22:09 A Tourist's Guide to Reversing Renesas M16C and the R8C, too!

by Christopher Hewitt and Niccolò Izzo

Ehilà, vicino!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the architecture of smaller devices as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with Renesas M16C or similar architectures might find it to be a useful refresher, while those of you new to the architecture will find that it really isn't as strange as you've been led to believe. If you've already reverse engineered binaries for any platform, even C-SKY CK803, you'll soon feel right at home.

We've written this guide using a device in the R8C/Tiny series for specific examples, but with minor differences it applies well enough to the R8C and M16C families as a whole. For larger Renesas parts, such as those used in engine control units and portable amateur radios, you might be better served by a different introduction. Either way, be sure to keep reading for a case study on applying power analysis and fault injection techniques to successfully recover firmware from an R8C/Tiny target with protected flash memory.

Architecture

Von Neumann
16-bit words

Registers

R0-R3: Data Registers (R0 and R1 as split 8-bit halves.)
A0-A1: Address Registers (A0 and A1 as combined 32-bit A1A0 register.)
FB: Frame Base
PC: 20-bit Program Counter
INTB: Interrupt Table (available as split 4-bit INTBH and 16-bit INTBL registers.)
USP: 16-bit User Stack Pointer
ISP: 16-bit Interrupt Stack Pointer
SB: 16-bit Static Base
FLG: 11-bit Flag register

Instructions

89 instructions, where instruction encoding is variable width
Opcode is 8-bit for the most frequently used
Opcode is 16-bit for the others

Instruction Set Basics

The first generation of R8C devices appeared in 2003 and were marketed as a cost-reduced alternative to Mitsubishi's M16C family, following the formation of Renesas as a joint venture between the semiconductor operations of Mitsubishi and Hitachi. The R8C family features the same 16-bit CISC architecture as M16C with binary level compatibility and the internal data bus reduced to 8 bits. These families are also compatible at the assembly level with the M32C family.

The instruction set is composed of 89 discrete instructions with many common instructions only requiring a single clock cycle. Instruction encoding has variable length and the opcode (8-bit for the most frequently used and 16-bit for the others) is followed by source and destination operands specified through different addressing modes.

Instruction mnemonics can be suffixed to prioritize one of the following four possible instruction formats with the assembler choosing an optimal format if one is not explicitly specified.

Generic (:G) Op-code (2 bytes), source (0-3 bytes), destination (0-3 bytes)

Quick (:Q) Op-code with immediate data (2 bytes), destination (0-2 bytes)

Short (:S) Op-code (1 byte), source (0-2 bytes), destination (0-2 bytes)

Zero (:Z) Op-code (1 byte), destination (0-2 bytes)

There are also numerous addressing modes categorized across three different types.

General Instruction Addressing Immediate, register direct, absolute, address register indirect, address register relative, SB relative, FB relative, SP relative

Special Instruction Addressing 20-bit absolute, address register relative with 20-bit displacement, 32-bit address register indirect, 32-bit register direct, control register (PC, INTB, USP, ISP, FLG) direct, PC relative

Bit Instruction Addressing Register direct, absolute, address register indirect, address register relative, SB relative, FB relative, FLG direct

Registers and Calling Convention

Be aware of different calling conventions depending on the compiler and options used. For example, the IAR C and C++ compiler for R8C and M16C supports a "normal" calling convention and a "simple" one. The normal (and default) calling convention is optimized to use registers as much as possible (with A0, R0, and R2 used as scratch registers), then defers to the stack for passing parameters. The simple calling convention, however, only passes the first parameter through R0L, R0, or R2R0 (depending on size), then defers to the stack for remaining parameters. R0 and R2R0 are also used for returning values from a function.

