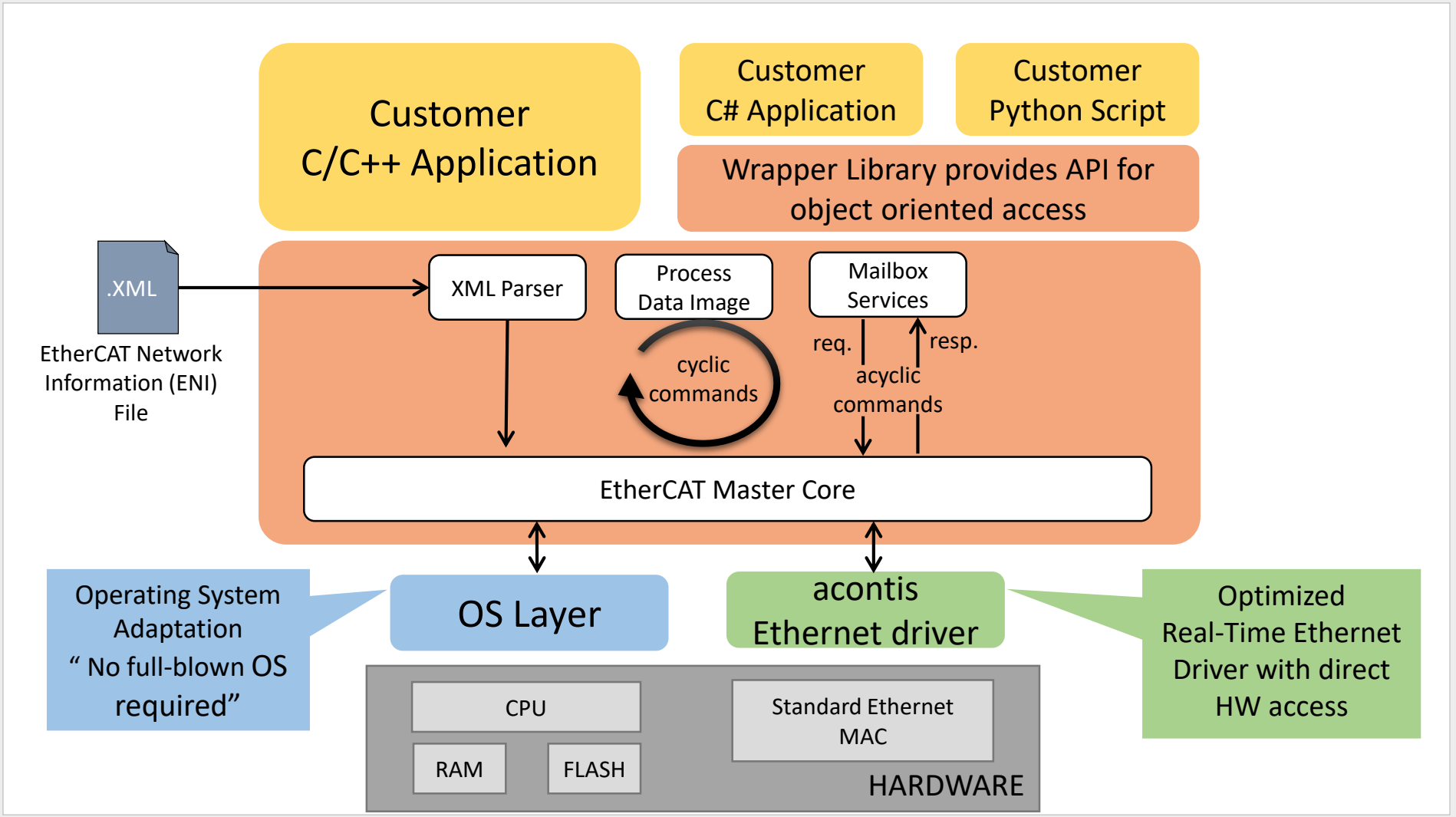
EtherCAT® Master Stack

Technical Presentation

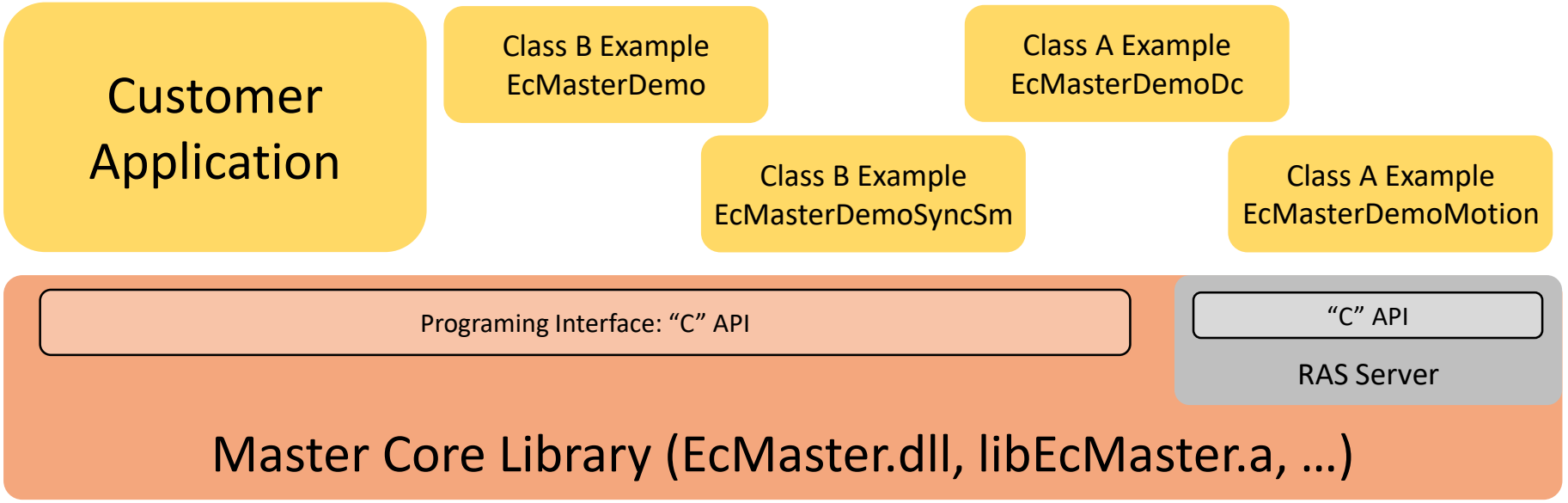
Application Programming Interfaces

acontis technologies

EC-Master Architecture



Programming the Master Core Library in C/C++



The EtherCAT Master Core Library and the RAS Server Module are implemented in C++. The API interfaces are C language interfaces, thus the master can be used in ANSI-C as well as in C++ environments.

C/C++ Example applications

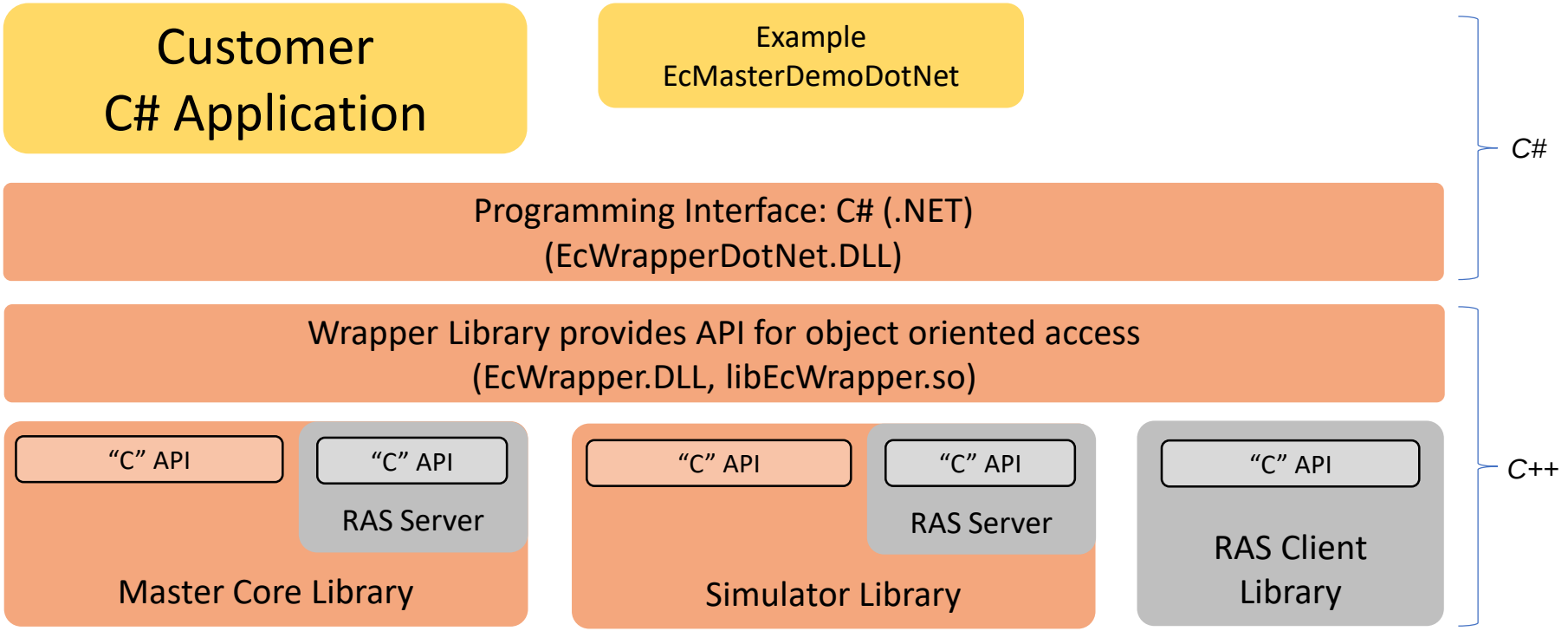
- **EcMasterDemo** is the basic example application for EC-Master. The example shows how to initialize the master and how to put the network into operational state. Based on the provided ENI file this example can handle all kind of EtherCAT Slaves.
- **EcMasterDemoSyncSm** demonstrates a different network timing which requires the interrupt from the Ethernet controller used by the master. This cyclic frame is transmitted at the begin of the cycle and the process data are updated immediately after the frame returns.
- **EcMasterDemoDc** is a good starting point for application requiring the accurate synchronization of slaves based on the Distributed Clocks (DC) technology. To synchronize the master controller with the slaves several modes can be selected.
- **EcMasterDemoMotion** comes with a simple motion control library to control drives implemented according to the profile CiA402 and the ETG Implementation Directive ETG.6010. The example supports the operation modes Cyclic Synchronous Position (CSP) and Cyclic Synchronous Velocity (CSV).

Programming the Master Core Library in C/C++
Operating systems and tools

EC Master

- Supported Operating Systems: All
- Supported Compilers: Microsoft, GNU, LLVM
- Supported IDE: Microsoft Visual Studio, Eclipse, WindRiver Workbench, QNX Momentics, IAR, Keil MDK

Programming in C# (1)



The example **EcMasterDemoDotNet** and the .NET wrapper are written in C#. The target platform is AnyCPU. The other libraries are written in C++ and are platform specific. The Wrapper Library is a helper that prepares the API for .NET. The demo runs in conjunction with Microsoft .NET framework.

