

## Coresight ETM трассировка на dragonboard 410c

1. Компиляция ядра linux с поддержкой Coresight.
2. Создание bootable SD карты, прошивка boot раздела новым ядром.
3. Компиляция perf с OpenCSD.
4. Трассировка на плате и отображение результатов в IDA.

### Компиляция ядра linux с поддержкой Coresight

Для компиляции ядра, perf'a и OpenCSD понадобится aarch64 тулчейн. В данном случае будет использоваться linaro тулчейн:

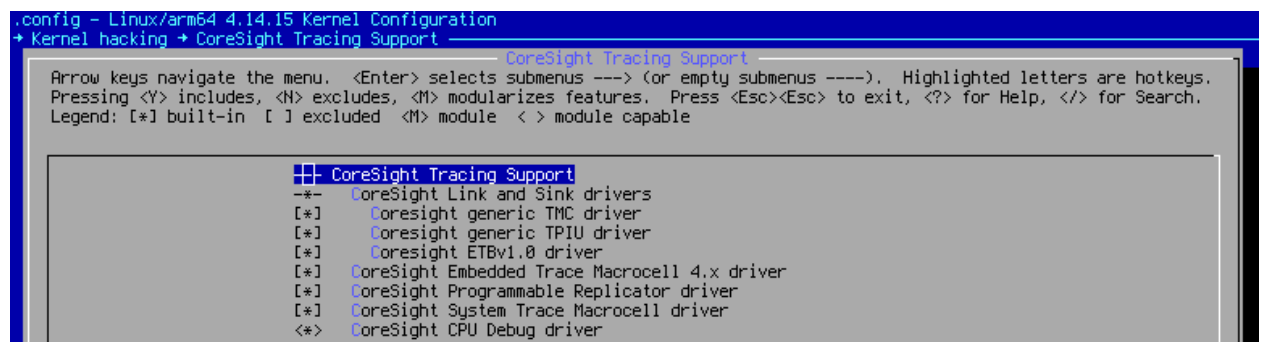
```
mkdir build
cd build
mkdir toolchain
wget releases.linaro.org/components/toolchain/binaries/latest/aarch64-linux-gnu/gcc-*-x86_64_aarch64-linux-gnu.tar.xz
tar -xvf gcc-*-x86_64_aarch64-linux-gnu.tar.xz -C ../toolchain
```

Скачивание исходников linux ядра для dragonboard 410c:

```
git clone http://git.linaro.org/landing-teams/working/qualcomm/kernel.git
cd kernel
```

Конфигурация ядра и компиляция:

```
export ARCH=arm64
export CROSS_COMPILE=../toolchain/bin/aarch64-linux-gnu-
make defconfig distro.config
make menuconfig
```



```
make -j$(nproc) Image.gz dtbs
make -j$(nproc) modules
```

В итоге будут скомпилированы сжатое ядро, dtb и модули.  
Теперь нужно добавить dtb к ядру:

```
cat arch/$ARCH/boot/Image.gz arch/$ARCH/boot/dts/qcom/apq8016-sbc.dtb  
> Image.gz+dtb
```

## Создание bootable SD карты, прошивка boot раздела новым ядром.

Для создания boot образа требуется ramdisk, но по факту ядром используется другой ramdisk (который обычно записан в /dev/\*\*\*)9), поэтому состав этого не имеет значения:

```
echo "not a ramdisk" > initrd.img
```

Теперь, используя ядро ramdisk, нужно создать boot раздел и записать его на карту. Я поступил следующим образом: скачал готовый образ bootable SD карты, а затем заменил в нем boot раздел:

```
wget  
https://releases.linaro.org/96boards/dragonboard410c/linaro/debian/latest/dragonboard-410c-sdcard-developer-buster-*.zip  
unzip dragonboard-410c-sdcard-developer-buster-*.zip  
cd dragonboard-410c-sdcard-developer-buster-*
```

Запись образа на SD карту (где /dev/sdb – SD карта):

```
dd if=dragonboard-410c-sdcard-developer-buster-*.img of=/dev/sdb  
status=progress
```

В данном образе есть девять разделов (/dev/sdb[1-9]), rootfs обычно находится на /dev/sdb9, а boot на /dev/sdb7. Лучше всего будет посмотреть параметры оригинального boot'a и собрать новый с аналогичными:

```
dd if=/dev/sdb7 of=img status=progress  
file img
```

```
> sudo dd if=/dev/sdb7 of=img status=progress  
58704384 bytes (59 MB, 56 MiB) copied, 3 s, 19.6 MB/s  
131072+0 records in  
131072+0 records out  
67108864 bytes (67 MB, 64 MiB) copied, 3.75234 s, 17.9 MB/s  
> file img  
img: Android bootimg, kernel, ramdisk, page size: 2048, cmdline (root=/dev/mmcblk1p9 rw rootwait console=ttyMSM0,115200n8)
```

Для создания образа используется программа mkbootimg:

```
mkbootimg --kernel Image.gz+dtb --ramdisk initrd.img --output boot-  
db410c.img --pagesize 2048 --base 0x80000000 --cmdline  
"root=/dev/mmcblk1p9 rw rootwait console=ttyMSM0,115200n8"
```

Запись нового boot'a:

```
dd if=boot-db410c.img of=/dev/sdb7 status=progress
```

Для того, чтобы загрузиться с SD карты, нужно установить свитч S6-2 в положение ON, а все остальные в положение OFF.

