18

Ethernet for Control Automation Technology

Gianluca Cena
National Research Council of Italy

Stefano Scanzio
National Research Council of Italy

Adriano Valenzano
National Research Council of Italy

Claudio Zunino
National Research Council of Italy

18.1 Introduction

Ethernet for Control Automation Technology (EtherCAT) [ETG12] is a high-performance Ethernet-based fieldbus system. The main reason for its development was the adoption of Ethernet in automation applications, where short cycle times and low communication jitters are required. Nowadays, EtherCAT is a popular solution for enabling access to field devices by control applications in industrial environments, including motion-control systems. Communication equipment and devices are available off-the-shelf, which are based on purposely developed hardware [ET1100].

The EtherCAT protocol is an open standard currently managed by the EtherCAT Technology Group (ETG). Its specification has been integrated into the international fieldbus standards IEC 61158 [IE158] and IEC 61784 [IE784]. EtherCAT is based on a master/slave approach and relies on a ring topology at the physical level. Only one master is allowed in the network, and this is suitable, for instance, to connect a control unit (e.g., a programmable logic controller [PLC]) to decentralized peripherals (sensors, actuators, drives, etc.). By using suitable gateways, EtherCAT can interoperate with both conventional Transmission Control Protocol (TCP)/Internet Protocol (IP)-based networks (intranets) and other real-time Ethernet (RTE) solutions, such as EtherNet/IP and/or PROFINET.

The master node is in complete control of the traffic exchanged over the EtherCAT network. In particular, it is the only device that can take the initiative in the communication; hence, it is responsible for initiating all data exchanges with the slaves. Each slave processes the received frame in order to extract/insert data from/into it. Then, the frame is forwarded to the next slave in the ring.

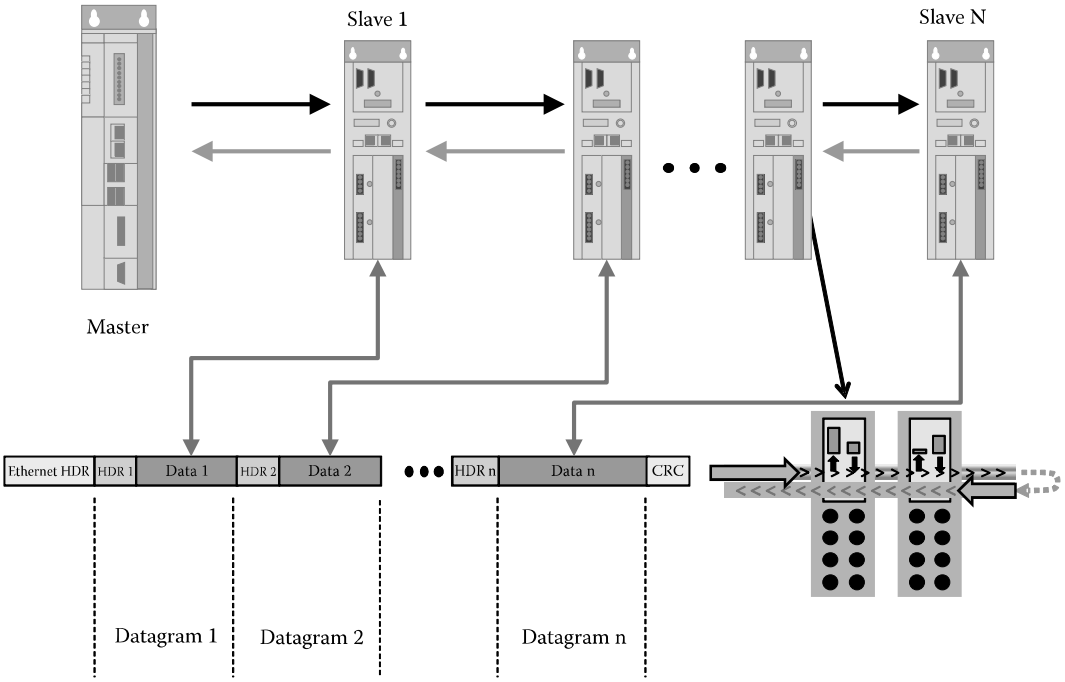

FIGURE 18.1 EtherCAT typical topology, with the *on-the-fly* frame processing.

Unlike Ethernet switches and bridges, slaves do not manage frames according to a conventional store-and-forward approach, which implies receiving the frame, decoding the related protocol control information, and sending the message out. Every frame, instead, is processed on-the-fly by the slave data link layer. In order to ensure high performance, frame processing and relaying take place at the same time so that these operations have to be carried out in hardware. This explains why specialized components are used for slaves, which are known as EtherCAT slave controllers (ESCs) [ET1100] (Figure 18.1).

18.2 Physical Layer

The physical layer of EtherCAT relies on the proven fast Ethernet transmission technology, which enables high data rates that are more than adequate to the communication needs of next-generation industrial plants. A low-cost solution, namely, EBUS, is also foreseen for modular devices, which is able to coexist with conventional Ethernet seamlessly and does not impair performance in any way. Although the use of switches is not recommended to ensure real-time behavior, the EtherCAT network architecture is quite flexible and can also stretch over wide areas.

18.2.1 Communication Support

Though EtherCAT is correctly listed among the industrial Ethernet solutions available today, it actually supports two different types of physical layers, namely, Ethernet and EBUS. The first alternative relies on the conventional 100 Mb/s full-duplex Ethernet technology (either 100BaseTX or 100BaseFX), according to the popular IEEE 802.3 standard [8023]. It is typically used for connecting the master to the network segment to which slaves are attached. Indeed, the whole EtherCAT segment (also known as *Type 12 segment*) is seen as a single, large Ethernet device by the master, which *concurrently* receives and sends Ethernet frames by exploiting full-duplex transmissions. However, this *device* does not consist of a single Ethernet controller but includes a (possibly very large) number of EtherCAT slaves connected

so as to form a ring topology. Ethernet is also used to interconnect slave devices, for instance, when they are located in distinct racks or fastened to different mounting rails.

The transmission medium, in this case, consists of a Cat 5 twisted pair (either shielded or unshielded, depending on the amount of electromagnetic interference) although higher category cables are also allowed. Both classic RJ45 (8P8C) and circular M12 D-code connectors can be used.

The Ethernet transceivers (PHYs) needed to interface ESCs to Ethernet have to satisfy a number of requirements in order to be used in EtherCAT. In particular, they must be able to provide either a standard media-independent-interface (MII) or a reduced one (RMII) and have to support the MII management interface, auto-negotiation, and auto-crossover.

Conversely, EBUS can be used only as a backplane bus and is not intended for wire connections. EBUS, in fact, was mainly conceived to interconnect modules in modular devices. It is worth noting that, unlike other fieldbuses that enable a modular design for devices, the sequence of logical bits that EtherCAT transmits over EBUS is exactly the same as for Ethernet. This means that switching from Ethernet to EBUS (and vice versa) can be done quickly, efficiently, and inexpensively (in practice, only transceivers have to be replaced).

EBUS is an inexpensive physical layer that features reduced pass-through delays inside the slaves. Typically, frames experience delays on the order of 120 ÷ 500 ns [PRY08,CEN10] when propagating through EBUS interfaces, whereas longer latencies (about 1 μs) are introduced by Ethernet interfaces. EBUS uses the Manchester (biphase L) encoding scheme and marks the start and end of Ethernet frames with a pair of delimiters, namely, the start of frame (SOF) and end of frame (EOF) identifiers. The beginning of a frame is defined by a Manchester code violation with a positive level following an idle condition. A bit at the logical value "1" follows, which is the first bit of the Ethernet preamble. Then, bits of the whole Ethernet frame are transmitted up to the cyclic redundancy check (CRC). The frame ends with a Manchester violation with negative level, followed by a bit at the logical value "0," which is, in practice, the first bit of the subsequent IDLE phase.

EBUS uses low-voltage differential signaling (LVDS), and the bit rate is selected equal to 100 Mbit/s to match the fast Ethernet specification. The EBUS protocol simply encapsulates Ethernet frames; thus, it can carry any kind of Ethernet message besides EtherCAT.