Programming in C# (2)

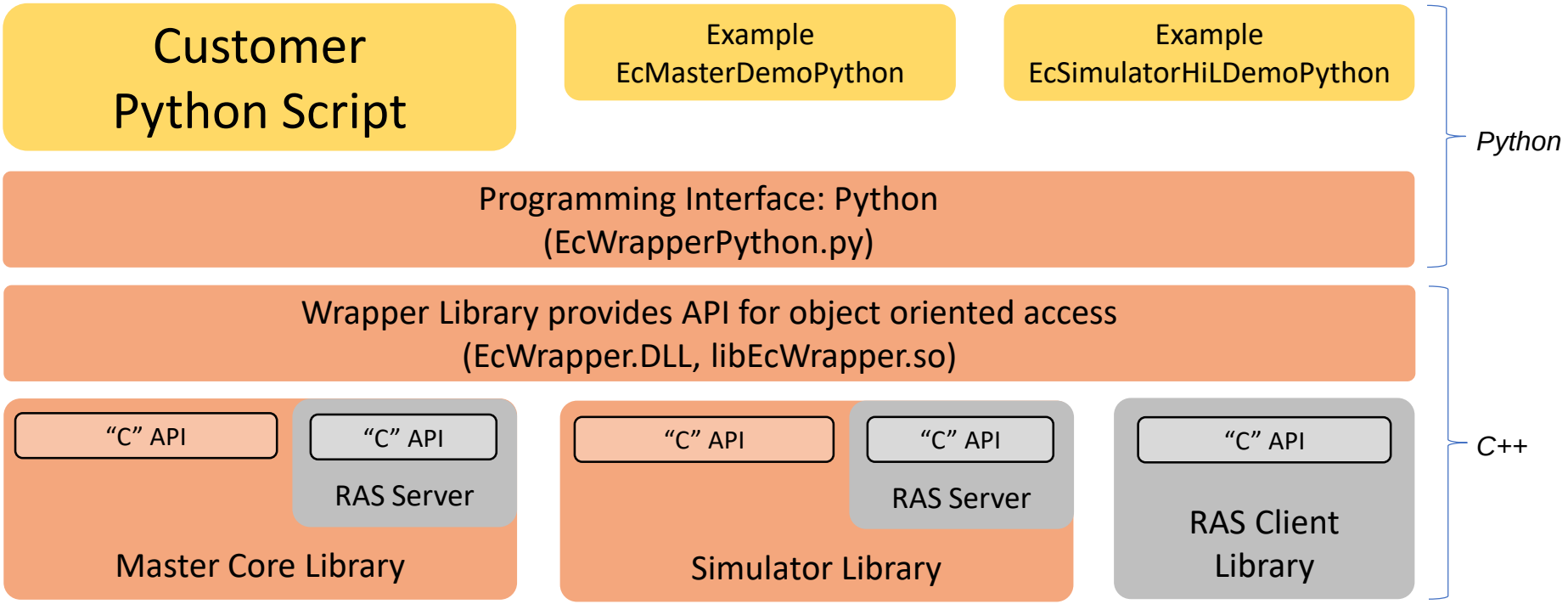
C# Example application

- **EcMasterDemoDotNet** is a basic Windows Forms (C#) GUI application for EC-Master. The example shows how to initialize the master and how to put the network into operational state. Based on the provided ENI file this example can handle all kind of EtherCAT Slaves. It is also shown how to read and write a variable. Even an example of how to read a register from the CoE object dictionary can be found.

Operating systems and tools

- Supported Operating Systems
  - Windows: MS .NET 2.0, MS .NET Standard 2.0
  - Linux: MS .NET Standard 3.0, Mono project
- Supported IDE: Microsoft Visual Studio (Code)
- Supported Compiler: Microsoft

Programming in Python (1)



The examples and the Python wrapper are written in Python. They are not platform specific. The other libraries are written in C++ and are platform specific. The EcWrapper is a helper that prepares the API for python.

Programming in Python (2)

Python Example applications

- Similar to the other demos the **EcMasterDemoPython** shows how to call the EtherCAT Master API. There is also a Python demo for the EC-Simulator. The python demos can also run in interactive mode e. g. to set an output of the EtherCAT network or something else. This is very useful to quickly test different behaviors of the EtherCAT network e. g. with the EC-Simulator.

Operating systems and tools

- Supported Operating Systems: Windows (Python 3.7), Linux (Python 3.7)
- Supported IDE: Python IDLE Shell, Microsoft Visual Studio Code

EtherCAT Network Timing

Interaction between customer application and EtherCAT network

acontis technologies

EC-Master without internal tasks

EtherCAT Master has no internal tasks:

From the application side it looks like a driver, which is activated by calling some simple functions, the so called **“Cyclic Jobs”**.

The benefits are:

- No synchronization issues between application and EtherCAT Master.
- Consistent process data without using any locks.
- Various network timings driven by the application possible
- Cyclic part may run within Interrupt Service Routine (ISR)
- Easy to integrate

Cyclic and acyclic EtherCAT frames

- Cyclic frames:
  - Contain process output and input data
  - Distributed Clocks (DC): Contain datagram to distribute network time
  - Typically sent by master in every cycle
  - Defined by the configuration tool (which data to read and to write)

- Acyclic frames:
  - Asynchronous, event triggered communication
  - Mailbox communication (CoE, FoE, EoE)
  - Status requests (e. g. read slave state information)
  - Raw EtherCAT datagrams requested by application

Cyclic jobs

P

EtherCAT Master: Refresh Inputs

eUsrJob_ProcessAllRxFrames: Process all received frames

S

EtherCAT Master: Write Outputs

eUsrJob_SendAllCycFrames: Send cyclic frames

MT

EtherCAT Master: Administration

eUsrJob_MasterTimer: Trigger master and slave state machines

AS

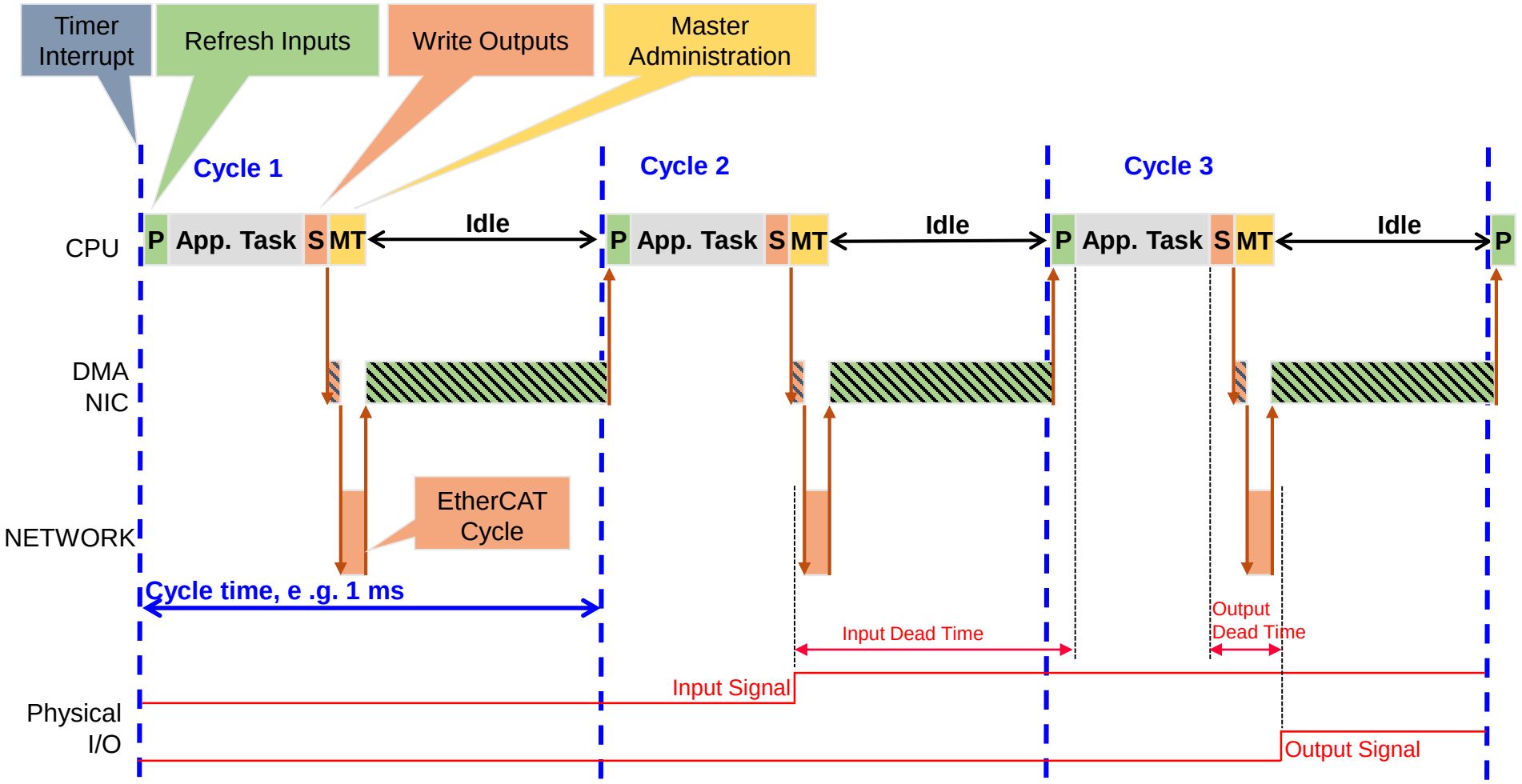
EtherCAT Master: Send acyclic datagrams/commands

eUsrJob_SendAcycFrames: Transmit pending acyclic frame(s).