There are also subtle differences between Renesas' own compilers such as the NC30 compiler used for the M16C and R8C families, and the NC308 compiler used for M32C and certain M16C family parts. For example, NC30 preserves registers during function calls on the caller side, while NC308 does so on the called side.[33]

Regardless of the compiler used, stack frame manipulation is evident by the presence of `ENTER` and `EXITD` instructions to build and deallocate stack frames respectively.

Memory Map

Note that different documents have conflicts. We used a Chinese language datasheet for our figure on page 41 where things differed.[34]

Also note that this article's PoC dumps actual code from a region that isn't supposed to be valid for this specific part number, but is valid for different catalog part numbers (likely sharing the same die).

Editors Note: We have included a die photo on page 45 taken from processing a R5F21194, for anyone who wishes to perform future comparisons to other catalog part numbers.

Code Protection

The Renesas R8C/Tiny series supports a couple of different mechanisms for flash protection. Serial programmer commands to access the flash, including erasing, are completely ignored if a custom 7-byte ID code was interleaved with entries in the interrupt vector table at offsets `0x0FFDF`, `0x0FFE3`, `0x0FFEB`, `0x0FFEF`, `0x0FFF3`, `0x0FFF7`, and `0x0FFFB`. An ID code consisting of all-ones (such as when flash cells are unprogrammed from the factory) is automatically unlocked by the boot ROM, while any other value requires manual unlocking with a successful ID code comparison to re-enable flash manipulation. Parallel programmer commands to access the flash are ignored through configuration of the Option Function Select (OFS) register located at offset `0x0FFFF` by setting `ROMCP1=0` and `ROMCR=1`.[35]

Fixed Interrupt Vector Table (Flash)

`0x0FFDC`	Undefined Instruction	ID1	(`0x0FFDF`)
`0x0FFE0`	Overflow	ID2	(`0x0FFE3`)
`0x0FFE4`	`BRK` Instruction		
`0x0FFE8`	Address Match	ID3	(`0x0FFEB`)
`0x0FFEC`	Single Step	ID4	(`0x0FFEF`)
`0x0FFF0`	Osc. stop, watchdog, VM2	ID5	(`0x0FFF3`)
`0x0FFF4`	Address Break	ID6	(`0x0FFF7`)
`0x0FFF8`	Reserved	ID7	(`0x0FFFB`)
`0x0FFFC`	Reset	OFS	(`0x0FFFF`)

Finding a Target

Renesas is one of the leading suppliers of microcontrollers in the world but it's not very common to see their microcontrollers used by electronics hobbyists in western countries. Mass-produced commercial designs spanning from inexpensive toys to fault-tolerant automotive engine control units are much more likely to include Renesas parts.[36]

One low-cost and readily accessible product containing an R8C/Tiny microcontroller is the SA868 radio module with integrated power amplifier. Limitations in the module's factory firmware make it an attractive target for modifications, but this first requires unlocking access to the protected contents.

[33] The documentation is confusing here, for further see `unzip pocorgtfo22.pdf m32c90-compiler.pdf` Page M-70 and `unzip pocorgtfo22.pdf m32-compiler.pdf` Page 59.

[34] `unzip pocorgtfo22.pdf r5r0c00cn.pdf`

[35] See `unzip pocorgtfo22.pdf r8c-hardware.pdf` section "ROM Code Protect Function" (Page 250)

[36] See *Bypassing the Renesas RH850/P1M-E read protection using fault injection* by Willem Melching.