18.2.2 Network Topology

The topology of a communication system is one of the crucial factors for its successful adoption in industrial environments, since it affects a number of key elements such as cabling, diagnostic features, redundancy options, and hot-plug-and-play features. The star topology, commonly used for switched Ethernet, implies significant cabling and infrastructure costs; hence, line or tree topologies are usually preferable in factory networks. Typically, slave devices in EtherCAT segments are connected in linear structures and exploit a daisy chain wiring scheme. Every slave is provided with (at least) two Ethernet ports, to connect downstream and upstream devices. The last slave in the segment performs a loopback function and returns the frame in the opposite direction to the master. The master is the headend of the structure and requires one Ethernet port only.

Each slave relays all frames it receives to the next device in the EtherCAT segment. Because of the daisy chain connection and loopback, all the slaves in the segment form an open ring. The master transmits frames at one of the ends of this open ring and receives them at the other end, after they have been processed by every slave. This means that on the whole the physical topology of an EtherCAT network is actually a ring.

Thanks to the full-duplex capabilities of Ethernet, which uses two pairs of wires housed in the same cable to carry out communications in both directions simultaneously, the resulting topology resembles, nevertheless, a physical line, as in most legacy fieldbuses. The reduction of wiring complexity helps in making the network deployment easier and lowers the installation costs at the same time.

In principle, branches can be introduced anywhere in an EtherCAT segment, by using devices equipped with three or more ports (EtherCAT couplers). This kind of devices are provided, for example,

with two Ethernet ports and one EBUS interface for direct connection of input/output (I/O) modules (also known as EtherCAT terminals) and can be used to enhance the basic line structure setting arbitrarily complex tree networks topologies. It is worth noting that, in this case, every slave device located at the end of a branch has to close the ring on its own using the loopback function.

The maximum number of addressable devices in EtherCAT is quite large, since 2^{16} nodes for each segment are allowed. The limit depends on the data link layer, and in particular on the address field, which is encoded on 16 bits. In the same way, the maximum network extension is usually able to satisfy the requirements of most real applications. The limitation, in this case, is mainly due to the maximum distance allowed between any two adjacent nodes (i.e., the length of the cable), which in turn depends on the underlying transmission support (up to 10 m for EBUS and up to 100 m for Ethernet connections). This means that the whole network size is practically unlimited: in theory, up to 2^{16} devices can be connected in daisy chain using 100 m Ethernet cable segments.

18.2.3 Device Architecture

Many asymmetric communication technologies conceived for industrial environments have different internal architectures for masters and slaves, and this is true for EtherCAT devices too.

EtherCAT masters (EMs) rely on standard communication hardware (full-duplex Ethernet network interface controllers) and dedicated software, even though open-source solutions based on Linux-like operating systems are also available. On the contrary, purposely designed hardware components (ESCs) have to be necessarily used for slaves. In order to reduce the implementation costs, frame processing by ESCs occurs in one direction only, which is known as *processing path*. The reverse direction, known as *forwarding path*, is needed to propagate frames in the ring. From a logical point of view, ESCs exhibit an active behavior only on the processing path (frame modifications on the forwarding path are not allowed), but at the physical layer, they behave as repeaters in both directions. Consequently, they are able to regenerate electrical signals so that network equipment like stand-alone repeaters is no longer necessary. This also reduces connection costs and complexity in large installations.

Most ESCs are internally equipped with two or more ports, depending on device complexity. For example, the Beckhoff ET1100 [ET1100] is provided with four separate ports, which can be individually configured to operate as either EBUS or MII. Port 0 is the *upstream* port whereas the others are used for *downstream* connections and to forward signals. Each port implements two functions, called *auto-forwarder* and *loopback*. The auto-forwarder block performs frame checks, such as CRC error detection, at the physical level and manages the error count. It is also responsible for taking timestamps on frame receptions, a mechanism which is needed, for instance, by the clock synchronization protocol. The loopback function, instead, forwards frames to the next logical port if the related link is not available. In this way, the ring is automatically closed in the case of faults affecting either devices or links.

18.3 Data Link Layer

The data link protocol of EtherCAT was designed to maximize the utilization of the Ethernet bandwidth and to grant a very high communication efficiency. As mentioned earlier, the access mechanism of EtherCAT is based on a master/slave approach, where the master node (typically the control unit, e.g., a PLC) sends Ethernet frames to slave nodes. Slaves, in their turn, either extract data from the frame payload or insert information by overwriting part(s) of the payload itself.

18.3.1 Frame Format

Messages sent over the network are standard Ethernet frames, with EtherCAT frames (also known as *Type 12 frames*) encapsulated in the data field. Consequently, they include the conventional fields:

preamble (8 bytes), destination and source MAC addresses (6 bytes each), EtherType (2 bytes, set to 0x88A4 to distinguish them from non-EtherCAT frames), frame check sequence (Fcs, 32 bits), and the interframe gap.

An EtherCAT frame, in turn, contains a frame header (2 bytes) and one or more EtherCAT datagrams, also known as *Type 12 Data Link Protocol Data Units* (DLPDU) according to the data link layer standard specifications. In this way, the large data field made available by conventional Ethernet can be better exploited to increase the communication efficiency. DLPDUs are packed together, one after the other, without intermediate gaps. The payload of the Ethernet frame ends with the last DLPDU, unless its overall size is 63 octets or less. In this case, the frame is padded to 64 octets in length, as required by the Ethernet specifications. The standard Ethernet CRC closes the frame and is used by each device (either master or slave) to check the integrity of the message. Thanks to the EtherType field, EtherCAT can coexist, in theory, with other Ethernet protocols.

Each DLPDU corresponds to a separate EtherCAT command and consists of three sections: header, data, and counter field. Commands are used to perform data exchanges: basically, they are issued by the master for reading or writing specific memory areas in the slave devices.

Ethernet frames, and in particular DLPDUs they embed, are processed in sequence by slaves. Each slave recognizes its commands of interest and executes them while the frames are passing through. Because of the physical ring topology, a frame is returned to the master after being processed by all the slaves. This procedure exploits the full-duplex mode of Ethernet, which means that the two communication directions can work independently.

Several DLPDUs can be embedded in the same Ethernet frame, each one addressing different devices and/or memory areas. As shown in Figure 18.2, DLPDUs are transported either (a) directly in the data field of the Ethernet frame or (b) within the data section of a datagram, by means of the User Datagram Protocol (UDP).

The first variant is limited to a single subnetwork, since Ethernet frames are not relayed by routers. Usually, this is not a limitation at all, for machine control applications. Direct Ethernet encapsulation is by far the most widespread EtherCAT solution at the shop floor of factory automation systems. In theory, multiple EtherCAT segments can be connected to a single master through one or more switches, and the MAC address of the first node in each segment is used for addressing the segment itself. However, this approach can affect the real-time properties of the communication.

On the one hand, the second variant, which relies on UDP and the IP, implies lightly larger overheads (because of the IP and UDP headers) and is also limited by routers, which are typically unable to guarantee a real-time behavior. On the other hand, this solution also enables IP routing; hence, it is suitable for applications having loose timing requirements, such as in process automation. Any standard UDP/IP implementation can be used in this case on the master side.

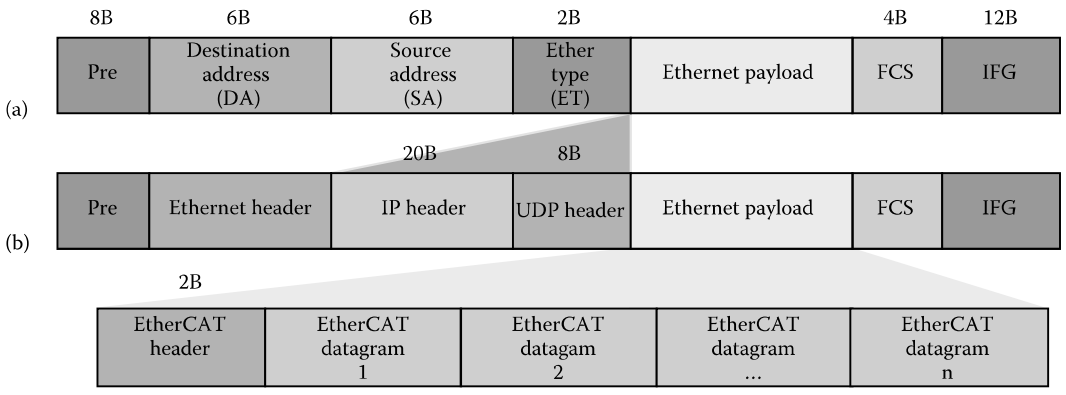

FIGURE 18.2 EtherCAT frame structure.

