



Intel Management Engine Deep Dive

Peter Bosch

About me



Peter Bosch

- CS / Astronomy student at Leiden University
- Email : me@pbx.sh
- Twitter: @peterbjornx
- GitHub: peterbjornx
- <https://pbx.sh/>



About me



Previous work:

- CVE-2019-11098: Intel Boot Guard bypass through TOCTOU attack on the SPI bus (Co-discovered by @qrs)

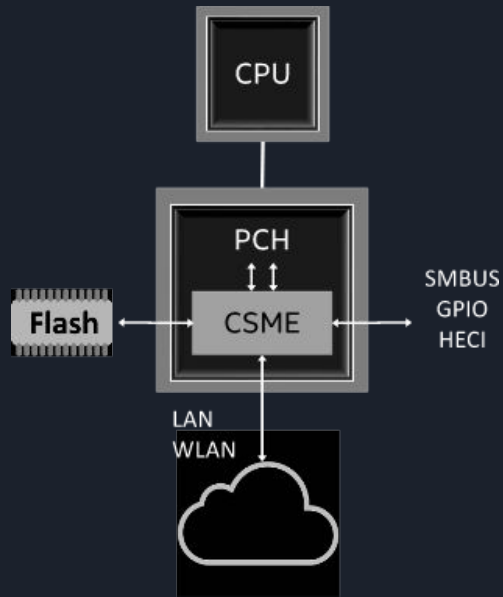


Outline

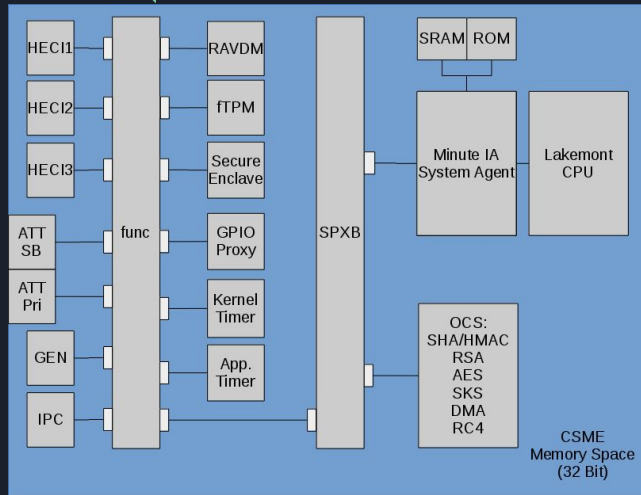
1. Introduction to the Management Engine Operating System
2. The Management Engine as part of the boot process
3. Possibilities for opening up development and security research on the ME

Additional materials will be uploaded to <https://pbx.sh/> in the days following the talk.

About the ME



About ME



- Full-featured embedded system within the PCH
 - 80486-derived core
 - 1.5MB SRAM
 - 128K mask ROM
 - Hardware cryptographic engine
 - Multiple sets of fuses.
 - Bus bridges to PCH global fabric
 - Access to host DRAM
 - Access to Ethernet, WLAN
- Responsible for
 - System bringup
 - Manageability
 - KVM
 - Security / DRM
 - Boot Guard
 - fTPM
 - Secure enclave

About ME



- Only runs Intel signed firmware
- Sophisticated , custom OS
 - Stored mostly in SPI flash
 - Microkernel
 - Higher level code largely from MINIX
 - Custom filesystems
 - Custom binary format
- Configurable
 - Factory programmed fuses
 - Field programmable fuses
 - SPI Flash
- Extensible
 - Native modules
 - JVM (DAL)



Scope of this talk

Intel ME version 11 , specifically looking at version 11.0.0.1205

Platforms:

- Sunrise Point (Core 6th, 7th generation SoC, Intel 100, 200 series chipset)
- Lewisburg (Intel C62x chipsets)



Disclaimer

- I am in no way affiliated with Intel Corporation.
- All information presented here was obtained from public documentation or by reverse engineering firmware extracted from hardware found “in the wild”.
- Because this presentation covers a very broad and scarcely documented subject I can not guarantee accuracy of the contents.
- The goal of this talk is to introduce people to the subject and introduce new tools, as such parts of the background information have been discovered/published by other researchers.



Working with ME firmware images

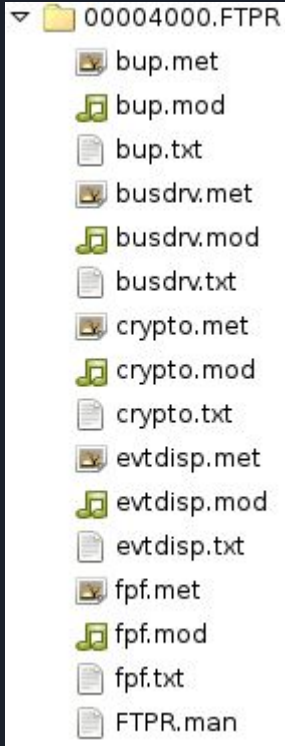
- File format already extensively documented by Positive Technologies team (Mark Ermolov, Dmitry Sklyarov, Maxim Goryachy)
 - <https://www.blackhat.com/docs/eu-17/materials/eu-17-Sklyarov-Intel-ME-Flash-File-System-Explained-wp.pdf>
 - https://www.troopers.de/downloads/troopers17/TR17_ME11_Static.pdf
- Ready to use tools are available
 - Unpacks code, metadata:
 - [ptresearch/unME11: Intel ME 11.x Firmware Images Unpacker](#)
 - Unpacks code, metadata, config archives, config FS
 - [platomav/MEAnalyzer: Intel Engine Firmware Analysis Tool](#)
 - Unpacks/Repacks config archives
 - [peterbiornx/meimagetool: Image manipulation tools for the Management Engine firmware](#)
- Flash Image Tool contains XML descriptions of formats that can be retrieved using binwalk

Understanding the ME: Firmware Partitions

IDX	NAME	START	SIZE	TYPE
1:	[FTPR]	1000:A7000		Code
2:	[FTUP]	110000:AC000		Code
3:	[DLMP]	0:0		Code
4:	[PSVN]	E00:200		Data
5:	[IVBP]	10C000:4000		Data
6:	[MFS]	A8000:64000		Data
7:	[NFTP]	110000:AC000		Code
8:	[ROMB]	0:0		Code
9:	[FLOG]	1BC000:1000		Data
10:	[UTOK]	1BD000:2000		Data
11:	[ISHC]	0:0		Code

- FTPR/NFTP
 - Read only filesystem
 - Contains firmware code
 - Mounted on /bin
 - FTPR is recovery/normal boot partition.
 - NFTP binaries not used during recovery.
- MFS
 - Read/write filesystem
 - Contains configuration data, state
 - Initialized by Flash Image Tool
- FLOG -> Crash log
- UTOK -> Unlock Token
- ROMB -> ROM Bypass

Understanding the ME: Code partitions



- Code Partitions contain modules
 - .mod files are loadable data/code (extension added by unME11)
 - .met files are metadata (Converted by unME11 to .txt)
- and the partition manifest
 - Filename : <partition>.man
 - Same general format as the metadata files, but has header prepended.



Understanding the ME: Metadata

- Type-Length-Value store, entries are called extensions
- Converted to human readable form by unME11
- Extensions:
 - Data module info
 - Code module info
 - Shared Library info
 - Process info
 - MMIO ranges
 - Device file definitions
 - ...and more...

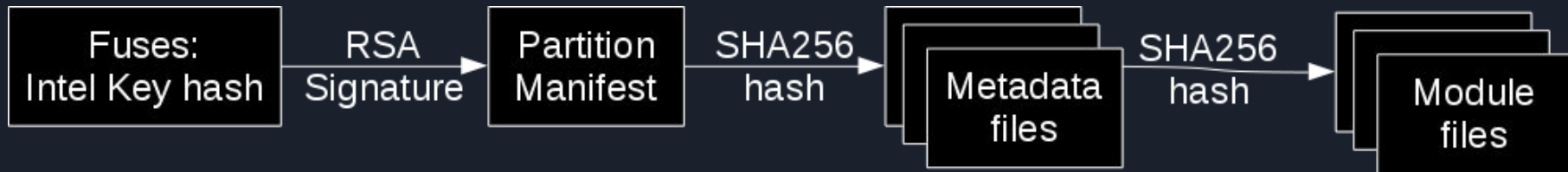
```
typedef struct {  
    uint32_t tag;  
    uint32_t length;  
} met_ext_t;
```

See also:

<https://github.com/peterbjornx/meloder/blob/master/include/manifest.h>



Code verification chain



ERRATUM (Added after talk): Intel Key hashes are in boot ROM, not fuses. Fuses only select which keys are actually trusted.

Analysing a simple module

- The module file itself is a flat binary
- Metadata contains memory space info
 - Base load address is easy to find, and usually does not vary across modules within a single firmware version

loc_3246E:

```
mov     dword ptr ds:36234h, 36228h
mov     dword ptr ds:36228h, 0
mov     dword ptr ds:3622Ch, 0
```

loc_3248C:

```
call    near ptr 1000h
push    218h
push    36578h
call    near ptr 9776h
push    36578h
```

Missing code section?

Missing code section?

Missing data section?



ME shared libraries

- No dynamic linker!
- Jump vector table with fixed address entry points
- Normal SysV i386 calling convention

```
9037    .jmp      sub 102A7
903C    ; j_toupper.
9041    ; j_toupper.
9046    ; j__exit.
904B    ; j_memcmp.
9050    ; j_memmove.
9055    ; j_memcpy_s.
905A    ; j_memmove_s.
905F    ; j_strcpy_s.
9064    ; j_strncpy_s.
9069    ; j_strcat_s.
906E    ; j_strncat_s.
9073    ; j_strlen_s.
9078    ; j_malloc.
907D    ; j_calloc.
9082    ; j_free.
```




ME shared libraries

- syslib.mod
 - Entry point addresses vary per firmware version
 - Contains
 - hosted libc
 - libsrv
 - libheci
 - crypto library
 - ...
- mask ROM
 - Entry point addresses fixed per chipset family (eg. SPT/LBG).
 - Base: 0x0000_1000
 - Contains
 - freestanding libc
 - MMIO
 - miscellaneous utility routines

Analysing a simple module

- The module file itself is a flat binary
- Metadata contains memory space info
 - Base load address is easy to find, and usually does not vary across modules within a single firmware version

loc_3246E:

```
mov     dword ptr ds:36234h, 36228h
mov     dword ptr ds:36228h, 0
mov     dword ptr ds:3622Ch, 0
```

loc_3248C:

```
call    near ptr 1000h
push    218h
push    36578h
call    near ptr 9776h
push    36578h
```

romlib_func_1000?

syslib: clear_ctx

Missing data section?

	entrypoint:	ME module entrypoint	MINIX 3 crtso	
0002D000	pop	ebx		
0002D000	pop	eax		
0002D001	push	eax		
0002D002	push	ebx		
0002D003	call	sub_32447		
0002D004	pop	eax		
0002D009	pop	eax		
0002D00A				
0002D00B				
0002D00B	crtso_:		crtso:	
0002D00B	xor	ebp, ebp	xor	ebp, ebp
0002D00D	mov	eax, [esp]	mov	eax, (esp)
0002D010	lea	edx, [esp+4]	lea	edx, 4(esp)
0002D014	lea	ecx, [esp+eax*4+8]	lea	ecx, 8(esp)(eax*4)
0002D018	mov	ebx, 3621Ch	mov	ebx, _environ
0002D01D	cmp	ebx, 36220h	cmp	ebx, __edata
0002D023	jnb	short loc_2D038	jae	0f
0002D025	test	bl, 3	testb	bl, 3
0002D028	jnz	short loc_2D038	jnz	0f
0002D02A	cmp	dword ptr [ebx], 53535353h	cmp	(ebx), 0x53535353
0002D030	jnz	short loc_2D038	jne	0f
0002D032	mov	ds:36218h, ebx	mov	(__penviron), ebx
0002D038				
0002D038	loc_2D038:			
0002D038	mov	ebx, ds:36218h	mov	ebx, (__penviron)
0002D03E	mov	[ebx], ecx	mov	(ebx), ecx
0002D040	push	ecx	push	ecx
0002D041	push	edx	push	edx
0002D042	push	eax	push	eax
0002D043	smsw	ax	smsw	ax
0002D047	test	al, 4	testb	al, 0x4
0002D049	setz	byte ptr ds:36224h	setz	(__fpu_present)
0002D050	call	sub_2D371	call	_main
0002D055	push	eax	push	eax
0002D056	call	near ptr 9046h	call	_exit
0002D05B	hlt			
0002D05C	;			
0002D05C	retn			

Data sections

- Initialized data is appended to .rodata
- Before crtso even runs it is copied over to ".bss"
- Addresses can be inferred from code or metadata.