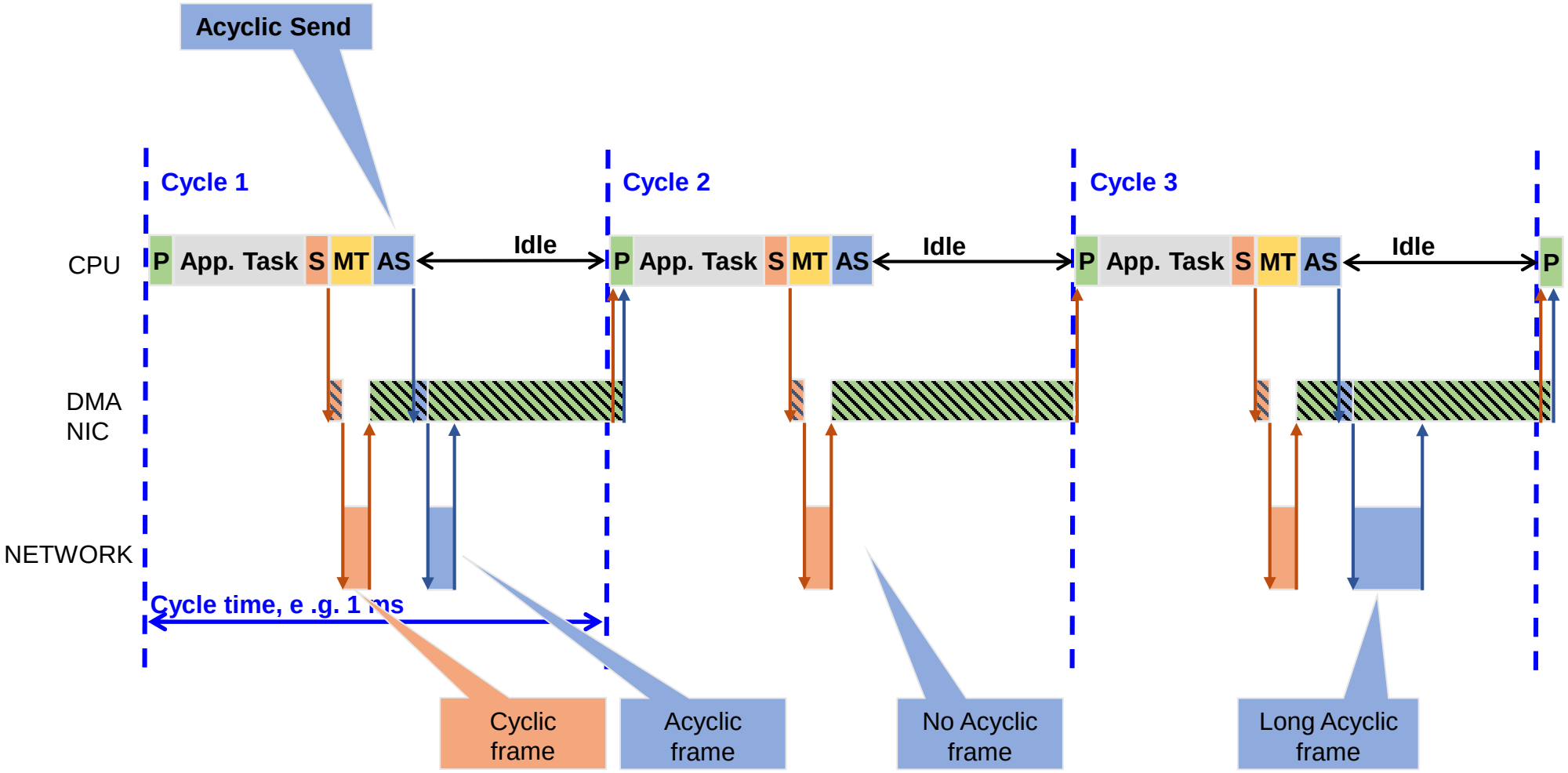
App

Application: Work on inputs and create output values

Standard Network Timing: Short output dead time Cyclic frames



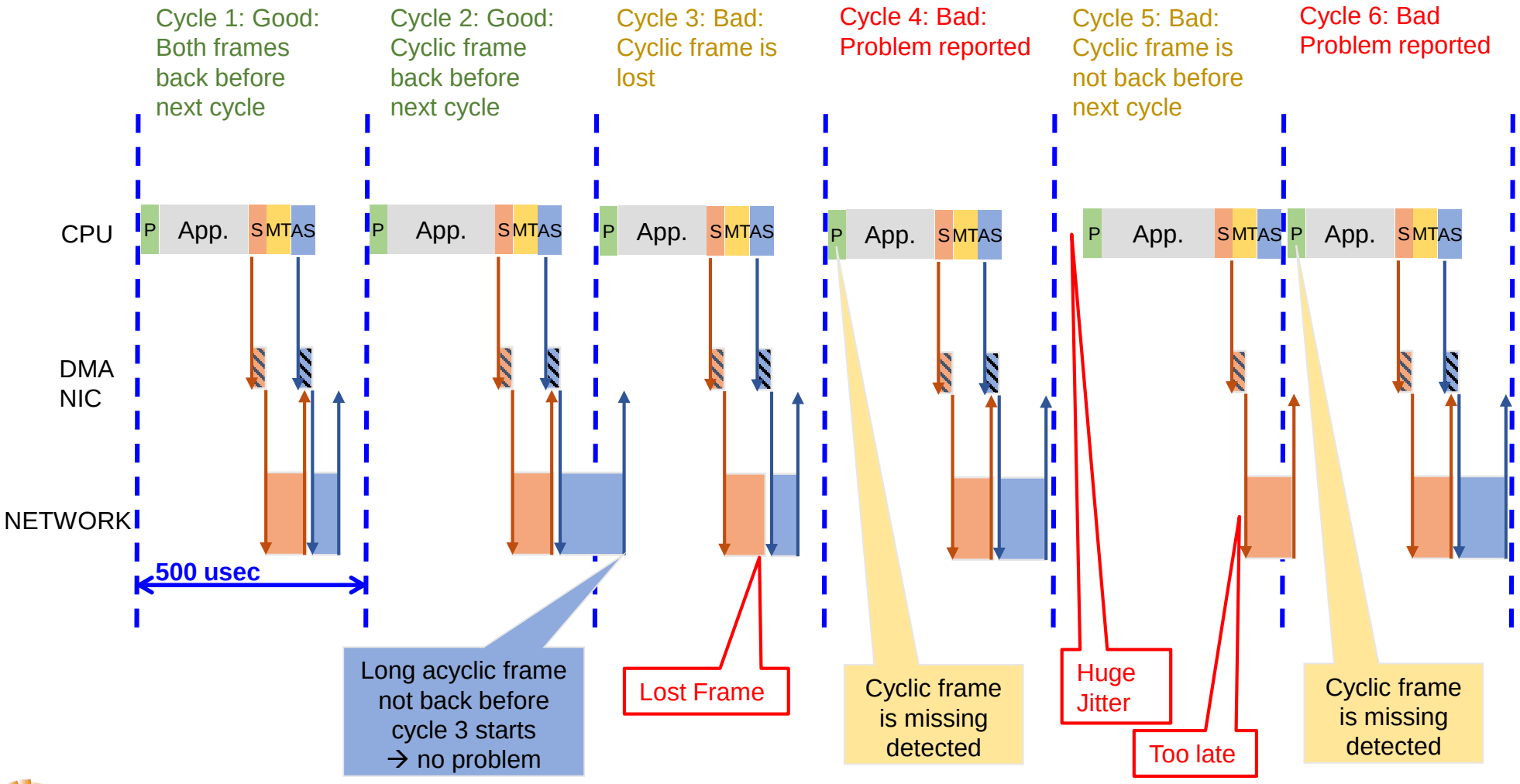
Standard Network Timing: Short output dead time
Cyclic and acyclic frames



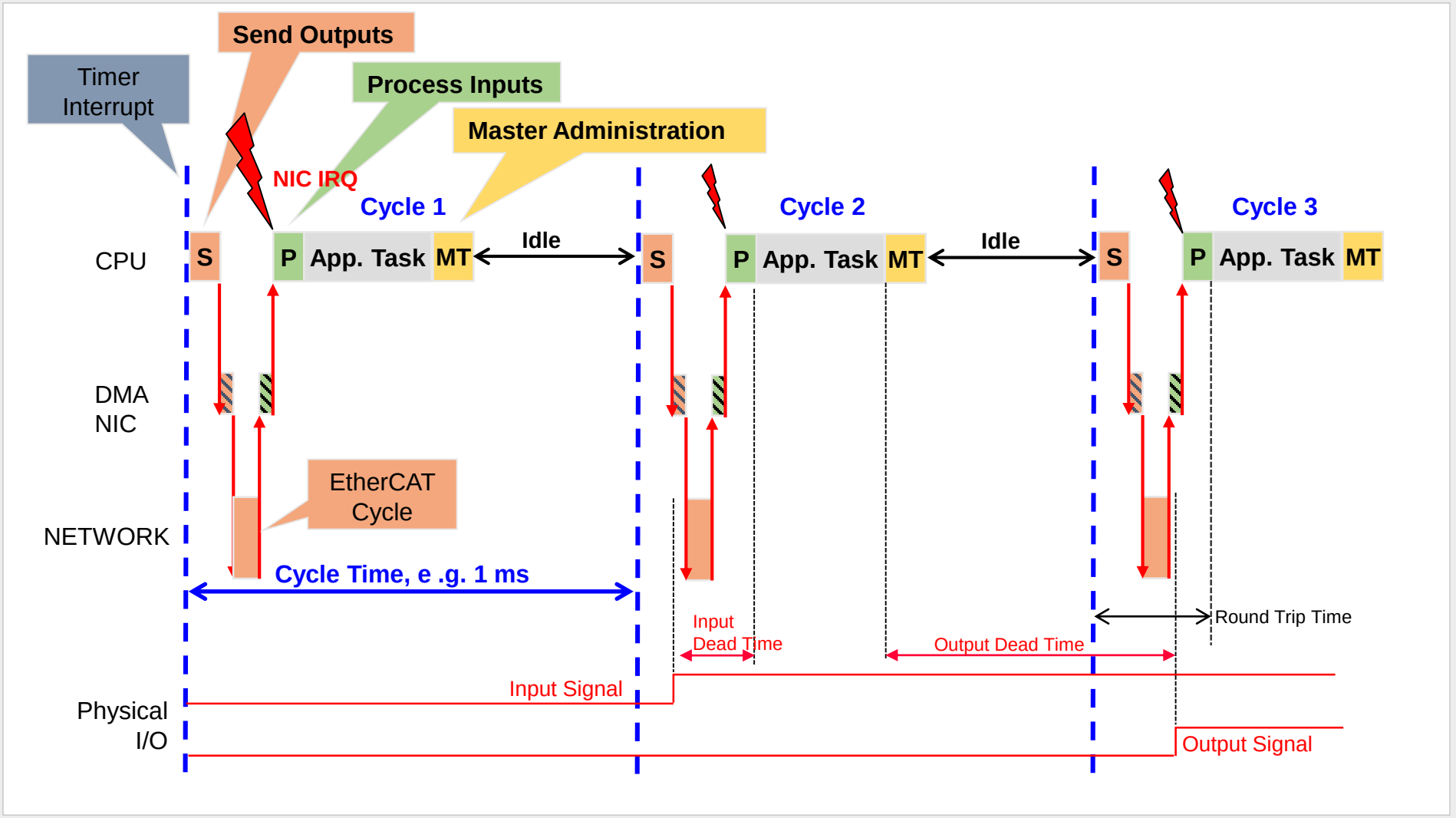
Standard Network Timing: Short output dead time Characteristics

- Long Input dead time (physical input to application)
- Short Output dead time (application to physical output)
- Cyclic frame is transmitted after the application
  → Higher jitter (can be compensated using Distributed Clocks)
- No interrupt from network controller (NIC) required
  → High performance, lower CPU load
  → Simple hardware configuration
- Cyclic task is idle while frame is running through the slaves
- Simple and robust implementation

Good and bad network cycles



Alternative Network Timing: Short Input dead time Cyclic frames



Alternative Network Timing: Short Input dead time Characteristics

- Short Input dead time (physical input to application)
- Long Output dead time (application to physical output)
- Cyclic frame is transmitted a the begin of the cycle
  → Less jitter
- Interrupt from network controller (NIC) required
  → Not available on all platforms
  → Cyclic task is interrupted and scheduled several times
- Cyclic task has to wait while frame is running through the slaves
- Higher implementation effort

EtherCAT Synchronization and Distributed Clocks

acontis technologies

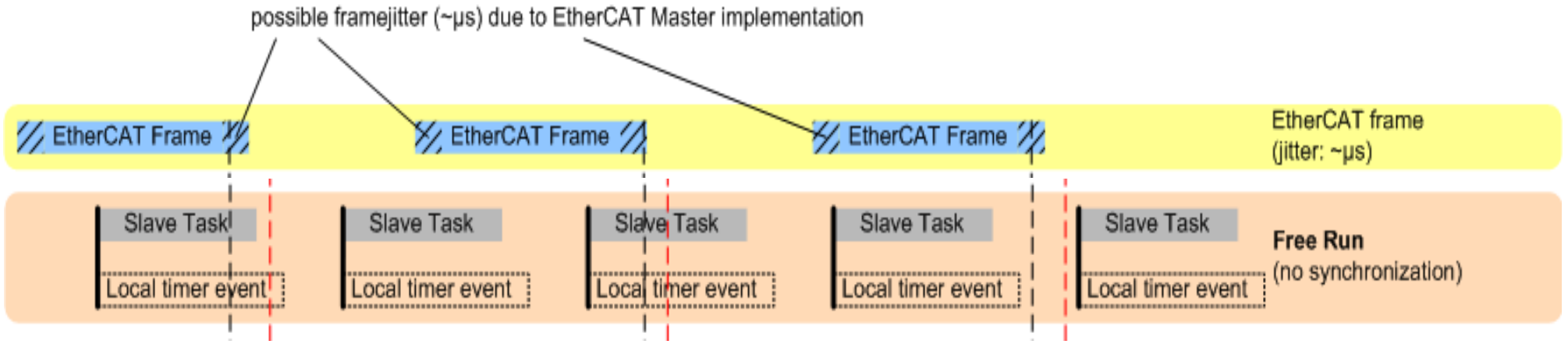
Synchronization modes in the slave

- Free Run:
  Slave's application is not synchronized to EtherCAT.

- Synchronous with Sync Manager (SM) Event:
  Slave's application is synchronized to the SM2/SM3 Event.

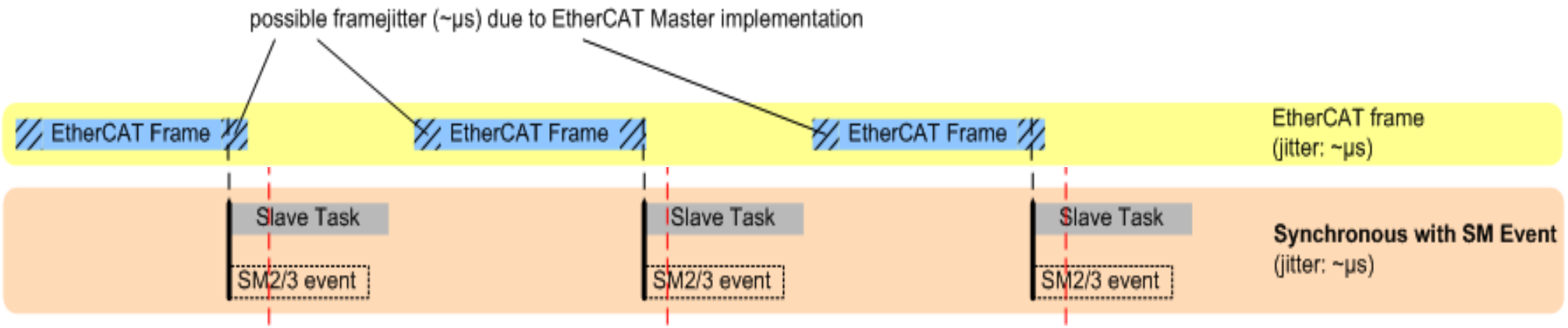
- Synchronous with DC SYNC Signal:
  Slave's application is synchronized to the SYNC0 or SYNC1 signal, which are based on the distributed clocks (DC) unit.