18.3.2 DLPDU Format

As shown in Figure 18.3, each DLPDU consists of a number of fields. The initial fields (up to IRQ included) can be assumed to belong to the header part, which has a fixed size (10 bytes). The variable-sized data area is placed immediately after the header and includes the information to be exchanged, often referred to as *data link service data unit* (DLSDU). The last field in the frame is the *working counter* (WKC), used mainly for checking whether a command has been successfully executed by the relevant slaves.

Table 18.1 summarizes sizes, formats, and meanings of the DLPDU fields.

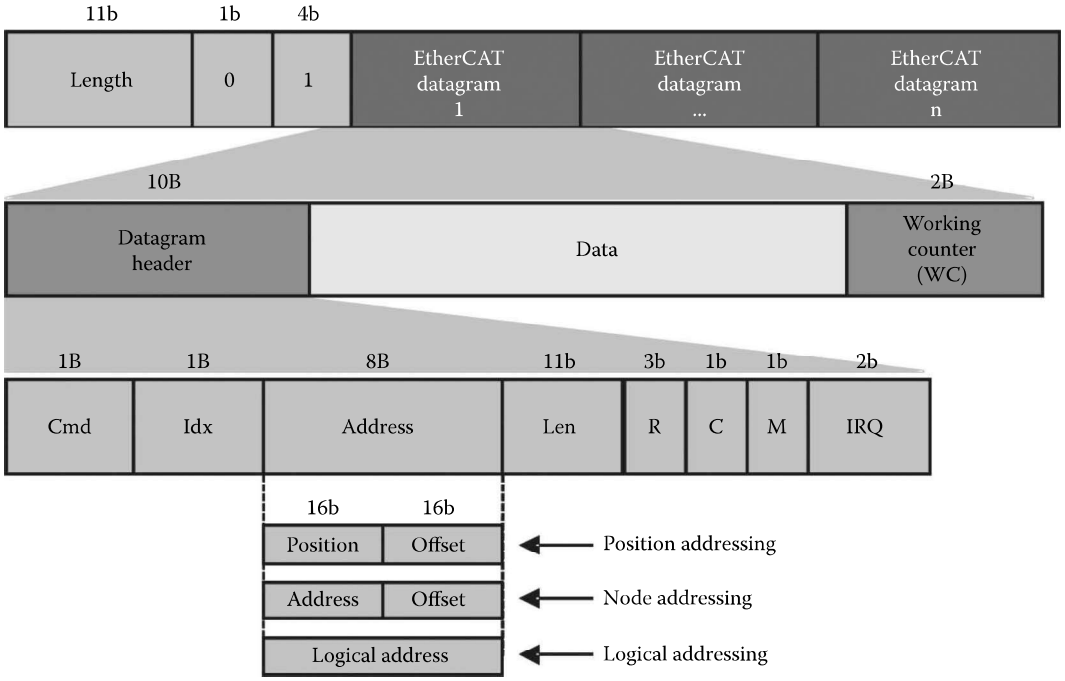

FIGURE 18.3 EtherCAT datagram structure.

TABLE 18.1 Generic Type 12 DLPDU Format

Field	Type	Meaning
CMD	Unsigned8	Service command (0–14, values from 15 to 255 are reserved).
IDX	Unsigned8	Index for identifying duplicate/lost DLPDUs.
Address	DWORD	Address (auto-increment, configured station address, broadcast, or logical address [see below]).
LEN	Unsigned11	Length of the DATA field.
R	Unsigned3	Reserved 0x00.
Circulating frame (C)	Unsigned1	0x00: Frame is not circulating.
		0x01: Frame has already circulated once.
NEXT (M)	Unsigned1	0x00: this is the last DLPDU in the EtherCAT frame.
		0x01: more DLPDUs follows in the EtherCAT frame.
IRQ	WORD	External Event Request registers of all slaves combined via OR.
DATA	OctetString LEN	Data (DLSDU).
WKC	WORD	Working counter.

The service command (1 byte) is encoded in the CMD parameter. Different types of command exist, which can be used to carry out highly optimized read and write operations on slave devices. Generally speaking, they can be grouped according to the access type:

- *Read* (RD) is used by the master to read memory areas or registers from slave devices.
- *Write* (WR) is used by the master to write to memory areas or registers of slave devices.
- *Read/Write* (RW) is used by the master to carry out both a read and a write operation at the same time; in this case, reading is performed by the slave before writing.
- *Read/Multiple Write* (RMW) is a quite peculiar service, where the addressed slave carries out a read operation while all other slaves are performing a write action.

The IDX field (1 byte) is used by the master to identify a given DLPDU. This is useful, for instance, to pair a frame returned on the forward path with those sent on the processing path, when more frames are allowed to travel along the ring at the same time. This field shall be left unchanged by the slaves.

The address field (4 bytes) can contain different information, encoded as either a single 32 bit double word or two 16 bit words, according to the addressing mode chosen for the DLPDU. A more detailed description is provided in the next section.

The parameter LEN (11 bits) is the size in bytes of the DATA field included in the DLPDU.

The *circulating frame* (C) bit is used to cope with failures of links between slaves. Because of the loop-back function (which forwards Ethernet frames to the next internal port if the current port is neither connected nor available), if a part of the network is cut off from the master, it becomes an insulated ring. If no action is taken, frames that are currently traveling there might continue to circulate. To prevent endless frame circulation, a slave with no active link to the master has to remove frames from the insulated ring according to the following rules:

- If the C bit of the DLPDU is 0, the C bit is set to 1.
- If the C bit of the DLPDU is 1, the frame is not processed but it is destroyed.

The NEXT field (also called Mbit) specifies whether the DLPDU is the last one in the frame or another DLPDU follows.

An important mechanism is implemented through the IRQ field. EtherCAT event requests are used to inform the master of slave events. On each slave, the *External Event* register is combined, using a logical "AND" operation, with a predefined *External Event Mask* register. This mask is used for selecting interrupts that are relevant to the master. The resulting bits are combined with the IRQ field in the incoming DLPDU using a logical "OR" operation, and then written into the IRQ field of the outgoing DLPDU. This implements a quasi-asynchronous event notification mechanism. It is worth noting that, when using the IRQ mechanism, the master is not able to single out what slave(s) originated the interrupt.

Process data that have to be read or written (DLSDU) are placed in the DATA field. When a slave modifies any field in the DLPDU (a quite frequent event), the CRC field has to be recomputed and updated so that it looks correct to the following slaves.

Finally, the WKC field is used as a further error detection mechanism. Basically, it counts the number of devices that were successfully addressed by the DLPDU. Each addressed slave, when reading/writing data to/from the DLPDU, increments the WKC in hardware. In most control applications, each DLPDU has an expected value for WKC, which is known in advance and depends on the number of involved slaves. Therefore, the correctness of datagram processing can be checked by the master by comparing the value of the WKC field in the received DLPDU with the expected value.

WKC is incremented by one in the case at least 1 byte/bit of the data field has been successfully accessed by the slave for read, write, and read/multiple write commands. In the case of read/write commands, instead, WKC is incremented by two for successfully written data and, additionally, by one for a successfully read operation.

18.3.3 Addressing

Different addressing modes are supported for accessing information in the slaves, as shown in Figure 18.3. The DLPDU header contains a 32 bit address field, which is used for both *physical* and *logical* addressing.

18.3.3.1 Physical Addressing

In physical addressing, the address field is split into two 16 bit words: the first one, the address position (ADP) encodes the address of a slave device in the EtherCAT segment, while the second, the address offset (ADO) refers to either a physical memory location or a register in the slave device.

This solution enables up to 2^{16} slave devices in the network, each one associated to a local memory as wide as 2^{16} cells. In other words, every slave is assigned its own 64 kB physical address space. The first 4 kB block (address range from 0x0000 to 0x0fff) is used for *registers* and user memory. The remaining space (from address 0x1000 onwards) is the 60 kB *process memory*. Its actual size, however, depends on the specific device considered. The address space of any ESC is directly addressable by both the EM and the attached microcontroller. In physical addressing, each DLPDU refers to a specific memory area (or register) in either a single slave or all slaves at the same time (broadcast). For this reason, this mechanism is mostly suitable for transferring parameterization information to devices.

