38

EtherCAT

Gianluca Cena
Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni

Adriano Valenzano
Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni

Claudio Zunino
Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni

38.1 Introduction

Ethernet for control automation technology (EtherCAT) [ETG04,ETG05] is a high-performance Ethernet-based fieldbus system. Its main development goal was to apply Ethernet to automation applications, which require short-cycle times and low-communication jitters. EtherCAT is nowadays a popular solution for connecting control applications to field devices in industrial environments, including motion-control applications, and communication equipment and devices can be easily found off-the-shelf [ESC20].

The EtherCAT protocol is an open standard currently managed by the EtherCAT technology group (ETG). Its specification has been recently integrated into the international fieldbus standards IEC 61158 [IE158] and IEC 61784 [IE784]. EtherCAT is based on a master/slave approach (where only one master is allowed in the network) and relies on a ring topology at the physical level. It can interoperate with both common TCP/IP-based networks and other Ethernet-based solutions, such as EtherNet/IP or PROFInet.

38.2 Physical Layer

The EtherCAT master processes data via standard hardware (full-duplex Ethernet network interface controllers) and dedicated software (e.g., Beckhoff TwinCAT), but it supports also open source solutions based on Linux-like operating systems. On the contrary, purposely designed hardware has to be used on slaves. The master node completely controls traffic over the network by initiating all transmissions. Every slave, when receiving a datagram, processes it (in hardware) and then forwards it to the next slave in the physical ring. Unlike Ethernet switches and bridges, frames are not managed according to

a store-and-forward scheme (i.e., received, interpreted, and then copied as process data at every connection). Instead, every frame is processed on-the-fly at the data-link layer.

To further improve communication efficiency, a fieldbus memory management unit (FMMU) is provided in each slave device that reads/writes portions of data included in the frame while it is being forwarded to the next device.

EtherCAT supports two different types of physical layer, namely Ethernet and EBUS. The former is used for connecting to an external Ethernet network according to IEEE 802.3; it is typically used for the connection between the master and the slave network segment. Indeed, an EtherCAT network can be seen as a single, large, Ethernet device that receives and sends Ethernet frames. However, this "device" is not made up of a single Ethernet controller but includes a (possibly large) number of EtherCAT slaves.

Conversely, EBUS can be used as a backplane bus (i.e., it is not qualified for wire connections).

In particular, EBUS is a physical layer designed to reduce pass-through delays inside the nodes; typically, frames are only delayed by 60÷500 ns [PRY08,BEC06], whereas a greater delay (about 1 μs) is introduced by Ethernet interfaces between segments. It uses the Manchester (biphase L) encoding and encapsulates frames between start-of-frame (SOF) and end-of-frame (EOF) identifiers. The beginning of a frame is defined by a Manchester violation with positive level followed by a "1" bit, which is also the first bit of the Ethernet preamble. Then, the whole Ethernet frame is transmitted (including Ethernet SOF at the end of the preamble, up to the CRC). The frame finishes with a Manchester violation with negative level followed by a "0" bit, which is also the first bit of the IDLE phase. It uses low-voltage differential signaling (LVDS), and the data rate is 100 Mbps to accomplish the fast Ethernet specification. The EBUS protocol simply encapsulates Ethernet frames; thus, EBUS can carry any Ethernet frame.

The maximum number of addressable devices is 2^{16} for each segment, while the distance between any two adjacent nodes (i.e., the cable length) can be up to 10 m for EBUS and (as usual) up to 100 m for Ethernet connections. The whole network size is practically unlimited (as, in theory, up to 2^{16} devices can be connected using 100 m Ethernet cables).

The topology of a communication system is one of the crucial factors for the successful application in automation environments. The topology has significant influence on cabling efforts, diagnostic features, redundancy options, and hot-plug-and-play features. Since the star topology commonly used for Ethernet leads to increased cabling efforts and infrastructure costs, a line or tree topology is preferable. Typically, the slave node arrangement represents an open ring. At one of the open ends, the master device sends frames, either directly or via Ethernet switches, and receives them at the other end after they have been processed. All frames are relayed from each slave to the next one.