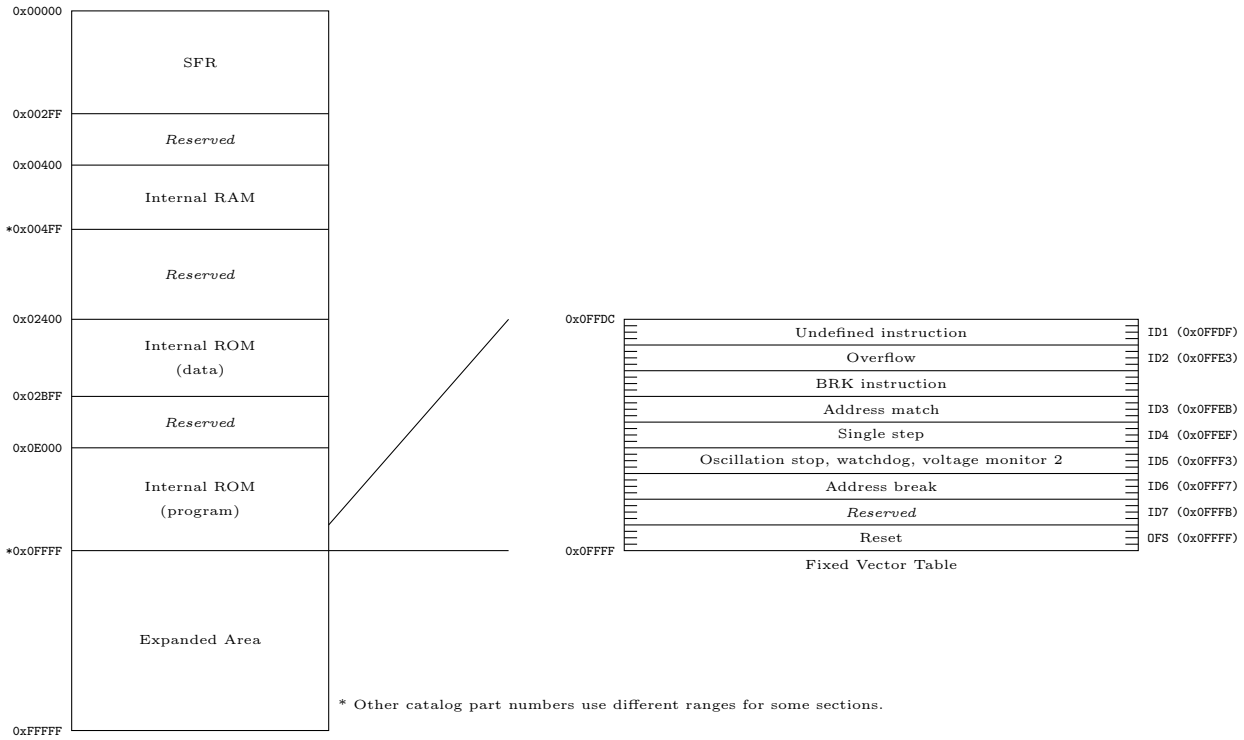
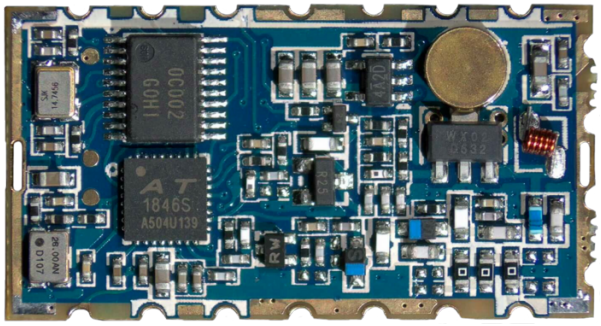


Figure 9: Memory Map Summary



Under the SA868 v1.1's metal shield sits a Renesas R5R0C002SN, an R8C/1B group compatible part that was only available to customers in Asia. A close approximation with a publicly available English language datasheet is the Renesas R5F211B2SP. The role of this microcontroller in the module is to expose a Hayes-style command set interface to control an Auctus AT1846S RF transceiver, which is the same part at the core of many low-cost amateur handheld radios including the ubiquitous UV-5R, GD-77, and MD-UV380.

