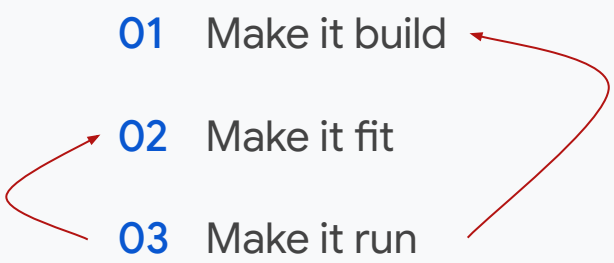
Migrating to Clang without seeing the source

The vendor was willing to rebuild their SDK with Clang but the code is proprietary and we couldn't see the code.

We couldn't rely on manual approach used with previous projects and had to leverage **automated solutions** where possible to aid migration.


Vendor
SDK
Google
Clang toolchain
+ instructions

Strategy



Does it build?

Clang ≠ GCC

Fixits are helpful but not always available.

LLVM libc ≠ newlib

We used Clang-Tidy checks to aid migration but need more.

LLD ≠ GNU ld

We need better diagnostics and documentation.

```
$ clang --target=riscv32-unknown-elf -march=rv32imafc -mabi=ilp32f … -lsdk -o app.elf
ld.lld: error: section can't have both LMA and a load region
…
```

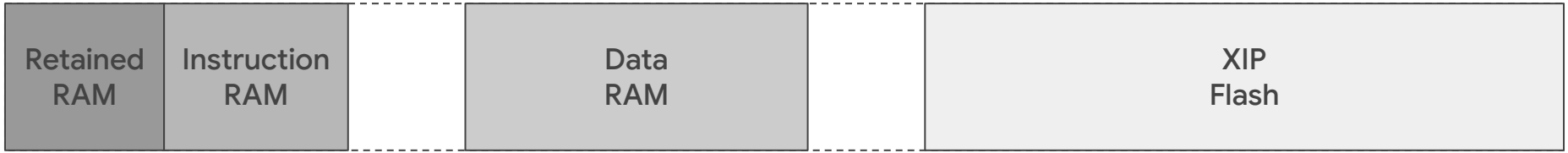
[RFC] Improve linker script handing in LLD

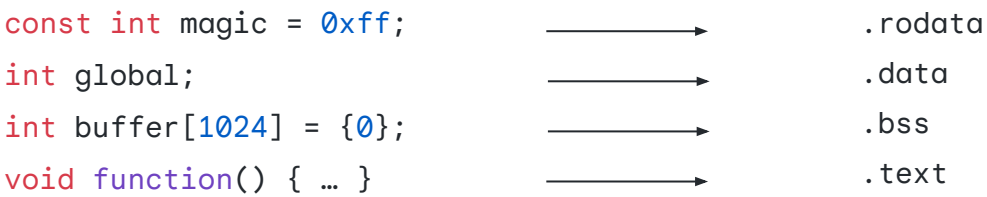
Does it fit?

The MCU uses heterogeneous memory layout:

- The placement of certain symbols is critical for correctness.
- The placement of performance-sensitive code may be critical for performance.

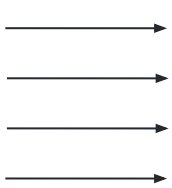
There's no support for heterogeneous memory in standard C/C++.




const int magic = 0xff;
int global;
int buffer[1024] = {0};
void function() { … }
.rodata
.data
.bss
.text

-fdata-sections

```
const int magic = 0xff;
int global;
int buffer[1024] = {0};
void function() { … }
```



```
.rodata.magic
.data.global
.bss.buffer
.text.function
```

-ffunction-sections

```
__attribute__((section(".retained"))) const int magic = 0xff;
__attribute__((section(".retained"))) int global;
__attribute__((section(".retained"))) int buffer[1024] = {0};
__attribute__((section(".retained"))) void function() { … }
```

```
MEMORY {
  RETAINED(rwxi) : ORIGIN = 0x00000000, LENGTH = 64k
  ...
}
SECTIONS {
  .retained : { *(.retained); } > RETAINED
  ...
}
```

```
__attribute__((section(".retained.magic"))) const int magic = 0xff;
__attribute__((section(".retained.global"))) int global;
__attribute__((section(".retained.buffer"))) int buffer[1024] = {0};
__attribute__((section(".retained.function"))) void function() { … }
```

Making things fit

Manual symbol placement overrides default compiler/linker logic.