The last slave in the network returns the frame back to the master. Thanks to the full-duplex capabilities of Ethernet, which uses two wire-pairs to carry out communications in both directions at the same time, the resulting topology visually resembles a physical line. Branches, which in principle are possible anywhere, can be used to enhance the basic line structure by making it possible to set up tree network topologies.

38.3 Communication Protocol

The EtherCAT communication protocol aims at maximizing the utilization of the Ethernet bandwidth, thanks to its very high communication efficiency. As mentioned above, the medium access mechanism (MAC) is based on the master/slave principle, where the master node (typically, the control system) sends Ethernet frames to the slave nodes over the physical ring, and each of the slave extracts data from (and inserts data into) the payload of these frames.

The frames sent over the network are standard Ethernet frames, whose data field encapsulates the EtherCAT frame. This one, in turn, is made up of a header and one or more EtherCAT datagrams or protocol data unit (PDUs), as to obtain a more efficient use of the large Ethernet data field available. EtherCAT PDUs are packed together without inter-PDU gaps. The frame is completed with the last EtherCAT PDU, unless the frame size is less than 64 octets, in which case the frame is padded to 64 octets in length. Each

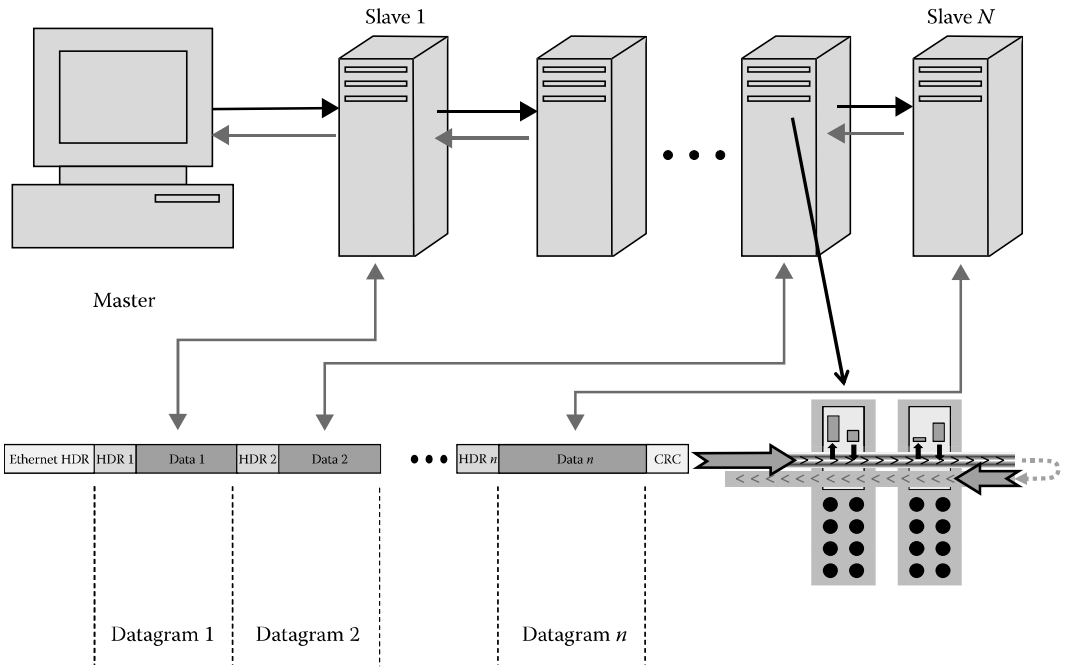

FIGURE 38.1 EtherCAT typical topology, with the "on-the-fly" frame processing.