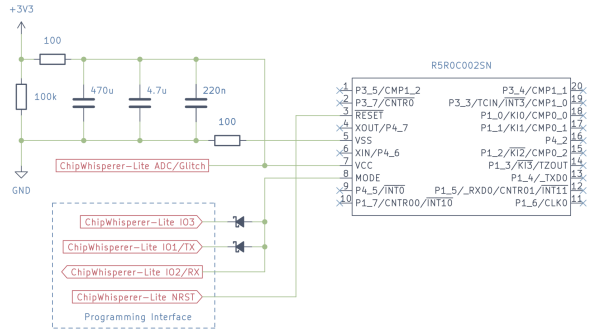
Without getting too deep into radio theory, the SA868 module has a lot more potential than it was designed for. While the module is strictly marketed for use in analog FM applications, the transceiver component is used in more sophisticated digital radios using 4-FSK modulation. Once the microcontroller's protected flash can be unlocked, it is possible to dump and patch the firmware or even replace it with a custom purpose-built one to support more useful and interesting digital voice and data protocols.[37]

[37] See Delorie 2009 page 5, `unzip pocorgtfo22.pdf renesasflash.pdf`

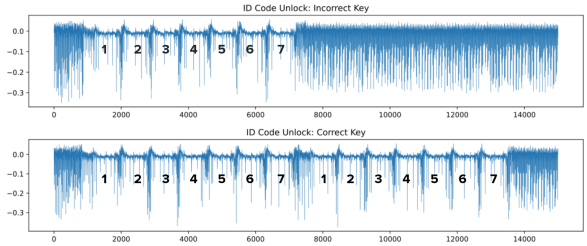
Extracting the Application

A previously published attack for a microcontroller in the M16C family described a successful timing attack against the boot ROM.[38] The authors demonstrated a measurable time delta between the last cycle of the serial programming clock (SCK) and an output pin (BUSY) asserted while servicing programmer commands that could be used to iteratively determine individual bytes of an ID code. This approach might have been viable for the R5R0C002SN since the R8C family is a close relative of the M16C family, if not for the lack of an equivalent pin indicating busy state. It is, however, still possible to demonstrate whether or not the timing attack is portable to this target by extracting the same information through power analysis of the ID code verification process.

A relationship to power consumption can be measured by removing the microcontroller from circuit and inserting a low value shunt resistor inline with the power supply. Experimentation with added capacitance or changing shunt resistor position helps establish which conditions provide the cleanest measurements. During any unsuccessful unlock attempt, voltage measurements at the supply pin expose seven evenly spaced segments, corresponding to each byte of the ID code. This observation suggests that the R5R0C002SN's boot ROM executes comparisons in constant time and is not vulnerable to the same timing attack. Brute force attempts are also discouraged by silently ignoring unlock requests after a few unsuccessful attempts. On a target with a known ID code, leakage from successful unlock attempts suggests that valid comparisons are performed twice, possibly to mitigate against power glitches.



Readers with experience in side channel analysis might be tempted to calculate Pearson correlation coefficients in order to match ID code attempts with power trace data in hopes of leaking bits or bytes from the real ID code, but the approach seemingly does not work here. Whether the result of high clock jitter, inadequate ADC resolution, or just bad luck, no apparent correlation between ID code attempts and resulting power trace can easily be identified.



Fault injection is another tool at our disposal for extracting protected flash memory contents. Long pioneered by satellite television enthusiasts exploring conditional-access modules, fault injection attacks traditionally manipulate clocks or supply voltages as a mechanism for introducing unintentional behavior to a system, such as causing instructions to be skipped or register contents to be modified. Devices like the ChipWhisperer-Lite have made these kinds of practical attacks significantly more approachable for hobbyists, but don't disregard the price point and flexibility of a simple microcontroller combined with a fast switching MOSFET to momentarily bridge a supply voltage to ground potential.