Free Run
Timing diagram



- Network update rate is slower than slave´s application cycle time
- Slave's application is not synchronized to EtherCAT. Slave task starts after local timer signal is generated.
- Drawback 1: Slave task may read input data twice
- Drawback 2: Slave task may miss input data, if cycle time of local timer signal is bigger than network cycle

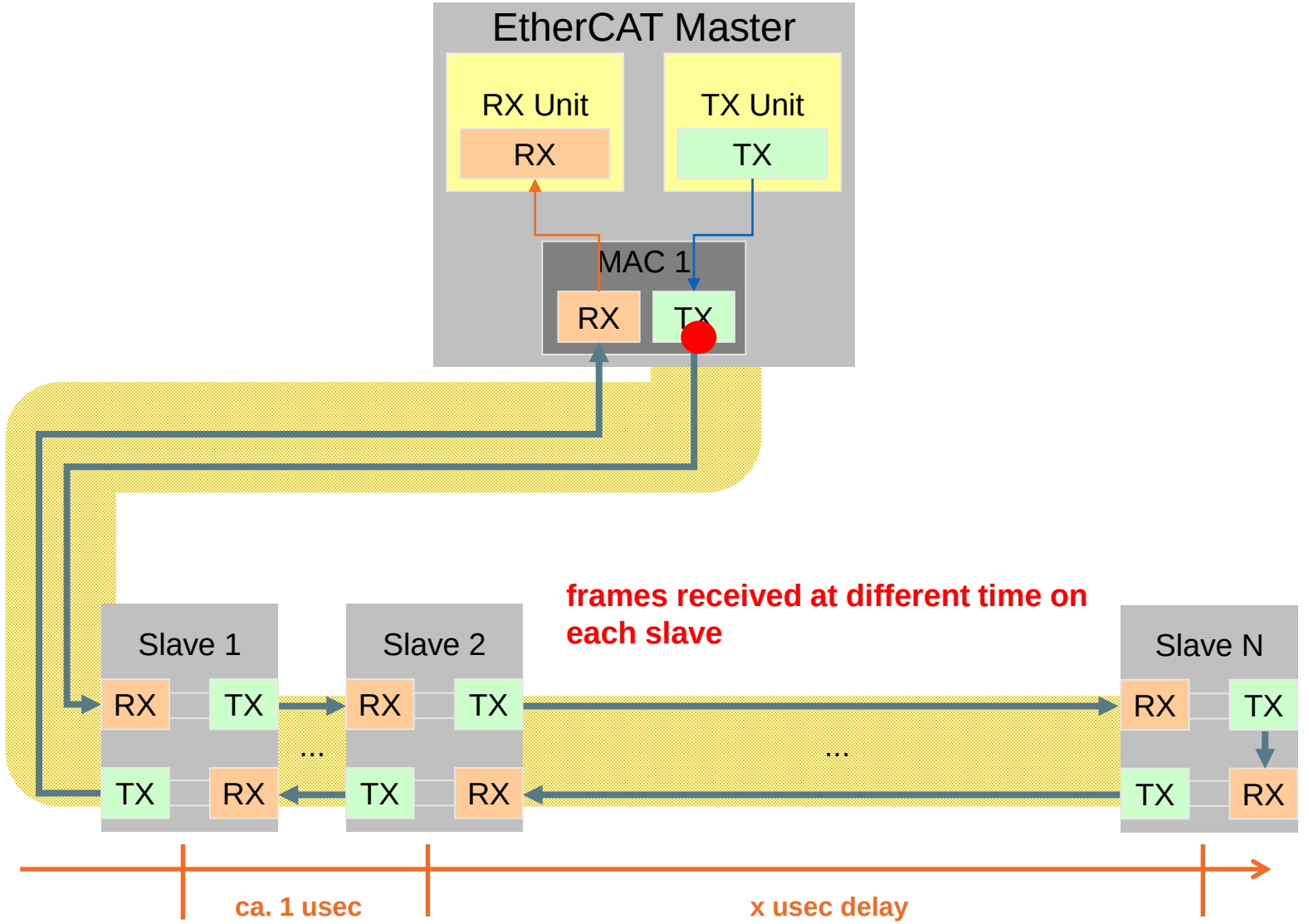
Synchronous with Sync Manager (SM) Event Timing diagram - Drawback Jitter



- Slave task starts after frame is received.
- Slave's application is synchronized to the SM2 event (if cyclic outputs are transmitted) or the SM3 Event (if only cyclic inputs are transmitted).
- Slave task has always "new" input data and do not miss data
- Drawback: This time can jitter in the range of a few microseconds due to the EtherCAT Master implementation (delay in Stack, PHY & MAC Delay, etc).

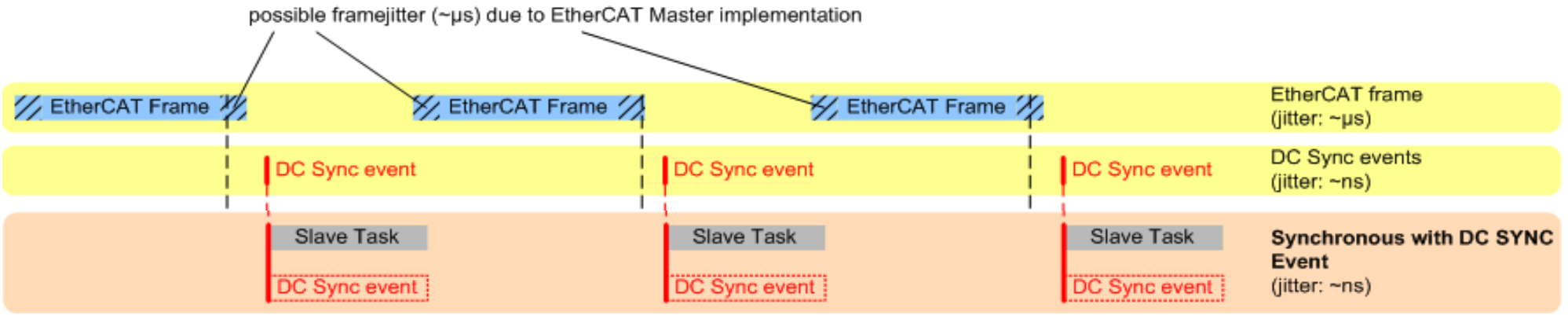
Synchronous with Sync Manager (SM) Event Drawback: Delay between slaves

EC Master



Synchronous with DC SYNC Signal Timing diagram

EC Master



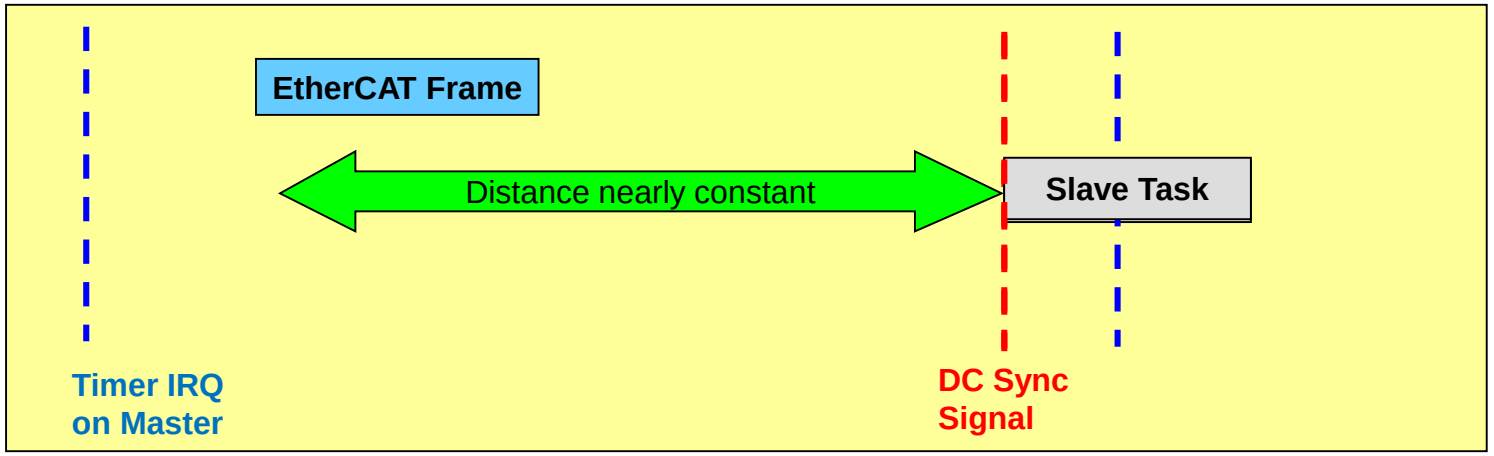
- Slave's application is synchronized to the SYNC0 or SYNC1 signal, which are based on the distributed clocks (DC) unit.
- Slave task is scheduled after DC-SYNC signal is raised
- The jitter could be reduced to a few nanoseconds.
- Drawback: Master has to support the DC features

DC Synchronization between Master and Slave Why important?

1. The DC reference clock provides the System (Network) Time.
   This time – a counter in the EtherCAT slave controller - is driven by an oscillator.
   Based on this time the DC-Sync-Signal (ESC hardware signal) is raised.
2. The cyclic frames, containing the process data, are sent by the EtherCAT Master within the cyclic task (job task). This cyclic task is scheduled by the operating system due to a timer interrupt on the master controller.
   This timer (counter) is driven by an oscillator.

→Due to the 2 oscillators a synchronization is required to avoid that the DC-SYNC-Signal is raised while the cyclic frame is arriving in the slaves.

Distributed Clocks Master (DCM) Synchronization Principle



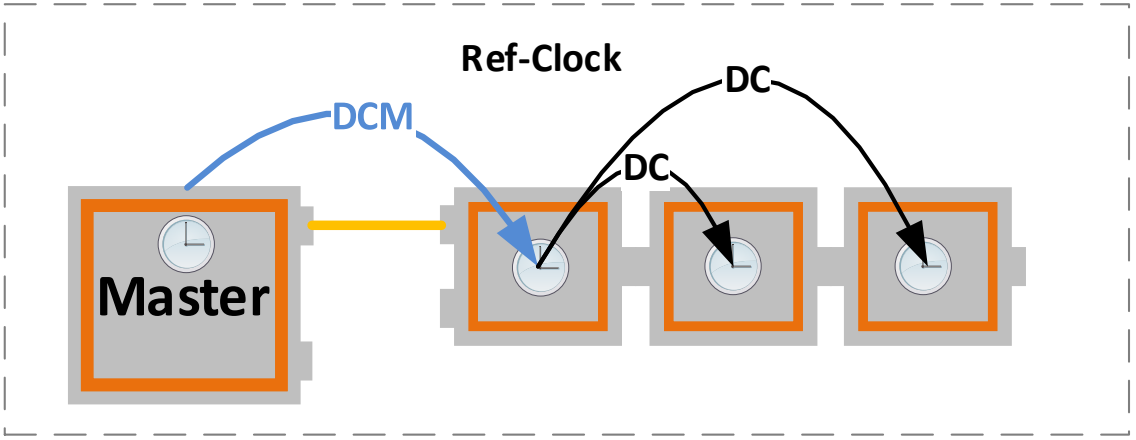
Blue line: Start of cyclic task (Job-Task) driven by oscillator on master controller

Red line: DC-Sync signal based on System Time driven by oscillator in the slave. Slave task is scheduled after DC-Sync is raised.

Without synchronizing both times, the EtherCAT frame (process data) may transmitted through the network while the slave task is scheduled.
This will cause the same issues as in free run mode.