EtherCAT PDU (that corresponds to a separate command) contains the header, data, and working counter (WC) fields. The standard Ethernet CRC is used to check the integrity of messages.

From the point of view of the master, the whole EtherCAT segment is seen as a single Ethernet device, which receives and sends standard Ethernet frames with the EtherType field set to 0x88A4 to distinguish it from other Ethernet frames. In this way, EtherCAT can coexist with other Ethernet protocols. The last EtherCAT slave device in the segment sends the fully processed frame back and the same occurs for each device. This procedure is based on the full-duplex mode of Ethernet: both communication directions are operated independently. In order to achieve the maximum performance, Ethernet frames have to be processed on-the-fly, as depicted in Figure 38.1. Slave nodes implemented in this way recognize relevant commands and execute them accordingly while the frames are passing through.

As said before, several EtherCAT datagrams can be embedded within the same Ethernet frame, each one addressing different devices and/or memory areas. As shown in Figure 38.2, the EtherCAT datagrams are transported either

1. Directly in the data area of the Ethernet frame or
2. Within the data section of a UDP datagram carried via IP

The first variant is limited to one Ethernet subnet since associated frames are not relayed by routers. For machine control applications, this usually is not a limitation. Multiple EtherCAT segments can be connected to one or several switches. The Ethernet MAC address of the first node in the segment is used for addressing the EtherCAT segment.

The second variant (via UDP/IP) implies a slightly larger overhead (because of the IP and UDP headers), but for less time-critical applications such as process control it enables the use of IP routing. On the master side, any standard UDP/IP implementation can be used in this case.

38.3.1 Commands

Each EtherCAT PDU (see Figure 38.3) consists of an EtherCAT header, the data area, and a subsequent counter area (WC).

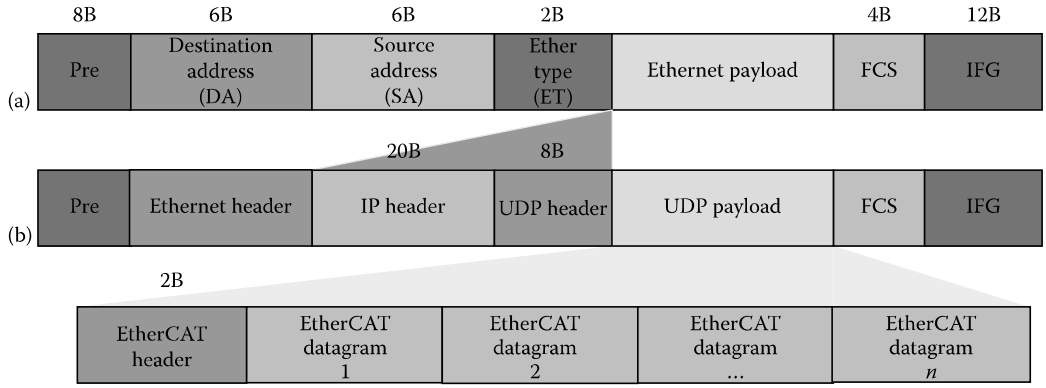

FIGURE 38.2 EtherCAT frame structure.