Each microcontroller in the R8C/Tiny family has an integrated ring oscillator running at roughly 8 MHz further divided by 32 for use as the system clock during boot ROM execution. This clock is not exposed externally, so clock glitching is not the most convenient approach. The high degree of jitter from this internal clock combined with the double check of the ID code observed in power analysis means that landing a voltage glitch twice during an unlock attempt, with the correct timing, might be excessively difficult. But let's consider for a moment if a successful ID code verification is even necessary prior to accessing flash programming commands. Maybe it's really only a formality intended for diligent engineers who rigidly follow the rules outlined in the hardware manual.

It is clear from the location of the fixed interrupt vector table that the ID code is stored on the last page of flash and clear from the programming guide that there is a flash page read command in the boot ROM. It's not unreasonable to at least try

[38] See *Hacking Toshiba Laptops* by Serge Bazanski and MichałKowalczyk.

repeatedly reading the final page of flash without any prior ID code verification while simultaneously sweeping glitches over the microcontroller's power supply with varied time offsets. The programming interface's serial transmit pin can even be used as a trigger to anchor glitches around the page read commands.

Some experimentation is required to find glitch pulse lengths and time offsets that don't stall the microcontroller yet still influence boot ROM behavior. Keep in mind that variations in capacitance and even temperature can easily impact results and repeatability. Since thousands of glitch attempts might be required for a single success, it's best to keep each attempt as short as possible: Skip unnecessary communication steps by directly using the boot ROM's flash programming protocol and only perform hard resets when the microcontroller is completely stalled and not responding. With a little luck, our trusty microcontroller confidently returns a full page of flash data, rather than nothing.[39]

Glitching an unknown programmed R5R0C002SN sample from AliExpress

```
[*] bootrom: VER.1.20
[*] injecting faults...
<omitted>
[?] dumped page - width: 37.890625 offset: -44.921875 ext_offset: 5420
0000FF00   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF10   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF20   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF30   ff ff ff ff  53 fb 00 00  53 fb 00 00  53 fb 00 00  ....S...S...S...
0000FF40   53 fb 00 00  53 fb 00 00  b4 fa 00 00  53 fb 00 00  S...S.......S...
0000FF50   53 fb 00 00  53 fb 00 00  aa f8 00 00  53 fb 00 00  S...S.......S...
0000FF60   53 fb 00 00  53 fb 00 00  53 fb 00 00  53 fb 00 00  S...S...S...S...
0000FF70   53 fb 00 00  1b fb 00 00  ff ff ff ff  ff ff ff ff  S...............
0000FF80   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FF90   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFA0   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFB0   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFC0   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000FFD0   ff ff ff ff  ff ff ff ff  ff ff ff ff  53 fb 00 4e  ............S..N
0000FFE0   53 fb 00 6a  53 fb 00 00  53 fb 00 46  53 fb 00 74  S..jS...S..FS..t
0000FFF0   50 fb 00 53  53 fb 00 59  53 fb 00 54  49 e0 00 f7  P..SS..YS..TI...
[!] valid idcode - 4e6a4674535954
[*] dumping entire flash...
[*] block 0 (0x0e000 - 0x0ffff):
0000E000   7b 60 5e 7c  65 3d 3f 70  7f 7d 77 2f  1b 6e 1f 17  {`^|e=?p.}w/.n..
0000E010   f4 85 0f f4  92 0f f4 ba  0f f4 d5 0f  f4 e1 0f f4  ................
0000E020   02 10 f4 ec  0f f4 0f 10  f4 2a 10 f4  56 10 f4 43  .........*..V..C
<omitted>
[*] block 1 (0x0c000 - 0x0dfff):
0000C000   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000C010   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
0000C020   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
<omitted>
[*] block a (0x02400 - 0x027ff):
00002400   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002410   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002420   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
<omitted>
[*] block b (0x02800 - 0x02bff):
00002800   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002810   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
00002820   ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff  ................
<omitted>
[*] done
```