- → Outputs (Master to Slave): Slave has no new data in one cycle, or will miss output data
- → Inputs (Slave to Master): Master has no new data, or will miss input data

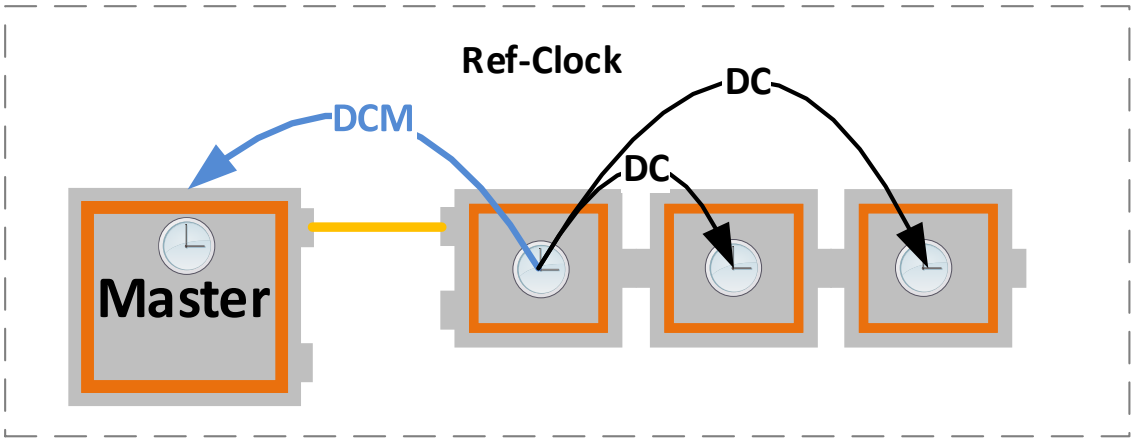
DCM: Bus Shift Mode (default)
Reference Clock controlled by Master/Controller Time



Bus Shift: DC Reference Clock follows the Master Clock/Timer

- Adjust the Bus Time Register of the DC Reference Clock. The Ref.clock converge to this time.
- The DC Ref. clocks time (EtherCAT system time) is distributed to the slaves behind the Ref.clock. The slaves will converge to the system time.

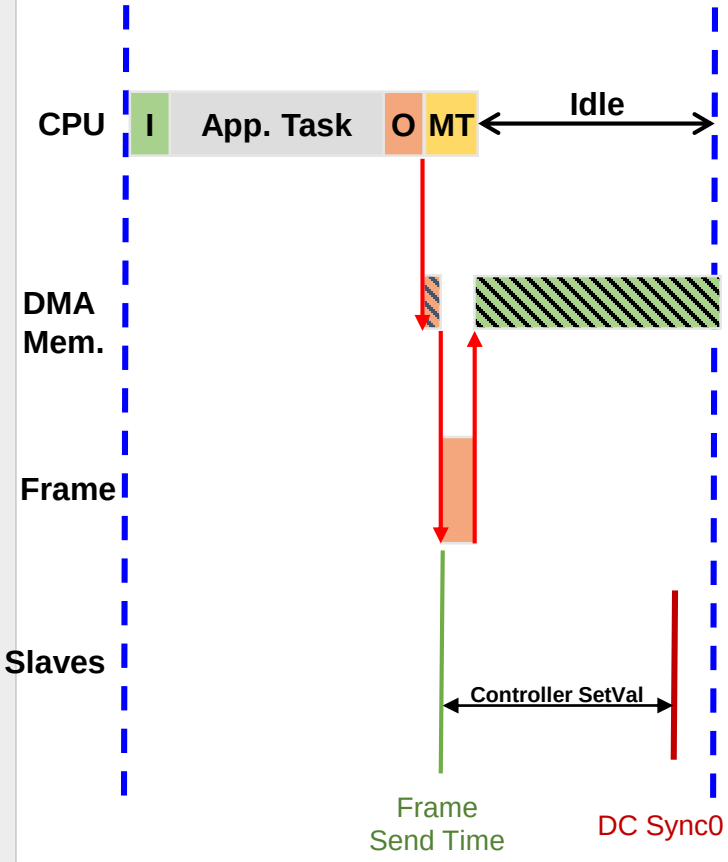
DCM: Master Shift Mode
Master/Controller Time controlled by Reference Clock



Master Shift: Master Clock/Timer follows the DC Reference Clock

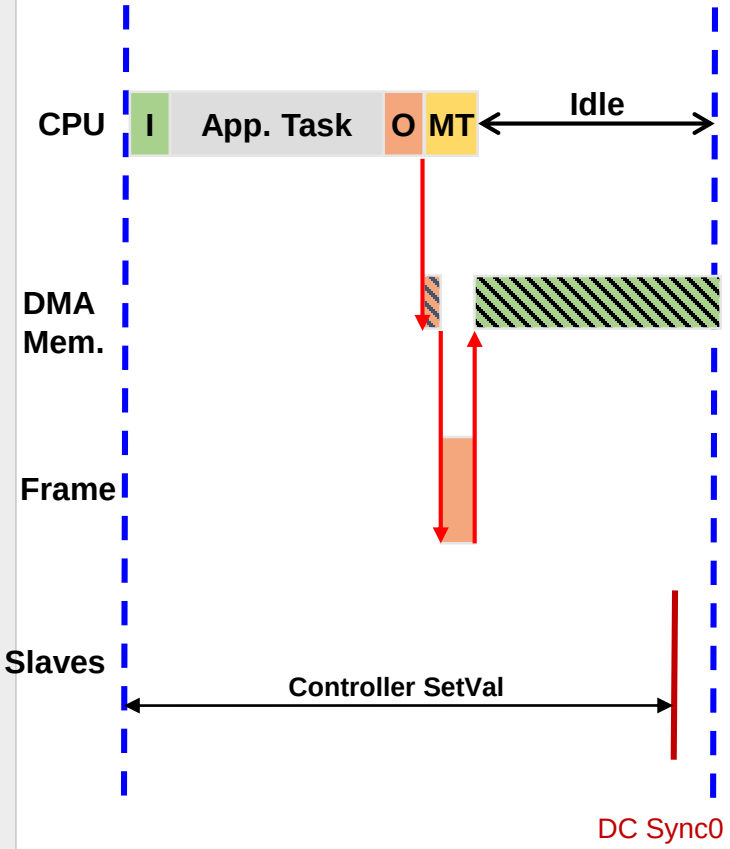
- The timer frequency on Master controller is adjusted.
  E. g., if the timer is to fast, slow down the timer for one cycle and then switch back to the original frequency.
- Pro: Reduced DCM controller error
- Pro: Quality independent from *cyclic frame send time jitter*
- Con: Requires enhanced OS-Layer:
  OsHwTimerGetInputFrequency(), OsHwTimerModifyInitialCount()
- Con: Not available on all operating systems!

DCM Controller Set Value (default)
“Controlling the time between cyclic frame and Sync0”



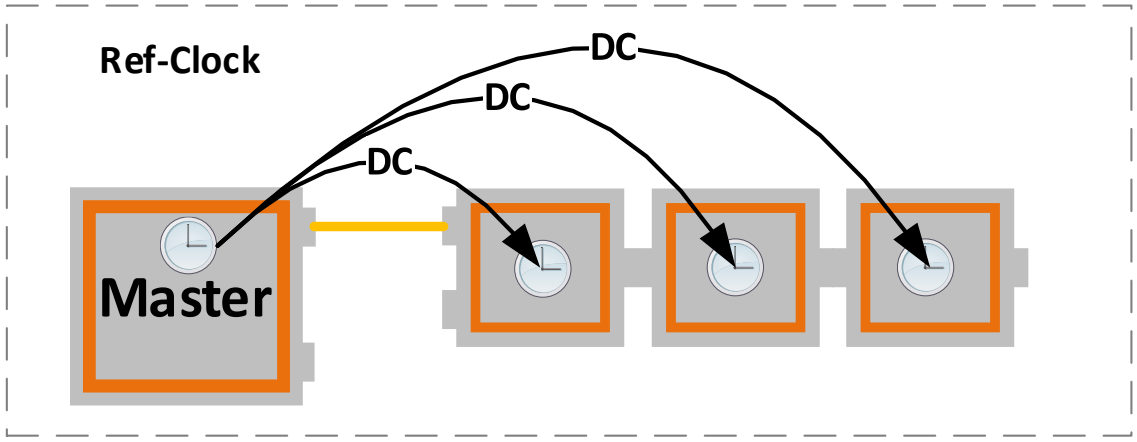
- Minimum controller error depending on *cyclic frame send time jitter*
  - Frame send time depends on application execution time
  - App execution time should be nearly constant
  - The DCM controller is using a filter algorithm to deal with frame send time jitter
- Works fine in most cases

DCM Controller Set Value (enhanced)
“Controlling the time between start of cycle and Sync0”



- Reduced DCM controller error
- Quality independent from *cyclic frame send time jitter*
- Requires enhance OS-Layer
  - OsHwTimerGetInputFrequency()
  - OsHwTimerGetCurrentCount()
- Not available on all operating systems!

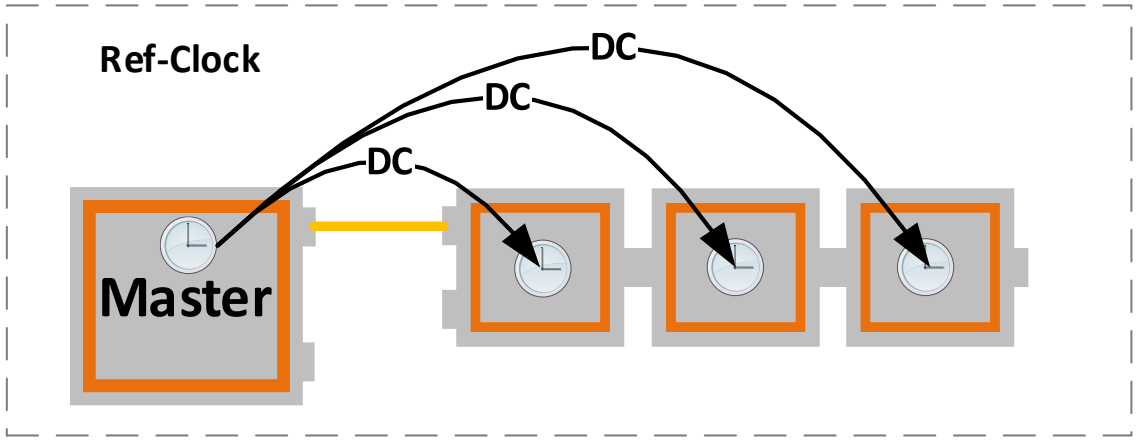
DCM: Master is Reference Clock Mode
Master/Controller time is used as reference clock



Master Ref Clock Mode

- No need for platform adaptation
- The DC reference time (register 0x0910) is provided by the controller
- Pro: No DCM controller required → Reduced CPU load compared to other modes
- Pro: Available on all platforms
- Con: Very long startup time compared to other modes, because the synchronization requires a minimum of 15.000 frame = 15.000 cycles

DCM: Link-Layer Reference Clock Mode
Master/Controller time is used as reference clock



Link-Layer Ref Clock Mode

- The DC reference time (register 0x0910) is provided by the controller
- Pro: No DCM controller required → Reduced CPU load compared to other modes
- Pro: Reduced DCM controller error
- Pro: Quality independent from cyclic frame send time jitter
- Con: Requires enhanced Link-Layer: EC_LINKIOCTL_GETTIME
- Con: Not available on all operating systems!

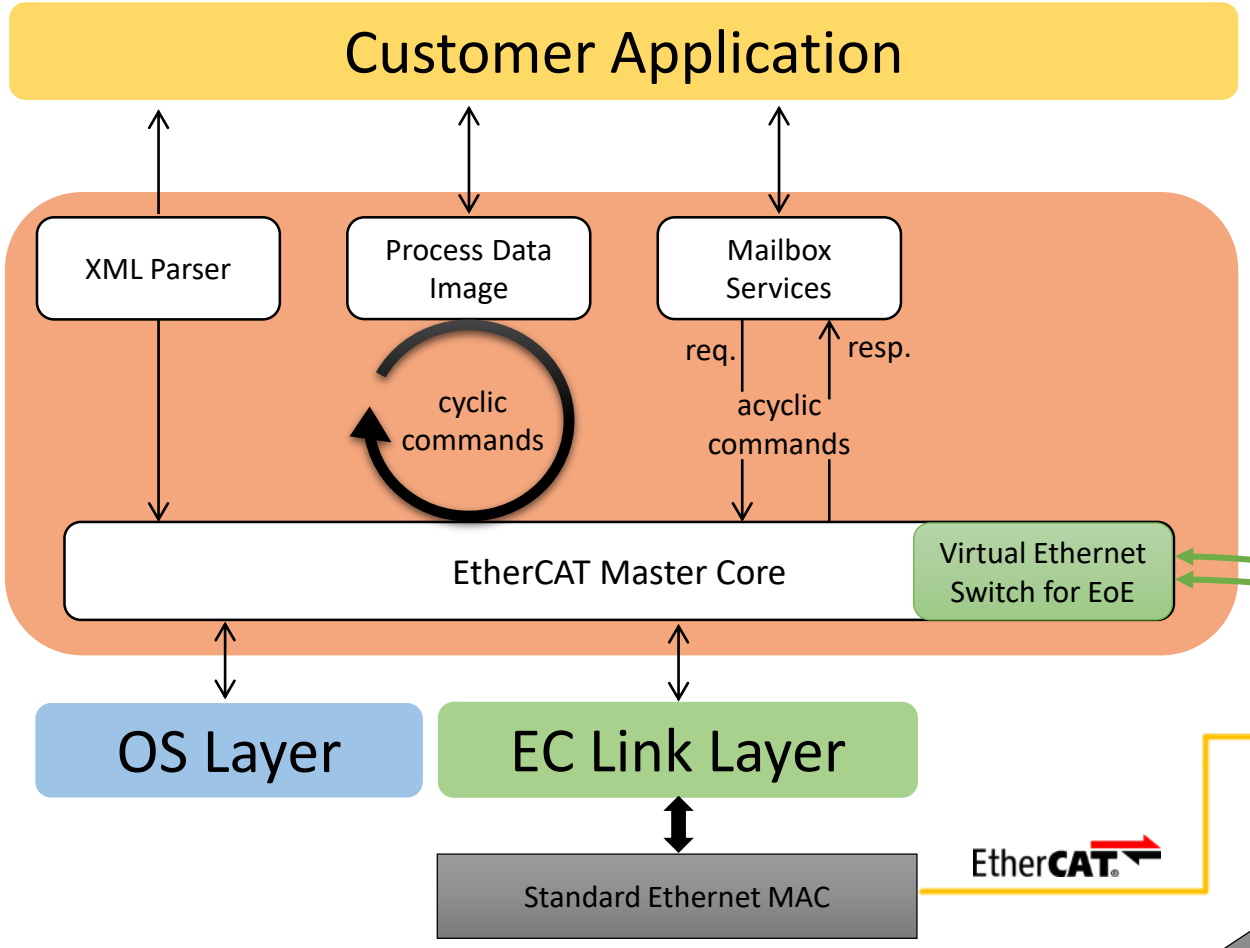
Distributed Clocks Master Sync
DCM Mode Selection Criteria