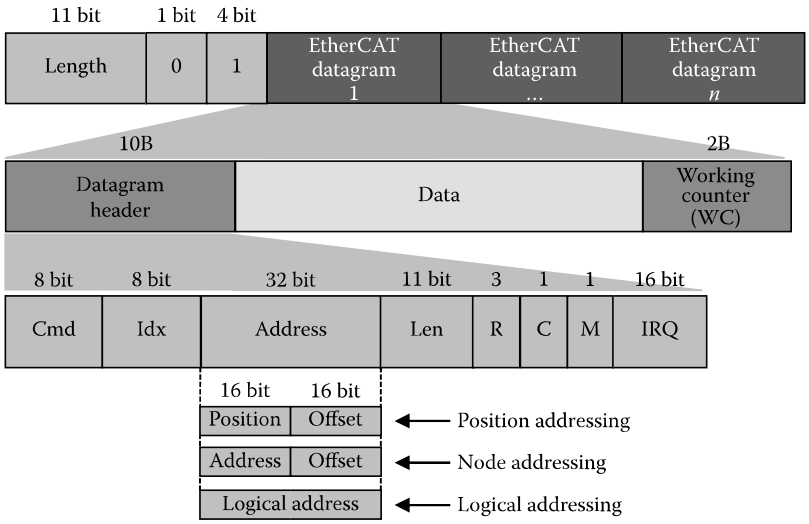

FIGURE 38.3 EtherCAT datagram structure.

In particular, each datagram contains the fields listed in Table 38.1.

The parameter "command" encodes the service command. Different types of commands can be used to optimize read and write operations on slave devices, which can be classified depending on the access type:

- *Broadcast read/write*: All slaves carry out a logical "OR" operation of data of the memory area and data of the incoming EtherCAT datagram, insert this information into the outgoing EtherCAT datagram, and write it back into the local memory area.
- *Read, write, or read/write actions*: Exchange of incoming data and local data. For read and write operations, the read operation is performed before the write operation.
- *Read multiple write actions* (*RMW*): Addressed slave will read while others will write.

The IDX field is the identifier of the datagram; it is left unchanged by the slaves. The address field can contain different information according to the addressing mode chosen (a more detailed description is provided in Section 38.4).

The parameter LEN shall contain the size in byte of the data to be read or written.

TABLE 38.1 EtherCAT Datagram Format

CMD	Unsigned8	EtherCAT command type
IDX	Unsigned8	Index for identification of duplicates(lost) datagrams
Address	DWORD	Address (autoincrement, configured station address, or logical address, see Section 38.4)
LEN	Unsigned11	Length of the DATA field
R	3 bit	Reserved 0x00
C	1 bit	Circulating frame 0x00: Frame is not circulating 0x01: Frame has circulated once
M	1 bit	0x00: Last EtherCAT PDU in EtherCAT frame 0x01: EtherCAT PDU in EtherCAT frame follows
IRQ	WORD	External EtherCAT event request registers of all slaves combined with a logical OR
DATA	OctetString LEN	Data
WC	WORD	Working counter

The field C is used in the case of a network with a link failure between two slaves; a frame currently traveling through the part of the ring on the isolated side of the network might start circulating. Indeed, each slave has a mechanism called loop-back function that forwards Ethernet frames to the next internal port if either there is no link at a port or the port is not available. This feature is needed to create the ring on the last slave. Thus, to prevent an endless frame from circulating, a slave with no active link through the master has to do the following:

- If the circulating bit of the EtherCAT datagram is 0, set the circulating bit to 1.
- If the circulating bit is 1, do not process the frame and destroy it.

M specifies whether or not the EtherCAT PDU is the only one in the frame.

An important mechanism is based on the IRQ field. EtherCAT event requests are used to inform the EtherCAT master of slave events. The EtherCAT event request register is combined using a logical AND operation with a predefined EtherCAT event mask register. This one is used for selecting the interrupts which are relevant for the EtherCAT master. The resulting bits are then combined with the EtherCAT IRQ field present on the incoming datagram with a logical OR operation and written into the EtherCAT IRQ field of the outgoing datagram. It is worth noting that the master is not able to distinguish which slave(s) was/were originating an interrupt.

Finally, process data that can be read or written are placed in the DATA field. In the case a slave is writing information in the data field, it must recompute and modify the CRC field at the end of the packet to make it correct for next slaves.

The last field is the WC. This field is used to count the number of devices that were successfully addressed by an EtherCAT datagram. Each addressed slave increments the WC in hardware. Because each datagram has an expected WC value computed by the master, the valid processing of datagrams can be checked by comparing the received WC with the expected value. The WC is increased if at least one byte/bit of the data field was successfully read and/or written. Complex operations like RMW commands are treated like either a read or a write command, depending on the address matching mechanism described above.