[39]See the Jupyter notebook `unzip pocorgtfo22.pdf r8c-glitch.ipynb`

Successful glitches don't always return meaningful data, but ID codes can be assembled from their expected offsets in the page, then verified through an unlock attempt. Eventually you'll find a match and, once successfully unlocked, the entire flash memory can be dumped or erased and reprogrammed. If it looks like some data is missing, try reading additional flash pages that aren't officially supported by the part number since there's a good chance the same internal die is used by several part numbers.[40]

Proceeding with Analysis

Once a firmware image is safely recovered, you'll almost certainly want to inspect how it works. M16C isn't as esoteric as it might seem and there are actually a few different options for analysis. IDA Pro provides a disassembler for the architecture and Binary Ninja has some support by way of a third-party plugin.[41] If you're averse to commercial software don't forget about GNU Binutils which supports M16C and R8C through the m32c-elf target.

```
;; Recovered firmware through m32-elf-
    objdump
0000fcca <.data+0xd8ca>:
  fcca: eb 40 32 06   ldc #1586,isp
  fcce: c7 02 0a 00   mov.b:s #2,0xa
  fcd2: b7 04 00      mov.b:z #0,0x4
  fcd5: b7 0a 00      mov.b:z #0,0xa
  fcd8: eb 30 80 00   ldc #128,flg
  fcdc: eb 50 b2 05   ldc #1458,sp
  fce0: eb 60 00 04   ldc #1024,sb
  fce4: eb 20 00 00   ldc #0,intbh
  fce8: eb 10 dc fe   ldc #-292,intbl
  fcec: fd 64 fc 00   jsr.a 0xfc64
  fcf0: 75 cf ba 04   mov.w:g #1586,0x4ba
  fcf4: 32 06
  fcf6: 75 cf bc 04   mov.w:g #128,0x4bc
  fcfa: 80 00
  fcfc: d9 0f be 04   mov.w:q #0,0x4be
  fd00: fd a2 fa 00   jsr.a 0xfaa2
  fd04: eb 70 00 00   ldc #0,fb
  fd08: fd 7a f5 00   jsr.a 0xf57a
  fd0c: f5 03 00      jsr.w 0xfd10
  fd0f: fb            reit
  fd10: d9 10         mov.w:q #1,r0
  fd12: 6e fd         jne 0xfd10
  fd14: f3            rts
```

Alternatively, a Ghidra third party plugin created recently is capable of disassembling most instructions and may help jumpstart new reverse engineering projects through integration with Ghidra's processor independent decompiler.[42]