## Компиляция perf с OpenCSD

OpenCSD – библиотека-декодер трассы Coresight'a.

Компиляция OpenCSD:

```
git clone -b master https://github.com/Linaro/OpenCSD.git
cd OpenCSD/decoder/build/linux
make LINUX64=1 DEBUG=0
```

Компиляция perf'a (P.S. Для хоста я использовал perf версии 4.17, а для таргета 4.8):

```
git clone -b perf-opencsd-4.8 https://github.com/Linaro/perf-opencsd
export CSTRACE_PATH=path/to/OpenCSD/decoder
```

Компиляция для хоста:

```
make
```

Также для того, чтобы иметь возможность декодировать трассу, понадобится установить пакет libpython/libpython-dev до компиляции (для хоста).

Компиляция для таргета:

```
make ARCH=arm64 CROSS_COMPILE=path/to/aarch64/toolchain/bin/aarch64-
linux-gnu-
```

Для статической компиляции нужно дописать в Makefile.perf строчку LDFLAGS=-static

В зависимости от версии perf'a и тулчейна, возможно, понадобится скомпилировать под ARM64 дополнительные библиотеки и добавить их в тулчейн, например, libelf.

## Трассировка на плате и отображение результатов в IDA

При правильной компиляции и установки ядра появятся следующие устройства:

```
> ls /sys/bus/coresight/devices
```

```
820000.tpiu 821000.funnel 824000.replicator 825000.etf 826000.etr
841000.funnel 85c000.etm 85d000.etm 85e000.etm 85f000.etm
```

dragonboard 410c поддерживает ETM трассировку. Трассировка выполняется следующей командой:

```
perf record -e cs_etm/@825000.etf/u --per-thread ./elf_name
```

где cs\_etm – тип события

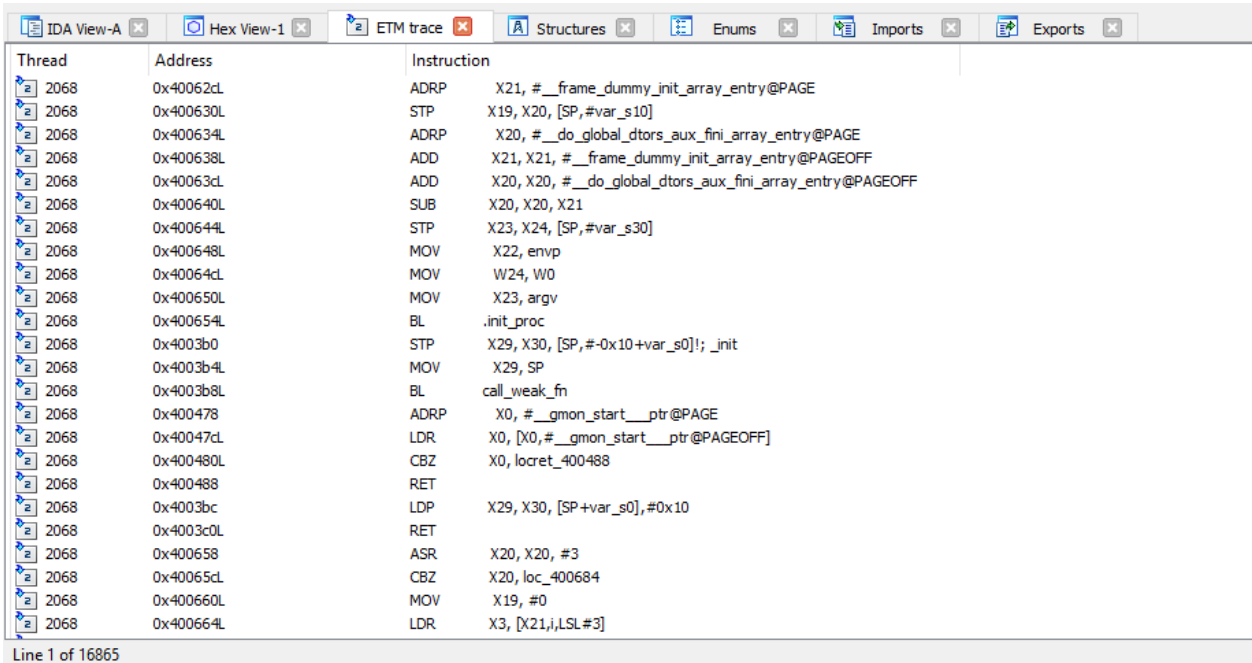
825000.etf – имя устройства

[illegible]

Декодированная трасса содержит записи о всех переходах, нарушающих последовательное исполнение программы.