38.4 Addressing

Different addressing modes are supported for accessing slaves, as shown in Figure 38.3. The header within the EtherCAT PDU contains a 32 bit address, which is used for both physical node addressing and logical addressing.

38.4.1 Physical Addressing

In the physical addressing scheme, the 32 bit address field within each EtherCAT PDU is split into a 16 bit slave device address and a 16 bit physical address within the slave device, thus leading to 2^{16} slave device addresses, each one with an associated 16 bit local address space. With device addressing, each EtherCAT PDU uniquely addresses one single slave device.

This mode is mostly suitable for transferring parameterization information for devices. There are two different device addressing mechanisms, namely, position addressing and node addressing. There is also a third mode for broadcast messages.

Now let us consider the node addressing mode (also defined as configured station address); because EtherCAT slaves can have up to two configured station addresses (if this functionality is enabled), the address may be either assigned by the master (configured station address) or read from the internal EEPROM that can be changed by the slave application (configured station alias address).

The following addressing alternatives are available:

- *Position address/autoincrement address*: Position addressing is used to address each slave device via its physical position within the segment. Each slave device increments the 16 bit address field as the datagram transits through the slave device; the device which receives a frame with an address field with value 0 is the one being addressed. This means that the field contains the negative position of the slave in the loop (the master position is marked "0"). Due to the mechanism employed to update this address while the frame is traversing the node, the slave device is said to have an autoincrement address. Position addressing should be used during the start-up phase of the EtherCAT system only to scan the fieldbus; it can be adopted occasionally to detect newly attached slaves because it can cause problems if loops are closed temporarily due to link problems.
- *Node address/configured station address and configured station alias*: In this case, the slaves are addressed via configured node addresses assigned by the master during the data-link start-up phase. This ensures that, even if the segment topology is changed or devices are added or removed, the slave devices can still be addressed via the same configured addresses. Node addressing is typically used for register access to individual and already identified devices.
- *Broadcast*: Each EtherCAT slave is addressed. Broadcast addressing is used for initialization of all slaves and for checking the status of all slaves if they are expected to be identical.

Each slave device has a 16 bit local address which is addressed via the offset field of the EtherCAT datagram.

38.4.2 Logical Addressing: FMMU

Using the datagram structure described above, several EtherCAT devices can be separately addressed via a single Ethernet frame, each one by means of one EtherCAT datagram, which leads to a significant improvement of the system bandwidth. However, for small-size input terminals, for example, with 2 bit input devices that map precisely 2 bit of user data, the overhead of a single EtherCAT command might be excessive.

The FMMU lessens this problem, and the available utilization ratio is more than 90%—even for devices with only 2 bits of user data, as mentioned before. Similar to the memory management units (MMUs) in modern processors, the FMMU converts a logical address into a physical one via an internal table. The FMMU is integrated in the EtherCAT slave architecture and enables individual address mapping for each device. In contrast to processor-internal MMUs that typically map complete memory pages, the FMMU also supports bit-wise mapping. This enables, for example, the 2 bits of an input terminal to be inserted individually anywhere within a logical address space. If an EtherCAT command is sent to read or write a certain memory area, instead of addressing a particular EtherCAT device, the 2 bit input terminal inserts its data at the right place within the data area. In the same way, other terminals