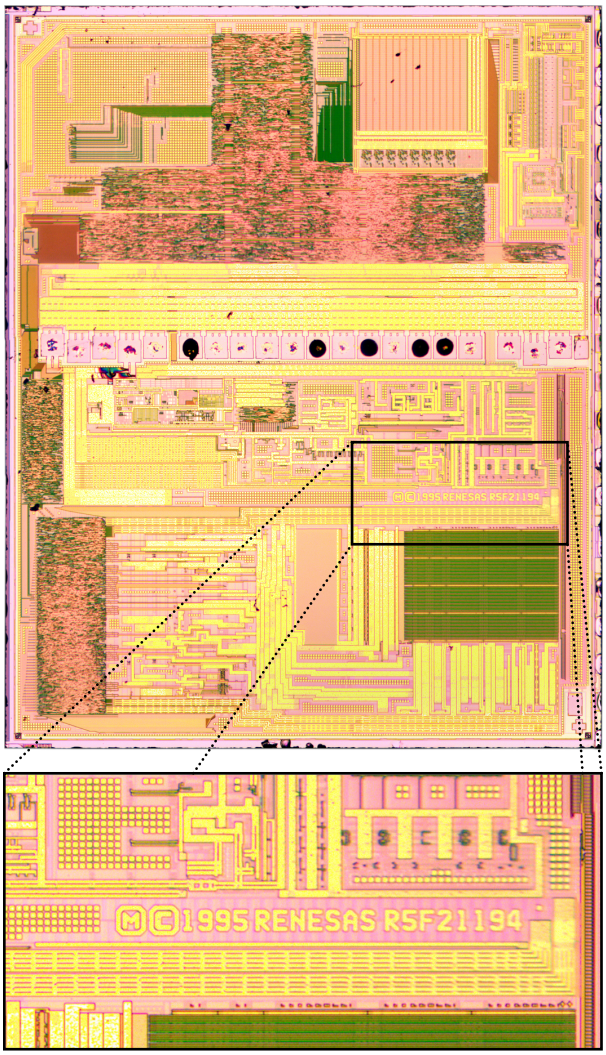
```
1  ;-----------------------------------
   ; after reset, this program will start
3  ;-----------------------------------
   ldc     #((topof istack)+(sizeof istack)),
       isp ;set istack pointer
5  mov.b   #02h,0ah
   mov.b   #00h,04h    ;set processor mode
7  mov.b   #00h,0ah
   .if __STACKSIZE__ != 0
9    ldc     #0080h,flg
     ; set stack pointer
11   ldc     #((topof stack)+(sizeof stack)),sp
   .else
13   ldc     #0000h,flg
   .endif
15   ldc     #__SB__,sb  ;set sb register

17   ; If the destination is INTBL or INTBH,
     ; make sure that bytes are sent in order
19   ldc     #((topof vector)>>16)&0FFFFh,INTBH
     ldc     #(topof vector)&0FFFFh,INTBL
21
   <omitted>
23
   ;===================================
25 ; Call main() function
   ;-----------------------------------
27 ldc     #0h,fb; for debugger

29 ; Remove the comment when you use
   ; global class object
31 ; Sections C$INIT will be generated
   ;   .glb      __CALL_INIT
33 ;   .call     __CALL_INIT,G
   ;   jsr.a     __CALL_INIT
35
   .glb     _main
37 .call      _main,G
   jsr.a      _main
```

Whichever option you pick, be sure to identify the correct entrypoint for the binary by referencing the reset vector in the fixed interrupt vector table at the very end of flash memory.

[40] See *The Secret of R8C/M11A and M12A* at the RVF/RC45 blog.
[41] `git clone https://github.com/whitequark/binja-m16c`
[42] `git clone https://github.com/silverchris/m16c`



Die photograph by Travis Goodspeed

An Unexpected Outcome

News of the effort to repurpose the SA868 with custom firmware eventually found its way to the company producing the radio modules, NiceRF Wireless Technology. An amateur radio enthusiast in China, Amo Xu, made a compelling case for the company to release an intentionally user-programmable variant of the module. Shortly after their discussion, the company began offering the SA868S Open Edition module. This variant is erased after quality assurance, guaranteeing the module is unlocked for reprogramming.

The new SA868S, version 2.0, is notably different from previous versions in that the microcontroller has been replaced with one from a different Renesas microcontroller architecture family, RL78, which is not vulnerable to the attack presented in this article. The RL78 family has, however, been explored in some detail in the context of the PlayStation 4 gaming console and several useful tools already exist for working with that platform, including an implementation of the debugging protocol and third party plugins for IDA Pro and Ghidra.[43]

While not as common as it should be, hardware reverse engineering occasionally leads to mutually beneficial outcomes with a manufacturer. A deep dive into an unfamiliar microcontroller architecture to improve a product's capabilities led to a manufacturer removing obstacles for experimentation. The availability of the SA868S Open Edition paves the way for user customizable firmware and has already motivated the creation of a free and open source alternative firmware granting complete control of all registers in the underlying transceiver chipset, enabling use of digital protocols such as APRS and M17.[44]

We hope that you've enjoyed this little guide to Renesas M16C and R8C, and that you'll keep it handy when reverse engineering firmware from those platforms.

[43] See *PS4 Aux Hax 2: Syscon* at Fail0verflow.
[44] `git clone https://github.com/OpenRTX/sa8x8-fw.git`