Basically, slaves can be physically addressed in two different modes, namely, position and node addressing, which rely on the information included in the ADP field. A third mode is also available for dealing with broadcast DLPDUs. In practice, the following alternatives are possible:

- *Position* (or *auto-increment*) *addressing*: In this case, each slave device is identified via its physical position in the EtherCAT segment. When processing a DLPDU that makes use of this kind of addressing, each slave increments the ADP field by one. When a DLPDU is received with the ADP field set to 0, the device assumes to be the intended destination. As a consequence, the master has to preset ADP to a negative value that corresponds to the absolute position of the target slave (the first slave, which is next to the master, is conventionally assumed to have position 0). For instance, an initial ADP value equal to –2 identifies the third slave in the segment. Because of the mechanism, which updates the address field while the frame is traversing a device, the DLPDU is said to adopt auto-increment addressing. Position addressing should be used only during the start-up phase to scan the EtherCAT segment and can be adopted occasionally to detect new slaves attached to the segment. This is because problems may arise if some network branches are temporarily closed because of link errors. In this case, in fact, a wrong count could lead to address a wrong slave, possibly disrupting the correct system operations.
- *Configured addressing*: In this case, slave devices are identified by means of node addresses (*configured station address* and *configured station alias*) that are assigned at start-up. This choice grants that, even if the segment topology is changed and/or devices are added to/removed from the network at runtime, every slave can nevertheless be safely accessed through the same predefined address. In this working mode, every slave compares the ADP field in the DLPDU with its own configured address(es). If a match is found, the device is the intended destination. Node addressing is typically used for accessing (already configured) devices individually. Each EtherCAT slave can be assigned a pair of station addresses. Assignment can be done by either the EM (configured station address), during the data link start-up phase, or the application running on the slave (configured station alias), for instance, by reading the address value from an internal nonvolatile memory during the device initialization.
- *Broadcast*: In this case, all slaves in the EtherCAT segment are addressed and shall execute the command encoded in the DLPDU. As for position addressing, the ADP field is incremented by one by each slave during the DLPDU propagation so that the master can eventually obtain the number of slaves that are present in the network. It is worth noting, however, that the ADP field is not checked by slaves in this case. Broadcast addressing is used, for instance, to initialize the same memory area in all slaves with the same value at the same time by means of a single command.

Specific areas in the local memory of the target node(s) are then addressed via the ADO field in the DLPDU. In particular, ADO represents an offset into the physical address space of the device and is useful to access a single block of data starting at a given memory location.

18.3.3.2 Logical Addressing

The EtherCAT data link communication services enable the individual addressing of several slaves by means of a single Ethernet frame carrying multiple DLPDUs. The main benefit is a significant improvement of the communication efficiency with respect to conventional solutions where every request (and the related reply) is encoded into a separate frame. The typical Ethernet overhead for frame transmission and protocol control information is quite large, and improvements are surely welcome. However, when dealing with simple devices characterized by very small process data (e.g., for 2 bit digital I/O modules), the overhead introduced by DLPDUs based on physical addressing might be still considered excessive.

This problem can be lessened by adopting the EtherCAT logical addressing mode. In logical addressing, the exchanged information is identified by using the 32 bits in the address field as a whole (ADR). In practice, a logical address space is defined for the EtherCAT segment, which is shared by all devices. When using the logical addressing, the resulting utilization can exceed 90% of the available bandwidth, even when data are exchanged with devices that produce/consume only 2 bits of user data.

Logical addressing requires that slaves include suitable *fieldbus memory management units* (FMMUs), which have to be implemented in hardware in the ESC. An FMMU is similar to memory management unit (MMU) of modern processors and takes care of translating the logical address to a physical address in the device local memory. FMMUs enable individual address mappings for each device.

In contrast to conventional MMUs, which typically map whole memory pages, FMMUs also support bit-wise mapping. This is the reason why, for instance, 2 bits of a specific input device can be mapped individually (and practically anywhere) in the logical address space. If an EtherCAT command is sent to read/write, the logical memory area assigned to the 2 bits (instead of addressing an EtherCAT device explicitly), the FMMU in the related device is able to transfer 2 bits of data from/to the memory and the right position in the payload of the DLPDU. The same happens to other slaves, when they use the logical address within the same DLPDU. Therefore, the combined adoption of FMMUs and logical addressing enables the exchange of data segments that span across several slave devices. As shown in Figure 18.4, a single command is able to address data arbitrarily distributed within several slaves. Clearly, this boosts the communication efficiency significantly.

The access type supported by each FMMU is configurable and can be read, write, or read/write. The configuration of FMMU entities in the slaves is carried out by the master during the data link start-up phase. In particular, the following items are configured for each FMMU to specify how the address translation has to be performed:

- (Bit-oriented) start address in the logical address space
- (Bit-oriented) start address in the device's physical memory
- Size of the translated memory area
- Direction of the mapping (input or output)

When a DLPDU with logical addressing is received, the slave looks for an address match via its FMMU. If any correspondence is found, it transfers data between the local memory position and the data field of the DLPDU.

18.3.4 Commands

Generally speaking, EtherCAT commands are obtained by combining an access type (read, write, read/write, or read/multiple write) and an addressing mode (auto-increment, configured, logical, or broadcast). However, not all possible combinations are actually allowed. The legal EtherCAT commands are listed in Table 18.2.

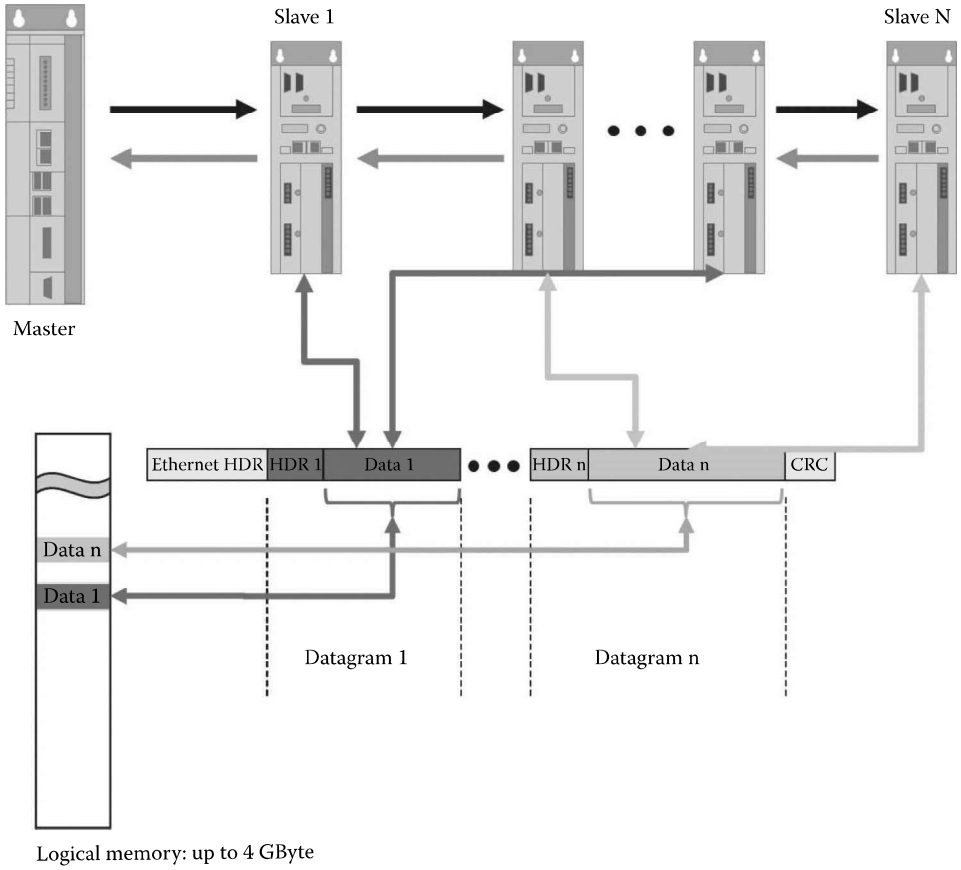

FIGURE 18.4 FMMU mapping example.

TABLE 18.2 EtherCAT Commands

Access Type	Command	CMD	Encoding
Read	Auto-increment physical read	APRD	1
	Configured-address physical read	FPRD	4
	Broadcast read	BRD	7
	Logical read	LRD	10
Write	Auto-increment physical write	APWR	2
	Configured-address physical write	FPWR	5
	Broadcast write	BWR	8
	Logical write	LWR	11
Read/Write	Auto-increment physical read/write	APRW	3
	Configured-address physical read/write	FPRW	6
	Broadcast read/write	BRW	9
	Logical read/write	LRW	12
Read/Multiple Write	Auto-increment physical read/multiple write	ARMW	13
	Configured-address physical read/multiple write	FRMW	14

Read commands (APRD, FPRD) are used to extract information at the specified (ADO) offset in the memory of the slave(s) addressed by ADP and to copy it in the data field of the DLPDU, which will eventually reach the master. Write commands (APWR, FPWR) carry out data exchanges in the opposite direction, while read/write commands (APRW, FPRW) are used for indivisible read and write operations. In this case, in fact, the contents of the DLPDU data field and the addressed memory location are swapped simultaneously.

Logical commands behave almost the same. It is worth remembering that multiple slaves can be involved in the execution of the same command when using logical addressing, depending on the FMMU configuration. As a consequence, the content of the data field in the DLPDU is either obtained by gathering information from the physical memory of several devices (LRD) or scattered over a number of different slaves (LWR). As mentioned before, the logical read/write command (LRW) carries out both operations at the same time.

Broadcast write commands (BWR) are straightforward: as expected, the same value overwrites the memory area addressed by ADO in all slaves. Broadcast read commands (BRD), instead, exploit Boolean operators; while processing an incoming DLPDU, every slave carries out a bitwise "OR" operation between the value found in its memory area addressed by ADO and the data field in the incoming datagram. Then, the result is inserted into the data field of the outgoing DLPDU.

The broadcast read/write command (BRW) combines read and write operations, so that the slave memory content is changed to reflect incoming data. However, according to data sheets [ET1100], this last service is typically not used.

Read/multiple write commands (ARMW and FRMW) complete the set of available services. Only singlecast physical addressing can be used in this case, namely, auto-increment or configured. Read/multiple write operations are quite peculiar and are exploited by EtherCAT in well-defined contexts (i.e., they are not intended for general-purpose data exchange services). In particular, the addressed slave carries out a read operation, whereas all other slaves execute a write operation triggered by the frame reception. If the addressed slave is the first one in the segment, such a mechanism permits a specific value to be read from its local memory and propagated to all the other slaves with a single command. This is profitably exploited, for instance, by the clock synchronization mechanism for distributing the reference time (further details will be discussed in the next sections).

18.3.5 SyncManager

The ESC memory is used for exchanging data between the EM and the application running on the slave. The master can access the memory through the network by using the data link layer services, whereas the local application makes use of the *process data interface* (PDI) provided by the ESC. As a consequence, problems may arise if concurrent accesses are carried out without any restriction. In particular, the consistency of data is not guaranteed by the basic data link communication services, unless a mechanism like semaphores is implemented in software for dealing with data exchanges in a coordinated way. Moreover, both the EM and the application running in the slave have to poll the memory explicitly, in order to determine when it is no longer used by the competing entity.

EtherCAT provides a mechanism for slave memory access control, which is based on *SyncManagers*, and was designed bearing in mind concurrency issues. SyncManagers are implemented in hardware in the ESC and enable consistent and secure data exchanges between the EM and the local application, together with the interrupt generation to notify both sides of changes.

SyncManagers are configured by the EM. The communication direction can be selected, as well as the communication mode. Each SyncManager uses a buffer in the local memory area for exchanging data and transparently controls all accesses to the buffer. The buffer must be accessed beginning with the start address; otherwise, the access is denied. Once access to the start location is granted, the whole buffer can be accessed, either as a whole or in a number of strokes. Accessing the last location also

concludes the whole operation. Buffer changes caused by the master are accepted by the SyncManager only if the frame FCS is correct. This also means that such buffer changes take effect immediately after the reception of the end of the frame.

SyncManagers support two communication modes:

1. *Buffered mode*: In this case, the interaction between the producer and the consumer of data is uncorrelated, and each entity can access the buffer at any time. The consumer is always provided with the newest data. In the case data are written into the buffer faster than they are read out, old data are simply discarded. The buffered mode is typically used for cyclic process data. This mechanism is also known as 3-buffer mode, because the SyncManager manages three buffers of identical size (denoted as 0, 1, and 2). One buffer is allocated to the producer (for writing), another buffer to the consumer (for reading), and a third buffer helps as intermediate storage. Reading or writing the last byte of the buffer results in an automatic buffer exchange. It is worth noting that both the EM and the local application must always refer to buffer 0 when accessing memory. It is up to the SyncManager redirecting accesses to the right buffer.
2. *Mailbox mode*: In this case, a handshake mechanism is implemented for data exchanges, which prevents buffer overwriting and ensures that no data will be lost. Just one buffer is allocated for each mailbox; moreover, reading and writing are enabled alternatively. The mechanism implemented by mailboxes is straightforward. At first the producer writes to the mailbox buffer. When done, the SyncManager locks it for writing and enables read access to the consumer. Only when the consumer has finished reading data out of the buffer, the producer is granted write access again. At the same time, the mailbox turns to the locked state for the consumer. The mailbox mode is typically used for application layer (AL) protocols, where the time taken to exchange information typically is not very relevant.

18.4 Distributed Clocks

An important mechanism included in the EtherCAT specification is the distributed clocks (DC) synchronization protocol, which enables all slave devices to share the same system time with high precision and accuracy. Synchronization errors are typically well below 1 μs [CEN12]; in this way, all devices can be synchronized, and consequently, distributed applications are synchronized as well. Possibly, the master can also be synchronized, even though this option requires additional capabilities.

18.4.1 Main Features

The DC mechanism provides a number of features that are very useful for distributed control applications, the most important being

- Synchronization of the slaves (and the master) clocks
- Generation of synchronous output signals (*SyncSignals*)
- Precise timestamping of input events (*LatchSignals*)
- Generation of synchronous interrupts
- Synchronous digital output updates
- Synchronous digital input sampling

DC is placed above the EtherCAT data link protocol, and its implementation is not mandatory. For this reason, both DC-enabled and non-DC-enabled devices can quietly coexist in the same network. It is worth noting that DC is not a general-purpose synchronization protocol, since it relies on specific features of EtherCAT, such as its ring topology, on-the-fly datagram processing, and hardware timestamping capabilities.

To better understand the DC mechanism, the following terms and definitions are needed:

- The *system time* is a 64 bit value representing the time (in nanoseconds) elapsed since 0:00 h of January 1, 2000.
- Typically, the *reference clock* is the slave device with DC capability that is connected closer to the master. Optionally, the reference clock can be adjusted to an extern *global* reference clock, for example, to an IEEE 1588 grandmaster clock. The reference clock provides the system time to all other devices in the EtherCAT segment.
- Each DC-enabled device has a *local clock* that, if not suitably controlled, runs independently of the reference clock.
- The *master clock* is used to initialize the reference clock. Typically, it is the clock of the EM, but it is also possible (and sometimes recommended) to bind it to a global clock reference (IEEE 1588, GPS, etc.), which is made available to the master either directly or through a specific EtherCAT slave.
- The *propagation delay* is the time spent by frames when passing through devices and cables.
- The *offset* is the difference between the local clock of a given slave device and the reference clock. It is due to several elements, such as the propagation delay from the device holding the reference clock to the considered slave, the initial difference of the local clocks resulting from the different instants at which devices were powered on, the skew between oscillator frequencies, and so on. The offset is compensated locally in each slave.
- Because the oscillator periods of local and reference clock sources are subject to small deviations (different quartzes are used), the resulting *drift* has to be compensated regularly.

18.4.2 DC Mechanism

The clock synchronization process consists of the following three main actions:

1. *Propagation delay measurement*: The master sends a synchronization DLPDU at certain time intervals, and each slave stores the time of its local clock; after collecting all timestamps, the master, which is aware of the network topology, computes the propagation delay for each segment.
2. *Offset compensation*: Because the local time of each device is a free-running counter, which typically does not have the same value as the reference clock, the master computes the offset between the reference and local clocks separately for each DC-enabled slave. Then, the offset is written to a specific register of the slave, in order to compensate differences individually. At the end of this step, all devices share the same absolute system time.
3. *Drift compensation*: After propagation delays have been measured and the offsets between clocks compensated, the drift of every local clock is corrected through a time control loop (TCL). This mechanism readjusts the local clock by regularly measuring its difference with the reference clock.

Details on these actions are provided in the following. For the sake of clarity, from now on, DLPDUs will be referred to simply as datagrams.

18.4.2.1 Propagation Delay Measurement

As mentioned before, each slave device introduces small frame processing/forwarding delays, caused by both internal and communication link mechanisms. For this reason, the propagation delay between the reference node and any slave has to be evaluated carefully. This procedure, which is the first action carried out by DC, consists of three main steps. First, the master sends a broadcast message (i.e., a datagram processed by all slaves). Second, each slave stores the value of its local clock when the first bit of that message is received. This operation is performed for each port of the device, on both the processing and forwarding paths (Figure 18.5). Finally, the master collects timestamps and computes the path delays, taking into account the topology of the network.

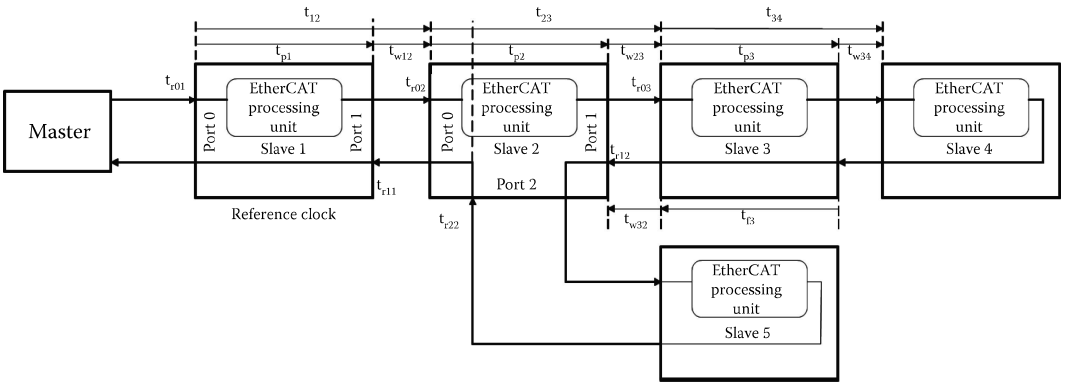

FIGURE 18.5 Time measurement in the DC mechanism.

A synchronization of slave devices is not necessary for the correct operation of the procedure, because only local clock values are used.

The computation of the propagation delay ($t_{PropagationDelay}$) for each slave is based on the differences between receive times at the device ports. Delays introduced by nodes without DC capabilities are taken into account as well. To better understand the propagation delay measurement, a sample EtherCAT network is shown in Figure 18.5. The node that is closer to the master (Slave 1) has been selected as the reference clock. Propagation delay computation makes use of the main elements summarized in Table 18.3.

The algorithm starts when the master sends a set of datagrams to reset the slave DC registers. After this initialization phase, the master transmits a pair of datagrams to each slave: the first message forces the ESC in the device to record timestamps related to the reception of the frame on its different ports, while the second is used to read timestamps back. For simple devices with only two ports, this means that timestamps are taken in both directions; that is, a first timestamp is acquired when the message is received on the processing path, while a second is recorded on the reception of the same frame on the forwarding path.

In sample network in Figure 18.5, Slave 2 records timestamp t_{r02} when the frame reaches its Port 0, t_{r12} when the frame is received back from Slave 3, and, finally, t_{r22} when the message is returned by Slave 5. Since timing information is taken using the same internal clock, differences between each pair of timestamps can be evaluated properly. Differences are then used to evaluate all delays (processing, forwarding, and wire contributions) and to compute the resulting propagation delay for the slave. It should be noted that the forwarding delay for the first slave (Slave 1 in Figure 18.5) cannot be computed in this way, as timestamps are recorded only for incoming messages. A possible solution is to assume that it is the same as the other t_{fx} or, alternatively, it can be measured once and for all (at least in theory) when the device is designed and manufactured. The compensation is then performed directly by the master, likely by using a predefined fixed value.

TABLE 18.3 Definition of DC Timing Parameters

t_{px}	Processing delay of slave x
t_{fx}	Forwarding delay of slave x
t_{xy}	Propagation delay from slave x to slave y
t_{wxy}	Wire propagation delay from slave x to slave y (symmetry assumption)
t_{rky}	Receive time on port k of slave y, recorded on the DC receive time register

18.4.2.2 Offset Compensation

The second main action carried out by DC is the offset compensation. Local clocks ($t_{LocalClock}$), if not synchronized, are free-running counters. At system start-up, they are likely to have different values from the reference clock so that an offset compensation is needed. The technique adopted in EtherCAT to evaluate offsets is based on the same timestamps collected by the master during propagation delay measurement. In fact, after the previous phase has finished, the master is also able to compute the offset of each local clock with respect to the reference time (t_{Offset}). This value is then written into a specific register (*system time offset*) of the slave and then used to adjust the local time. At this point, when the initialization phase is concluded, each DC slave can determine its own copy of the system time autonomously, by using the local time and offset values.

18.4.2.3 Drift Compensation

Finally, DC has to deal with oscillator drifts. The natural drift of every local clock (which depends on variations between the reference and local clock quartz) is corrected by means of a TCL algorithm implemented into each ESC. The master periodically distributes the system time ($t_{ReceivedSystemTime}$, as read from the reference clock) to all slaves in the network. The algorithm in each slave compares the received system time to its local copy. In evaluating the difference Δt, propagation delays have also to be taken into account:

$$\Delta t = (t_{LocalClock} + t_{Offset} - t_{PropagationDelay}) - t_{ReceivedSystemTime}$$

In automation systems, adjusting the clock with abrupt increments (or, even worse, decrements) is typically not tolerable. The clock discipline algorithm defined by DC, namely, TCL, is also able to cope with this requirement. If Δt is positive, the local time is running faster than the system time and has to slow down. On the contrary, when Δt is negative, the local time is too slow and needs to speed up. In this way, the monotonic property of time is assured for each DC-enabled slave. After the initialization of delays and offsets, the master tries to compensate the static clock deviations quickly, that is, by sending a high number of commands (e.g., fifteen thousands) in separate frames. In other words, first the control mechanism takes care of the static deviations to obtain synchronized DCs, then compensation frames are sent cyclically to correct dynamic clock drifts.

18.4.3 External Synchronization

The ability to synchronize to an external time source, so as to define clear timing hierarchies for the whole control system, is quite desirable in current industrial plants. This feature has been recently introduced in EtherCAT so that the time alignment of separate EtherCAT systems is now possible. The basic idea is having the DC time controlled directly by an external device. A possible solution makes use of special devices that provide a common DC reference time to different network segments. In practice, the EBUS is cross-connected to an Ethernet segment, and the user can decide what side (either EBUS or Ethernet) hosts the reference clock with the highest priority. The connecting device is then responsible for sending reference clock correction values to the EM with the lower priority.

Moreover, when the reference time in an EtherCAT system has to be adjusted to a higher-level clock, an external synchronization device can be used that supports synchronization protocols different from DC. For example, some devices are now available that can work with an external Precision Time Protocol (PTP) clock source. In principle, several other popular synchronization protocols and clock sources can be also used, for example, GPS, radio clocks, Network Time Protocol (NTP), and Simple Network Time Protocol (SNTP).

Another remarkable feature has been recently introduced in EtherCAT to cope with systems that demand increased availability; it consists of an optional redundancy mechanism that enables the on-the-fly replacement of devices without having to shut down the network and makes the system resilient to slave failures.

18.5 Application Layer

The AL of EtherCAT implements a state machine, which describes the behavior of a device by means of its states and events that trigger transitions between states. In particular, the state machine is responsible for coordinating master and slave applications during the start-up and operational phases. Depending on the current state, different functions are enabled in the EtherCAT slave. Different commands have also to be sent to the device in each state by the EM, in particular, during the boot sequence of the slave. Commands are acknowledged by the local application after the involved operations have been completed. Unsolicited changes of the local application state are also possible. Moreover, simpler devices, which do not include a microcontroller, can be configured to follow the state machine logic through an emulation mechanism. In this case, any state change has to be accepted and acknowledged.

The state machine is controlled and monitored using some registers included in the slave. The master controls the state transitions by writing to the AL control register. In turn, the slave updates information about its current state by writing in the AL status register, which is also used for error notification by means of suitable error codes written in the register itself.

As Figure 18.6 shows, an EtherCAT slave shall support four basic states and, possibly, one optional state:

- *Init*: EtherCAT slaves enter this state at power-on. In this situation, the master initializes the SyncManager channels for mailbox communications.
- *Preoperational*: Mailbox communications are enabled in the preoperational state, but process data communications are not. The EM initializes the SyncManager channels for process data, the FMMUs and the PDO mapping mechanism, if supported.
- *Safe operational*: In this state, mailbox and process data communications are enabled, but the slave outputs are kept in a safe state, while inputs are updated cyclically.
- *Operational*: In this state, slaves can transfer data between the network and their I/O logic. Mailbox and process data communications are completely enabled. The operational state is the normal working condition for slaves after completing the bootstrap sequence.
- *Bootstrap* (optional): The bootstrap state is mainly aimed at downloading the device firmware. In the bootstrap state, mailboxes are active but restricted to file access via EtherCAT services.

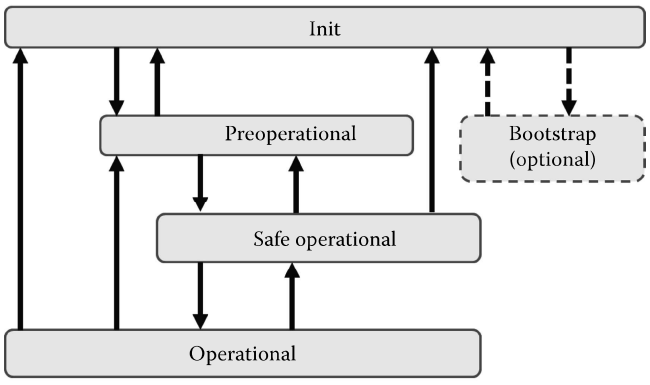

FIGURE 18.6 EtherCAT state machine.

18.5.1 Application Protocols

An important characteristic of EtherCAT is its ability to support multiprotocol higher-level communications using standardized mailboxes. This aspect is particularly appealing when options are offered for popular solutions such as the following:

- *CANopen over EtherCAT* (CoE): This option offers a way to access a CANopen object dictionary (OD) and to exchange CANopen messages according to event-driven mechanisms.
- *Ethernet over EtherCAT* (EoE): This option allows to tunneling standard Ethernet Frames in EtherCAT.
- *File access over EtherCAT* (FoE): This option enables the download/upload of firmware and other files.
- *Servo drive profile over EtherCAT* (SoE): This option is useful to grant access to the device profile of SERCOS.

Supporting popular communication protocols helps in improving compatibility and efficiency of data exchanges between new and old components in automation systems; to this purpose, EtherCAT makes use of well-known and established technologies. For instance, the *CoE* protocol enables the adoption of the complete CANopen profile family in EtherCAT networks. Besides this feature, the service data object (SDO) transport protocol allows the transmission of objects of any size and is equivalent to its CANopen counterpart so that it is possible to reuse existing protocol stacks. Data are organized in process data objects (PDOs), which are transferred using the efficient support of EtherCAT. Moreover, an enhanced mode is defined that overcomes the 8 byte limitation of CAN and enables the readability of the whole object list.

Another appealing feature an industrial Ethernet solution should provide is the support to standard IP-based communication protocols (i.e., TCP/IP and UDP/IP) and all higher-level protocols that rely on them, such as HTTP, FTP, SNMP, etc. To this purpose, the *EoE* feature exploits a mechanism where Ethernet datagrams are tunneled and reassembled in a device, before being relayed as complete Ethernet frames. This procedure has no impact on the achievable cycle time, because the size of fragments can be optimized according to the available bandwidth.

The *FoE* is an EtherCAT service that can be used to download a file from a client to a server or to upload it in the opposite direction. The protocol is similar to the trivial file transfer protocol (TFTP), and both sides are allowed to initiate a read or write request via the corresponding command. This service is typically used to update the device firmware.

Finally, the *Servo drive profile over EtherCAT* (SoE) service enables the use of the SERCOS device profile and is suitable for demanding applications that rely on popular drive technology.

18.6 EtherCAT Master Characteristics

Functionalities of the EM are usually implemented in software. EM coordinates activities of slaves through a control application, mainly by reading and writing data stored in their local memory. In addition, EM manages some network mechanisms (concerning, for instance, clock synchronization) and checks the availability of the system by monitoring the state of slaves (for both detection and active reaction to faults). The master is also responsible for the slave configuration during the network start-up, and it often offers additional features such as the automatic discovery of the network topology and/or support to make the network configuration and reconfiguration easier. Other useful features provided by some EMs are high-level functionalities, such as communications between EMs (carried out at the cell level) that allow interconnecting separate subsystems (at the field level), each one consisting of one EM and a number of connected slaves.

The most critical task carried out by EM is probably dealing with the control loop, which is entered after the configuration of slaves. This is because, in an EtherCAT network, the level of determinism achieved by the master directly affects the real-time performance of the whole control application.

In the following subsections, the concept of control loop is briefly introduced. Then, the main features of two categories of EMs, namely, commercial and open source, as well as their points of strength and weakness, are discussed.

18.6.1 Control Loop

The master enters the cyclic control loop at the end of the configuration phase, after fundamental operations such as the discovery of the network topology and the transmission of configuration data to slaves have been completed. The control loop is a cyclic task with its own predefined period (T_{CYCLE}), which, independently of its complexity, has unavoidably to read and write data from/to the slaves according to the (well-defined) control algorithm. Besides datagrams (i.e., DLPDUs) needed by the control application, additional messages are exchanged over the network under the EM control, to provide essential network services such as synchronization and fault detection.

A distinguishing feature of EtherCAT is its ability to enable very short cycle times T_{CYCLE} (that can be as short as 500 μs or even less), so this technology is also suitable for motion control and other high-precision applications.

Figure 18.7 shows a very basic structure for an EM control loop, where pseudo-code is used to highlight its main phases. The `receive_frame` function is where EtherCAT frames, received through the EM network interface, are made available to the application. This involves a specific sequence of actions: extracting datagrams from the frame, extracting process data from each datagram, and, finally, storing process data into the relevant variables into the process image (collectively represented by the "`data`" variable in the picture). The functional block `execute_control_application` is where the control algorithm is executed, so as to compute new values (`new_data`) for commands and state variables. Data to be sent to actuators are first encoded into suitable datagrams and then collected into a single frame, which may also include other datagrams added by EM for management purposes. At this point, the frame is sent over the EtherCAT network (function `send_frame`). The `wait` function in Figure 18.7 reminds that the execution of the control loop must occur exactly once on every T_{CYCLE}, so some waiting time could be necessary before starting a new iteration.

The timing diagram in Figure 18.8 depicts some iterations of the control loop. In the diagram, t_{CPU} represents the time spent by the CPU in the master to execute the code of the control loop (i.e., the `receive_frame`, `execute_control_application`, and `send_frame` blocks in Figure 18.7). The t_{NET} interval, instead, is the overall time spent by the network interface to send an EtherCAT frame over the network, including the time needed by the frame to traverse all the slaves on the processing path and come back on the forwarding path (propagation delay). To keep the diagram simple, the time needed to store the frame received from the network interface into the EM memory is also included in t_{NET}.

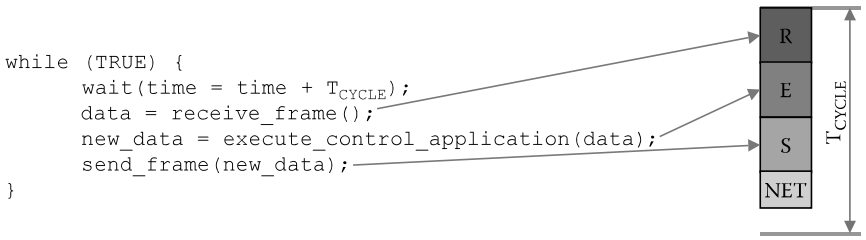

FIGURE 18.7 Pseudo-code of a typical EM control loop.