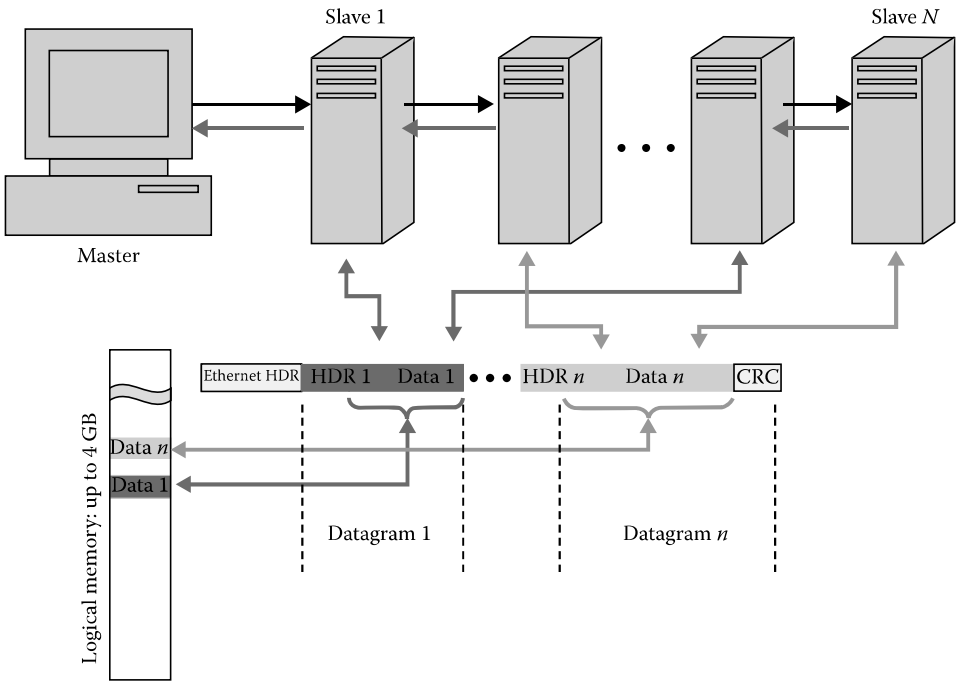

FIGURE 38.4 FMMU mapping example.

that are addressed in the FMMU also insert their data, so FMMUs allow to use logical addressing for data segments that span several slave devices: one command addresses data within several arbitrarily distributed slaves, as shown in Figure 38.4. The access type supported by an FMMU is configurable to be read, write, or read/write. Configuration of the FMMU entities is performed by the master device and transferred to the slave devices during the data-link start-up phase. For each FMMU entity, the following items are configured: a logical, bit-oriented start address, a physical memory start address, a length, and a type that specifies the direction of the mapping (input or output). When an EtherCAT datagram with logical addressing is received, the slave device checks whether one of its FMMU entities exhibits an address match. If appropriate, it inserts data at the associated position of the data field into the datagram (read operation) or extracts data from the associated position (write operation).

38.5 SyncManager

EtherCAT provides a mechanism to synchronize slave memory access. Because the memory of a slave can be used for exchanging data between the master and the local application without any restrictions, some problems can arise. First of all, data consistency is not guaranteed, and a mechanism like semaphores has to be implemented for exchanging data in a coordinated way. Moreover, both master and slave applications have to access the memory in order to know when the memory is no longer used on the other side.

Thus, SyncManager enables consistent and secure data exchanges between the EtherCAT master and the local application, generating interrupts to inform both sides of changes. It is organized in channels, and each channel defines a consistent area of the memory.

The SyncManager system is configured by the EtherCAT master. The communication direction is configurable, as well as the communication mode (buffered mode and mailbox mode). The SyncManager system uses a buffer located in the memory area for exchanging data and controls the access to this buffer. All accesses to the buffer begin from the start address, otherwise the access is denied. After the access to

the start address, the whole buffer can be accessed, either as a whole or in several strokes. A buffer access finishes by accessing the end address.

Two communication modes are supported by SyncManagers:

- *Buffered mode*. The interaction between the producer and the consumer of data is uncorrelated, and each entity can perform the access at any time, always providing the consumer with the newest data. In the case the buffer is written faster than it is read out, old data are dropped. The buffered mode is typically used for cyclic process data. The mechanism is also known as three-buffer mode because three buffers of identical size are used. One buffer is allocated to the producer (for writing), one buffer to the consumer (for reading), and a spare buffer helps as intermediate store. Reading the last byte or writing the last byte results in an automatic buffer exchange.
- *Mailbox mode*. The mailbox mode implements a handshake mechanism for data exchange, so that no data will be lost. One entity fills data in and cannot access the area until the other entity reads out the data. At first, the producer writes to the buffer and locks it for writing until the consumer has read it out. Afterward, the producer has write access again, while the buffer is locked for the consumer. The mailbox mode is typically used for application layer (AL) protocols.