- Placing all symbols into the same section breaks linker GC.
- You need to manually choose the right section type.

We added `-fseparate-named-sections` option to support linker GC without needing to modify the source code.

```
__attribute__((section(".retained.rodata"))) const int magic = 0xff;
__attribute__((section(".retained.data"))) int global;
__attribute__((section(".retained.bss"))) int buffer[1024] = {0};
__attribute__((section(".retained.text"))) void function() { … }
```

```
__attribute__((memory("retained"))) const int magic = 0xff;
__attribute__((memory("retained"))) int global;
__attribute__((memory("retained"))) int buffer[1024] = {0};
__attribute__((memory("retained"))) void function() { … }
```

[RFC: Support for Memory Regions in ELF]

```
MEMORY {
  RETAINED(rwxi) : ORIGIN = 0x00000000, LENGTH = 64k
  ...
}
SECTIONS {
  .retained.text : { INPUT_SECTION_MEMORY(retained) *(.text .text.*); } > RETAINED
  ...
}
```

[RFC: Support for Memory Regions in ELF]

Making things smaller

We build all toolchain runtime libraries using `-ffat-lto-objects` to support LTO.

- We need to ensure that ABI is compatible across all object files.
- Using LTO with runtime libraries remains a challenge.

For now we only use LTO for the application code.

LT-Uh-Oh: Adventures trying to LTO libc Paul Kirth & Daniel Thornburgh

Does it run?

The application has strict timing requirements.

- MCU supports misaligned reads and writes, but they're 4× slower.
- Compilers have discretion for alignment and memory placement.

We used `-fsanitize=alignment` with a custom runtime to **find misaligned reads and writes** but the instrumentation overhead is a challenge.

[Usability Improvements for the Undefined Behavior Sanitizer]

Finding performance bottlenecks

Predicting performance of code can be difficult.

- Placing a hot function inside a source file prevents inlining.
- Placing a cold inline function inside a header may lead to bloat.