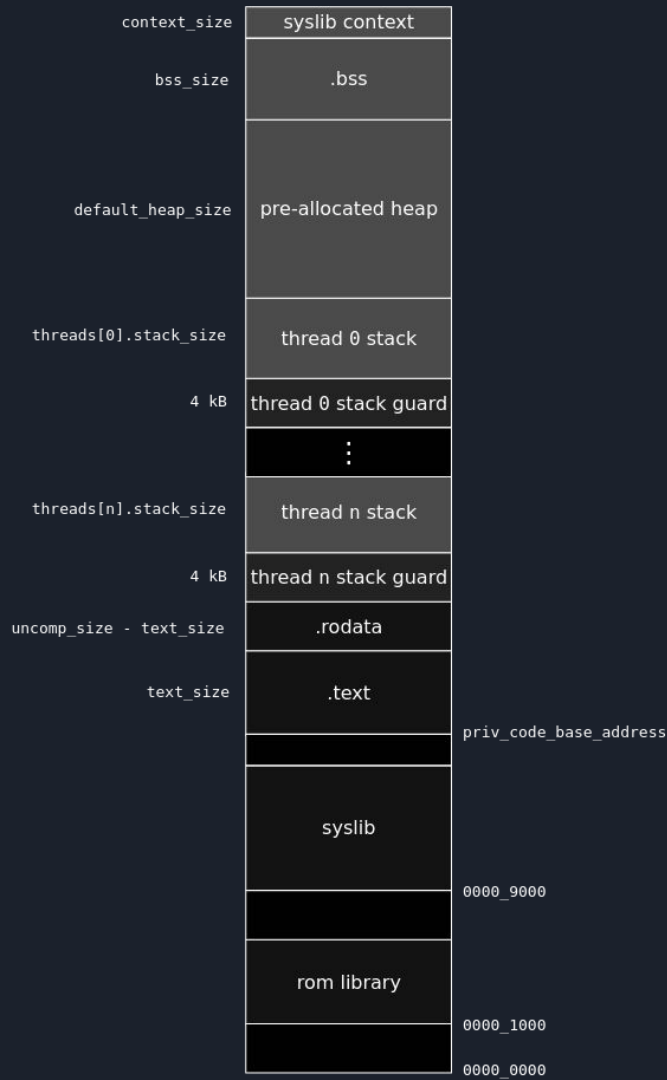
```
00032428 sub_32428      proc near
00032428      push      ebp
00032429      mov       edx, offset dword_33000
0003242E      mov       ebp, esp
00032430      mov       eax, 36200h
00032435
00032435 loc_32435:
00032435      cmp       eax, 36220h
0003243A      jz         short loc_32445
0003243C      inc       eax
0003243D      mov       cl, [edx]
0003243F      inc       edx
00032440      mov       [eax-1], cl
00032443      jmp       short loc_32435
00032445 ; -----
00032445
00032445 loc_32445:
00032445      pop       ebp
00032446      retn
00032446 sub_32428      endp
```


Data sections

- Processes use flat 32-bit memory model
- Base address and various area sizes are stored in metadata.
- System library state resides in program-specified area.

For a minimal working implementation of this, see:

GitHub meloader repo: [user/loader/map.c](#)





Familiar APIs

ME provides many familiar POSIX APIs:

- **libc:**
 - `read()`, `write()`, `close()`, `open()`, `fcntl()`, `ioctl()`, `select()`
 - `chdir()`, `stat()`,
 - nearly everything in `string.h`
 - `exit()`
 - `malloc()`, `free()`, `calloc()`
- **pthread**s
 - `pthread_create()`, ...
 - `pthread_mutex_{lock,unlock}`
 - ...

Example driver main() function

```
cookie = cookie_value;
memset(&funcs, 0, 40);
funcs.open = VdmDrvOpenCallback;
...
funcs.select = VdmDrvSelectCallback;
funcs.var_1C = sub_2D062;
sven_init(7);
...
event_fd = open("/dev/events", 2);
...
if ( srv_init(&g_srv_ctx, 24, &funcs, 0, 5) ) {
    sven_catalog0i(2, 0x320033);
    goto LABEL_7;
}
return srv_task(&g_srv_ctx);
```

libsrv callbacks

POSIX file IO

libsrv init

SVEN tracing

libsrv main loop



Trace output: SVEN

- *Intel Software Visible Event Nexus*
- Trace print format strings are replaced by message IDs
 - These are reasonably stable for given platform/major version.
- Output goes to Trace Hub
 - Can be read back from host using memory trace
 - Can be read over debug interface EVEN WITHOUT UNLOCK
- Intel System Studio used to contain decoder and dictionary
 - GREEN dictionary is not very useful, only has a handful of messages
 - System Studio 2018 beta had a nearly complete one for LBG

```
void sven_catalog<n>( int level, int id, ... );  
void sven_printf( const char *fmt, ... );  
void sven_printf_l( int level, const char *fmt, ... );  
void sven_init( int mmio );
```




ME driver overview: device files

- Unix-style special files under /dev
 - One major number per module
 - Major, minor numbers and names specified in metadata
 - Drivers implement read(), write(), open(), close(), ioctl() for device files
 - Not just for device drivers, used for all high-level services.
- syslib contains convenient framework for implementing this
 - Implementation details hidden, just provide callbacks

```
Ext#9 SpecialFileProducer[3]: major_number=0x0018
```

```
1: vdm_gde    access_mode:0660, user_id:0x0074 group_id:0x0037 minor_number:00
2: vdm_pavp   access_mode:0660, user_id:0x0074 group_id:0x018B minor_number:01
3: vdm_rosm   access_mode:0660, user_id:0x0074 group_id:0x018C minor_number:02
```




ME driver overview: libsrv

Framework for drivers, allows driver to only implement simple callbacks.

- `open()`, `close()` implementations return their status,
- `read()`, `write()`, `ioctl()` call a reply function with their result data and status.
- libsrv also allows handling hardware interrupts and power state changes.

```
typedef int (*ioctl_cb)(int info, int fd, int gtid, int request, void *par);
typedef int (*open_cb)( srvctx_t *ctx, int minor, int gtid, int *p_fd, void *ok);
typedef int (*close_cb)( srvctx_t *ctx, int fd);

int srv_task( srvctx_t *ctx )
int srv_init( srvctx_t *ctx, int major, srvfuncs_t *ops,
              srvcli_t *clients, int maxclients )
int srv_ioctl_reply( srvctx_t *ctx, int fd, int gtid, int ?, int status, void *p
)
```


Accessing hardware

```
void sub_4EA5E()
```

```
{
```

```
    write_seg_32(0xDF, 88, 0xF80);
```

```
    write_seg_32(0xDF, 92, 0xFED40080);
```

```
    write_seg_32(0xDF, 96, 0);
```

```
    write_seg_32(0xDF, 100, 0xF80);
```

```
    write_seg_32(0xDF, 104, 0xFED40080);
```

```
    write_seg_32(0xDF, 108, 0);
```

```
}
```

What's this?

```
. Ext#8 MmioRanges[41]:
```

```
...
```

```
CF    base:F00A0000, size:00006000, flags:00000003  RAVDM
```

```
D7    base:F5050000, size:00010000, flags:00000003  ICC_CONTROLLER
```

```
DF    base:F0090000, size:00006000, flags:00000003  FTPM
```



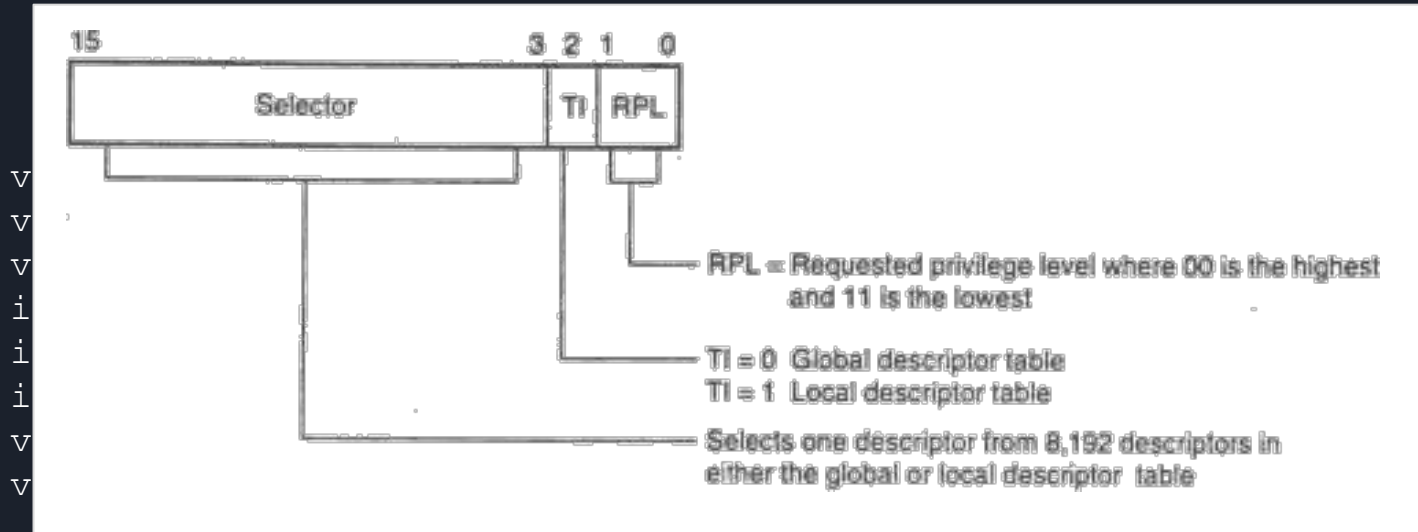

Accessing hardware

- MMIOs are accessed through ROM library functions
- The MMIO ranges are defined in the manifest
- `mmio = (mmio_list_index * 8) | 7`
 - Seem familiar to anyone?

```
. Ext#8 MmioRanges[41]:  
...  
CF  base:F00A0000, size:00006000, flags:00000003 RAVDM  
D7  base:F5050000, size:00010000, flags:00000003 ICC_CONTROLLER  
DF  base:F0090000, size:00006000, flags:00000003 FTPM
```


Accessing hardware

- MMIOs are accessed through ROM library functions
- The MMIO ranges are defined in the manifest
- `mmio = (mmio_list_index * 8) | 7`
 - Seem familiar to anyone?



```
count);  
);
```


Accessing hardware

```
void sub_4EA5E()
```

```
{
```

```
    write_seg_32(0xDF, 88, 0xF80);  
    write_seg_32(0xDF, 92, 0xFED40080);  
    write_seg_32(0xDF, 96, 0);  
    write_seg_32(0xDF, 100, 0xF80);  
    write_seg_32(0xDF, 104, 0xFED40080);  
    write_seg_32(0xDF, 108, 0);
```

```
}
```

What's this?

```
void write_seg_32(int mmio, int offset, int value);  
void write_seg_16(int mmio, int offset, short value);  
void write_seg_8 (int mmio, int offset, char value);  
int  read_seg_32 (int mmio, int offset);  
int  read_seg_16 (int mmio, int offset);  
int  read_seg_8  (int mmio, int offset);  
void write_seg   (int mmio, int offset, const void *buffer, int  
count);
```




The levels below the POSIX-like environment

- Kernel implements IPC primitives and MMIO access
 - Message passing
 - Memory grants
 - DMA buffers
 - MMIO mappings
 - Memory protection
- VFS/Process Manager server implement POSIX calls
 - Accessed through kernel IPC
- Drivers and high level servers implement device files



Message Passing: Basics

- Used to implement server-based “syscalls” and other low level IPC
- Not often directly used by modules
- Mostly MINIX derived
- Fixed message header structure, variable body.
- `int ipc_sendrec(int who, syscall_msg *msg)`
 - Sends a message, and immediately does a blocking receive
 - Used for server calls
- `int ipc_send (int who, syscall_msg *msg)`
 - Sends a message, blocks until it is received
- `int ipc_notify (int who)`
 - Asynchronously sends a notify event to a process



Memory Grants

- Also MINIX derived(safecopies), relatively new feature in MINIX.
- Dynamic resource and memory access control
- Allows a process to register a global name for a memory buffer or MMIO range
- Referenced as (gtid, id) pair
 - Memory grant ID is not global, but always combined with the GTID of the owner process
- Granted to a single process.
- Either refers to
 - Granter memory space
 - (pointer, size)
 - MMIO resource:
 - (MMIO, offset, size)

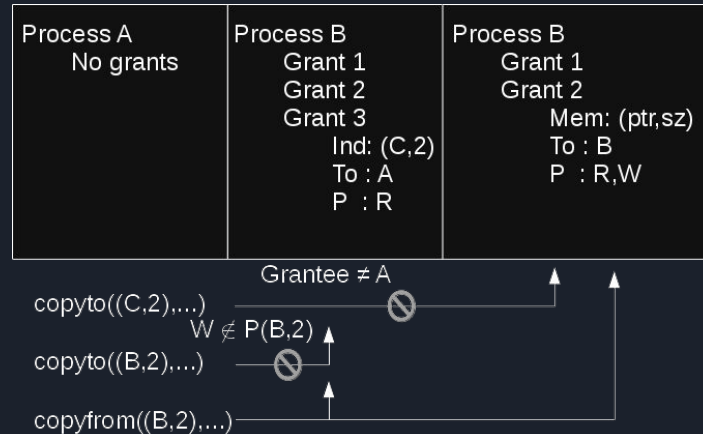


Memory Grants

- Grantee operations:
 - `mg_copyto (MG, offset, data, size)`
 - `mg_copyfrom(MG, offset, data, size)`
- Owner operations:
 - `mg_getbuf(MG)`
 - `mg_revoke(mg)`
 - `mg_create(MMIO/memory, grantee GTID)`

Memory Grants: Indirect Grants

- Refer not to memory but to a grant given to the owner.
- Allow grantee to further delegate grants
- Permissions are the intersection of those in the chain






ME optimizations to MINIX IPC: IOs

- Direct IPC between process and drivers is impossible in MINIX
- ME OS has a solution: kernel is aware of fd's
- Memory can be granted to fd's owners
- Messages targeted to GTID 0 go to fd driver.

```
void io_close(int pid, int fd);  
void io_open (int io, int s_gtid, int s_fd,  
              int c_gtid, int c_fd, int minor, int sel, int flags);
```

ME optimizations to MINIX IPC: select_receive()

- select() was moved into kernel and combined with ipc_receive() as

```
int select_receive(  
    int nfd,  
    __int64 *readfds,  
    __int64 *writefds,  
    __int64 *exceptfds,  
    timeval *timeout,  
    int from_gtid,  
    syscall_msg *msg_out,  
    int *have_msg );  
  
void io_notify( int fd, int notbits );
```




DMA Locks

- Processes can request MGs to be locked in memory for DMA
- Separate in (device->ram) and out (ram->device) mappings

```
int sys_mem_dma_lock(
    short out_tid, char out_flags, int out_mg, int out_offset,
    short in_tid, char in_flags, int in_mg, int in_offset,
    int size,
    /*out*/ uint32_t *out_paddr,
    /*out*/ uint32_t *in_paddr,
    /*out*/ int *dl_hnd);
```

```
int sys_mem_dma_unlock( int dl_hnd );
```