38.6 Distributed Clock

An important mechanism present in the EtherCAT protocol is the distributed clock synchronization that enables all devices (master and slaves) to share the same system time with a precision smaller than 1 μs. In this way, all devices can be synchronized, and consequently, slave applications are synchronized as well. Main features of this mechanism are

- Generation of synchronous output signals
- Precise time stamping of input events
- Synchronous digital output updates
- Synchronous digital input sampling

Typically, the clock reference (system time) is the first slave with distributed clock capability after the master within one segment. This system time is used as the reference clock to synchronize the DC slave clocks of other devices and of the master itself. The propagation delays, local clock source drifts, and local clock offsets are taken into account for the clock synchronization. All settings are done during the clock synchronization process made of three steps:

1. *Propagation delay measurement*: The master sends a synchronization datagram at certain intervals (as required in order to avoid the slave clock diverging beyond application-specific limits), and each slave stores the time of its local clock twice—once when the message is received and once when it is sent back (remember EtherCAT has a ring topology). The master reads all time stamps and computes the propagation delays between all slaves. It is worth noting that clocks are not synchronized for the delay measurement, and only local clock values are used. So the propagation delay calculation is based on receive time differences.
2. *Offset compensation to reference clock (system time)*: Because the local time of each device is a free running clock that has not the same time as the reference clock, a compensation mechanism is necessary. To achieve the same absolute time, all device difference between the reference clock and every slave device's clock is computed by the master. Then the offset time is written to a specific slave register "system time offset." Each slave computes its local copy of the system time using its local time and the local offset value. Moreover, small offset errors are eliminated by the drift compensation.
3. *Drift compensation to reference clock*: After the delay time between the reference clock and the slave clocks has been measured, and the offset between both clocks has been compensated, the natural drift of every local clock has to be compensated by a time-control loop. This mechanism readjusts the local clock by regularly measuring the differences.

38.7 Application Layer

The EtherCAT state machine is responsible for the coordination of master and slave applications at start-up and during operation.

State changes are typically initiated by requests of the master. They are acknowledged by the local application after the associated operations have been executed. Unsolicited state changes of the local application are also possible. Simple devices without a microcontroller can be configured to use EtherCAT state machine emulation. These devices simply accept and acknowledge any state change automatically.

The state machine is controlled and monitored using some registers present on the slave. The master requests state changes by writing to the AL control register. The slave indicates its state writing in the AL status register and puts possible error codes into the AL status code register.

As shown in Figure 38.5, there are four basic states an EtherCAT slave shall support, plus one optional state:

- Init
- Pre-operational
- Safe-operational
- Operational
- Bootstrap (optional)

All state changes are possible except for the "Init" state and for the "Pre-operational": for the former the only possible transition is the one to the "Pre-operational" while for the latter there is no direct state change to "Operational."

State changes are normally requested by the master. The master requests a write to the AL control register, the slave responds through a local AL status register write operation after a successful or a failed state change. If the requested state change failed, the slave also responds by setting an error flag.

The Bootstrap state is optional, and there is only a transition from or to the Init state. The only purpose of this state is to download the device's firmware. In the Bootstrap state, the mailbox is active but restricted to the file access over EtherCAT service (FoE) protocol.