We used IR PGO to find hot and cold functions and Remarks to guide developers.

```
$ clang … -fprofile-generate -mllvm -profile-correlate=binary -mllvm -disable-vp drivers/device.c -c -o drivers/device.c.obj

$ lldb -o 'gdb-remote localhost:1111' app.elf

(lldb) memory read -b --outfile app.profraw &__llvm_profile_start &__llvm_profile_end
```

```
+ #include <stddef.h>
+ #include <stdint.h>
+ #include <string.h>
+
+ #include <profile/InstrProfData.inc>
+
+ #pragma clang attribute push(__attribute__((no_profile_instrument_function)), apply_to = function)
+
+ typedef intptr_t IntPtrT;
+
+ enum ValueKind {
+ #define VALUE_PROF_KIND(Enumerator, Value, Descr) Enumerator = Value,
+ #include <profile/InstrProfData.inc>
+ };
+
+ struct __llvm_profile_header {
+ #define INSTR_PROF_RAW_HEADER(Type, Name, Init) Type Name;
+ #include <profile/InstrProfData.inc>
+ #undef INSTR_PROF_RAW_HEADER
+ };
+
+ struct __llvm_profile_data {
+ #define INSTR_PROF_DATA(Type, LLVMType, Name, Init) Type Name;
+ #include <profile/InstrProfData.inc>
+ };
+
+ extern uint64_t CountersStart __asm__(INSTR_PROF_QUOTE(INSTR_PROF_SECT_START(INSTR_PROF_CNTS_COMMON)));
+ extern uint64_t CountersEnd __asm__(INSTR_PROF_QUOTE(INSTR_PROF_SECT_STOP(INSTR_PROF_CNTS_COMMON)));
+
+ static inline uint64_t __llvm_profile_get_magic() { return INSTR_PROF_RAW_MAGIC_32; }
+
+ __attribute__((weak)) const uint64_t INSTR_PROF_RAW_VERSION_VAR = INSTR_PROF_RAW_VERSION;
+
+ static inline uint64_t __llvm_profile_get_version() { return INSTR_PROF_RAW_VERSION_VAR; }
+
+ static inline uint64_t __llvm_write_binary_ids(void* ignored) { return 0; }
+
+ const int INSTR_PROF_PROFILE_RUNTIME_VAR = 0;
+
+ __attribute__((section("__llvm_prf_hdr"), used)) struct __llvm_profile_header Header;
+
+ void __llvm_profile_initialize(void) {
+  const uint64_t NumData = 0;
+  const uint64_t PaddingBytesBeforeCounters = 0;
+  const uint64_t NumCounters = &CountersEnd - &CountersStart;
+  const uint64_t PaddingBytesAfterCounters = 0;
+  const uint64_t NumBitmapBytes = 0;
+  const uint64_t PaddingBytesAfterBitmapBytes = 0;
+  const uint64_t NamesSize = 0;
+  const uint64_t CountersBegin = (uint64_t)&CountersStart;
+  const uint64_t BitmapBegin = 0;
+  const uint64_t DataBegin = (uint64_t)&CountersStart;
+  const uint64_t NamesBegin = 0;
+  const uint64_t NumVTables = 0;
+  const uint64_t VNamesSize = 0;
+
+ #define INSTR_PROF_RAW_HEADER(Type, Name, Init) Header.Name = Init;
+ #include <profile/InstrProfData.inc>
+ #undef INSTR_PROF_RAW_HEADER
+ }
+
+ // These need to defined by the linker script.
+ const uintptr_t __llvm_profile_start, __llvm_profile_end;
+
+ #pragma clang attribute pop
```

```
+    /* Zero the profile in SRAM. */
+_INIT_LLVM_PROFILE:
+    la     t2, __llvm_profile_start
+    la     t3, __llvm_profile_end
+_INIT_LLVM_PROFILE_LOOP:
+    bleu   t3, t2, _MAIN_FUNC
+    sw     zero, 0(t2)
+    addi   t2, t2, 4
+    j      _INIT_LLVM_PROFILE_LOOP

+    . = ALIGN(8);
+    __llvm_prf_hdr {
+        HIDDEN(__llvm_profile_start = .);
+        KEEP(*(__llvm_prf_hdr));
+        . = ALIGN(8);
+    }
+    __llvm_prf_cnts {
+        PROVIDE_HIDDEN(__start___llvm_prf_cnts = .);
+        *(__llvm_prf_cnts);
+        PROVIDE_HIDDEN(__stop___llvm_prf_cnts = .);
+    }
+    HIDDEN(__llvm_profile_end = .);
+
+    __llvm_covdata (TYPE = SHT_PROGBITS) : {
+        *(__llvm_covdata);
+    }
+    __llvm_covnames (TYPE = SHT_PROGBITS) : {
+        *(__llvm_covnames);
+    }
```

```
$ clang … -fprofile-use=app.profdata -Rpass-missed=inline -fdiagnostics-show-hotness
-fdiagnostics-hotness-threshold=1 drivers/device.c -c -o drivers/device.c.obj
...
drivers/device.c:42:26: remark: read_register will not be inlined into device_stop because
its definition is unavailable (hotness: 1000) [-Rpass-missed=inline]
   42 |         write_register(0x78, read_register(0x78) & 0xfe);
      |
```

Debugging issues

Baremetal projects often use assembly with either missing or incorrect DWARF CFI.

- Compiler automatically generates CFI directives from high-level languages.
- In a handwritten assembly, you must write these directives manually.
- Incorrect CFI directives result in malformed frames.

We implemented MC layer checker for validating CFI directives.

[RFC] DWARF CFI validation

```
pushq   %rdi
.cfi_adjust_cfa_offset 8
.cfi_rel_offset %rdi, 0
pushq   %rsi
.cfi_adjust_cfa_offset 8
.cfi_rel_offset %rsi, 0

callq func

popq    %rsi
.cfi_adjust_cfa_offset -8
.cfi_same_value %rdi # error: loading %rdi from wrong location
popq    %rdi
# error: changed register RDI, that register RDI's unwinding rule uses, but CFI directive is missing
.cfi_adjust_cfa_offset -8
.cfi_same_value %rsi # error: loading %rsi from wrong location
```

Was the effort worth it?

- The same MCU may be used across several generations of the product.
- We came up with many ideas for new features and improvements.
- Automated tools are beneficial for other embedded projects.

How to get involved?

There's an active and growing embedded community within LLVM.

- **LLVM Embedded Toolchains Working Group** Thursday 9am PST every 4 weeks
- **LLVM Embedded Toolchains Workshop** as part of LLVM Developers' Meeting

`embedded` label in GitHub as a way to find issues related to baremetal uses.