ME Hardware





Understanding the address space

- MMIO metadata refers to physical addresses, but HW is nonstandard and configurable
- However,...

```
$ strings busdrv.mod -n 12
```

```
...
```

```
HECI1_PCIPF_IBDF
```

```
HECI2_PCIPF_IBDF
```

```
FTPM_PCIPF_IBDF
```

```
SECURE_ENCLAVE_PCIPF_IBDF
```

```
RAVDM_PCIPF_IBDF
```

```
ATT_PCIPF_IBDF
```

```
GEN_PCIPF_IBDF
```

```
GPIO_PROXY_PCIPF_IBDF
```

```
KERNEL_TIMER_PCIPF_IBDF
```

```
...
```




The bus driver: `busdrv`

- Power gating
- PCI configuration space access
- Sideband bus access
- Physical resource mapping (BARs, ATTs)

- Old SPT builds have lots of debug strings
- Holds table containing system address and bus map

The table in human readable form

	Name	Type	CFG base	Bus	Dev	Func	SAI	SKU flags
0	HECI1_PCIPF_IBDF	PRIM_PCIFIXED	F1000000	0	0	0		——0
1	HECI2_PCIPF_IBDF	PRIM_PCIFIXED	F1001000	0	0	1		——0
2	FTPM_PCIPF_IBDF	PRIM_PCIFIXED	F1002000	0	0	2		——0
3	SECURE_ENCLAVE_PCIPF_IBDF	PRIM_PCIFIXED	F1003000	0	0	3		——0
4	RAVDM_PCIPF_IBDF	PRIM_PCIFIXED	F1004000	0	0	4		——0
5	ATT_PCIPF_IBDF	PRIM_PCIFIXED	F1005000	0	0	5		——0
6	GEN_PCIPF_IBDF	PRIM_PCIFIXED	F1006000	0	0	6		——0
7	GPIO_PROXY_PCIPF_IBDF	PRIM_PCIFIXED	F1007000	0	0	7		——0
8	KERNEL_TIMER_PCIPF_IBDF	PRIM_PCIFIXED	F1008000	0	1	0		——0
9	APP_TIMER_PCIPF_IBDF	PRIM_PCIFIXED	F1009000	0	1	1		——0
10	IPC_PCIPF_IBDF	PRIM_PCIFIXED	F100A000	0	1	2		——0
11	HECI3_PCIPF_IBDF	PRIM_PCIFIXED	F100B000	0	1	3		——0

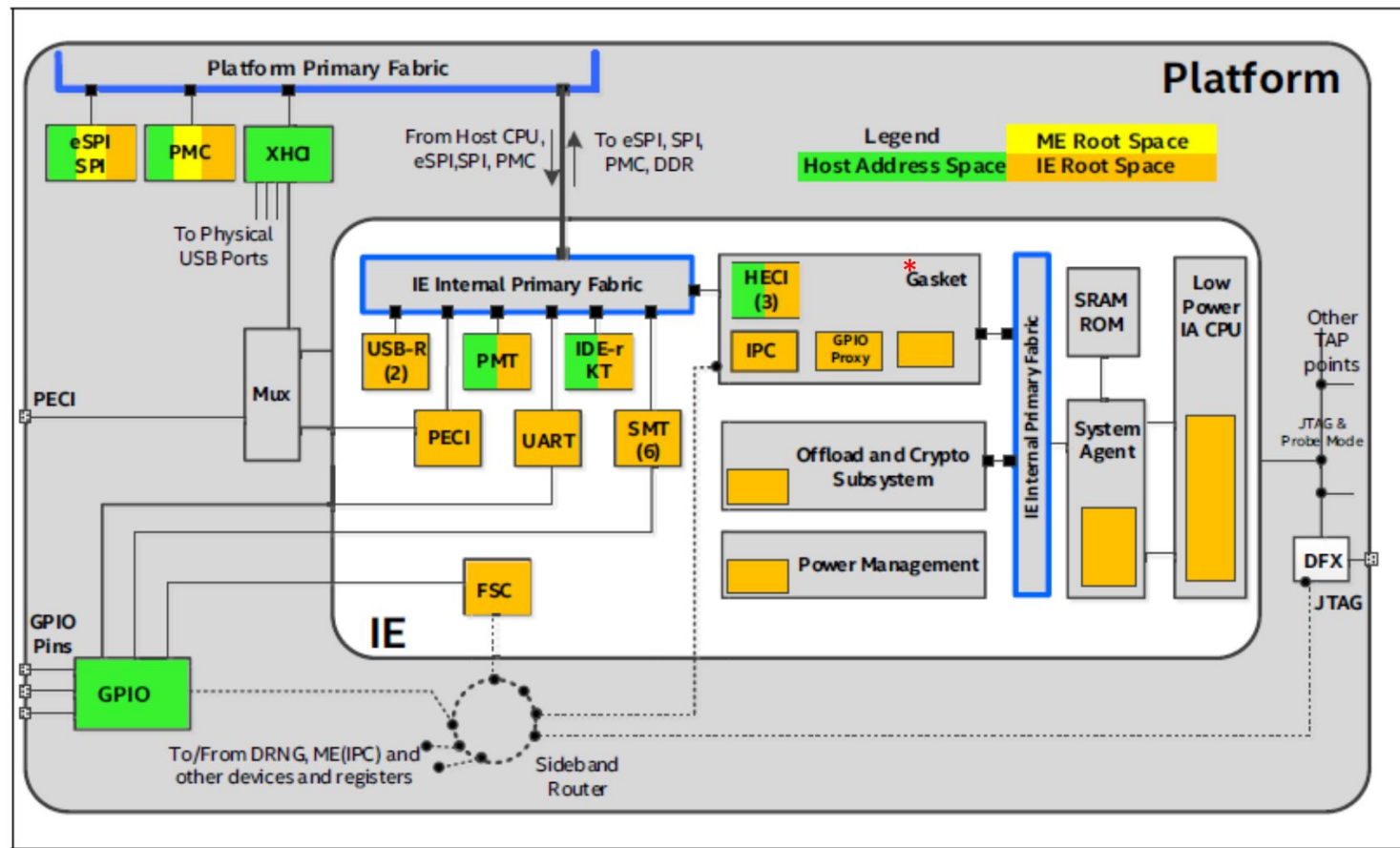


	Name	Type	CFG base	Bus	Dev	Func	SAI	SKU flags
	ROM MINUTE_IA_SA_IBDF	ROM Early Init	E0000000	0	0	0	?	
	ROM CRYPTO_ENGINE_IBDF	ROM Init	E0008000	0	1	0	?	
17	KVM_PCIP_IBDF	PRIM_PCIP	E0040000	0	8	0	52	——0
18	USB0_PCIP_IBDF	PRIM_PCIP	E0048000	0	9	0	54	——0
19	USB0_PCIP_IBDF	PRIM_PCIP	E0049000	0	9	1	6E	——0
20	SMT0_PCIP_IBDF	PRIM_PCIP	E0050000	0	10	0	56	——0
21	SMT1_PCIP_IBDF	PRIM_PCIP	E0051000	0	10	1	56	——0
22	SMT2_PCIP_IBDF	PRIM_PCIP	E0052000	0	10	2	56	——0
23	SMT3_PCIP_IBDF	PRIM_PCIP	E0053000	0	10	3	56	——3—0
24	SMT4_PCIP_IBDF	PRIM_PCIP	E0054000	0	10	4	56	——3—0
25	SMT5_PCIP_IBDF	PRIM_PCIP	E0055000	0	10	5	56	——3—0
14	CLINK_PCIP_IBDF	PRIM_PCIP	E0058000	0	11	0	5C	——0
26	SST_PCIP_IBDF	PRIM_PCIP	E0060000	0	12	0	5E	——3—0
15	PTIO_IDER_PCIP_IBDF	PRIM_PCIP	E0068000	0	13	0	60	——0
16	PTIO_KT_PCIP_IBDF	PRIM_PCIP	E0069000	0	13	1	62	——0
27	PMT_PCIP_IBDF	PRIM_PCIP	E0070000	0	14	0	64	——0
31	HDAU_PCIP_IBDF	PRIM_PCIP	E00C0000	0	24	0	46	——0
13	SPI_PCIP_IBDF	PRIM_PCIP	E00C8000	0	25	0	40	——0
28	ESPI_PCIP_IBDF	PRIM_PCIP	E00C9000	0	25	1	40	——0
12	PMC_PCIP_IBDF	PRIM_PCIP	E00D0000	0	26	0	2A	——0
29	GBE_PCIP_IBDF	PRIM_PCIP	E00D8000	0	27	0	44	——0
30	WLAN_PCIP_IBDF	PRIM_PCIP	E0100000	1	0	0	5C	——0



Other information sources on HW

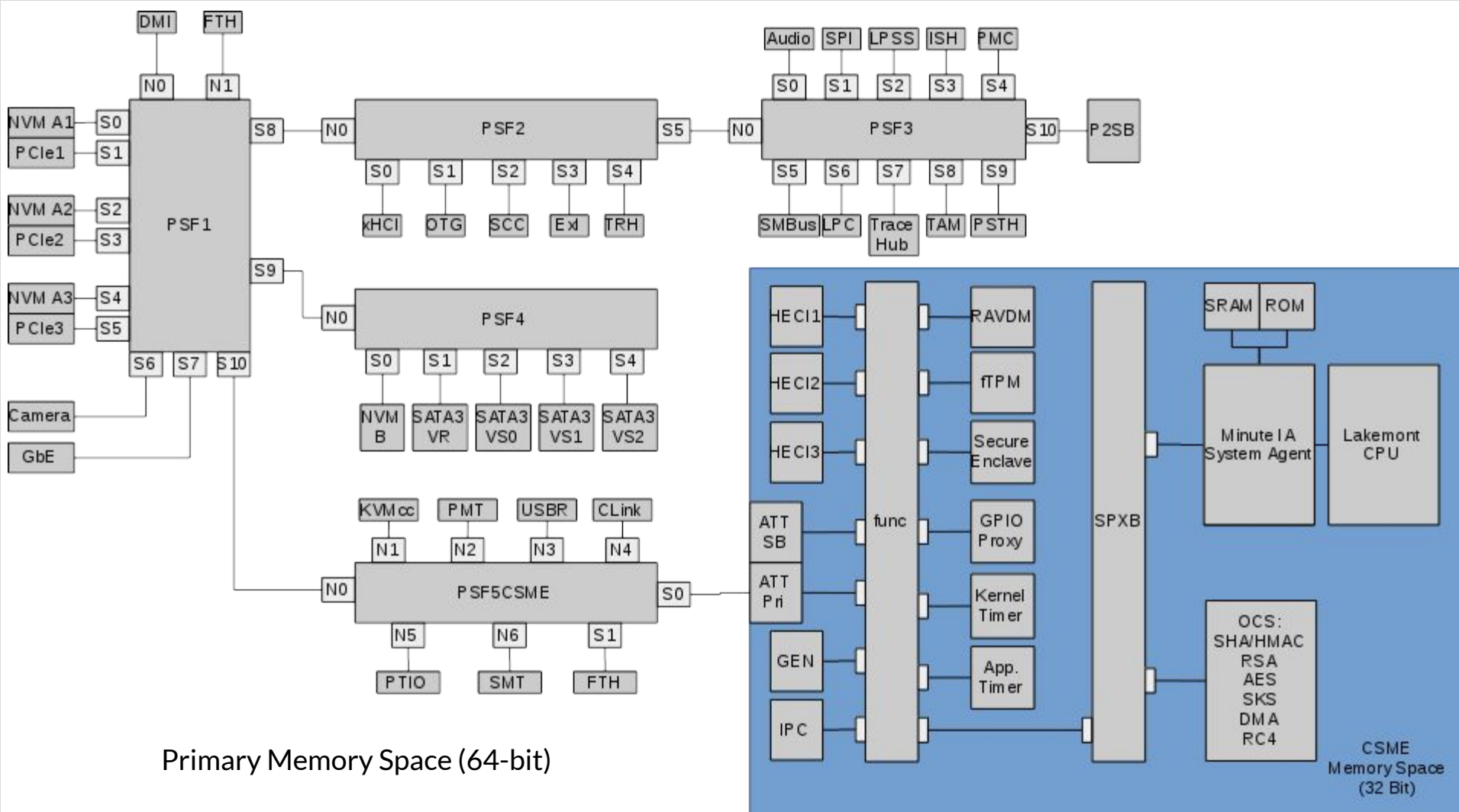
- My ME emulator:
 - <https://github.com/peterbiornx/meloader>
- Various files in old Intel System Studio versions
 - See [Intel VISA: Through the Rabbit Hole](#) (Goryachy, Ermolov) for info on extracting
 - https://github.com/peterbiornx/iss_tools Tools for parsing some of the XML config
- Innovation Engine firmware by HP
- Pentium N and J Series Datasheets
 - [Intel® Pentium® and Celeron® Processor N and J Series: Datasheet 3](#)