Criteria	Bus Shift Mode	Master Shift Mode	Master Ref Clock Mode	Link-Layer Ref Clock Mode
Accuracy of SYNC Signals	+/- 1000 nanosec jitter	+/- 20 nanosec jitter	+/- 20 nanosec jitter	+/- 20 nanosec jitter
Maximum acceptable drift between master and reference clock	600 ppm → very precise timer in control system required	no physical limitation	600 ppm → very precise timer in control system required	600 ppm → very precise timer in control system required
Multiple EtherCAT segments	possible	not possible	possible	possible
Implementation	possible	not possible on all operating systems	possible	not possible on all operating systems
Startup time	Short	Short	Long	Short

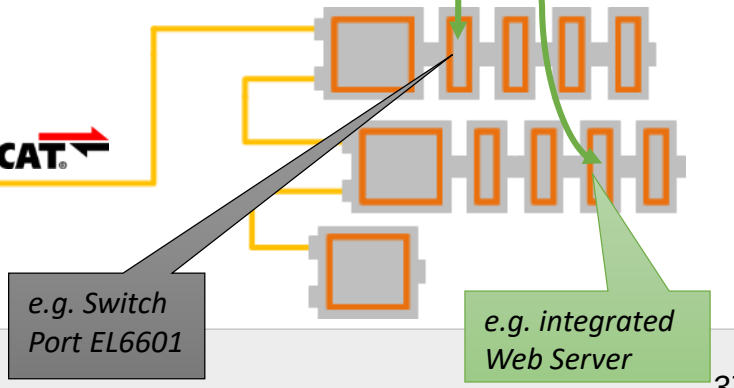
Ethernet over EtherCAT® (EoE)

acontis technologies

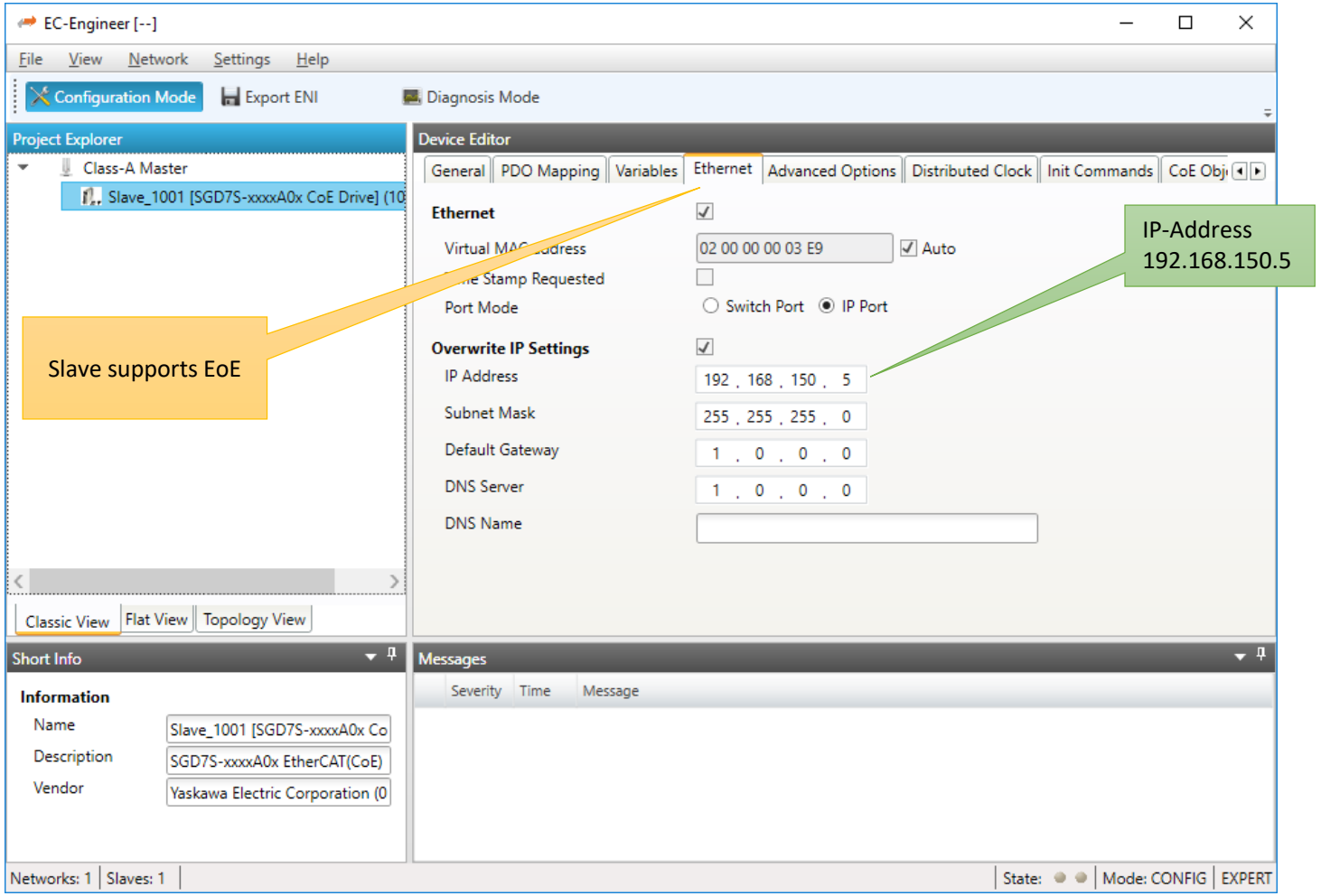
EC-Master Class A and B: Virtual Switch for EoE included



EC-Master includes a „Virtual Ethernet Switch“ to transport the EoE protocol between different slaves



Each slave using EoE requires an own IP address
Setting IP address in EC-Engineer



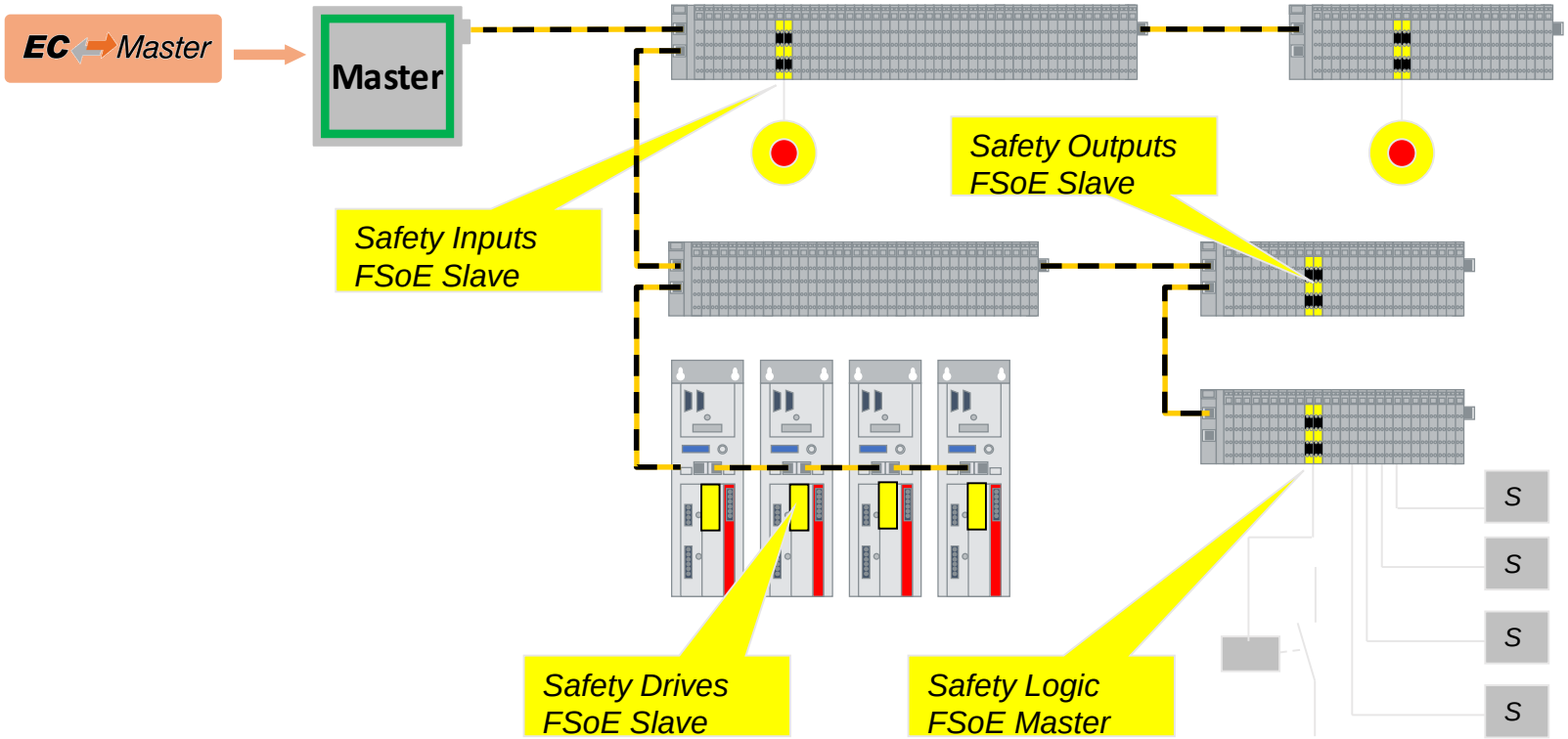
Functional Safety over EtherCAT (FSoE)