38.7.1 Mailbox Services

Another important characteristic of EtherCAT is the possibility to feature multiprotocol capability, consolidating high- or same-level protocols in a standardized mailbox. In particular, the following capabilities are supplied:

- *Ethernet over EtherCAT (EoE)*: tunnels standard Ethernet frames over EtherCAT
- *CANopen over EtherCAT (CoE)*: access to CANopen devices and their objects, with also an optional event-driven PDO (process data object) message mechanism

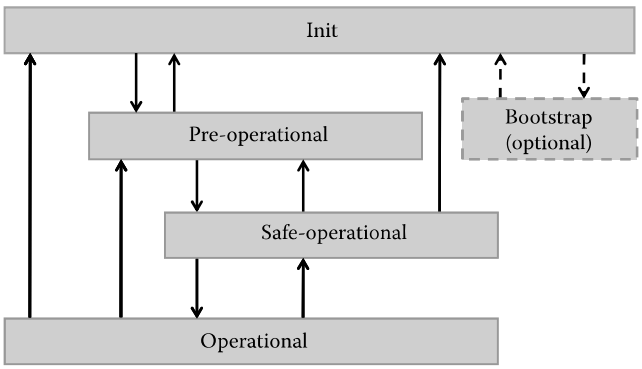

FIGURE 38.5 EtherCAT state machine.

- *File access over EtherCAT (FoE)*: download and upload of firmware and other files
- *Servo drive over EtherCAT (SoE)*: access to servo profile identifier (IDN)

The EoE and FoE protocols provide options for integrating a Web server using standard IP-based protocols such as TCP/IP, UDP/IP, and all higher protocols based on these (HTTP, FTP, SNMP, etc.). For example, EtherCAT can use a mechanism in which the Ethernet datagrams are tunneled and reassembled in a associated device, before being relayed as complete Ethernet datagrams. This procedure does not restrict the achievable cycle time since the fragments can be optimized according to the available bandwidth.

Another important aspect for a fieldbus system is to support communication protocols used for compatibility and efficient data exchange between a controller and a drive; for these purposes, EtherCAT uses well-known and established technologies. Indeed, the "CANopen over EtherCAT" (CoE) protocol enables the complete CANopen profile family to be used via EtherCAT. The SDO (service data object) protocol is used directly, so that existing CANopen stacks can be left practically unchanged. The process data are organized in PDOs, which are transferred using the efficient support of EtherCAT. Moreover, optional extensions are defined that lift the 8 byte limit and enable complete readability of the object list.

Another protocol supported is the "Servo profile over EtherCAT" (SoE). This capability enables the proven servo profile, which is specialized for demanding drive technology, to be used. The servo service channel, and therefore access to all parameters and functions residing in the drive, is mapped to the EtherCAT mailbox.

References

[BEC06] Beckoff Automation GmbH, EtherCAT per Motion Control, presented at MOTION CONTROL for, ilB2B.it, Bologna, Italy, February 23, 2006, [Online], available: http://www.ilb2b.it/mc4_2006/atti/BECKHOFF.pdf

[ESC20] Hardware Data Sheet ESC20 Slave Controller, [Online], available: Beckhoff website http://www.beckhoff.com/

[ETG04] ETG.1000.4: Data link protocol specification, available on EtherCAT Technology Group: http://www.ethercat.org/

[ETG05] ETG.1000.5: Application layer service definition, available on EtherCAT Technology Group: http://www.ethercat.org/

[IE158] International Electrotechnical Commission, Industrial communication networks—Fieldbus specifications—Part 3-12: Data-link layer service definition—Part 4-12: Datalink layer protocol specification—Type 12 elements, IEC 61158-3/4-12 (Ed.1.0).

[IE784] IEC 61784-2, Industrial communication networks—Profiles—Part 2: Additional fieldbus profiles for real-time networks based on ISO/IEC 8802-3.

[PRY08] Prytz, G., A performance analysis of EtherCAT and PROFINET IRT, *IEEE International Conference on Emerging Technologies and Factory Automation, 2008 (ETFA 2008)*, Hamburg, Germany, September 15–18, 2008, pp. 408–415.