Buses

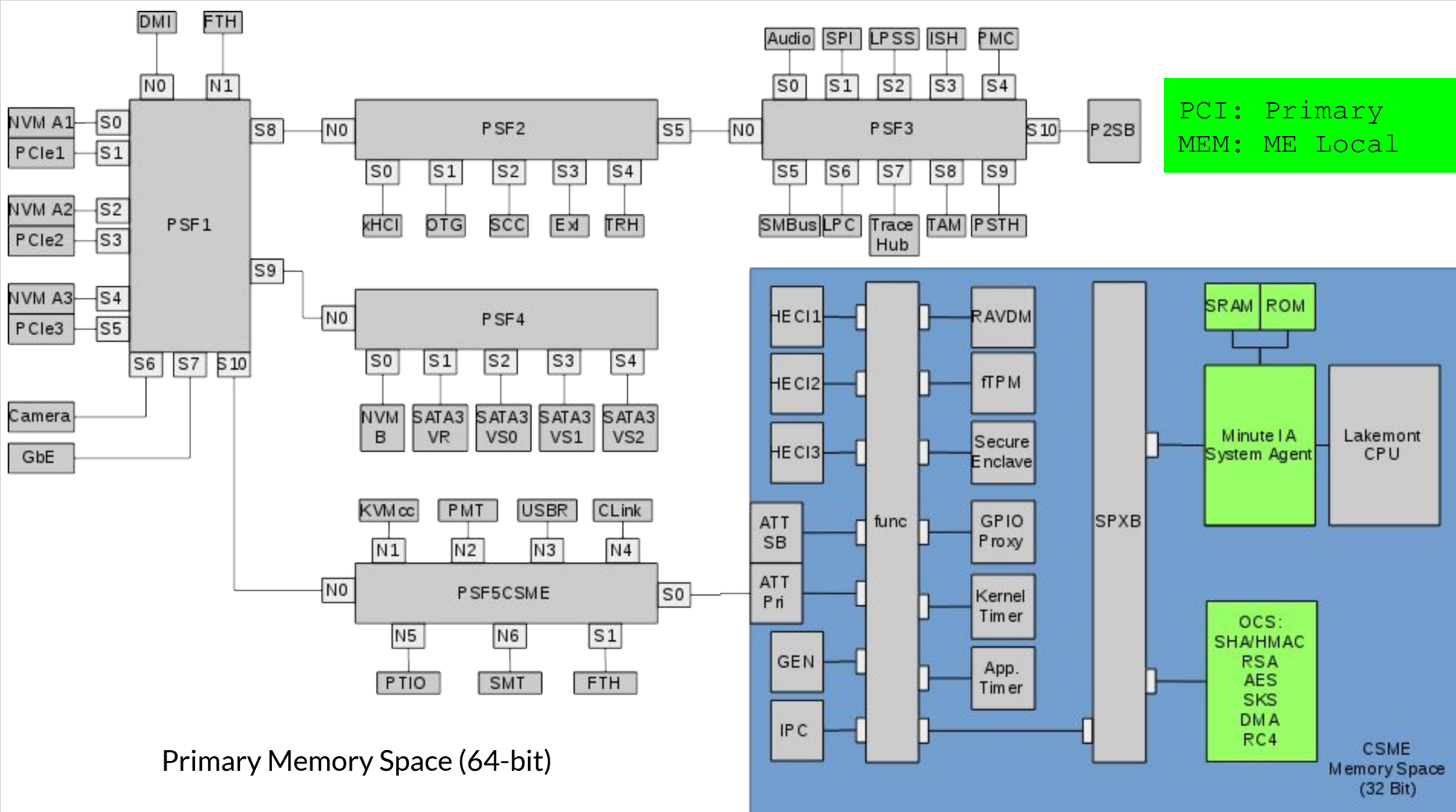
- Primary
- Side Band
- DFx
- External

* IE Gasket has implied bridges



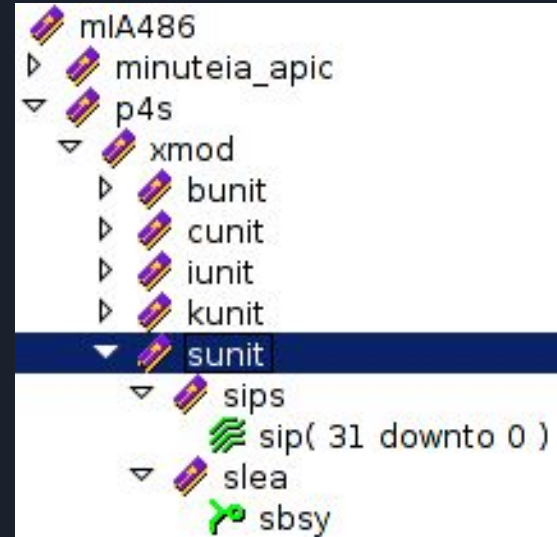
Primary Memory Space (64-bit)

CSME
Memory Space
(32 Bit)



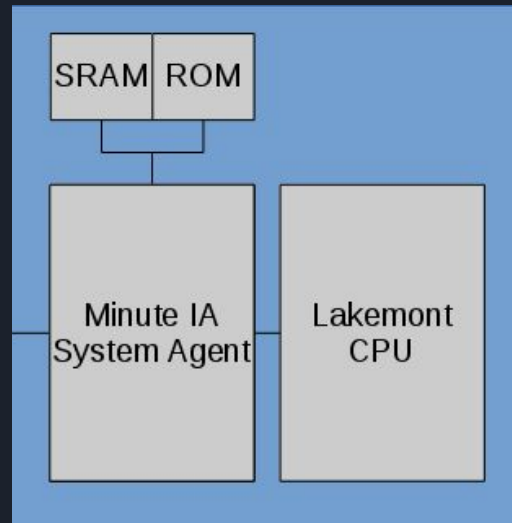
Processor

- Lakemont microarchitecture
 - “Minute IA”
 - 486 derived
 - Same as Quark MCUs
 - Run-Control documentation is public
 - Supported by OpenOCD
- Modern ISA extensions
 - MSRs
 - CPUID
- Only MSI interrupts used



Custom host bridge: Minute IA System Agent

- Similar to some Quark devices
- Partial documentation available:
 - [Intel® Pentium® and Celeron® Processor N and J Series: Datasheet 3](#)
- IO address space seems to be unused!
- Implements
 - SRAM / ROM controller
 - IOMMU for fabric->memory requests
 - PCI configuration space access
 - Bus firewall
 - and more



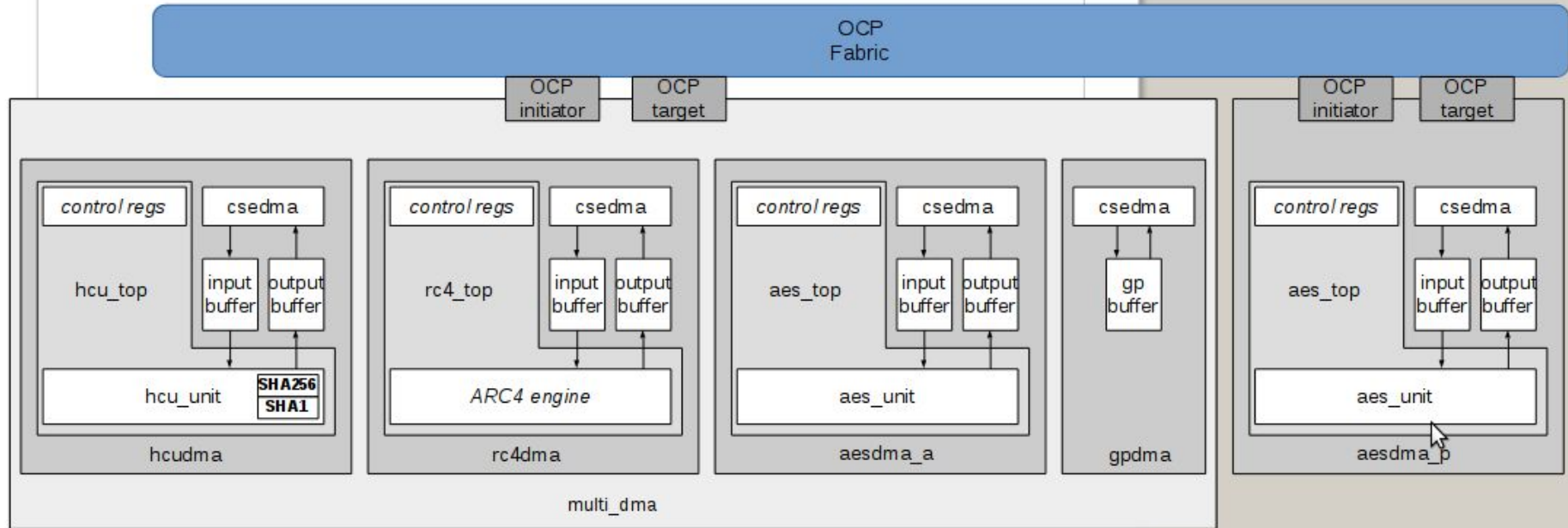


Hardware Cryptographic Accelerator

- Referred to in various places as OCS
- Hardware implementations of
 - SHA1
 - SHA256
 - SHA256 HMAC
 - AES (2 cores)
 - RSA
 - RC4
- Multiple DMA engines
- Secure Key Storage

Base	Name	DMA
+8000	<u>AES</u>	X
+A000	<u>AES</u>	X
+B000	Hash	X
+C000	? (RC4 in <u>IE</u>)	X
+D000	? (GP in <u>IE</u>)	X
+E000	<u>RSA</u>	
+F000	SKS	

Hardware Cryptographic Accelerator IP blocks (partial)

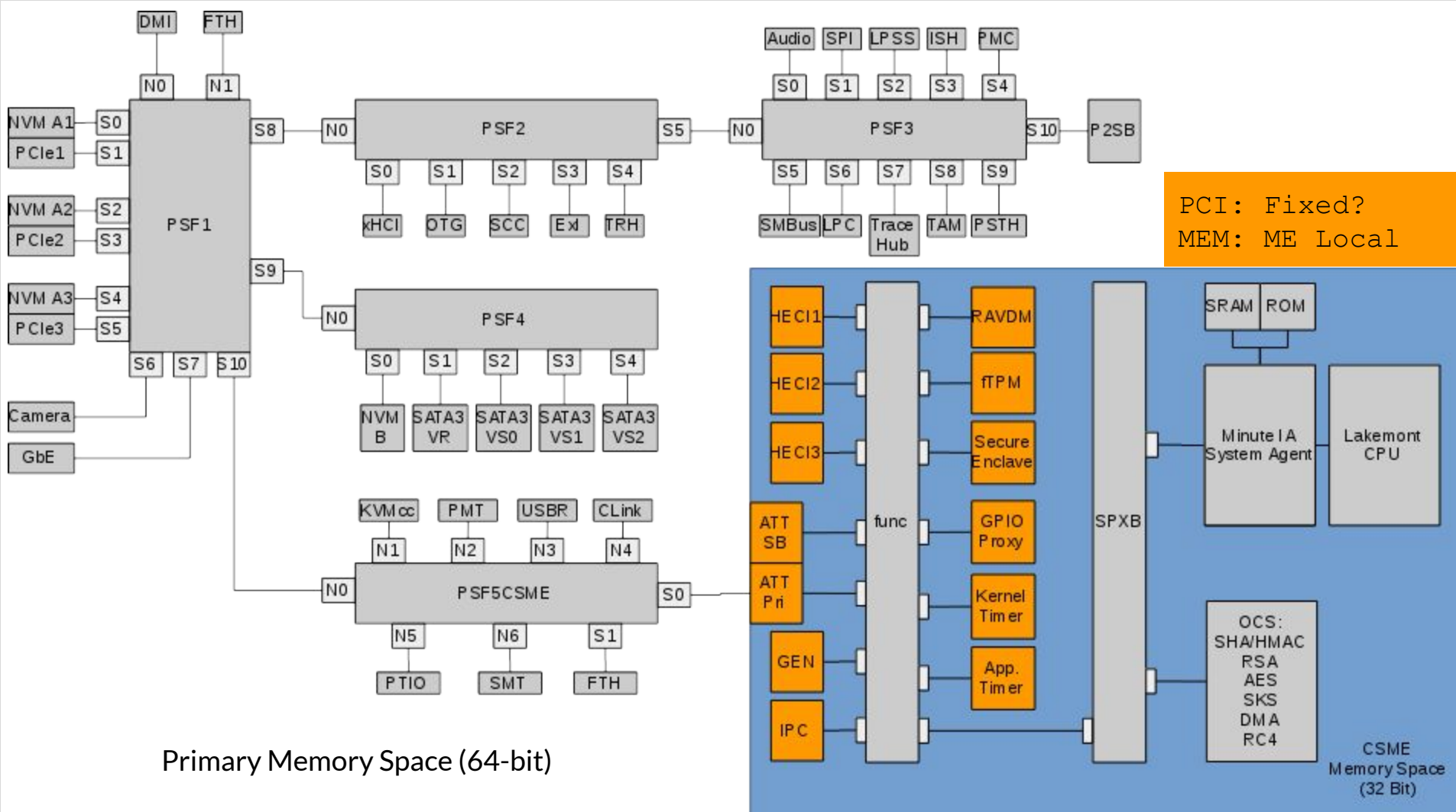




Crypto: DMA Engines

- At offset 400h in HCU sub devices
- Used for general purpose DMA
- Src/Dst = 0 targets internal buffer

id	name	Description
+400	SRC_ADDR	Source address of the DMA transfer
+404	DST_ADDR	Destination address of the DMA transfer
+408	SRC_SIZE	Size of the source buffer
+40C	DST_SIZE	Size of the destination buffer
+410	CONTROL	Transfer control bits
+428	STATUS	Status of the DMA engine





Host-Embedded Controller Interface (HECI)

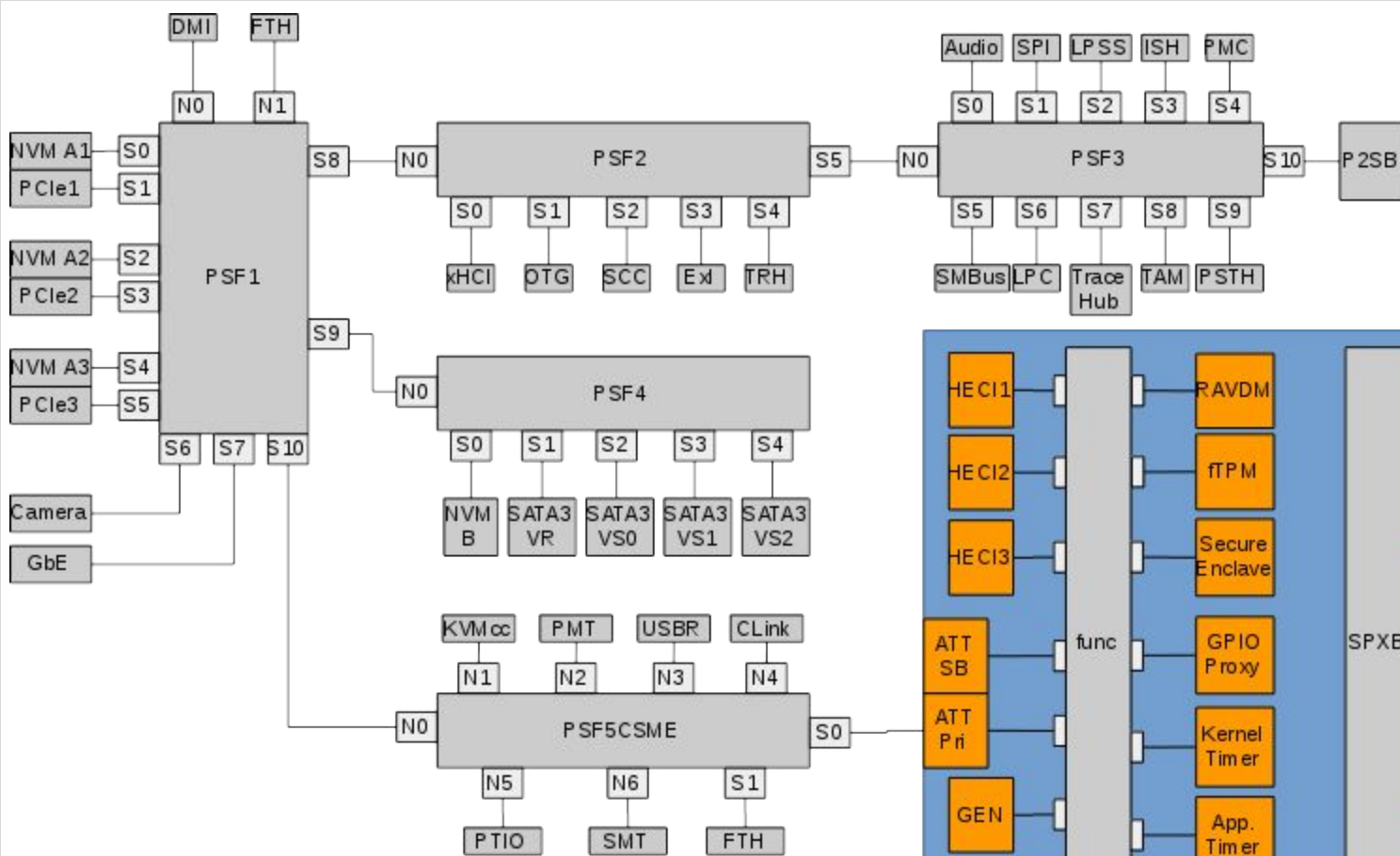
- Misleading name
 - also known as Management Engine Interface (MEI)
- Command interface between Host and ME
- Firmware Status Registers
 - Written by ME
 - Read by host.
 - See <https://github.com/peterbjornx/meloder/tree/master/periph/gasket/heci>
 - and [intel/skylake: Display ME firmware status before os boot \(1a511c4f3\) · Gerrit Code Review](#)
 - and the MEINFO tool in the vendor package.



Primary Address Translation Table

- Maps ME memory cycles onto primary fabric
- Used for both ME and host root spaces
- Not fully understood yet, config is pretty much hardcoded:

Slot	ME Address	Size	Primary address	Control	Descriptions
0	F2000000	2000000	00000000_F2000000	12040007	ME peripherals
1	F4600000	200000	00000000_F4600000	12040007	ME peripherals
2	D0000000	4000000	00000000_00000000	080E0003	UMA
3	F7000000	800000	00000000_F7000000	12040007	TraceHub
4	BC000000	2000000	00000000_00000000	01040003	Host DRAM!
5	C0000000	2000000	00000000_00000000	03440003	
6	C4000000	2000000	00000000_00000000	03440003	
7	C8000000	2000000	00000000_C8000000	12040003	
8	CA000000	2000000	00000000_00000000	03040003	

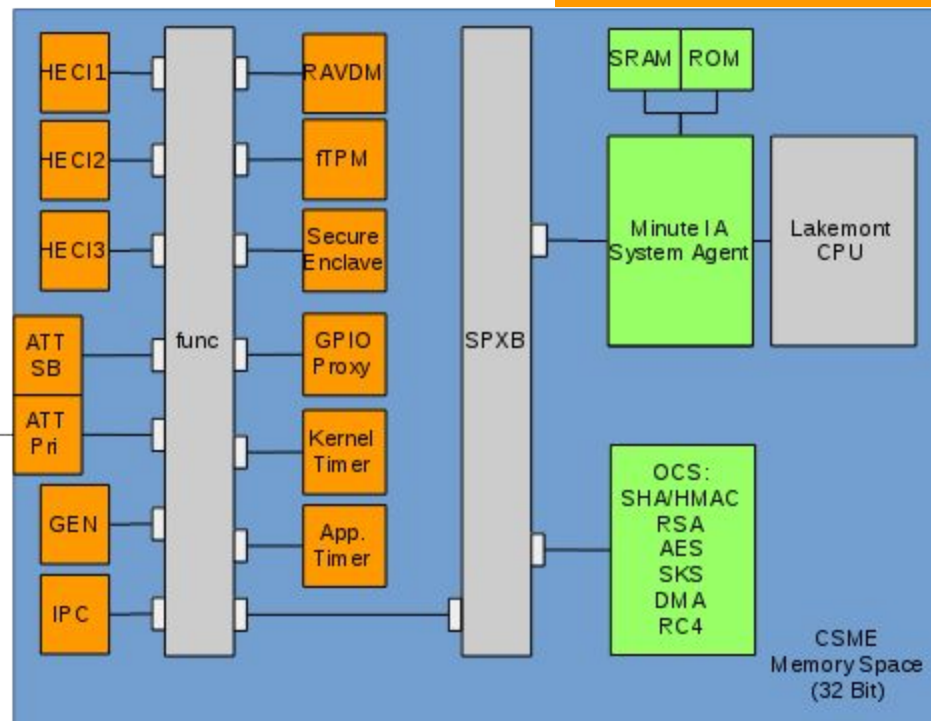


PCI: Primary
MEM: Behind ATT

PCI: Primary
MEM: ME Local

PCI: Fixed?
MEM: ME Local

Primary Memory Space (64-bit)



CSME
Memory Space
(32 Bit)



Root spaces

- Some peripherals expose different PCI functions to different hosts
- Example: SPI controller, documented at:
 - [Intel® Pentium® and Celeron® Processor N and J Series: Datasheet 3](#)



Sideband Fabric

- Packet switched network
- Endpoint IDs instead of PCI BDF
- Accessible from both ME and host
- PCI-like opcodes:
 - Register R/W
 - Configuration R/W
 - Memory R/W
- Addressed by:
 - (Opcode, Endpoint, Root Space, Function Number, BAR number)
- Security model based around SAI numbers
- Spec partially public as patent application US 2013 0138858A1

Sideband Address Translation Table

Maps sideband devices as memory space

id	name
+00	INT_BA
+04	INT_SIZE
+08	CONTROL
+10	unknown
+14	unknown
+18	SB_ADDRESS
+1C	SB_ADDRESS_HI

start	end	name	Description
0	7	endpoint	The sideband endpoint number
8	15	read_op	The read opcode to use
16	23	write_op	The write opcode to use
24	27	bar	The base address register index

start	end	name	Description
0	7	function	The function ID being addressed
8	10	rootspace	The root space

Some sideband addresses for LBG/SPT

BROADCAST1 = 0xFF,	LDO = 0x14,	NPK = 0xB6,	CSME3 = 0x93, //FSC	RUMAIN = 0x3B,
BROADCAST2 = 0xFE,	DSP = 0xD7,	MIMP0 = 0xB0,	CSME2 = 0x92, //USB-RSAI	EC = 0x20,
DMI = 0xEF,	FUSE = 0xD5,	GPIOCOM0 = 0xAF,	CSME0 = 0x90, //CSE	CPM2 = 0x38,
ESPISPI = 0xEE,	FSPROX0 = 0xD4,	GPIOCOM1 = 0xAE,	CSME_PSF = 0x8F, //MEPSF	CPM1 = 0x37,
ICLK = 0xED,	DRNG = 0xD2,	GPIOCOM2 = 0xAD,	CSMERTC = 0x8E,	CPM0 = 0x0C,
MODPHY4 = 0xEB,	FIA = 0xCF,	GPIOCOM3 = 0xAC,	IEUART = 0x80,	VSPTHERM = 0x25,
MODPHY5 = 0x10,	FIAWM26 = 0x13,	GPIOCOM4 = 0xAB,	IEHOTHAM = 0x7F,	VSP2SB = 0x24,
MODPHY1 = 0xE9,	USB2 = 0xCA,	GPIOCOM5 = 0xA1,	IEPMT = 0x7E,	VSPFPK = 0x22,
PMC = 0xE8,	LPC = 0xC7,	MODPHY2 = 0xA9,	IESSTPECI = 0x7D,	VSPCPM2 = 0x35,
XHCI = 0xE6,	SMB = 0xC6,	MODPHY3 = 0xA8,	IEFSC = 0x7C,	VSPCPM1 = 0x34,
OTG = 0xE5,	P2S = 0xC5,	PNCRC = 0xA5,	IESMT5 = 0x7B,	VSPCPM0 = 0x33,
SPE = 0xE4,	ITSS = 0xC4,	PNCRB = 0xA4,	IESMT4 = 0x7A,	MSMROM = 0x08,
SPD = 0xE3,	RTC = 0xC3,	PNCRA = 0xA3,	IESMT3 = 0x79,	PSTH = 0x89
SPC = 0xE2,	PSF5 = 0x8F,	PNCRO = 0xA2,	IESMT2 = 0x78,	
SPB = 0xE1,	PSF6 = 0x70,	CSME15 = 0x9F, //SMS2	IESMT1 = 0x77,	
SPA = 0xE0,	PSF7 = 0x01,	CSME14 = 0x9E, //SMS1	IESMT0 = 0x76,	
UPSX8 = 0x06,	PSF8 = 0x29,	CSME13 = 0x9D, //PMT	IEUSBR = 0x74,	
UPSX16 = 0x07,	PSF9 = 0x21,	CSME12 = 0x9C, //PTIO	IEPTIO = 0x73,	
TAP2IOSFSB1 = 0xDF,	PSF10 = 0x36,	CSME11 = 0x9B, //PECI	IEIOSFGASKET = 0x72,	
TRSB = 0xDD,	PSF4 = 0xBD,	CSME9 = 0x99, //SMT6	IEPSF = 0x70,	
ICC = 0xDC,	PSF3 = 0xBC,	CSME8 = 0x98, //SMT5	FPK = 0x0A,	
GBE = 0xDB,	PSF2 = 0xBB,	CSME7 = 0x97, //SMT4	MPOKR = 0x3C,	
SATA = 0xD9,	PSF1 = 0xBA,	CSME6 = 0x96, //SMT3	MP1KR = 0x3E,	
SSATA = 0x0F,	HOTHARM = 0xB9,	CSME5 = 0x95, //SMT2	RUAUX = 0x0B,	
LDO = 0x14,	DCI = 0xB8,	CSME4 = 0x94, //SMT1		
	DFXAGG = 0xB7,			

Source: Intel VISA: Through the Rabbit Hole (Ermolov, Goryachy at BlackHat Asia 2019)

Dynamic analysis



ME JTAG How-To

Arbitrary code execution in the BUP module (CVE-2017-5705,6,7)



Activation of RED UNLOCK without Intel keys




JTAG access to ME core



Full control over the target



ME is no longer a "black box"



Developing an exploit for CVE-2017-5705,6,7

- Determine stack location
- Craft payload to turn stack variable overflow into arbitrary write
- Determine return pointer address
- Find ROP gadgets
- Turn on debug access / chainload custom firmware



meloader: *WINE* for the ME

- Runs unmodified ME usermode binaries under Linux
- Built to run *bup*, not to be an accurate emulation of HW

<https://github.com/peterbjornx/meloader>

Features:

- ME binary loader
- Hooks for syslib, romlib
- Syscall stubs
- MMIO peripheral emulation
- Bus emulation
- MMIO passthrough to external programs
- Configurable hardware configuration and initial state
- SVEN decoder

meloader as a debugger

- Get *meloader* to run *bup* to the vulnerable part of its code



Peter Bosch @peterbjornx · Apr 21

I've gotten my ME loader to the point where it will load the Trace Hub config file, which means that I can now easily debug an SA-00086 exploit and hopefully get JTAG on a real ME other than the TXT targetted by the PT proof of concept



- Develop exploit against *bup* running in *meloader*
- Forget to add `--one-file-system` to `rm` command and lose `homedir`



Unfortunately there wasn't a way to scan or probe for devices so I had to generate an xml with all possible device paths in the jtag chain and clear out those with an invalid idcode then did an idrscan 0x2 on valid TAP devices until I found the processor id of the LMT device.

In the end, it works :)

Thanks for all the help I got from here!

After I'm done with the rest of my ROPs to get the main CPU booting, I'll release everything with a blog post on the whole process, in the meantime, you get to see the important offsets in that screenshot for those who need them :)



Peter Bosch @peterbjornx · Sep 1

Haven't yet pushed everything for this yet, but here's my ME emulator booting BUP to the point where it does the unlock logic, EXI logic and tracehub config (/home/bup/ct).

```
[METRC] sven: 0056000D8086000200000002
[ERROR] cse_sa: Bus error while reading from primary bus addr F00B1050 size 4
[METRC] sven: CSE zeroing register 00000000
[METRC] sven: DFX consent register 00000000
[METRC] sven: DFX personality register 00000000
[METRC] sven: DFX status register (low dword) 00000000
[METRC] sven: DFX status register (high dword) 00000000
[METRC] sven: DFX PUID register (low dword) 78ABCDEF
[METRC] sven: DFX PUID register (high dword) 00123456
[ERROR] libc: syscall( 25, 12, 0x0005CE78) wrong size
[ERROR] libc: syscall( 25, 12, 0x0005CE78) wrong size
[METRC] sven: "No secure token present"

[METRC] sven: "[HECI1_CSE_GS1] write data = 0x1000000, mask data = 0xF000000."
[DEBUG] exi: Read emecc register: 00000000
[DEBUG] exi: Write emecc register: 00000000 (ExI disabled)
[DEBUG] exi: Read emecc register: 00000000
[DEBUG] exi: Read ectrl register: 00000100
[DEBUG] dfxagg: Write consent register: 00000001

[ERROR] pmc: Write to unimplemented register 00000218 size 4
[DEBUG] thub: Read SCRPD0: 0x01000000
[ERROR] cse_sa: Bus error while reading from primary bus addr F0080018 size 4
```



↻ 3

♡ 30





Peter Bosch @peterbjornx · Sep 1

Finally got around to re-implementing and testing an exploit for the ME buffer overflow while parsing /home/bup/ct. Haven't tested against real HW yet, but it works in the emulator: Enabling DCI followed by RED unlock. The code path I used is the arb. write via mg_copyto.

```
[ERROR] pmc: Read to unimplemented register 00000260 size 4
[DEBUG] pmc: Write PMC RTCPMCFG: 00000040
[ERROR] pmc: Read to unimplemented register 00000010 size 4
[ERROR] pmc: Read to unimplemented register 00000018 size 4
[ERROR] pmc: Read to unimplemented register 00000218 size 4
[DEBUG] pmc: Write PMC HESTS0: 00000000
[DEBUG] pmc: Write PMC HESTS1: 00000000
[DEBUG] pmc: Write PMC IEWS: 00000000
[DEBUG] thub: Read SCRPD0: 0x00000000
[DEBUG] exi: Read emecc register: 00000000
[DEBUG] exi: Write emecc register: 00000010 (ExI enabled)
[DEBUG] exi: Read emecc register: 00000010
[DEBUG] dfxagg: Write personality register: 00000003
```

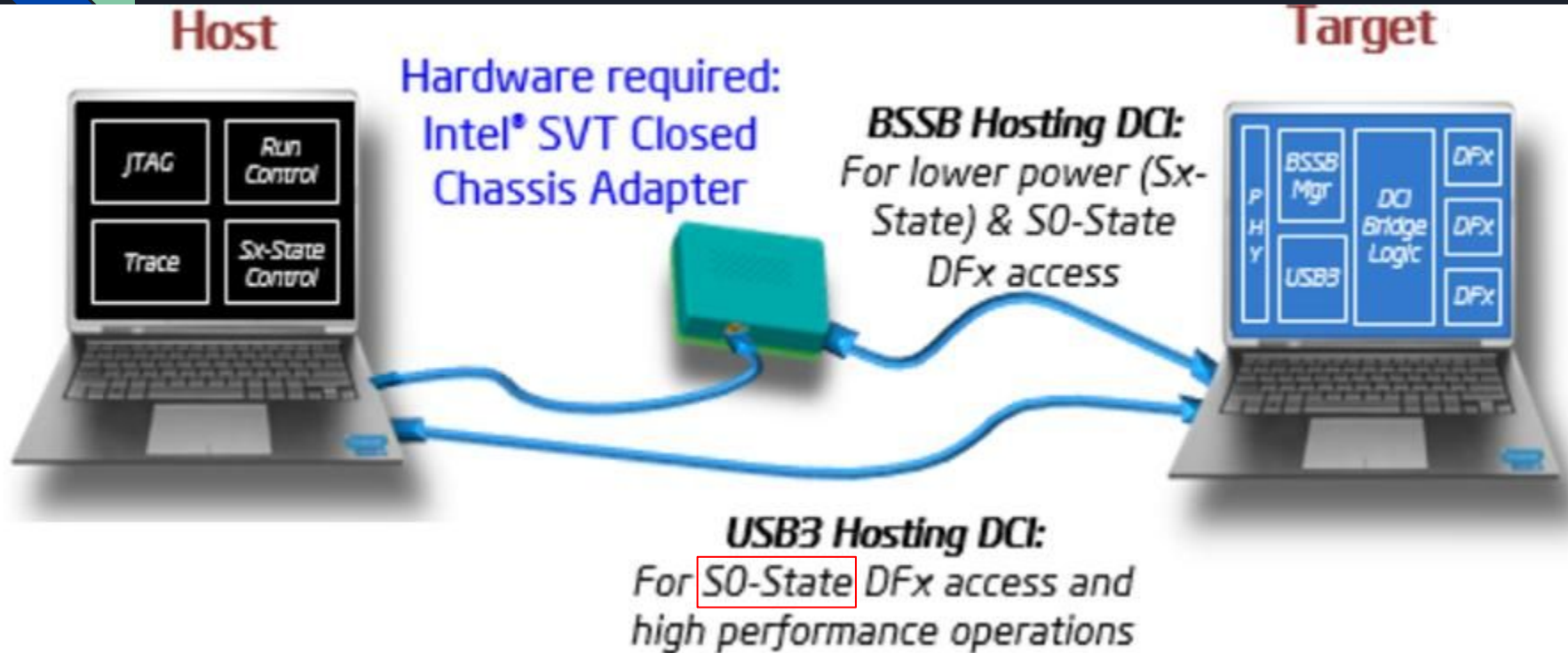
ROPS:

```
write_u( a, 4, 0x515cc )
write_u( a, 4, 0x11b9 )
write_u( a, 4, 0x50bed )
write_u( a, 4, 0x0010f )
write_u( a, 4, 0x00000 )
write_u( a, 4, 0x00003 )
write_u( a, 4, 0x12757 )
```



https://github.com/peterbjornx/me_sa86_exploit

Getting JTAG access



Expensive: € 456,30



<https://eu.mouser.com/ProductDetail/Intel/EXIBSSBADAPTOR?qs=byeeYqUIh0P5fUdWOCXn8A%3D%3D>

System boot process



ME Boot Process

- Microkernel bootstrap problem: the bup module
 - Has integrated versions of server functionality.
 - Had very high privileges up to ME 12
 - Is responsible for starting host CPU.
 - Starts all servers

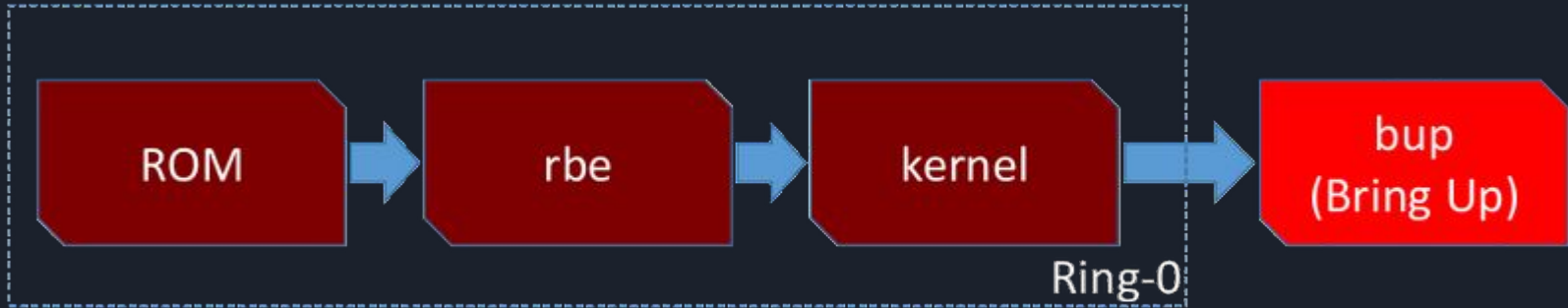
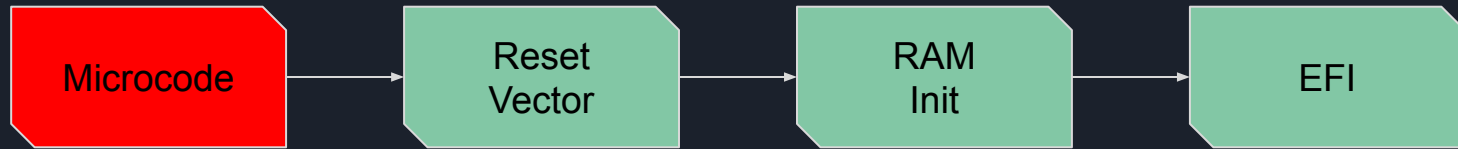


Image credit: "Intel ME: The Way of the Static Analysis." Ermolov, Goryachy, Sklyarov (2017)



Host Boot Process



Host Boot Process: Boot Guard

Host CPU

Micro
code

Boot Guard
ACM

Reset
Vector



Host Boot Process

Host CPU

Micro
code

Boot Guard
ACM

Reset
Vector

Update
PMC
FW

Host
init

Signal
PMC

CSME

CPU
Power
Up

Deassert
CPU RST

PMC



The Power Management Controller

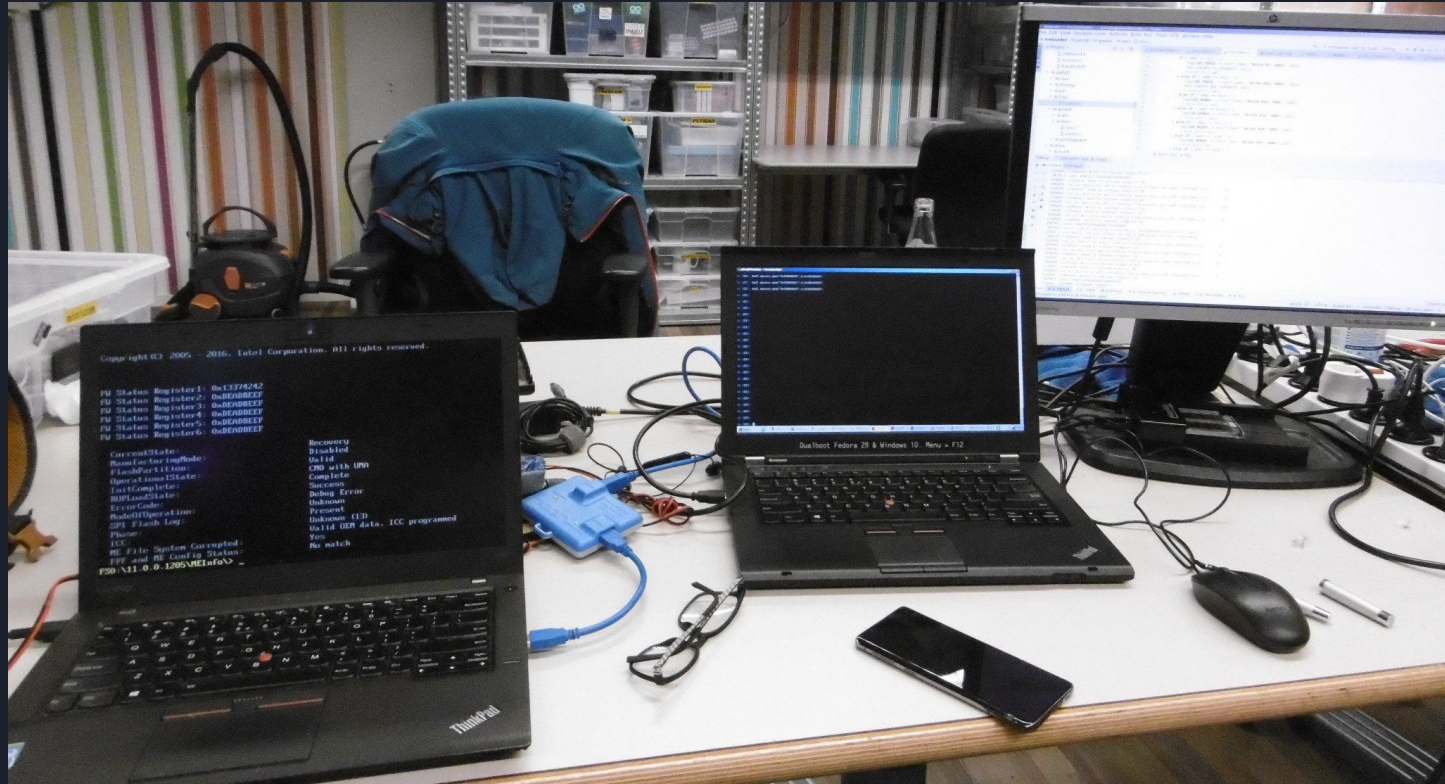
- 8051 based MCU
- Runs CMX RTOS “Copyright (c) CMX Co. 1999. All Rights Reserved”
- On SPT, Firmware in ROM but patches written from CSME
- On LBG, Firmware loaded from CSME
- Presents register based interface to the CSME
- Controls power gating and reset of IP blocks and CPU

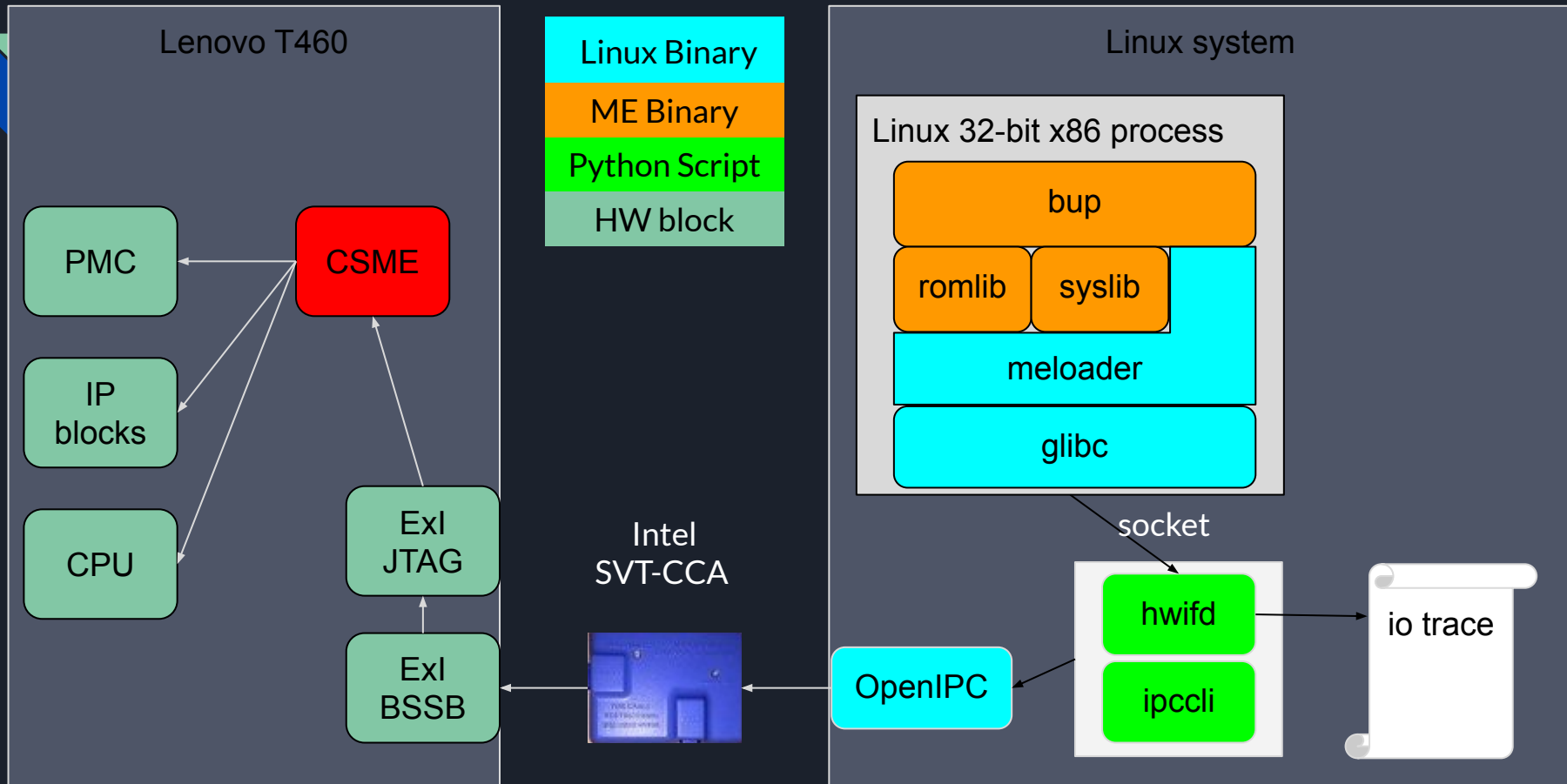


Host Initialization: ME tasks

1. Boot guard configuration load
2. Clock controller setup
3. PMC CPU power ungate
4. PSF Fabric configuration
5. CPU out of reset

Getting to the minimal viable implementation

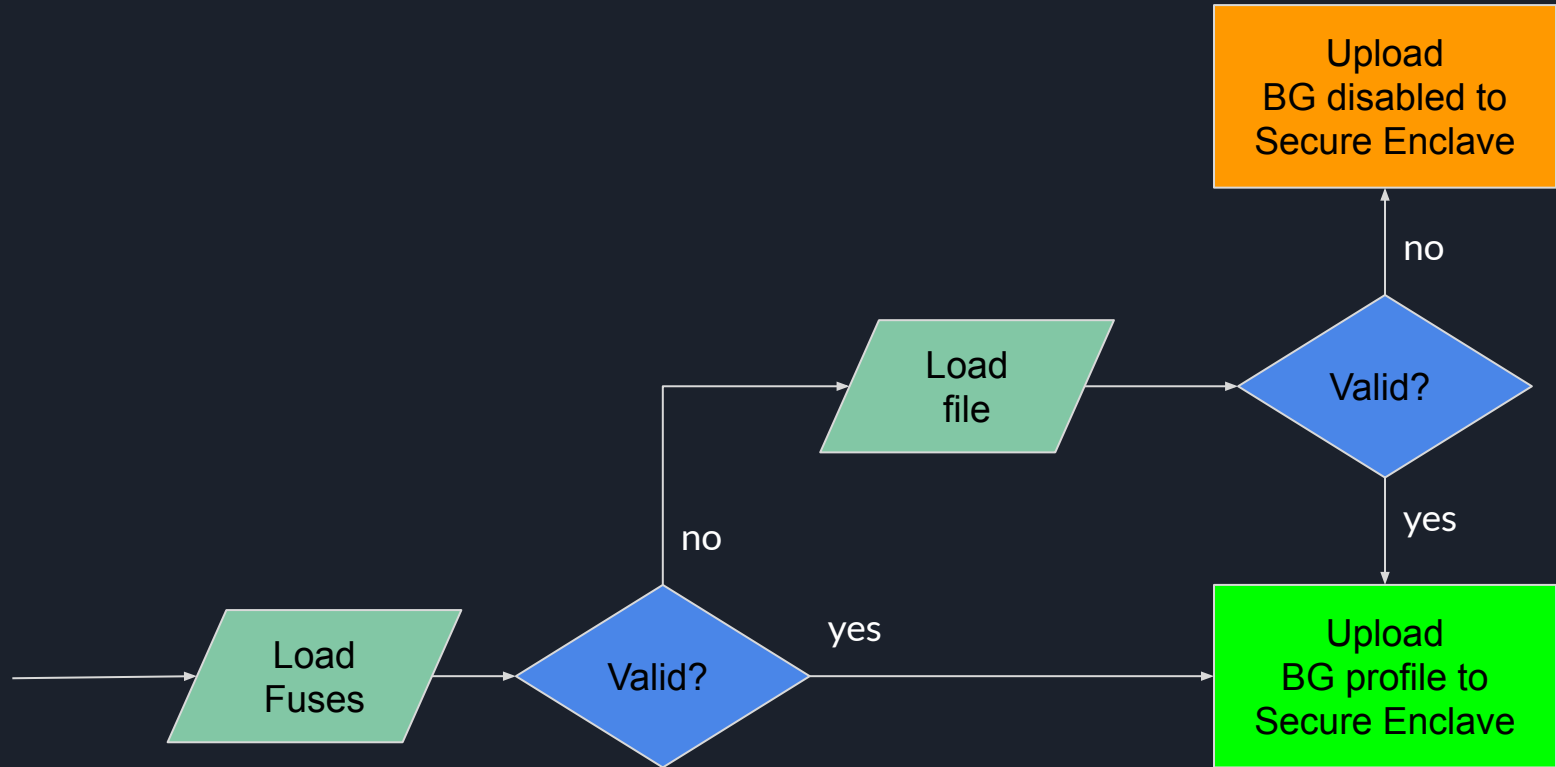




DEMO: meloader boots real HW



Boot Guard Configuration





Boot Guard Configuration

CPU

- Profile in MSRs
- ACM verifies
- Result to MMIO device
- Result to MSRs

CSME

- Profile in Secure Enclave device
- Respond to status of Secure Enclave
- Shutdown timer in software

Boot Guard Configuration

Minimal viable implementation

```
#00000001 : enum BOOTPOLRES, mappedto_191, bitfield # Secure Enclave Registers
#00000001 SB_FBGAcMEn = 1 ENC_UNK00 =0xF0099000
#00000002 SB_CpuDebugEn = 2 ENC_BOOTPOL =0xF0099040
#00000004 SB_BspInitEn = 4 ENC_SUNKMID =0xF0099044
#00000008 SB_ProtectBiosEn = 8 ENC_PUBKEY =0xF0099048
#00000001 : enum BOOTPOLTYPE, mappedto_192, bitfield
#00000001 SB_MeasuredBootEn = 1
#00000002 SB_VerifiedBootEn = 2

def enclave_init(hwif):
    hwif.memory_write( ENC_BOOTPOL , 4, 0x00040100 ) # bootpoltype . bootpolres
    hwif.memory_write( ENC_SUNKMID , 4, 0x00000000 ) # kmid . svn_bsmm . svn_acm_km
    hwif.memory_write( ENC_PUBKEY*0x00, 4, 0x00000000 ) # public key hash[0]
    hwif.memory_write( ENC_PUBKEY*0x04, 4, 0x00000000 ) # public key hash[1]
    hwif.memory_write( ENC_PUBKEY*0x08, 4, 0x00000000 ) # public key hash[2]
    hwif.memory_write( ENC_PUBKEY*0x0C, 4, 0x00000000 ) # public key hash[3]
    hwif.memory_write( ENC_PUBKEY*0x10, 4, 0x00000000 ) # public key hash[4]
    hwif.memory_write( ENC_PUBKEY*0x14, 4, 0x00000000 ) # public key hash[5]
    hwif.memory_write( ENC_PUBKEY*0x18, 4, 0x00000000 ) # public key hash[6]
    hwif.memory_write( ENC_PUBKEY*0x1C, 4, 0x00000000 ) # public key hash[7]
    hwif.memory_write( ENC_UNK00 , 4, 0x00000040 ) # constant?
```


Boot Guard Configuration

Minimal viable implementation

- Also opens up host-side firmware replacement for machines with Boot Guard enabled

```
def enclave_init(hwif):  
    hwif.memory_write( ENC_BOOTPOL      , 4, 0x00040100 ) # bootpoltype . bootpolres  
    hwif.memory_write( ENC_SUNKMID      , 4, 0x00000000 ) # kmid . svn_bsmm . svn_acm_km  
    hwif.memory_write( ENC_PUBKEY+0x00 , 4, 0x00000000 ) # public key hash[0]  
    hwif.memory_write( ENC_PUBKEY+0x04 , 4, 0x00000000 ) # public key hash[1]  
    hwif.memory_write( ENC_PUBKEY+0x08 , 4, 0x00000000 ) # public key hash[2]  
    hwif.memory_write( ENC_PUBKEY+0x0C , 4, 0x00000000 ) # public key hash[3]  
    hwif.memory_write( ENC_PUBKEY+0x10 , 4, 0x00000000 ) # public key hash[4]  
    hwif.memory_write( ENC_PUBKEY+0x14 , 4, 0x00000000 ) # public key hash[5]  
    hwif.memory_write( ENC_PUBKEY+0x18 , 4, 0x00000000 ) # public key hash[6]  
    hwif.memory_write( ENC_PUBKEY+0x1C , 4, 0x00000000 ) # public key hash[7]  
    hwif.memory_write( ENC_UNK00        , 4, 0x00000040 ) # constant?
```




Future goals

- Escalate to Ring 0
 - Either through “modchip” on debugger interface or
 - through kernel vulnerability.
- Implement bootloader for custom firmware
- and minimal bringup firmware.
- Add Exl support to openocd
- Clone Intel CCA
- Research post-boot power management: Sleep, Reboot, Shutdown
- Research PMC firmware
- Research other peripherals



Acknowledgements

- @noopwafel for lending me her Intel SVT-CCA
- Igor Skochinsky for information that helped me get started on this project
- Mark Ermolov for helping me out when I got stuck
- RevSpace (The Hague hackerspace) for access to a well-equipped electronics lab.

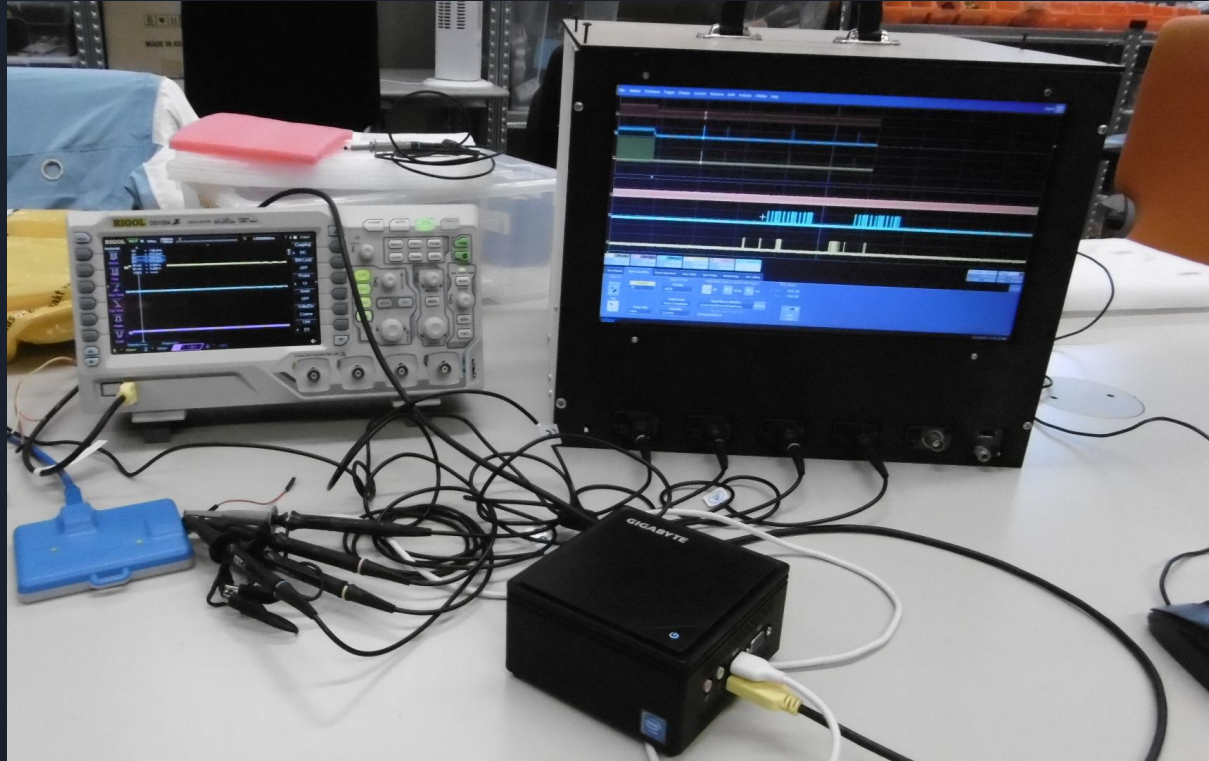
Questions?



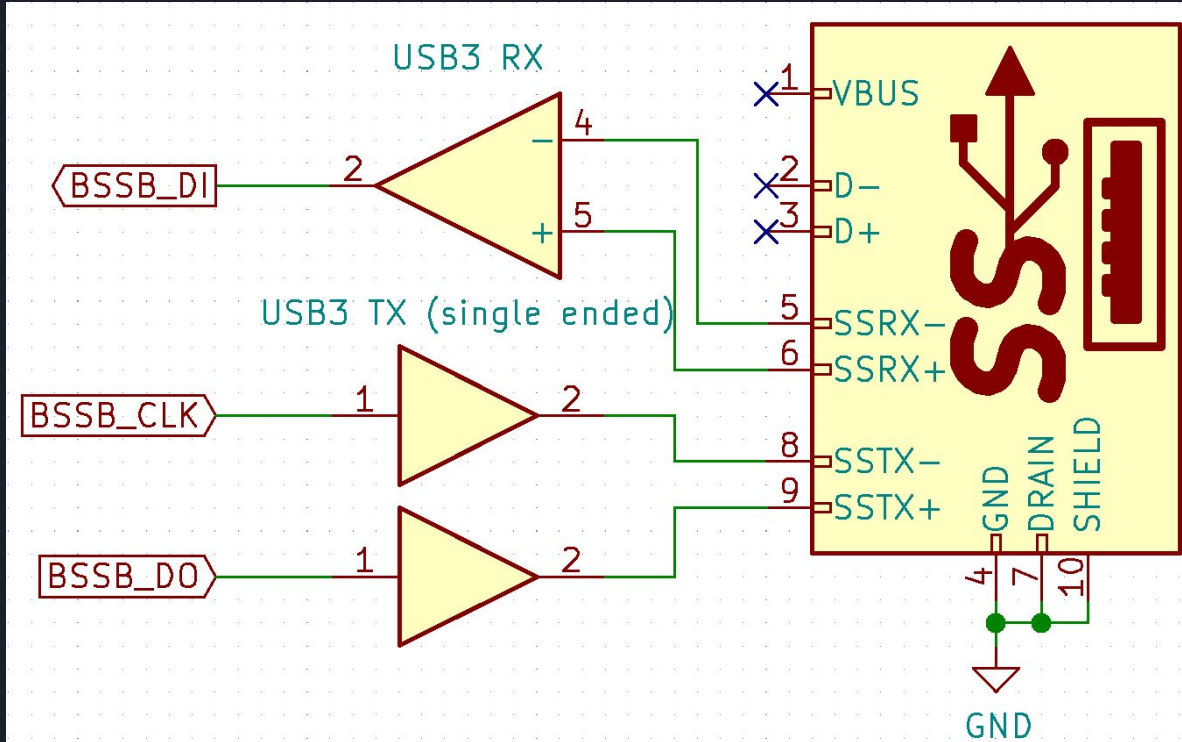
Cloning the CCA



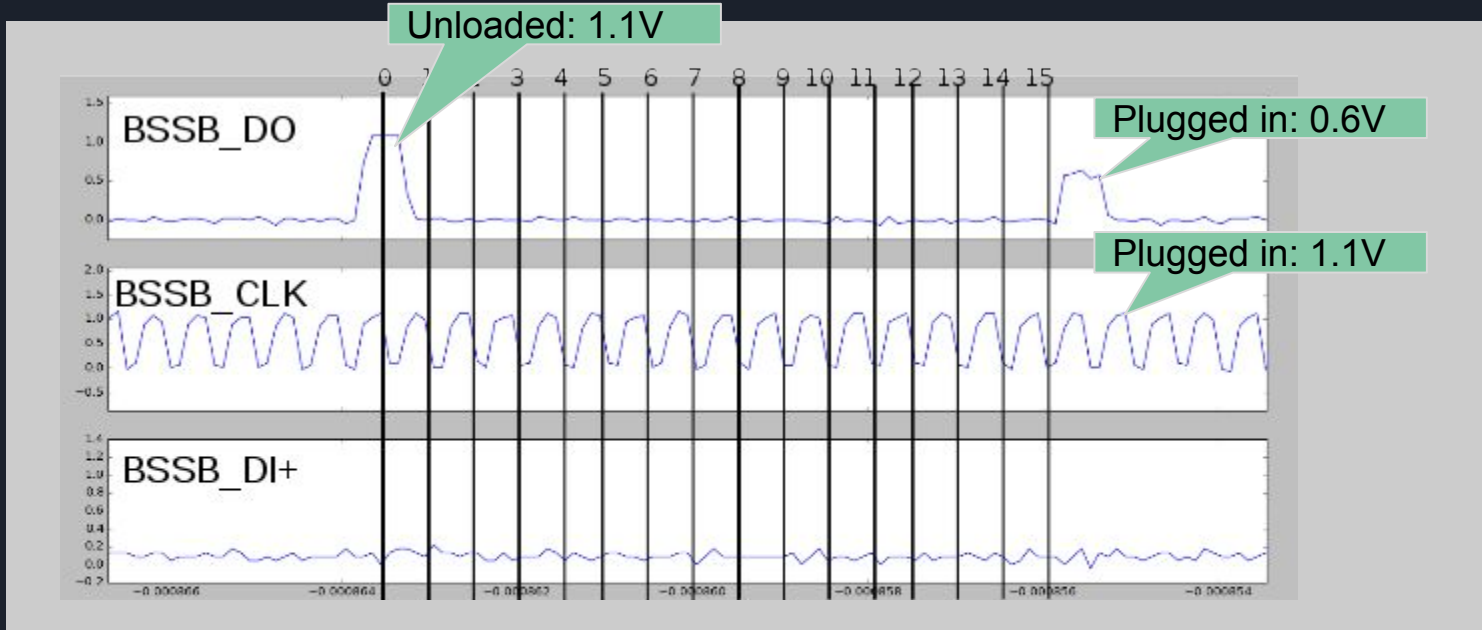
Debugging Intel systems: BSSB physical layer



Debugging Intel systems: BSSB physical layer

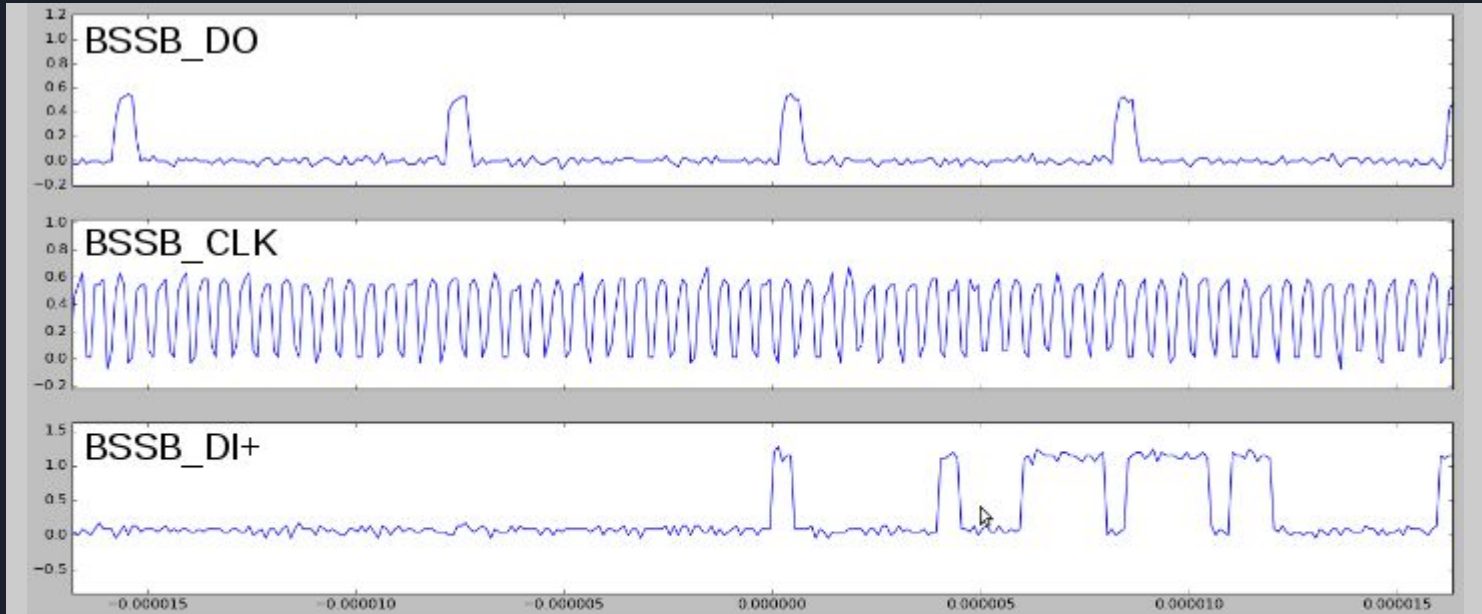


BSSB waveforms: Sync



BSSB_DO (to DUT) sampled on BSSB_CLK falling edge, data order LSb first
Sync word is 0x0001

BSSB waveforms: First DUT->Host packet



BSSB_DI (from DUT) sampled on BSSB_CLK rising edge, data order LSb first

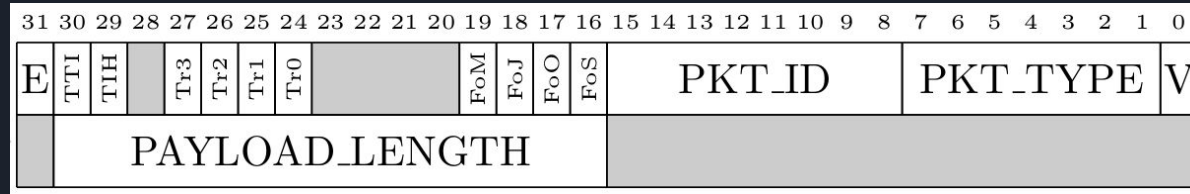


BSSB packets

- 64 bytes long
- Little Endian
- Same protocol as USB based ExI
 - CCA does handle some vendor requests

Outbound ExI packets

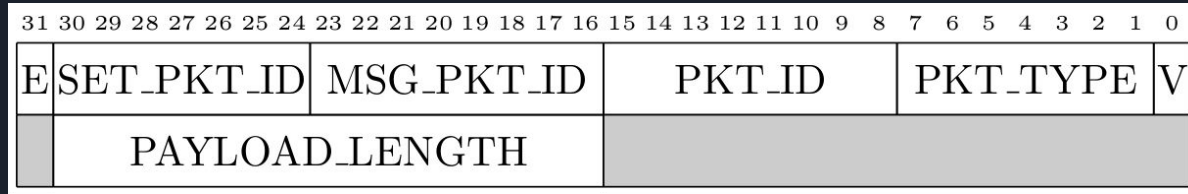
- DUT to Host
- Payload length only sent if E(xtended Header) is set





Inbound ExI packets

- Host to DUT
- Payload length only sent if E(xtended Header) is set



100C cookie_value

ROM API Entrypoints

1010 tstamp_read

1015 atoi1

101A atoi2

101F atoi1

1024 memchr

1029 memcmp

102E memcpy

1033 memmove

1038 memchr

103D memset

1042 strcat

1047 strchr

104C strcmp

1051 strcpy

1056 strlen

105B strncat

1060 strncmp

1065 strncpy

106A strlen

106F strrchr

1074 strstr

1079 strtoll

107E strtoll

1083 strtol2

1088 memcasecmp

108D strcasecmp

1092 strncasecmp

1097 itoa

109C itoa

10A1 utoa

10D3 bw_clr_lsb

10D8 bw_clr_msb

10DD bw_set_lsb

10E2 bw_set_msb

10E7 bw_find_hi_lsb

10EC bw_find_hi_msb

10F1 bw_find_lo_lsb

10F6 bw_find_lo_msb

10FB bw_count_ones

1100 bit_fill_clear

1105 bit_fill_set

110A bit_set

110F bit_clear

1114 bit_range_set

1119 bit_range_clear

111E bit_test

1123 bit_and

1128 bit_or

112D bit_inv

1132 bit_xor

1137 bit_find_set_lsb

113C bit_find_set_msb

1141 bit_find_clr_lsb

1146 bit_find_clr_msb

114B bit_csub_set_lsb

1150 bit_csub_set_msb

1155 bit_ssub_clr_lsb

115A bit_ssub_clr_msb

115F bit_fsub_set_lsb

1164 bit_fsub_set_msb

1169 bit_fsub_clr_lsb

116E bit_fsub_clr_msb

1173 bit_count_sub_ones

1178 bit_sc_and

117D base64_size

1182 base64_dec

1191 shl64

1196 shr_s64

11A0 shr_u64

11AA mul_s64

11AF div64

11B4 mod64

11B9 write_seg_32

11BE write_seg_16

11C3 write_seg_8

11C8 read_seg_32

11CD read_seg_16

11D2 read_seg_8

11D7 write_seg

11DC read_seg

11FA crc8

1209 memcmp_ct



Useful filenames

system_studio_2016.1.028.exe

system_studio_2016.2.040.exe

system_studio_2016.3.043.exe

system_studio_2016.4.046.exe

system_studio_2017.1.045.exe

system_studio_2017.2.050.exe

system_studio_2017.3.057.exe

system_studio_2017_beta.0.028.exe

system_studio_2019.0.033_ultimate_edition_windows_target.exe

system_studio_2019.1.050_ultimate_edition_windows_target.exe

system_studio_2019.2.057_ultimate_edition_windows_target.exe

system_studio_2019.4.077_ultimate_edition_windows_target.exe

system_studio_2019_beta.0.014_ultimate_edition_windows_target.exe

system_studio_2019_update_3_ultimate_edition.exe

w_cemdb_2014.0.026.exe

w_cemdb_p_2013.0.013.exe