acontis technologies

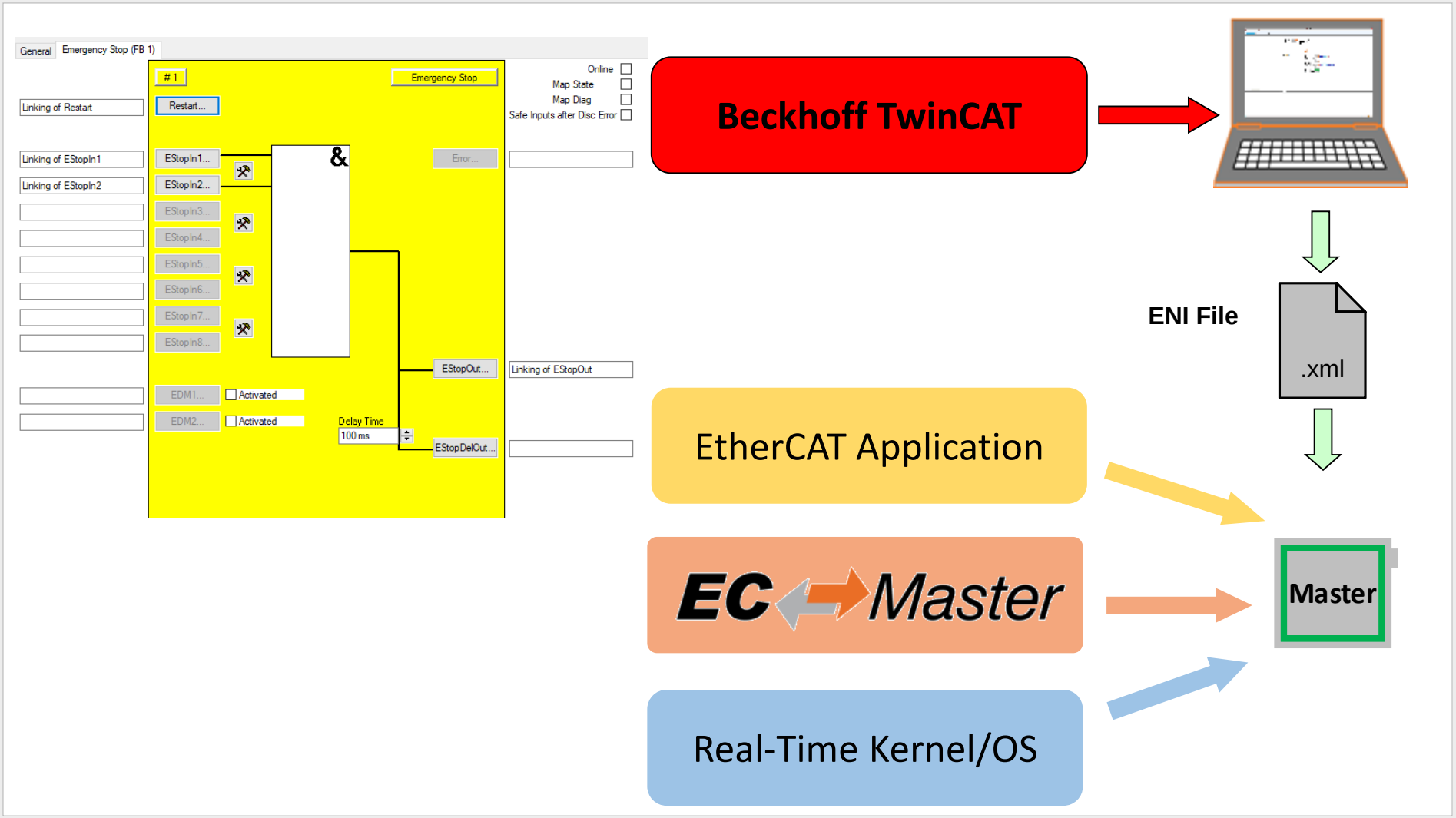
Safety over EtherCAT: Implementation Example

EC Master

- Fully integrated solution: Safe and standard communication in one channel
- Decentralized Safety-Logic
- Standard Master controller routes the safety messages

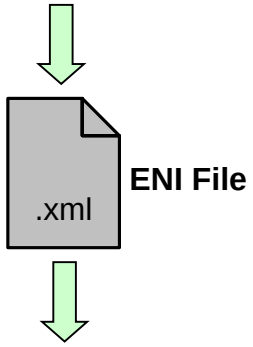


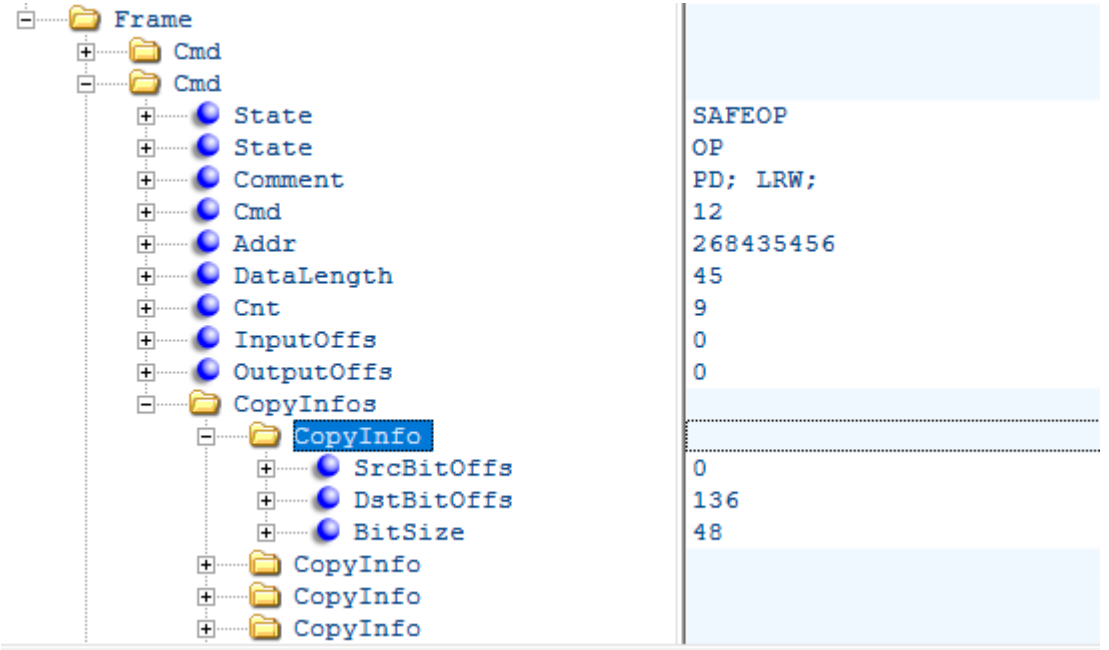
Example: Safety configuration with Beckhoff TwinCAT



Safety requires Slave-to-Slave Communication

- Information is given in ENI file
- Copying of the data will be handled by master stack automatically
  - In the example below 48 bits from process data input memory starting at offset 0 are copied to process data output memory at offset 136





Performance Measurements

acontis technologies

ARM Cortex-A8, 32-Bit, 600 MHz
CPU load depending on number of slaves

Device: TI AM3359 Industrial Communications Engine
CPU: TI Sitara AM3359 (Cortex-A8), 600 Mhz
Software: EC-Master V3.1.1.01 TI-RTOS, Link Layer ICSS-PRU
Features: Distributed Clocks with DCM Bus Shift

Number of Slaves	16	32	64
Network cycle time	250 usec	500 usec	1000 usec
Payload	128 Bytes	256 Bytes	512 Bytes
EC-Master Function			
Process Inputs [usec]	34.2	35.1	50.2
Send Outputs [usec]	17.0	19.1	33.1
Administration [usec]	11.7	14.2	28.0
Send Acyclic Frame [usec]	16.2	14.9	14.5
Total Time [usec]	79.1	83.3	125.8
CPU Load [percent]	31.6 %	16.7 %	12.5 %

ARM Cortex-A72, 64-Bit, 1000 MHz
CPU load depending on number of slaves

Device: Toradex Apalis iMX8
CPU: NXP i.MX 8QuadMax 2x Cortex™-A72, 4x Cortex™-A53, 1000 Mhz
Software: EC-Master V3.1.1.01 Linux_aarch64, Link Layer FSLFEC
Features: Distributed Clocks with DCM Bus Shift

Number of Slaves	16	32	64
Network cycle time	250 usec	500 usec	1000 usec
Payload	128 Bytes	256 Bytes	512 Bytes
EC-Master Function			
Process Inputs [usec]	17.1	18.8	27.1
Send Outputs [usec]	5.2	5.3	6.1
Administration [usec]	3.6	5.0	12.0
Send Acyclic Frame [usec]	3.6	3.6	3.6
Total Time [usec]	29.5	32.7	48.8
CPU Load [percent]	11.8 %	6.6 %	4.8%

Intel x64, 64-Bit, 1600 MHz
CPU load depending on number of slaves

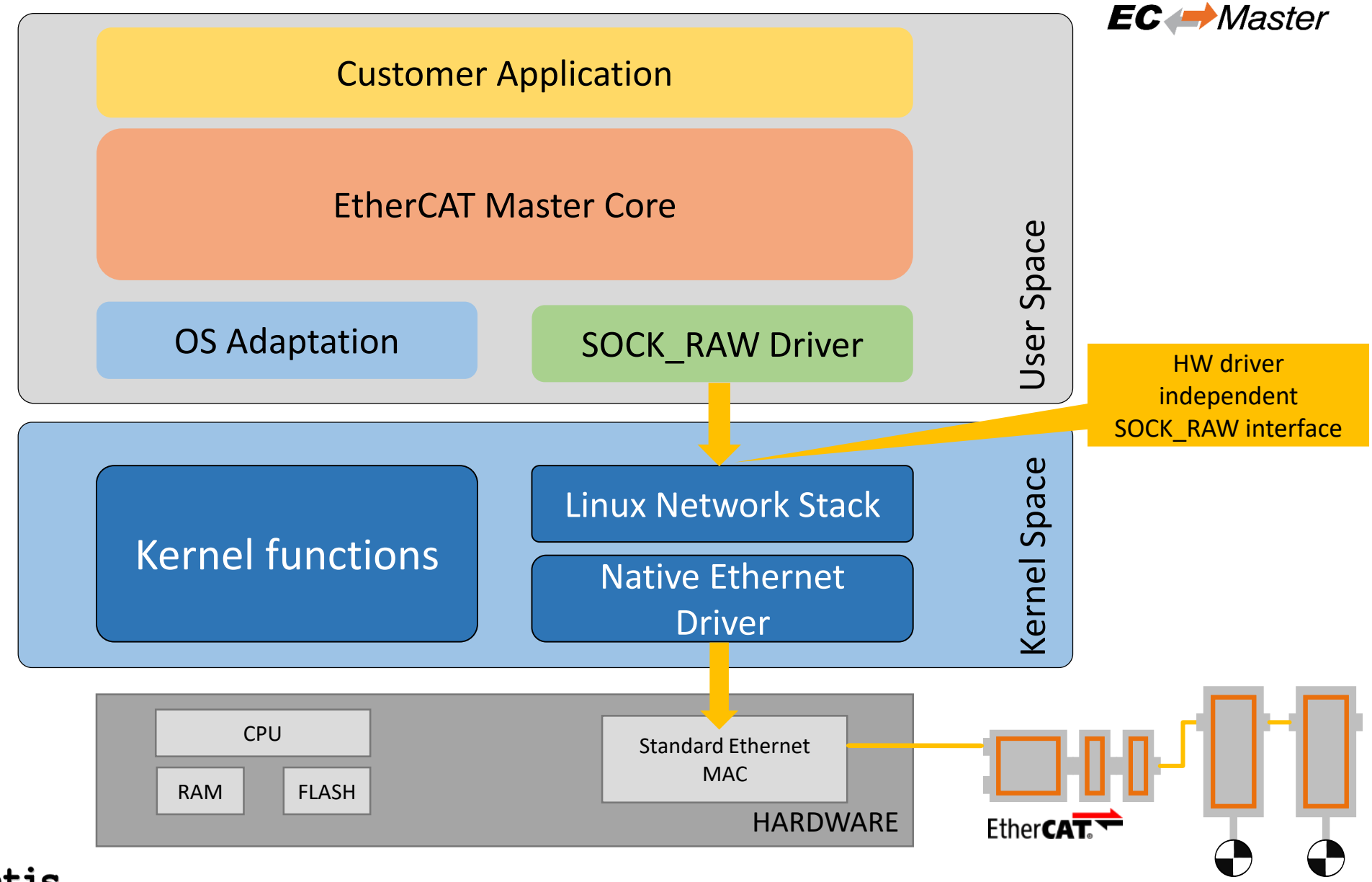
Device: Industrial Board IB891-D5 (EC1)
CPU: Intel Atom D510 1600MHz
Software: EC-Master V3.1.1.01 for QNX_x64 , Link Layer Realtek 8169
Features: Distributed Clocks with DCM Bus Shift

Number of Slaves	16	32	64
Network cycle time	250 usec	500 usec	1000 usec
Payload	128 Bytes	256 Bytes	512 Bytes
EC-Master Function			
Process Inputs [usec]	11.3	11.5	13.5
Send Outputs [usec]	6.3	6.4	6.9
Administration [usec]	5.8	8.0	15.7
Send Acyclic Frame [usec]	4.2	4.4	4.1
Total Time [usec]	27.6	30.3	40.2
CPU Load [percent]	11.0 %	6.0 %	4.0 %