Имя трассируемый образ и декодированную трассу, можно отобразить все в IDA с помощью плагина.

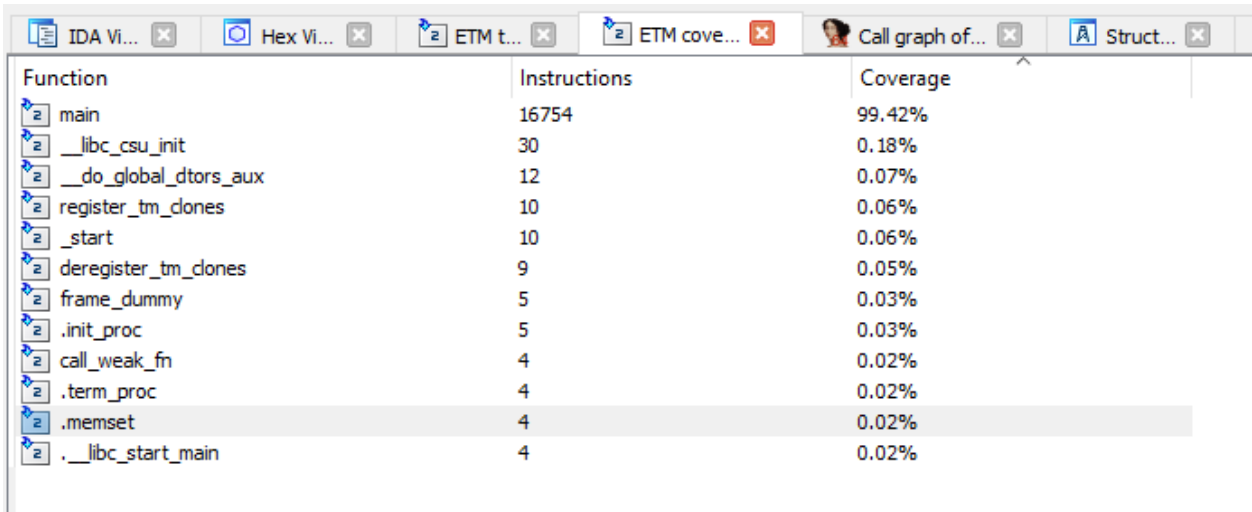
Трасса:



Thread	Address	Instruction
2068	0x40062cL	ADRP X21, #__frame_dummy_init_array_entry@PAGE
2068	0x400630L	STP X19, X20, [SP, #var_s10]
2068	0x400634L	ADRP X20, #__do_global_dtors_aux_fini_array_entry@PAGE
2068	0x400638L	ADD X21, X21, #__frame_dummy_init_array_entry@PAGEOFF
2068	0x40063cL	ADD X20, X20, #__do_global_dtors_aux_fini_array_entry@PAGEOFF
2068	0x400640L	SUB X20, X20, X21
2068	0x400644L	STP X23, X24, [SP, #var_s30]
2068	0x400648L	MOV X22, envp
2068	0x40064cL	MOV W24, W0
2068	0x400650L	MOV X23, argv
2068	0x400654L	BL .init_proc
2068	0x4003b0	STP X29, X30, [SP, #-0x10+var_s0]!; .init
2068	0x4003b4L	MOV X29, SP
2068	0x4003b8L	BL call_weak_fn
2068	0x400478	ADRP X0, #_gmon_start_ptr@PAGE
2068	0x40047cL	LDR X0, [X0, #_gmon_start_ptr@PAGEOFF]
2068	0x400480L	CBZ X0, locret_400488
2068	0x400488	RET
2068	0x4003bc	LDP X29, X30, [SP+var_s0], #0x10
2068	0x4003c0L	RET
2068	0x400658	ASR X20, X20, #3
2068	0x40065cL	CBZ X20, loc_400684
2068	0x400660L	MOV X19, #0
2068	0x400664L	LDR X3, [X21,i,LSL#3]

Line 1 of 16865

Покрытие кода:



Function	Instructions	Coverage
main	16754	99.42%
__libc_csu_init	30	0.18%
__do_global_dtors_aux	12	0.07%
register_tm_clones	10	0.06%
_start	10	0.06%
deregister_tm_clones	9	0.05%
frame_dummy	5	0.03%
.init_proc	5	0.03%
call_weak_fn	4	0.02%
.term_proc	4	0.02%
.memset	4	0.02%
__libc_start_main	4	0.02%

Последовательность вызовов функций:

