emsys USB stack for IMC

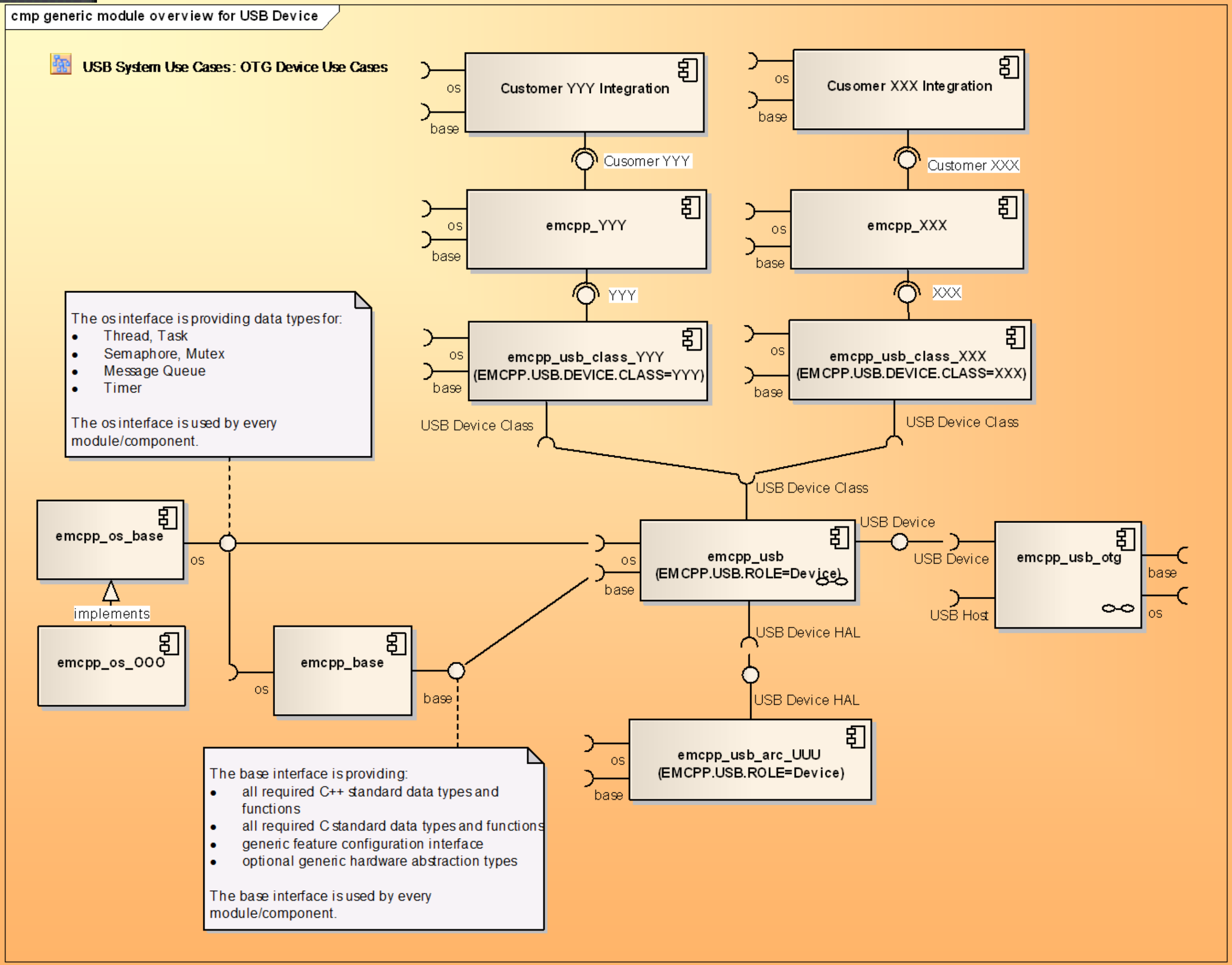
Dipl.Ing. Stefan Schulze

emsys Embedded Systems GmbH, Ilmenau

e-mail: stefan.schulze@emsys.de

Architecture

Generic Module Overview USB Device



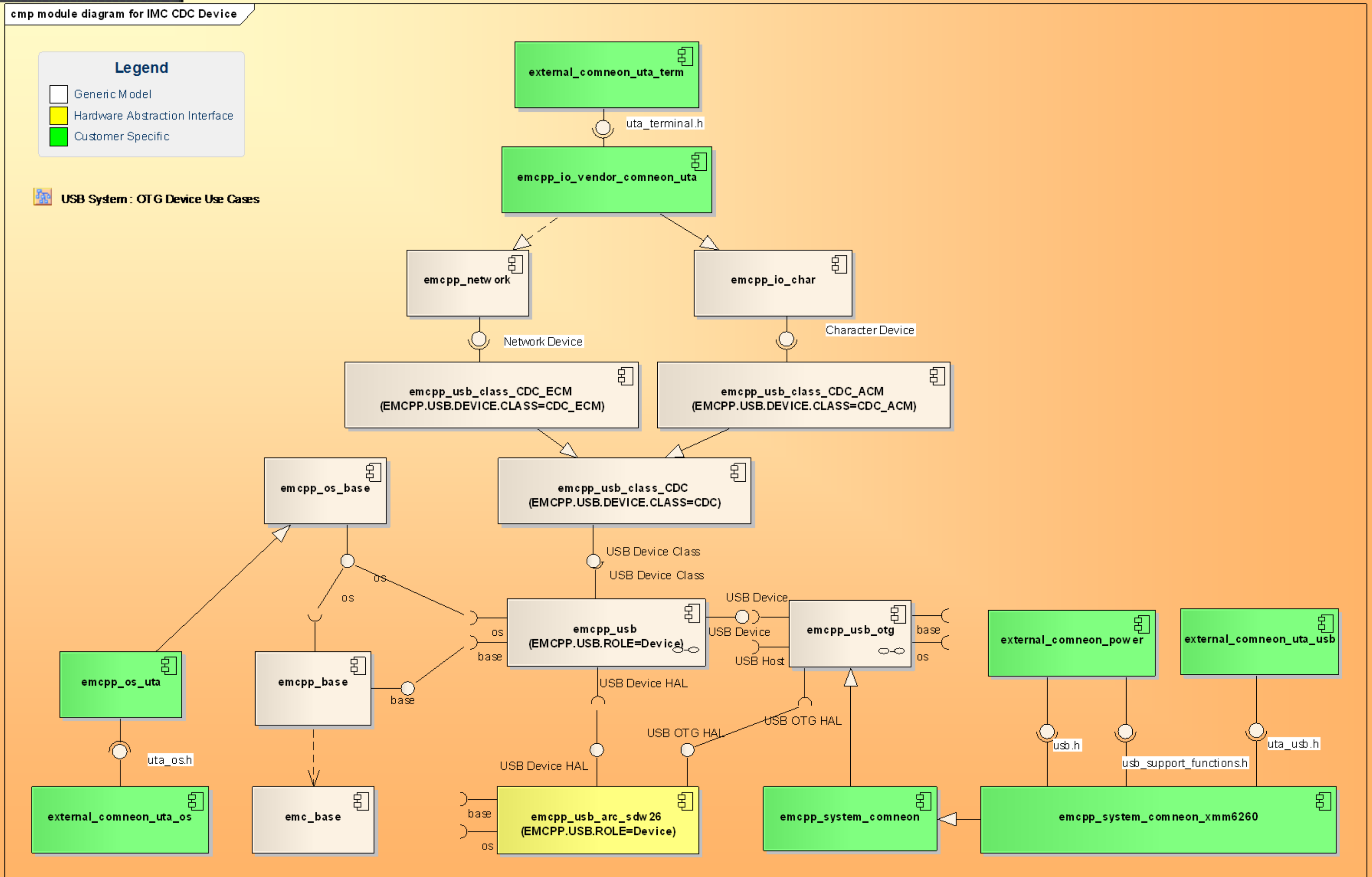
Generic Module Overview (2)

- The emcpp_usb is the central core module implementing the USB protocol. It is configured in the Device role.
- The emcpp_usb_arch_UUU is implementing the hardware abstraction layer. It is also configured in Device role. UUU is a synonym e.g. for SDW26 (Synopsys Design Ware Component 2.6x - 2.9x).
- The emcpp_usb_otg module is controlling the physical hardware which detects the connection as USB Device.

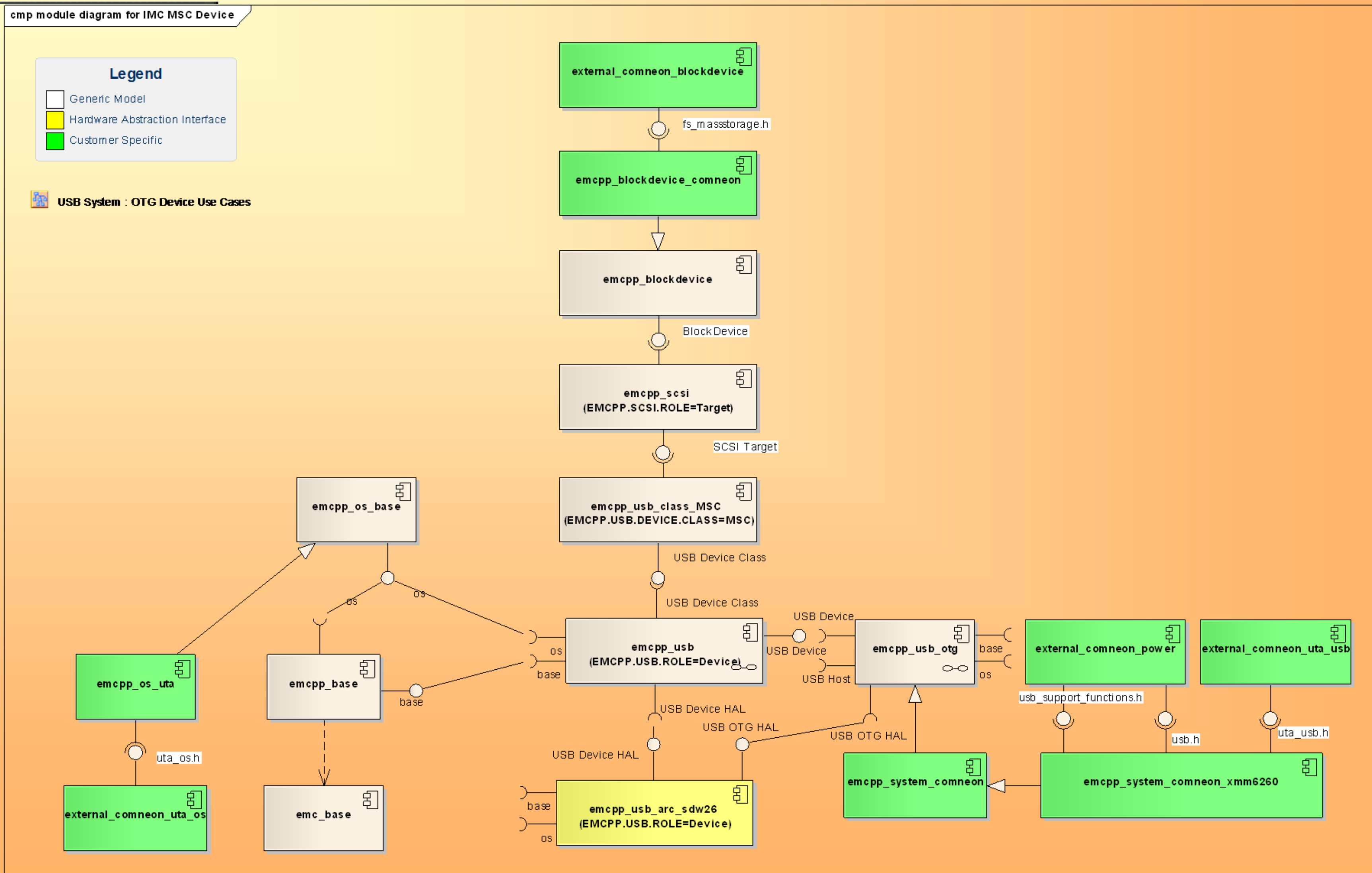
Generic Module Overview (3)

- emcpp_base is providing all C and C++ standard types and operations as well as basic generic data structures
- emcpp_os_base is defining an abstract operating system interface
- For a specific os (e.g. for Nucleus or UTA) the generic os interface is implemented by emcpp_os_OOO (e.g. emcpp_os_nucleus or emcpp_os_uta)
- Every use case requires typically one or more usb device classes. This is done by the module_usb_class_XXX or YYY (e.g. for Mass Storage or Communications Device Class). These modules are using the USB Device Class Interface of the emcpp_usb stack.
- The most USB device classes are wrappers around protocols (e.g. SCSI in Mass Storage case). These protocol specific modules (emcpp_XXX and emcpp_YYY) are interacting with the USB device class modules and will be adopted by a customer specific wrapper module on a customer API.

CDC Device Use Case



MSC Device Use Case



Interfaces USB stack/IMC

- emsys_os_uta uses
  - uta_os.h
  - iui.h
- emcpp_system_comneon uses
  - usb.h (power driver, charging)
  - usb_support_functions.h (new: actually contains serial number API)
  - uta_usb.h (application interface to control USB use cases and to get usb state change notification)
- emcpp_io_vendor_comneon_uta uses
  - uta_terminal.h (serial API for ACM/ECM use cases)
- emcpp_blockdevice_comneon uses
  - fs_massstorage.h (block-device API for MSC use case)
- emcpp_ethernet (directly used by IMC)

Dynamic Behavior

USB Stack Dynamic Model

- ⇨ Allocation of required resources (RAM, tasks) only at the time where it is required
- ⇨ Don‘t use resources if USB connection is not established
- ⇨ Synchronization between tasks during start/stop of USB stack required
- ⇨ Task overview:

Task Name	*Default Stack Size*	*Default Priority*	*Owner*
USB_Otg	*1024*	*EMCPP_OS_FEATURE_DEFAULT_PRIORITY_MEDIUM*	*Singleton*
USB_Dev	*1024*	*EMCPP_OS_FEATURE_DEFAULT_PRIORITY_MEDIUM*	*OTGStack*
USB_Ctrl	*1024*	*EMCPP_OS_FEATURE_DEFAULT_PRIORITY_MEDIUM*	*USBDevice*
USB_Ifc[n]	*Depends on use case*	*Depends on use case*	*USBDevice*

Dynamic Behaviour at USB Connect

- Basically, there is always one task running (USB_Otg)
- There exist several OTG Stacks
  - Responsible for handling Vbus detection/removal
  - Create/destroy required child-tasks
- VBus detection not part of USB stack, uses callback registration function as defined in usb.h:

```
USBPOW_Error
USBPOW_RegisterVbusStatusCb(USBPOW_InterfaceAdapter * p_context,
                            USBPOW_VbusUserHandle * p_handle);
```

- Done at the very early beginning within emcpp::OTGstack::start()
- From callback context, an event is sent to USB_Otg task:
  - OTGStack::CONNECTED_TO_HOST: start USB Device
  - OTGStack::DISCONNECTED_FROM_HOST: stop USB Device

Dynamic Behaviour at USB Connect

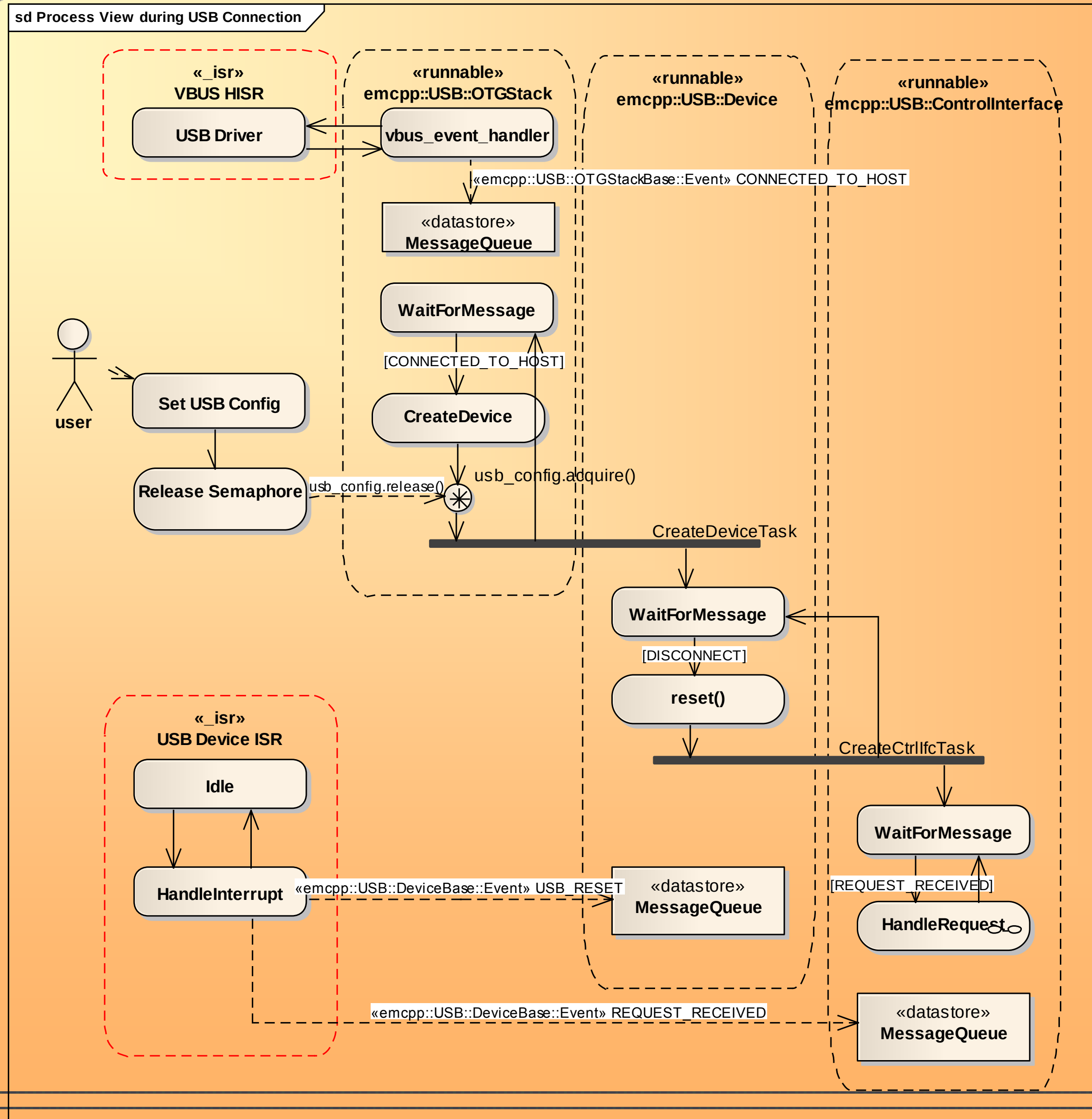
- USB_Otg task is waiting at message box
- If event OTGStack::CONNECTED_TO_HOST is received, it creates the USB Device instance
- Leads to calling the following functions at OTGStack:
  - `emcpp::OTGStack::start_device_prefix`
  - **`emcpp::OTGStack::start_device_infix`**
  - `emcpp::OTGStack::start_device_postfix`
- A Semaphore (usb_config) is acquired in start_device_infix if no USB configuration is selected yet. Allows implementation of „ask-mode“ at Uta-USB:

```
DeviceBase* OTGStack::start_device_infix()
{
  ..
  if (usb_config() == UTA_USB_ENUM_CONFIG_NONE) {
    /* asking mode: wait until a configuration is set (the UtaUsbSetConfig() is
     * called with a proper parameter value) e.g. as a result of user
     * interaction with the device...*/
    usb_config_acquire();
  }
```

USB Connect Sequence

- ⇨ If USB config is selected, semaphore can be acquired, and the USB_Otg task continues creating the USB Device task (USB_Dev) and USB Control Interface Task (USB_Ctrl)
- ⇨ Leads to turning on D+ pull-up resistor
- ⇨ Device detected by host and enumerated

USB Connect Sequence



USB Disconnect Sequence

- Disconnect can be forced via cable disconnect or UtaUsbSetConfig(UTA_USB_ENUM_CONFIG_NONE)
  - Cable disconnect: From callback context, the event OTGStack::DISCONNECTED_FROM_HOST is sent to USB_Otg task
  - UTA_USB_ENUM_CONFIG_NONE: calls soft_disconnect() function in OTGStack, which also sends the OTGStack::DISCONNECTED_FROM_HOST to USB_Otg task
- If event OTGStack:: DISCONNECTED_FROM_HOST is received, it destroys the USB Device instance
- Leads to calling the following functions at OTGStack:
  - **`emcpp::OTGStack::stop_device_prefix`**
  - `emcpp::OTGStack::stop_device_infix`
  - `emcpp::OTGStack::stop_device_postfix`

USB Disconnect Sequence

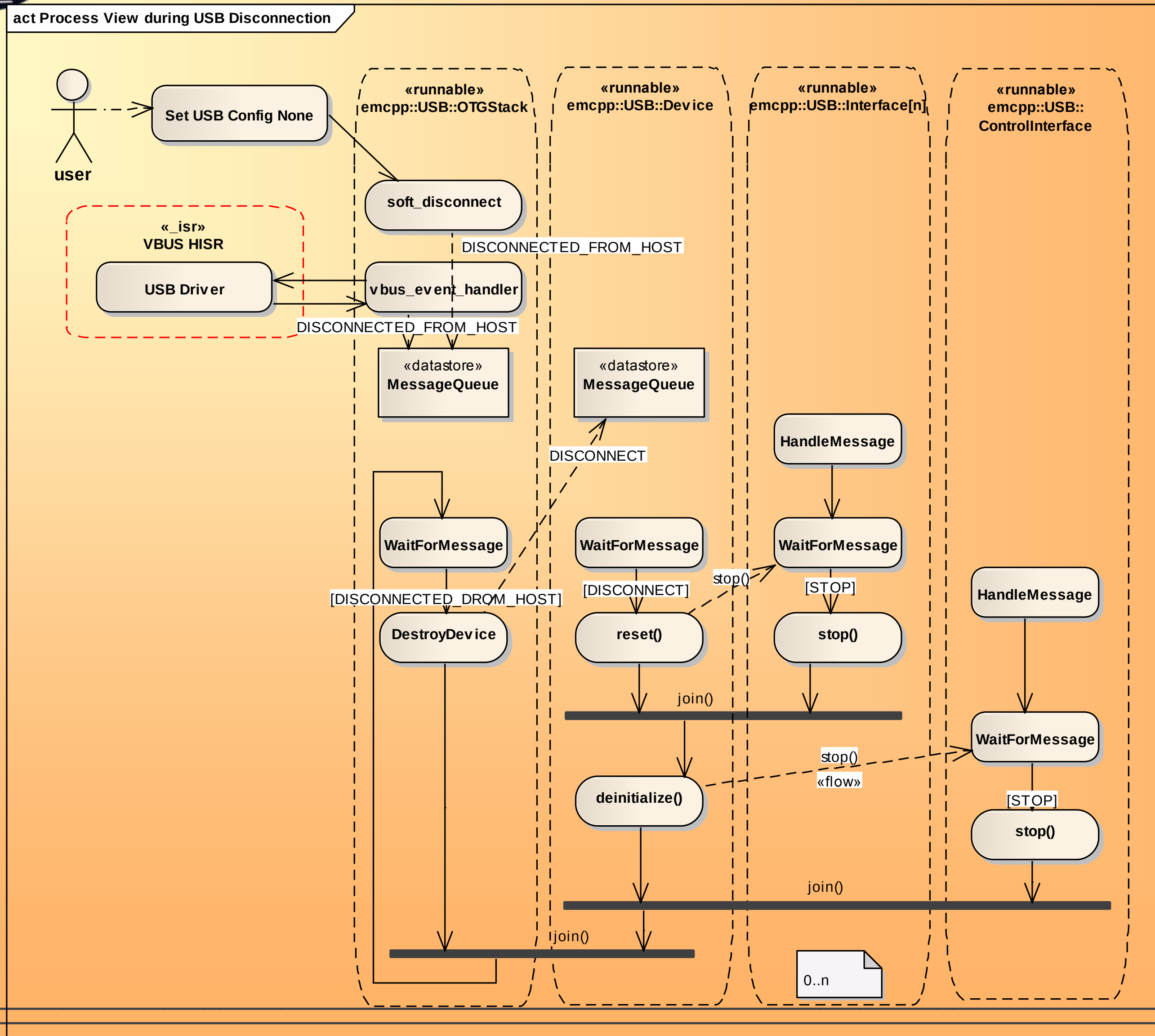
- DISCONNECT message sent to Device task. The disconnect() function call is blocking until the event was handled

```
bool OTGStack::stop_device_prefix()
{
  if (device()) {
    UtaUsbNotificationCb(UTA_USB_ENUM_EVENT_DISCONNECTED);
    device()->disconnect();
  }
  safe_delete(_device_thread);

  return true;
}
```

- All child tasks are finished in the opposite direction as they were created

USB Disconnect Sequence



Changes for 70xx/63xx

- New USB Stack architecture introduced for 70xx/63xx and following platforms
- emsys OS abstraction revised to fit the needs of asynchronous design
- Requirements and goals:
  - Synchronization via messages (avoid using Semaphores for synchronization)
  - Reduce number of required threads (e.g. for CDC interfaces)
  - Reduce CPU load
  - Limit USB hardware access to only one thread (and from ISR context)
  - C-HAL module for USB hardware access (already introduced on 223x)
  - Profile management

Emsys USB Stack

Debugging

Debugging the emsys USB Stack

- ⇨ Most helpful debugging tools to debug USB issues: USB analyzer and memory logs
- ⇨ Most challenging task is to correlate USB trace with memory logs

Logging

Part 1: Legacy emsys memory logging system

Why „Memory logging“?

- ⇨ Besides to the memory logging, all emsys USB sources contain traces which can be activated during compile time
- ⇨ Used log-macros can be redirected to simple „fprintf“, log4c traces, …
- ⇨ A log4c appender exists at emsys side which can re-direct these traces as gate-level message to MobileAnalyser
- ⇨ If active, this has always a not acceptable impact on
  - – the timing of the program execution
  - – Task stack load
  - – if traced via USB COM port to MobileAnalyser, the tracing of USB information via USB can lead to dead-locks
- ⇨ Usually, this logging concept is not active because of the above impact
- ⇨ Logging to memory allows tracing of most important information with almost no impact on timing and used task stack load

Emsys Memory Logging Concept

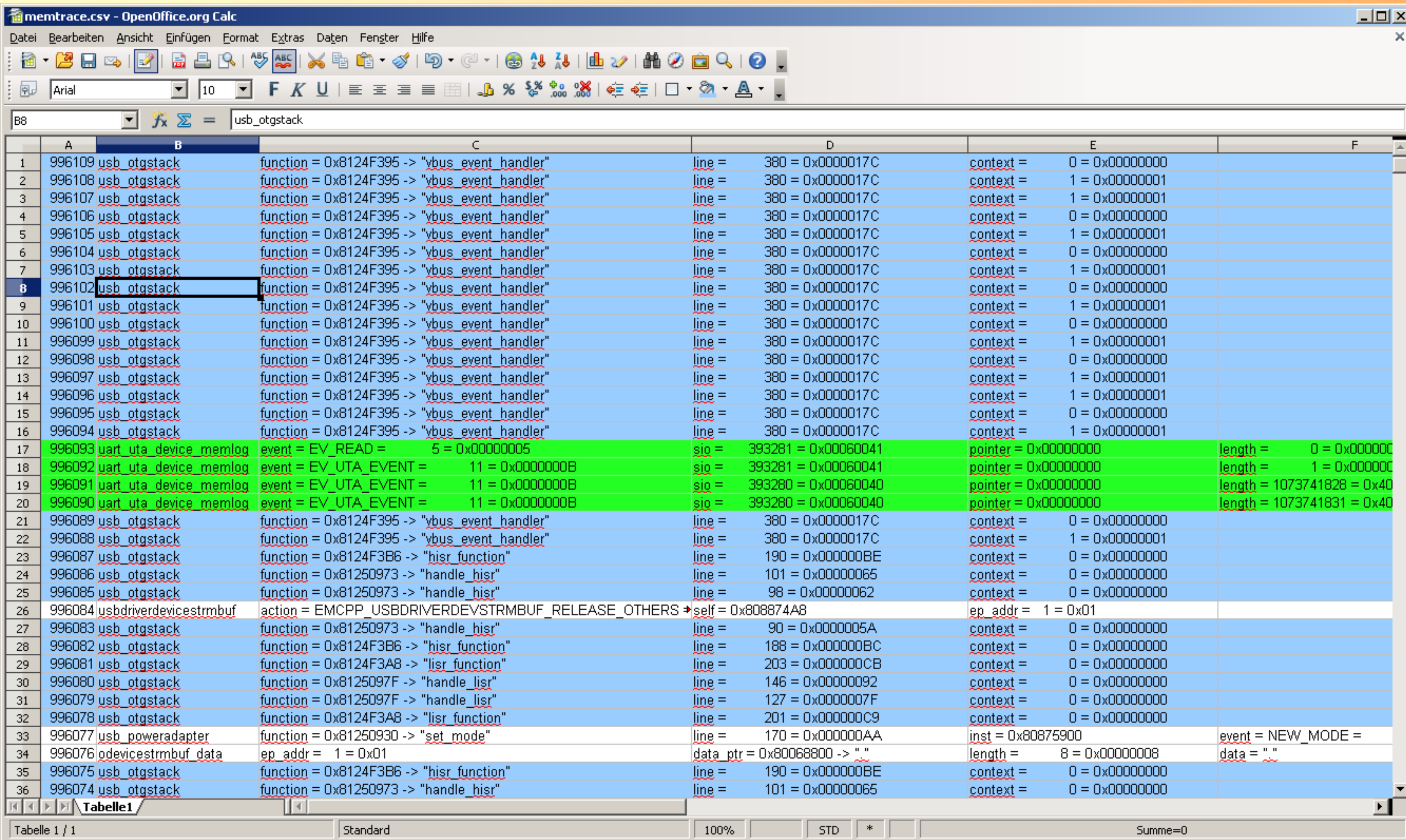
- Every module contains memory logs which can be activated/deactivated at compile time
- Is always a ring-buffer of **module-specific** structures
- Requires most of the time structure declaration to understand and interpret memory logs
- Note: No Synchronisation! (because can also be called from ISR context)
- Contains unique log count to allow sorting the memory logs of different modules in a chronological order
- The symbol name does always include a pattern “emc[pp]*_memlog”
- Example of emcpp::USB::DeviceBase event memory log:

```
emcpp::USB::emcpp_memlog_usb_devicebase_event         = (
  (log_count =         54 = 0x00000036, event = SUSPEND =           5 = 0x00000005, location = 0x810C2788 -> "send"),
  (log_count =         58 = 0x0000003A, event = SUSPEND =           5 = 0x00000005, location = 0x810C25A0 -> "run"),
  (log_count =         86 = 0x00000056, event = RESET =           2 = 0x00000002, location = 0x810C2788 -> "send"),
  (log_count =         90 = 0x0000005A, event = RESET =           2 = 0x00000002, location = 0x810C25A0 -> "run"),
  (log_count =        135 = 0x00000087, event = REQUEST_RECEIVED =           6 = 0x00000006, location = 0x810C2788 -> "send"),
  (log_count =        139 = 0x0000008B, event = REQUEST_RECEIVED =           6 = 0x00000006, location = 0x810C25A0 -> "run"),
  (log_count =        174 = 0x000000AE, event = STATUS_STAGE_FINISHED =           7 = 0x00000007, location = 0x810C2788 -> "send"),
  (log_count =        175 = 0x000000AF, event = STATUS_STAGE_FINISHED =           7 = 0x00000007, location = 0x810C25A0 -> "run"),
```

Emsys Memory Logging Concept

- Analyzing an issue sometimes requires to have all available memory logs (e.g. connect/disconnect/suspend/resume issues)
- Dumping all memory logs possible with emsys cmm-script “memlog.cmm”
  - create directory c:\tmp\t32_log
  - in T32: >do memlog save
  - zip’d t32_log folder can be forward to emsys. If ever possible with correlating USB trace
- A complete set of memory logs can be post-processed using a proprietary tool “ProcessMemlogFiles.exe”. To be executed in the same folder where the memory logs are located
- Result is a new file “memtrace.csv”, which is a chronological sorted coma-separated list of log entries

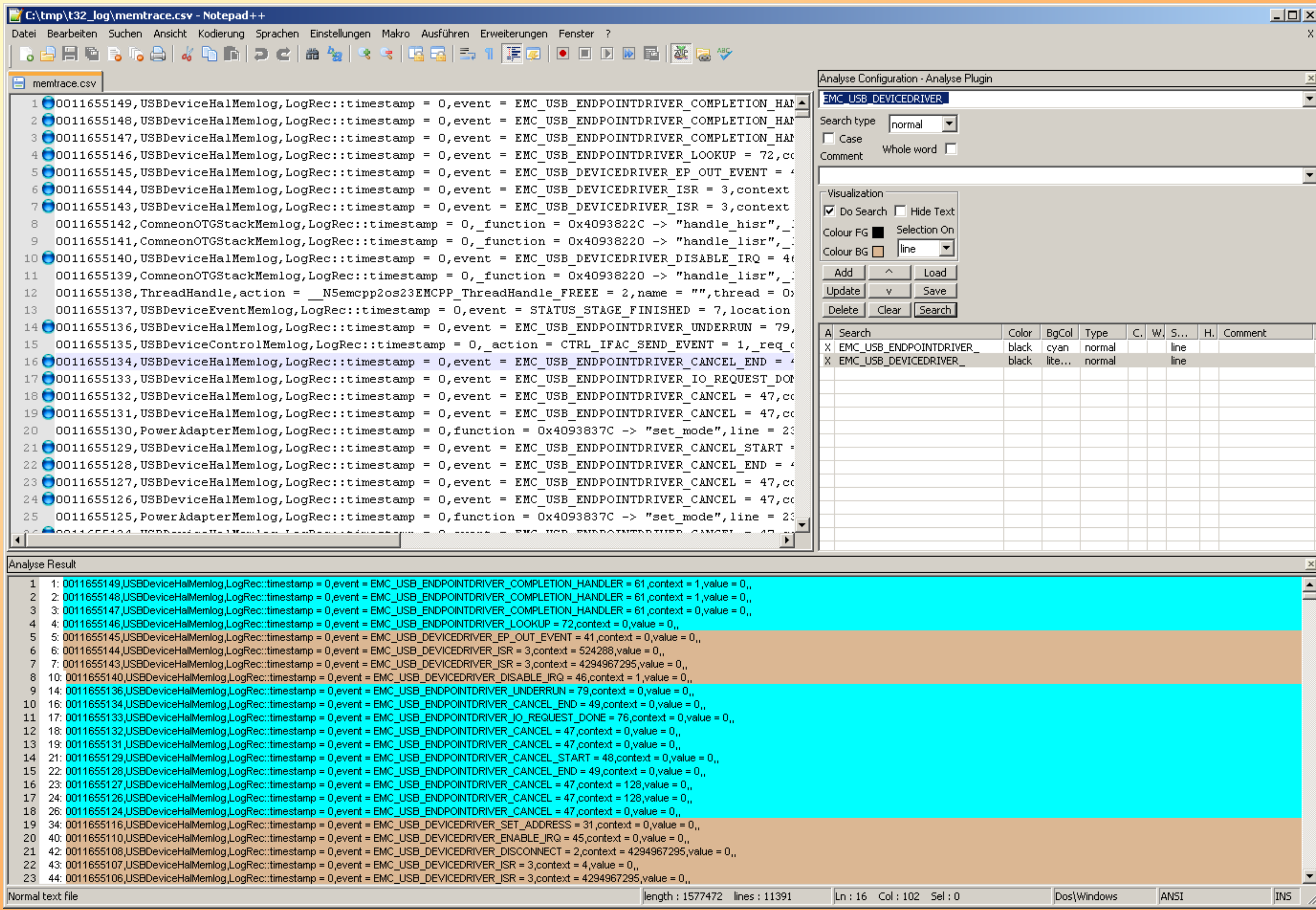
Analysis with “Openoffice Calc”



	A	B	C	D	E	F
1	996109	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
2	996108	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
3	996107	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
4	996106	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
5	996105	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
6	996104	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
7	996103	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
8	996102	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
9	996101	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
10	996100	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
11	996099	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
12	996098	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
13	996097	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
14	996096	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
15	996095	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
16	996094	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
17	996093	uart_uta_device_memlog	event = EV_READ = 5 = 0x00000005	sio = 393281 = 0x00060041	pointer = 0x00000000	length = 0 = 0x000000C
18	996092	uart_uta_device_memlog	event = EV_UTA_EVENT = 11 = 0x0000000B	sio = 393281 = 0x00060041	pointer = 0x00000000	length = 1 = 0x000000C
19	996091	uart_uta_device_memlog	event = EV_UTA_EVENT = 11 = 0x0000000B	sio = 393280 = 0x00060040	pointer = 0x00000000	length = 1073741828 = 0x40
20	996090	uart_uta_device_memlog	event = EV_UTA_EVENT = 11 = 0x0000000B	sio = 393280 = 0x00060040	pointer = 0x00000000	length = 1073741831 = 0x40
21	996089	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 0 = 0x00000000	
22	996088	usb_otgstack	function = 0x8124F395 -> "vbus_event_handler"	line = 380 = 0x0000017C	context = 1 = 0x00000001	
23	996087	usb_otgstack	function = 0x8124F3B6 -> "hisr_function"	line = 190 = 0x000000BE	context = 0 = 0x00000000	
24	996086	usb_otgstack	function = 0x81250973 -> "handle_hisr"	line = 101 = 0x00000065	context = 0 = 0x00000000	
25	996085	usb_otgstack	function = 0x81250973 -> "handle_hisr"	line = 98 = 0x00000062	context = 0 = 0x00000000	
26	996084	usbdriverdevicestrmbuf	action = EMCPP_USBDRIVERDEVSTRMBUF_RELEASE_OTHERS	self = 0x808874A8	ep_addr = 1 = 0x01	
27	996083	usb_otgstack	function = 0x81250973 -> "handle_hisr"	line = 90 = 0x0000005A	context = 0 = 0x00000000	
28	996082	usb_otgstack	function = 0x8124F3B6 -> "hisr_function"	line = 188 = 0x000000BC	context = 0 = 0x00000000	
29	996081	usb_otgstack	function = 0x8124F3A8 -> "lisr_function"	line = 203 = 0x000000CB	context = 0 = 0x00000000	
30	996080	usb_otgstack	function = 0x8125097F -> "handle_lisr"	line = 146 = 0x00000092	context = 0 = 0x00000000	
31	996079	usb_otgstack	function = 0x8125097F -> "handle_lisr"	line = 127 = 0x0000007F	context = 0 = 0x00000000	
32	996078	usb_otgstack	function = 0x8124F3A8 -> "lisr_function"	line = 201 = 0x000000C9	context = 0 = 0x00000000	
33	996077	usb_poweradapter	function = 0x81250930 -> "set_mode"	line = 170 = 0x000000AA	inst = 0x80875900	event = NEW_MODE =
34	996076	odevicestrmbuf_data	ep_addr = 1 = 0x01	data_ptr = 0x80068800 -> "."	length = 8 = 0x00000008	data = "."
35	996075	usb_otgstack	function = 0x8124F3B6 -> "hisr_function"	line = 190 = 0x000000BE	context = 0 = 0x00000000	
36	996074	usb_otgstack	function = 0x81250973 -> "handle_hisr"	line = 101 = 0x00000065	context = 0 = 0x00000000	

Analysis with Notepad++

- ⇨ Usage of "Analyse Plug-in"
- ⇨ Allows definition of filter rules
- ⇨ Filter rules can be saved and loaded



Logging

Part 2: emsys Tlog memory logging system

Tlog: Basics

- Messages logged to memory
- String messages saved as 32bit hashes
- Post-processing required for human-readable format
- Extraction script and post-processing tool provided by emsys

Tlog: Involved files

- Memory dumps
  - Several binary files (**`*.bin`**)
  - Extracted via CMM script from Lauterbach Trace32
- Helper files
  - **`log_hash_file.txt`** – maps hash values to messages
    (re-generated upon user request before build process, part of every delivery)
  - **`hierarchy.txt`** – describes loggers and their hierarchy
    (generated by CMM script when extracting memory dumps)
  - **AXF** file – optionally helps to decode enumeration values and static strings
- Output file
  - Human-readable plain text file (e.g., **`usb.log`**)
  - Generated by post-process tool from above input files

Tlog: Plain text extraction workflow

1

Lauterbach Trace32
with emsys tlog.cmm script

AXF

log_hash_file.txt

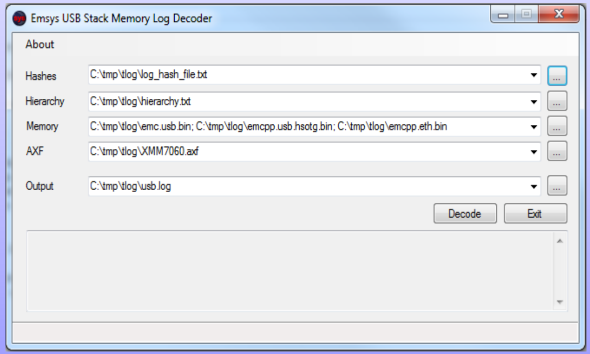
memory dumps (*.bin)

hierarchy.txt

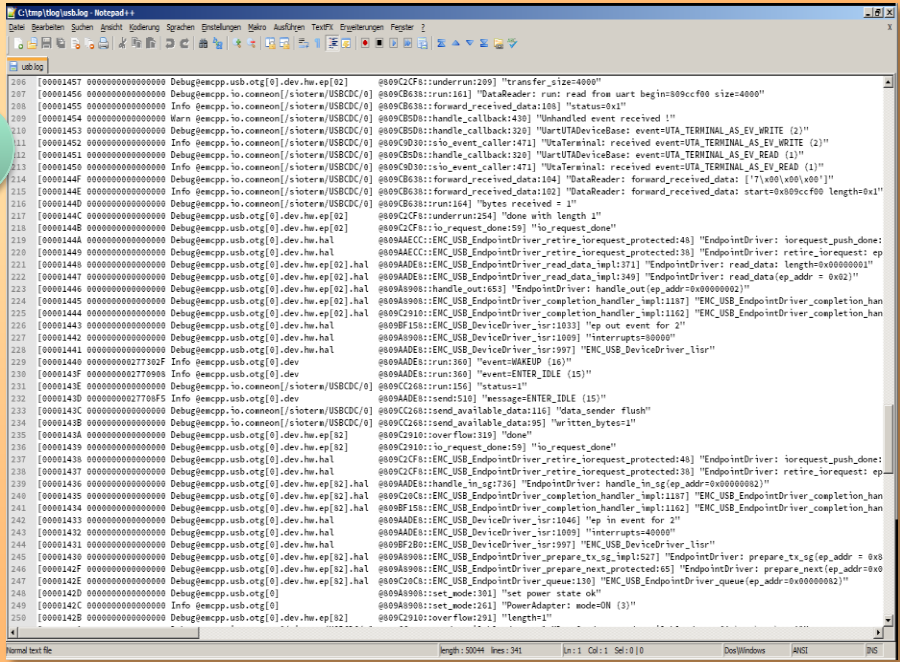
2

Tlog post-process tool

GUI
or
CLI



3



plain text file (usb.log)

Tlog documentation

Up-to-date user's guide:

https://www.emsys.de/polarion/#/project/Global/wiki/Logging

Emsys USB Stack

UtaTerminal Interface

UtaTerminal Interface

- ⇨ as defined in uta_inc/core/uta_terminal.h
- ⇨ 2 separate tasks are created to allow full-duplex communication
  - – DataReader task (DtaRdr)
  - – DataSender task (DtaSndr)
- ⇨ Tasks are created after CDC Data interface was created, and they are stopped and deleted if the CDC Data interface was removed
- ⇨ Zero-copy mechanism used
- ⇨ Callback registered at UtaTerminal for event handling:
  `UartUTADeviceBase::handle_callback()`

DataReader Program Flow

- DataReader is responsible for the direction PC->UtaTerminal (**reads** data from USB)
- 2 modes are supported:
  - UTA_TERMINAL_MODE_STREAMING (default mode, explained in detail)
  - UTA_TERMINAL_MODE_FW_BLOCK
  - UTA_TERMINAL_MODE_xxx (new zero copy mode, implementation in progress)
- For streaming mode, the DataReader task first allocates DMA capable memory via ptf_malloc_dma_buffer() . Size of buffer is configurable in CustomerDescriptorSet (default 16k)
- DataReader task always tries to read data from USB (blocking function call due to synchronous USB Stack API)
- If data received, the data is forwarded to UtaTerminal

DataReader Program Flow

```
void DataReader::run()
{
 ..
 if(UTA_TERMINAL_MODE_STREAMING == write_param.mode) {
    _streamingmode_buffer = (byte*)ptf_malloc_dma_buffer(_uart_buffer_size);
    EMCPP_ASSERT(_streamingmode_buffer, BadAlloc());
    while(!isCanceled()) {
      EMCPP_HW_NS::BufferHandle bh(_streamingmode_buffer, _uart_buffer_size);
      EMCPP_IO_UART_UTA_DEVICE_MEMLOG(UartUTADeviceBase::DATA_READER__DATA_READ_FROM_UART,
                       _device.uta_handle(), bh.begin(),bh.size());
      size_t read_bytes = uart.read(bh); // read data from USB
      EMCPP_IO_COMNEON_LOG1(TRACE, logger,"DataReader::run: got %d bytes from USB", read_bytes);
      if(0 == read_bytes) {
        // transmit error (e.g. disconnect...)
        EMCPP_IO_UART_UTA_DEVICE_MEMLOG(UartUTADeviceBase::DATA_READER__ABORT_1,
            _device.uta_handle(), 0,0);
        break;
      }
      if(forward_received_data(_streamingmode_buffer, read_bytes) < read_bytes) {
        EMCPP_IO_UART_UTA_DEVICE_MEMLOG(UartUTADeviceBase::DATA_READER__ABORT_2,
            _device.uta_handle(), 0,0);
        // write error (e.g. terminal closed...)
        break;
      }
    }
  ..
  }
```

DataReader Program Flow

```
size_t DataReader::forward_received_data(byte* buffer, size_t buffer_length)
{
  if (0 == buffer) {
    return 0;
  }
  UtaInt32 status = UTA_SUCCESS;
  size_t written_bytes = 0;
  while((buffer_length - written_bytes) > 0 && !isCanceled()) {
    UtaTerminalResult watermark;
    byte* start = buffer + written_bytes;
    size_t length = buffer_length - written_bytes;
    ..
    status = UtaTerminalWWrite(_device.uta_handle(), start, length, &watermark);
    SysProfBW(USB_WWrite, status);
    EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_READER__FORWARD_RECEIVED_DATA,
            _device.uta_handle(), start, (word32)status);
    .. // error handling
    written_bytes += status;
  }
  return written_bytes;
}
```

DataSender Program Flow

- ⇨ DataSender is responsible for the direction UtaTerminal->PC (**sends** data to USB)
- ⇨ no dedicated buffer allocated, because the DMA-capable memory is provided by UtaTerminal
- ⇨ 2 UtaTerminal callback events are evaluated for the DataSender:
  - UTA_TERMINAL_AS_EV_READ, a „SEND_DATA“ event is put to the DataSender message queue
  - UTA_TERMINAL_AS_EV_TX_RESET, a „CANCEL_END“ event is put to the DataSender message queue
- ⇨ If DataSender gets “SEND_DATA” event, this data is fetched from UtaTerminal and forwarded via USB
- ⇨ If DataSender gets “CANCEL_END” event, pending USB transfers are stopped

DataSender Program Flow

```
void DataSender::run()
{
  while (!isCanceled()) {
    Event ev(_mq.get());
    EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_EVENT,
          _device.uta_handle(),0,ev);
    if(CANCEL_END == ev){
      EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_CANCEL_END,
          _device.uta_handle(),0,0);
      _device.uart().reset();
      _cancel_transfer=false;
      continue;
    }
    if(SEND_DATA == ev) {
      EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_GET_SEND_DATA_EVENT,
           _device.uta_handle(),0,0);
      EMCPP_IO_COMNEON_LOG0(TRACE, logger,"DataSender::run: got SEND_DATA event");
      int status = send_available_data(_device.uta_handle(), _device.uart(), _uart_buffer_size);
      EMCPP_IO_COMNEON_LOG1(TRACE, logger,"DataSender::run: data sent, status=%x",status);
      if(status < 0  && !_cancel_transfer) {
        EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_STATUS,
           _device.uta_handle(),0,status);
        break;
      }
    } else {
      break;
    }
  }
}
```

DataSender Program Flow

```
void DataSender::run()
{
  while (!isCanceled()) {
    Event ev(_mq.get());
    EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_EVENT,
          _device.uta_handle(),0,ev);
    if(CANCEL_END == ev){
      EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_CANCEL_END,
          _device.uta_handle(),0,0);
      _device.uart().reset();
      _cancel_transfer=false;
      continue;
    }
    if(SEND_DATA == ev) {
      EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_GET_SEND_DATA_EVENT,
           _device.uta_handle(),0,0);
      EMCPP_IO_COMNEON_LOG0(TRACE, logger,"DataSender::run: got SEND_DATA event");
      int status = send_available_data(_device.uta_handle(), _device.uart(), _uart_buffer_size);
      EMCPP_IO_COMNEON_LOG1(TRACE, logger,"DataSender::run: data sent, status=%x",status);
      if(status < 0  && !_cancel_transfer) {
        EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__RUN_STATUS,
           _device.uta_handle(),0,status);
        break;
      }
    } else {
      break;
    }
  }
}
```

DataSender Program Flow

```
int DataSender::send_available_data(UtaIoHdl term_handle, Uart& uart, size_t uart_buffer_size)
{
  size_t transferred_bytes = 0;
  void* buffer = NULL;
  UtaInt32 read_bytes = 0;
  EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__SEND_AVAILABLE_DATA_START,
           term_handle, 0, 0);
  while(0 < (read_bytes = UtaTerminalPRead(term_handle, &buffer, uart_buffer_size)) &&
            !isCanceled()) {
    SysProfBW(USB_PRead, read_bytes);
    EMCPP_ASSERT(buffer, BadError());
    EMCPP_HW_NS::BufferHandle bh(buffer, (size_t)read_bytes);
    EMCPP_IO_UART_UTA_DEVICE_MEMLOG(device_type::DATA_SENDER__SEND_AVAILABLE_DATA,term_handle,
           buffer, read_bytes);
    size_t written_bytes =0;
    if(!_cancel_transfer){
     written_bytes=uart.write(bh);
    }
    transferred_bytes += written_bytes;
    if(STATIC_CAST(UtaInt32,written_bytes) != read_bytes && !_cancel_transfer) {
      // transmit error (e.g. disconnect...)
      return -1;
    }
  }
```

UtaTerminal Memory Logging

- Memory logging is activated by default
- can be deactivated at compile-time by setting
  - EMCPP_IO_COMNEON_UART_UTA_DEVICE_USE_MEMLOG 0
- configurable to create memlog instances per channel
  - EMCPP_IO_COMNEON_UART_UTA_DEVICE_MEMLOG_MULTI_COUNT *n*
  - *n* defines the number of CDC channels. E.g. 7 for 7CDC use case, or 11 for 7CDC use case + HSIC with 4CDC

CDC Test Modes

- ⇨ data consistency and performance can be checked easily using CDC Test modes
- ⇨ 2 Test modes exist:
  - – CDC Bulk Loop-back test mode (emsys)
  - – SIO Loop-back test mode (IMC)
- ⇨ CDC Bulk Loop-back test mode activated by default and available in every platform delivery (not supported on 2130)
- ⇨ SIO Loop-back test mode availability has to be configured at IMC

Emsys USB Stack

Interface to Ethernet-devices

and

CDC/NCM-Implementation

CDC/NCM

- The Communication Class Network Control Model (NCM) Subclass is a protocol by which USB hosts and devices can efficiently exchange Ethernet-Frames.
- It builds upon CDC/ECM protocol and extends it by following features:
  - Multiple Ethernet frames can be aggregated into single USB transfer
  - CDC/NCM functions (devices) can specify how Ethernet frames may be best placed within the transfer to minimize overhead
- Related Documents
  - http://www.usb.org/developers/devclass_docs/NCM10_012011.zip

Interface to Ethernet-devices

- ⇨ Requirements
  - – Asynchronous operation
    - Initiation and completion of operations are separate
    - Queueing and cancellation of pending operations
  - – Zero-Copy(ZC) for incoming traffic
  - – Multi-buffering for input buffers
  - – Buffer constrains propagation and checking
  - – Scatter/Gather (S/G) for outgoing frames
  - – Aggregation (grouping) of outgoing frames
  - – Device management
    - Device identification
    - Dynamic device addition/removal
  - – Power management

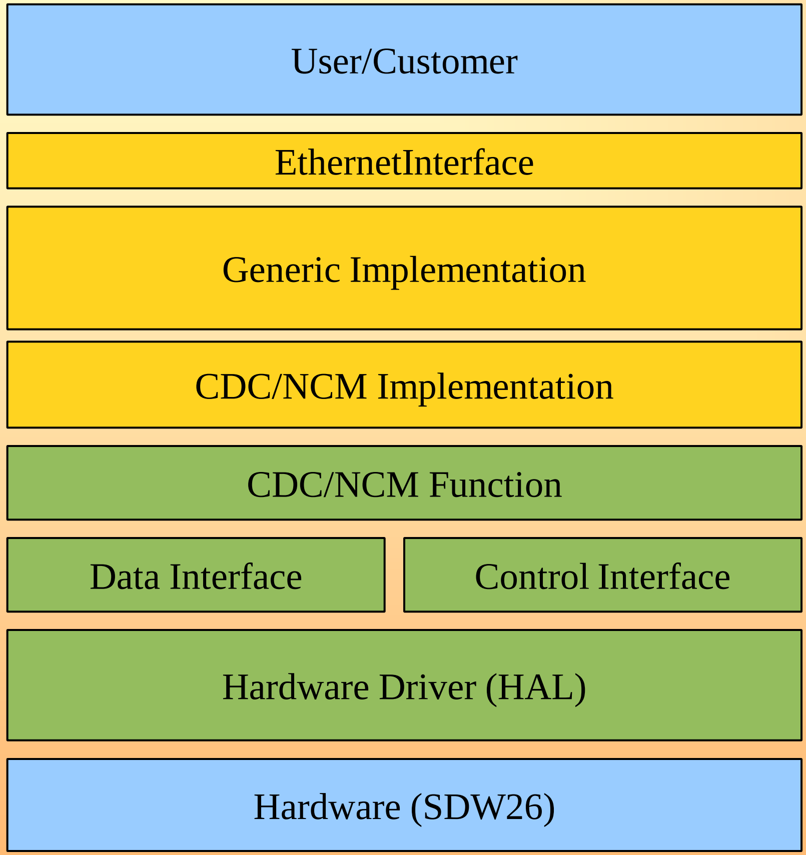
Interface definition (1)

- ⇨ Defined in **emcpp/ethernet/EthernetInterface.hpp**
- ⇨ Defines abstract interfaces
  - – **emcpp::ethernet::EthernetInterface**
    - frame-aggregation and initiation of output transfers
    - initiation of input transfers
    - (de)registering signal-handlers/observers
  - – **emcpp::ethernet::EthernetInterface::Observer**
    - receiving signals on completion of transfers and changes of device-status

Interface definition (2)

- … abstract interfaces (continued)
  - **emcpp::ethernet::EthernetInterface::Registry**
    - Enumerating devices in the system
  - **emcpp::ethernet::EthernetInterface::Registry::Observer**
    - Receiving signals whenever a device was added or removed to/from the system
- Defines (abstract) types
  - **emcpp::ethernet::EthernetInterface::EthernetFrame**
    - A head of a chain of buffers (for S/G stacks)
  - **emcpp::ethernet::EthernetInterface::EthernetFrame::Buffer**
    - A reference to memory chunk holding (a part of) payload bytes

System Overview



- **User/Customer layer** represents an adapter to specific protocol stack or packet routing framework (Packet Buffer Manager = PBM)
- **Generic Implementation** layer represents generic reusable part (EthernetInterfaceAdapter)
- **CDC/NCM Implementation** layer represents specific CDC/NCM protocol implementation
- **CDC/NCM Function** is a *Façade* connecting to the USB-Stack

Layered Design

- **Goal:** Separate the generic, reusable part from the concrete part.
- Generic base-class, implements **emcpp::ethernet::EthernetInterface** abstract interface.
  - defines a less abstract, lower-level interface
  - **(emcpp::ethernet::EthernetInterfaceAdapter)**
- Concrete implementations, deriving from generic base-class.
  - CDC/NCM implementation
  - **(emcpp::USB::CDC::NCM::NcmEthernetInterfaceAdapter)**
  - More concrete implementations will (probably) follow...

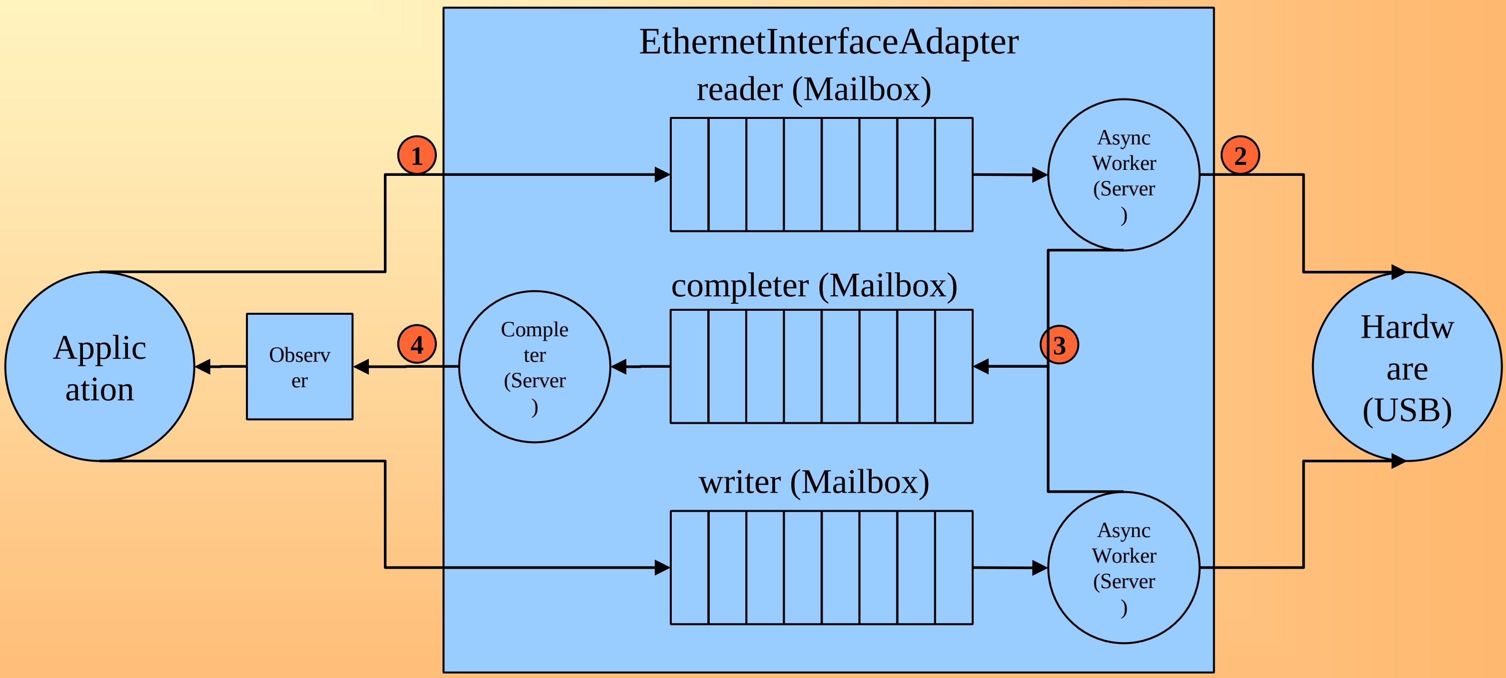
Generic Implementation (1)

- ⇨ Uses the *Mailbox* abstraction (concept from the new USB-stack design)
  - *Mailbox* realize a generic **message-queue for arbitrary messages.**
  - Defines an interface for a **message-handler**, *Server*.
  - It can also optionally control the **thread** running one or more *Servers* or connect to an existing thread.
  - Furthermore *Mailbox* defines a way to look ahead in the message-queue to implement **message-precedence** rules.
  - *Mailbox* supports realizing of **producer/consumer scenarios**, when messages are forwarded between multiple *Mailboxes*.
  - *Mailboxes* can be linked to implement **error-handling and clean-up**
- ⇨ See **emcpp/os/Mailbox.hpp**

Generic Implementation (2)

- **Goal:** Overlap **I/O** (Hardware) and **completion** (CPU) phases of an asynchronous operation (request) as much as possible
  - Progressing I/O has higher priority (short operation) as completion (longer operation).
- **EthernetInterfaceAdapter** uses three *Mailboxes*...
  - one for queueing input-requests (reader),
  - one for queueing output-requests (writer) and
  - one for queueing requests, that are waiting for completion (completer)
- **EthernetInterfaceAdapter::AsyncWorker** is a *I/O-Server*
- **EthernetInterfaceAdapter::Completer** is a *Completion-Server*
- The *Servers* are **generic**, and Requests are **concrete**.
- See **emcpp/ethernet/private/EthernetInterfaceAdapter.hpp**

Generic Implementation (3)



CDC/NCM Implementation

- ⇨ Defines parameters
  - – Buffer (and other) constrains
- ⇨ Extends generic methods
  - – Cancellation of pending requests
  - – Link-Status changing
- ⇨ Provides implementation for abstract methods
  - – Creation of I/O-requests
- ⇨ Implements Requests
  - – **emcpp::USB::CDC::NCM::InputRequest**
  - – **emcpp::USB::CDC::NCM::NcmOutputRequest**
- ⇨ See **emcpp_usb_cdc_ncm_NcmEthernetInterfaceAdapter.cpp**

CDC/NCM Implementation (2)

- The CDC/NCM Ethernet-interface uses a *Function* to communicate with the USB-Stack.
- The *CDC/NCM Function* provides an interface
  - to access configuration parameters,
  - to access the data input/output endpoints and
  - the event/interrupt endpoint.
- The *Function* hides the CDC/NCM control specific parts from the CDC/NCM Ethernet-device, which implements the protocol parts.
- See **.../NetworkControlModelCommunicationsFunction.hpp**

Testing ethernet-devices

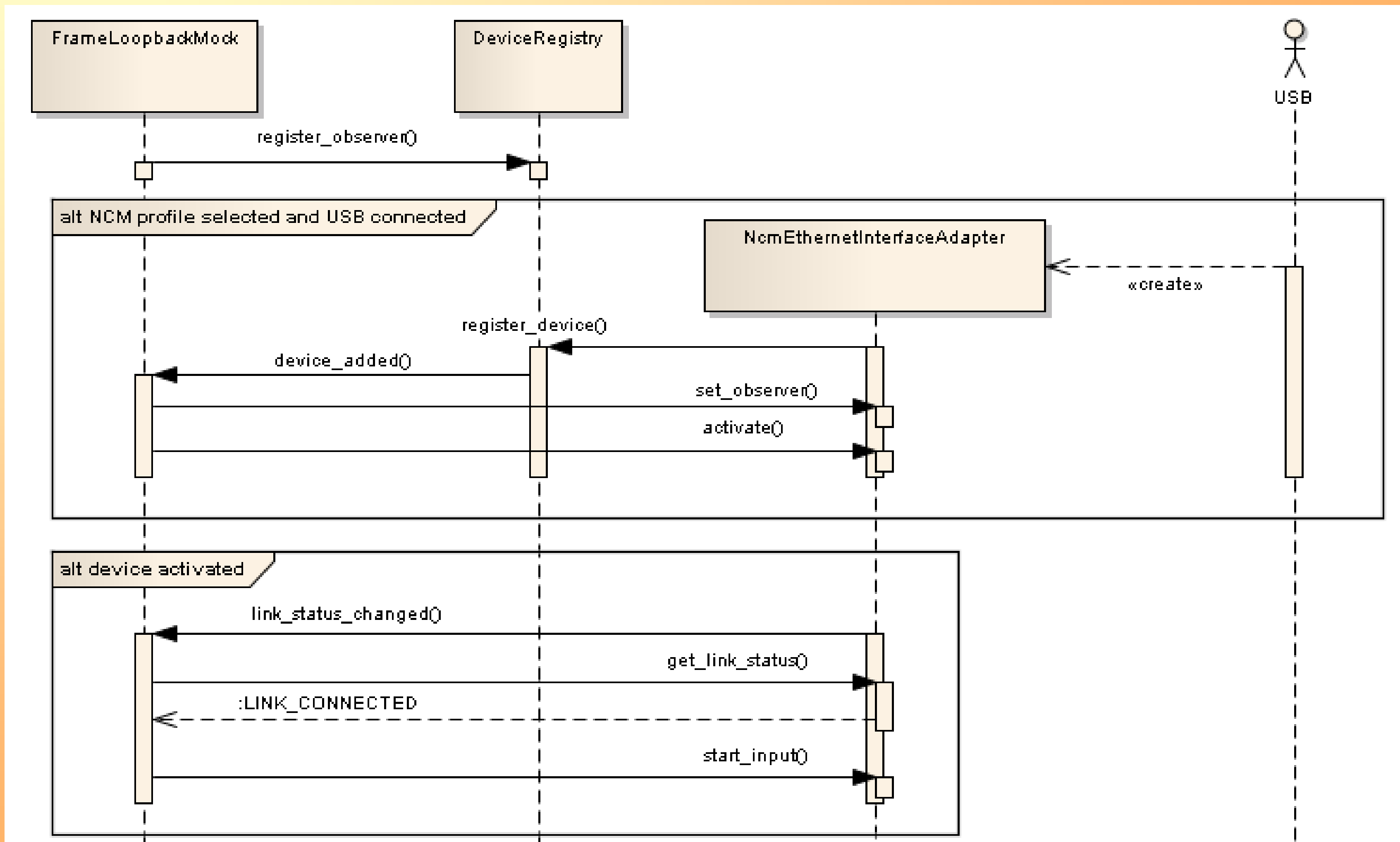
- Emsys provides adapters (mocks) connecting to Ethernet-devices to provide test and diagnostic functionality
  - **FrameLoopbackMock**: loops incoming frames back to the network (host) by exchanging the source and the destination addresses. It intercepts only specific test-frames and drops all other frames. This mock is used for **functional tests**.
  - **SourceSinkMock**: tries to receive all incoming traffic as fast as possible (multi-buffering) and generates outgoing traffic to be intercepted by the host. This mock is used for **bandwidth tests**.
- Adapters also document the usage of the interface

Interfacing to Ethernet-devices

⇨ Usecases

- UC1: Lookup and initialization when new device was added
- UC2: Handling of completion of input-request
- UC3: Queueing of outgoing frames
- UC4: Handling of completion of an output-request
- UC5: Device removal

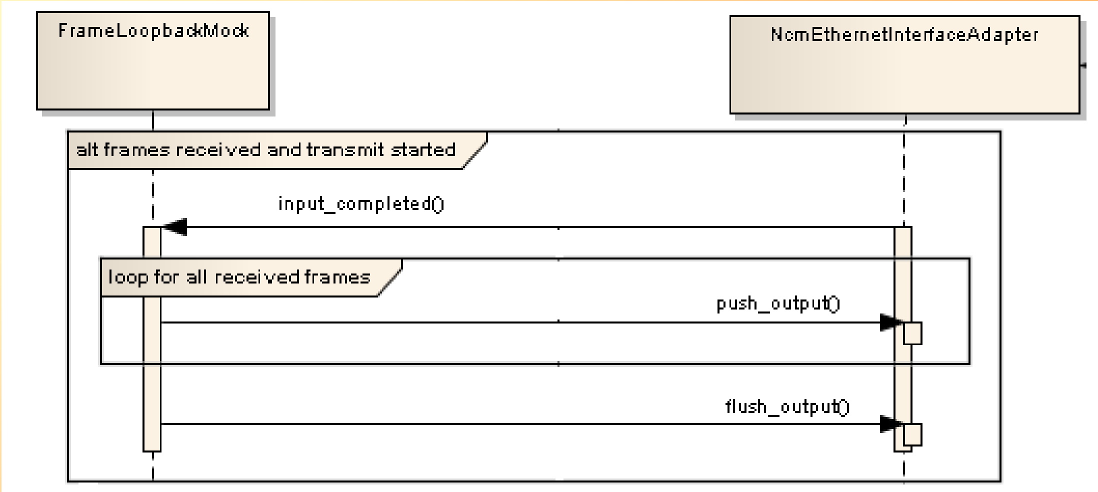
UC1: Lookup and Initialization (1)



UC1: Lookup and Initialization (2)

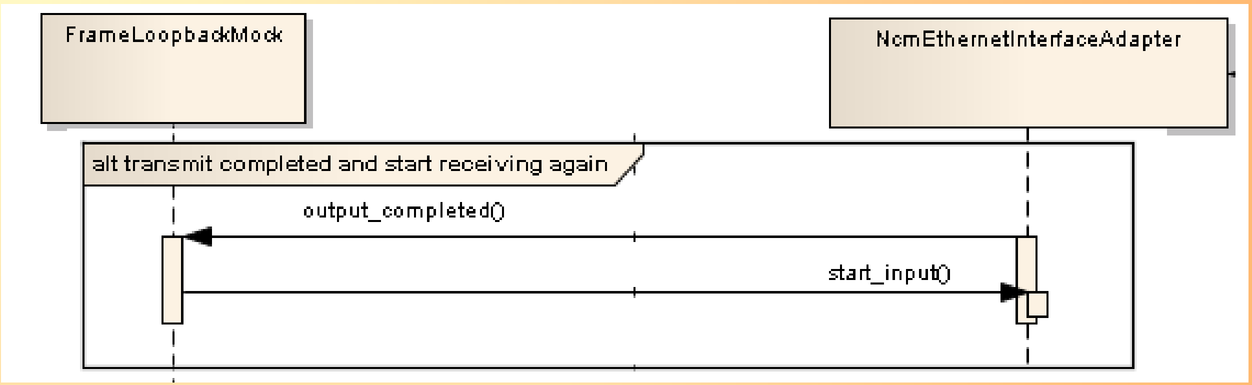
- ⇨ Get a reference (pointer) to specific **EthernetInterface**
  - Use **EthernetInterface::Registry** to enumerate devices in the system
  - When device wasn't found in the registry, register an **EthernetInterface::Registry::Observer** to get notified when devices are added to the system
  - Register **EthernetInterface::Observer** to the device to get notified upon device events.
  - Activate device with **EthernetInterface::activate**
  - Upon **link_status_changed** notification, make device ready for receiving incoming frames by providing one or more input-buffers to **start_input**.

UC2 and UC3: Loopback



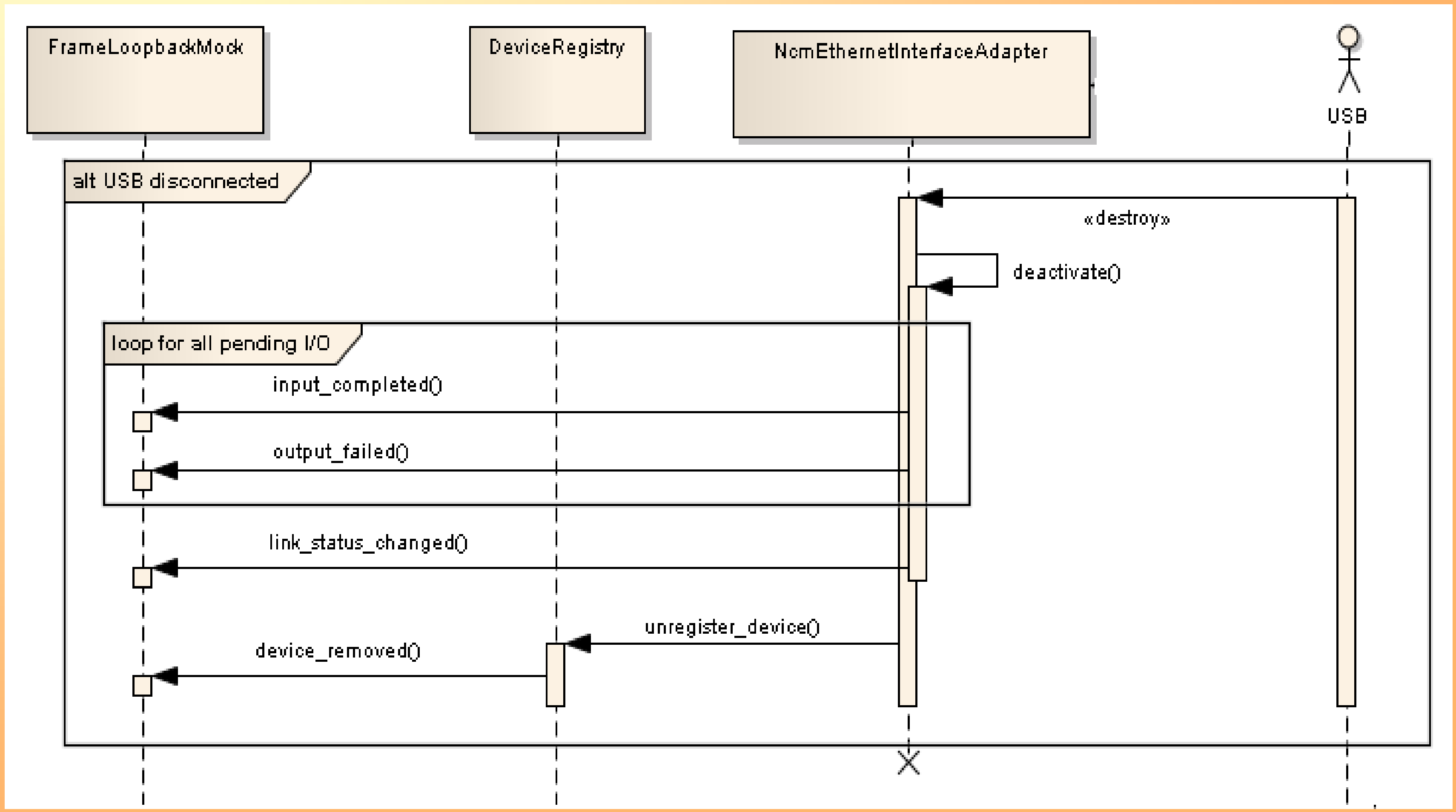
- Upon completion of an input-request (initiated by **start_input**) the **input_completed** gets called on the registered observer object.
  - This provides an *Iterator* to frames, put in specific input-buffer.
- In case of **FrameLoopbackMock,** received frames will be pushed to the output-queue by calling **push_output** and **flush_output** after the last frame, which will result in one (or more) output-requests.

UC4: Completion of Output-Request



- Upon successful completion of an output-request (initiated by **push_output** when an *autoflush* condition was met or **flush_output**) the **output_completed** gets called on the registered observer object.
  - This provides an *Iterator* to frames that were successfully put on the network.
- In case of **FrameLoopbackMock,** the input-buffer can be reused only when all frames was successfully looped back. Then, the input-buffer can be re-used for a new input-request, which can be initiated by **start_input** (zero-copy loopback).

UC5: Device Removal (1)



UC5: Device Removal (2)

- ⇨ Upon device removal the device will...
  - giveback all pending input-buffers (releasing the memory back to the user/initiator) by calling **input_completed**,
  - giveback all pending output frames by calling **output_failed,**
  - notify the registered observer object by calling **link_status_changed** about being “disconnected” and not ready to start any more input-transfers or accept outgoing frames.
- ⇨ Before the device reference gets invalid the **EthernetInterface::Registry** will notify all registered **EthernetInterface::Registry::Observer** about this by calling **device_removed.**
- ⇨ Afterwards, it should be safe to delete the device an release all resources acquired for its function.

Emsys USB Stack

Block-device Interface

Block-device Interface

- IMC Block-device interface (fs_massstorage.h) is used in emcpp_blockdevice_comneon module
- calls to block-device interface are running from SCSI task context, which is using the Mass-Storage Interface task resource (e.g. USB_Ifc0)
- no calls from block-device to USB stack (synchronous API, no callbacks)
- *FS_MS_Slave_GetDeviceType()* function call to determine the type of the block device:
  - *FS_MMCSD_DEVICE*: external MMC/SD device
  - *FS_INT_USER_DEVICE*: internal drive, e.g. FAT partition
  - *FS_USBRO_DEVICE*: read-only device, e.g. contianing a CDROM image for required for auto-install capability

Block-device Interface

➩ Memory logs available for at function calls (read/write/open/close/…) to allow the tracing of the status of these calls

```
emcpp::bd::emcpp_memlog_bd_comneon = (
   (
     log_count =       2604 = 0x00000A2C,
     timestamp =                 39192 = 0x0000000000009918,
     event     = __N5emcpp2bd34PhysicalGenericDrive_do_initializeE =          17 = 0x00000011,
     lun       =             0 = 0x00000000,
     context   =             0 = 0x00000000),
   (
     log_count =       2605 = 0x00000A2D,
     timestamp =                 39192 = 0x0000000000009918,
     event     = __N5emcpp2bd39PhysicalGenericDrive_do_initialize_doneE =          18 = 0x00000012,
     lun       =             0 = 0x00000000,
     context   =             0 = 0x00000000),
   (
     log_count =       2606 = 0x00000A2E,
     timestamp =                 39192 = 0x0000000000009918,
     event     = __N5emcpp2bd19COMNEON_FFS_MS_OPENE =           1 = 0x00000001,
     lun       =             0 = 0x00000000,
     context   =             0 = 0x00000000),
   (
     log_count =       2607 = 0x00000A2F,
     timestamp =                 39193 = 0x0000000000009919,
     event     = __N5emcpp2bd26COMNEON_FFS_MS_OPEN_RESULTE =           2 = 0x00000002,
     lun       =             0 = 0x00000000,
     context   =             0 = 0x00000000),
```

Emsys USB Stack

Interface to USB Power Module

Interface to USB Power Module

- ⇨ defined in mhw_drv_inc/inc/connectivity/usb.h
- ⇨ Used within emsys "PowerAdapter" module
- ⇨ Used to control power states of USB Core(s) in case of suspend/resume/connect/disconnect
- ⇨ emsys stack must ensure to access USB registers only if USB power driver is set to USBPOW_ON
- ⇨ Actually, this is ensured by emsys PowerSharedGuards used by USB register classes
  - access to hardware **only** made using register classes
  - so it is ensured that registers will only be accessed if call to power module with USBPOW_ON was made before

Interface to USB Power Module

⇨ Memory logs available

```
emcpp::System::Comneon::emcpp_memlog_usb_poweradapter = (
    (
      log_count =          2 = 0x00000002,
      function  = 0x81250258 -> "set_mode",
      line      =        170 = 0x000000AA,
      inst      = 0x808749A8,
      event     = NEW_MODE =           1 = 0x00000001,
      mode      = IDLE =           1 = 0x00000001,
      success   = TRUE ),
    (
      log_count =         12 = 0x0000000C,
      function  = 0x81250258 -> "set_mode",
      line      =        170 = 0x000000AA,
      inst      = 0x808749A8,
      event     = NEW_MODE =           1 = 0x00000001,
      mode      = ON =           3 = 0x00000003,
      success   = TRUE ),
```

Emsys Module Responsibility

Emsys CEO: Dr. Karsten Pahnke

USB HSIC
Enrico Schmidtke

USB Stack
Ralf Oberländer
Stefan Schulze
Paul Kunysch

USB IC-USB
Ralf Oberländer

USB MSC
Stefan Schulze

USB CDC ACM
Toralf Henze

USB SIC/PictBridge
Stefan Schulze

USB CDC ACM
Toralf Henze

USB CDC NCM
Pavol Kurina

USB HAL
Stefan Schulze
Enrico Schmidtke