EtherCAT® Master on Linux

acontis technologies

Architecture 1: Linux Network Driver



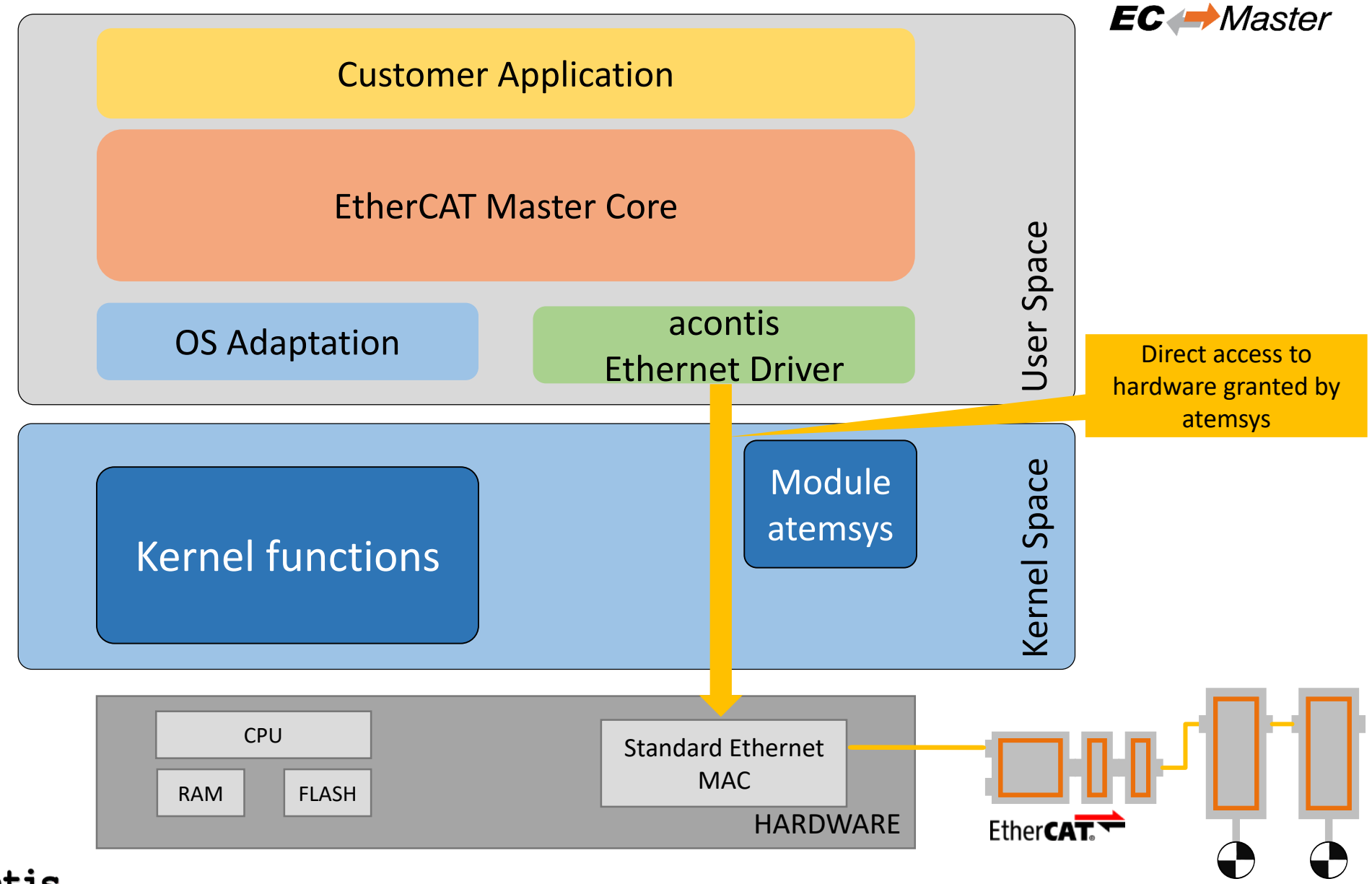
Architecture 1: Linux Network Driver

The EC-Master stack communicates with the slaves by sending and receiving EtherCAT frames using an Standard Ethernet network adapter (MAC).

The Linux system includes different network drivers for the different adapter types. The Linux network driver can be used by means of an abstracted network adapter type independent interface called SOCK_RAW to send and receive frames, although the performance may be poor and real-time constraints are typically not kept due to Linux network stack inclusion and driver code that is not optimized for high performance cyclic operation.

Because SOCK_RAW is typically available on every Linux Distributions' Kernel, the pre-compiled EcMasterDemo can be used for evaluation of the EC-Master library's general functionality without compiling any host specific files. For discreet slaves that e.g. implement Distributed Clocks, real-time constraints must be kept. In this case SockRaw is very likely not able to send and receive frames in time and the EtherCAT slaves will refuse the operation. Therefore SOCK_RAW should be only used for initial evaluation purpose and be replaced with the acontis real-time driver as explained below.

Architecture 2: acontis Real-time Driver



Architecture 2: acontis Real-time Driver

The acontis Real-time Driver replaces the standard Linux Ethernet network adapter (MAC) driver for real-time EtherCAT usage.

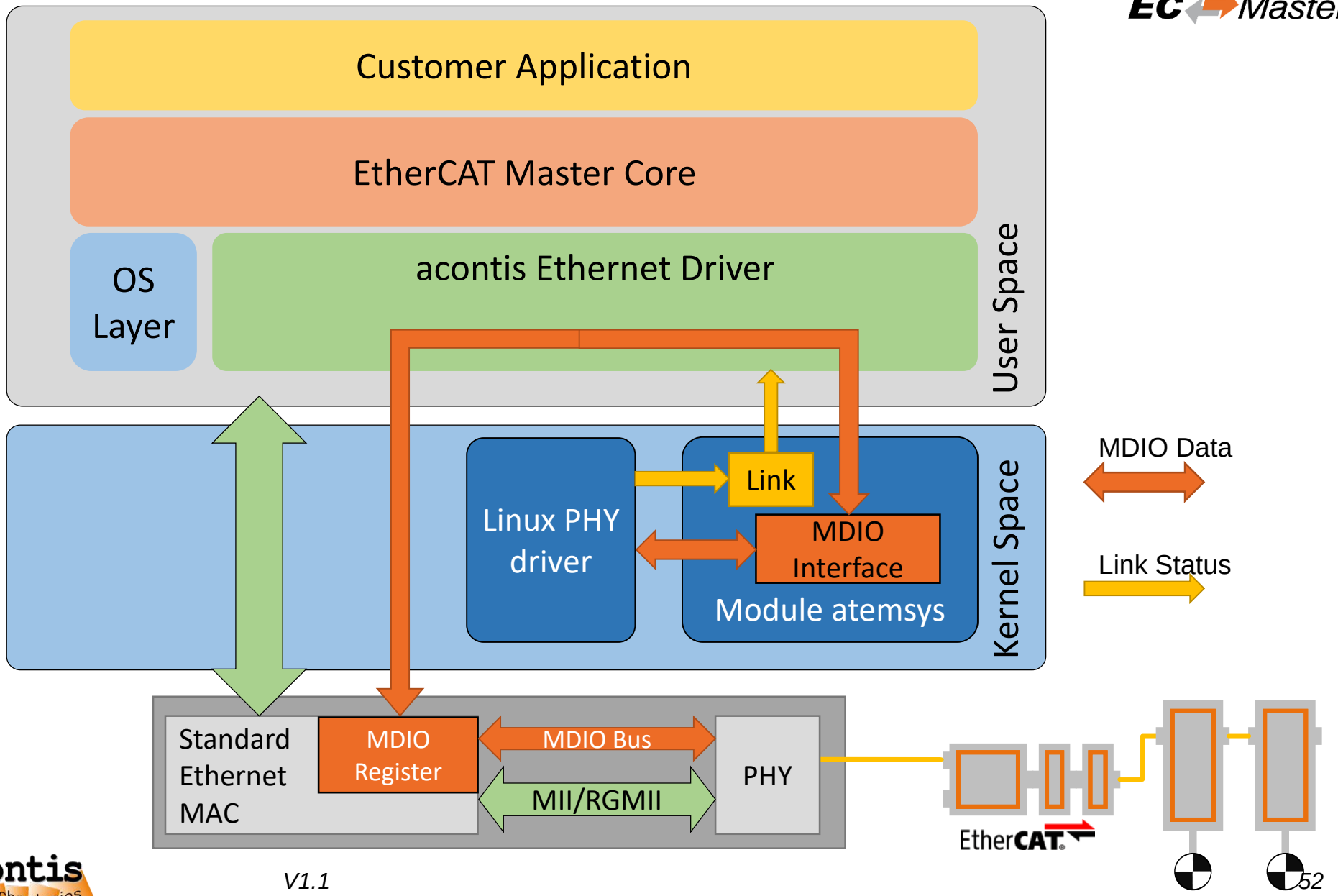
The driver runs in User Space and handles the MAC directly for high performance cyclic operation. It needs direct access to the MAC granted by an GPL‘ed Kernel Module called atemsys. The Kernel module must be compiled to match the running Kernel. Methods for Kernel module compilation can be found in the Linux Distribution‘s documentation. Additionally a build recipe for Yocto Linux is included.

The Real-time Driver is not GPL‘ed and must be licensed. It needs the atemsys loaded at the Linux target for running. With the granted access to the MAC HW it operates fast on sending and receiving frames by completely bypassing the Linux network stack.

The standard Linux network adapter driver may not operate at the same time on the same instance as the acontis Real-time Driver. It is therefore needed to unbind the network adapter instance by removing the driver from the system, unbinding the instance using the virtual sys-fs or in case of embedded devices by modifying the Linux Device Tree.

The atemsys is shipped with the EC-Master and should be updated if the EC-Master contains a newer version.

Architecture 3: Using Linux device tree



Architecture 3: acontis Real-time Driver with Linux device tree support

The Linux operating system provides drivers for most common Ethernet controllers and the related physical transceivers (PHY). The manufacturer specific PHY circuit is handled by an dedicated kernel driver.

To make use of this infrastructure, the acontis kernel module atemsys has to be included in the Linux device tree as an official driver for the Ethernet controller. As a result atemsys can interact with Linux drivers.

Depending on the hardware architecture, atemsys can grants access to the MDIO bus to the Linux drivers, or request MDIO operations by Linux drivers.

The PHY “OS Driver” functionality is configured exclusively through the Linux device tree and doesn’t required any additional configuration at the application level.

Features according to ETG.1500 Master Classes

acontis technologies

Master Core Features (1)

Feature name	Master Class A	Master Class B
Basic Features		
Service Commands, IRQ field in datagram, Slaves with Device Emulation, EtherCAT State Machine, Error Handling, EtherCAT Frame Types	✓	✓
Sophisticated error detection and diagnosis: Lost cable connection, missing/wrong, slave response, slave operation monitoring, Ethernet link layer debug messages, > 200 error codes	✓	✓
VLAN	✓	--
Process Data Exchange		
Cyclic PDO (High performance up to 50 us cycle time), Multiple Tasks	✓	✓
Network Configuration		
Online scanning, Reading ENI, Compare Network configuration, Explicit Device identification, Station Alias Addressing, Read and Write to EEPROM	✓	✓

Master Core Features (2)

Feature name	Master Class A	Master Class B
Mailbox Support		
State change check: Logical and physical polling	✓	✓
Resilient Layer (repeating mailbox communication)	✓	✓
Multiple mailbox channels (multiple protocols in parallel)	✓	✓
CoE Mailbox Protocol		
SDO Up- and Download, Normal, Expedited and Segmented Transfer, SDO Info service (Read Object Dictionary), Complete Access, Emergency Message	✓	✓
EoE Mailbox Protocol		
Services for tunneling Ethernet frames. includes all specified EoE services Virtual Switch	✓	✓
EoE Endpoint interface to Operating Systems	FP	FP
FoE Mailbox Protocol		
FoE Services	✓	✓
Firmware Up- and Download	✓	✓
Boot State	✓	✓

Master Core Features (3)

Feature name	Master Class A	Master Class B
SoE Mailbox Protocol		
SoE Services	✓	✓
AoE Mailbox Protocol		
AoE Services	✓	✓
VoE Mailbox Protocol		
VoE Services	✓	✓
Synchronization with Distributed Clock (DC)		
Initial propagation delay measurement, Offset compensation, Set start time, Continuous drift compensation, Sync window monitoring	✓	--
Master must synchronize itself on the reference clock. DC master synchronization (DCM).	✓	--
Slave-to-Slave Communication		
via Master, required for safety devices	✓	✓