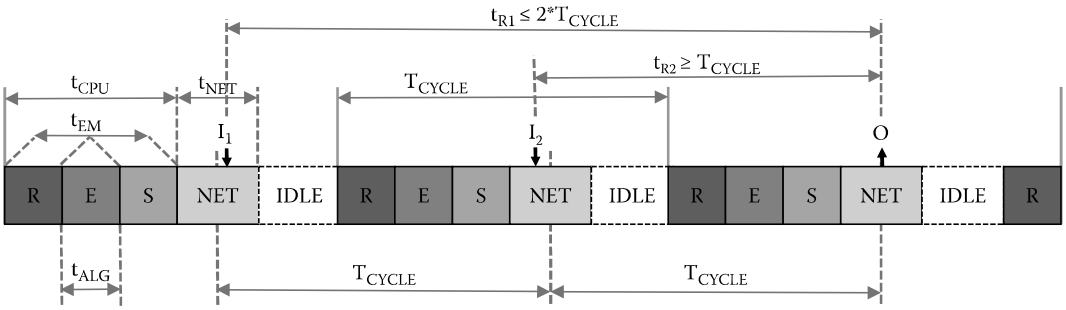

FIGURE 18.8 EtherCAT control loop timing diagram.

Slaves copy sampled input data from their local memory to EtherCAT datagrams in a time interval included in t_{NET}, and the same happens for output data, which have to be transferred from datagrams to slaves' registers.

The *reaction time* t_R is the time required by the control application to react to a variation of sampled input data, by writing commanded output data accordingly. Figure 18.8 shows that t_R is less than or equal to 2^*T_{CYCLE} (the time interval between I_1 and O in Figure 18.8 depicts the worst case). However, when the sampled input signal is made available by the slave in time for being included in the datagram containing the read command, t_R is just a bit larger than T_{CYCLE} (time interval between I_2 and O in Figure 18.8). Summing up, the reaction time t_R is bound by the inequality $T_{CYCLE} < t_R \leq 2^*T_{CYCLE}$.

It is worth remarking that the results hold if the input and output operations are performed by the same slave and the execution time of the control algorithm is constant. On the contrary, other factors have to be taken into account, such as the propagation delay on the processing path for the portion of EtherCAT network segment between the involved slaves, and/or the amount of indeterminism introduced by the control algorithm. For instance, when the cycle time is T_{CYCLE} = 500 μs, the maximum reaction time is t_R = 1 ms, but this is true if and only if the inequality $t_{CPU} + t_{NET} \leq T_{CYCLE}$ holds.

When DC is used to synchronize the clock of EtherCAT slaves, they can execute a given action at a predefined instant in time. In this case, the maximum reaction time actually increases, because all slaves must receive the related command (included in one or more the datagrams) before the programmed actuation time.

Because of the intrinsic determinism of EtherCAT, the time t_{NET} does not vary across subsequent iterations of the control loop. In practice, t_{NET}, which depends on the frame size and other parameters, can be computed analytically [CEN10b].

Unfortunately, the t_{CPU} time interval is not deterministic. Its duration basically depends on two factors, that is, the time spent by the CPU to execute the control algorithm (t_{ALG}) and delays introduced by EM and its operating system (t_{EM}). t_{ALG} is mostly affected by the code implementing the control algorithm. In most cases, an upper bound on the execution time can be evaluated, either analytically or experimentally. On the contrary, no reasonable bound can be found a priori for t_{EM}. For this reason, the inequality $t_{CPU} + t_{NET} \leq T_{CYCLE}$ might not hold in some circumstances.

t_{EM} takes into account the time needed by the protocol stack to send and receive EtherCAT frames and also includes operating system latencies caused by interacting tasks, context switching, and scheduling. To keep t_{EM} bounded, EM implementations usually rely on hard real-time operating systems and protocol stacks. Software drivers for interfacing to the Ethernet controller are often rewritten, in order to ensure that such a requirement is met.

Besides the functionalities a given EM is able to provide, the quality of an EM largely depends on its ability to transfer EtherCAT frames with high determinism. The EM performance analysis presented in [CER11], for instance, shows that jitters on the order of 10 μs can be obtained for a nominal cycle time equal to 1 ms, provided that hard real-time operating systems and protocol stacks are used. EMs based

on non real-time operating systems and protocol stacks are usually not suitable for EtherCAT, since in such conditions t_{EM} is not bounded, and delays larger than 1 ms can be commonly experienced. In these cases, the actual determinism featured by EtherCAT can be even worse than non real-time industrial Ethernet solutions, such as EtherNet/IP.

18.6.2 Commercial versus Open-Source Implementations

As already pointed out, the most important requirement for EM is determinism. Even though the determinism achieved by an EtherCAT network is largely enabled by the slave hardware (i.e., by the precision of the DC synchronization mechanism and the ability of slaves to process datagrams on-the-fly), the master plays an important role as well, because all actions summarized in Figure 18.7 have to be completed in a single iteration of the control loop in order to ensure a predetermined and constant cycle time.

In addition, a wide range of other features differentiate EM products today available off-the-shelf. Even though hardware prototype implementations that are based on field programmable gate array (FPGA) boards are described in the scientific literature, EMs are usually implemented as software components. In [MAR12], for example, a mixed hardware-and-software-based architecture was presented that enables the transmission of EtherCAT frames with jitters shorter than 10 ns. The whole system behavior was also sped up significantly, by performing several time-consuming EM operations directly in hardware. Although promising, FPGA-based EM architectures are just prototypes, and, as far as we know, they are not used in real-world EtherCAT applications at the present day. For this reason, the remaining part of this section focuses on software-based EMs.

A number of software EMs are currently available, and designers and systems integrators can choose between commercial and open-source licenses. Commercial products are usually more user-friendly, bug-free, full-featured, and well documented, but open-source products can also be selected, in some situations, for a number of complementary reasons.

On the one hand, Beckhoff TwinCAT [TWIN3] is the most popular commercial EM solution, produced by the company that developed EtherCAT. Other interesting commercial products are NI EM and KPA EM, produced by National Instrument and Koenig-pa GmbH, respectively.

On the other hand, the most popular open-source EM is certainly EtherLab, developed by Ingenieurgemeinschaft IgH and freely downloadable [ELAB]. Another popular open-source solution is the Simple Open EtherCAT Master (SOEM). Unfortunately, SOEM cannot be used in applications demanding high determinism and low T_{CYCLE}, since its implementation is not real-time.

Regardless of the specific implementation, the two EM categories exhibit very different characteristics. In Table 18.4, the most significant differences between commercial and open-source EMs are highlighted, pointing out strengths (+) and weaknesses (–). The absence of weak points for commercial EMs in the table does not mean absence of defects, since open-source EM strengths have to be considered as weaknesses for commercial EMs and vice versa. A more detailed comparison between the two categories can also be found in [SCA12].

TABLE 18.4 Commercial versus Open-Source EMs

Commercial	Open Source
+ Easy-to-use	+ Cost (less expensive)
+ Programming languages compliant with IEC 61131-3	– (Usually) nonstandard programming languages
+ Many libraries	+ Customizable code and (usually) better performance
+ Documentation	– Limited hardware support
+ Integrability	– Some EtherCAT features not implemented

Commercial EMs are usually easy to use, if compared with their open-source counterparts. The network configuration, programming interface, and all configurable features are accessible via a graphical user interface (GUI), which allows reducing the time needed to learn the programming and configuration environment also for not experienced programmers and, consequently, shortens the *time-to-market* of applications.

The biggest advantage of open-source EMs is that they are generally free of charge. For this reason, if a large number of control applications have to be developed, costs for the longer learning time can be compensated by adopting the open licensing scheme.

At the beginning of 2013, the new version of the IEC standard 61131-3 [IEC13] was released, which defines five programming languages to be used with PLCs. This edition replaces the previous version published in 2003. All most important commercial tools for EtherCAT programming (and for the vast majority of other industrial Ethernet protocols as well) are now IEC 61131-3 compliant.

Two textual and three graphical languages are included in the IEC 61131-3 standard:

- *Instruction list* (IL) is a low-level programming language, with a level of abstraction similar to assembler.
- *Structured text* (ST) resembles high-level programming languages like C or C++.
- *Ladder diagram* (LD) allows to represent the program as a sequence of digital circuits, including contacts, coils, and specific blocks like counters, timers, etc.
- *Function block diagram* (FBD) is based on block diagrams, where blocks are connected through oriented arcs. It describes how inputs, through the blocks, are transformed into outputs.
- *Sequential function chart* (SFC) represents the control program as a sequence of states (*steps*) with associated *actions* activated by given conditions (*transitions*).

ST and LD are currently the most used languages. ST has a higher level of abstraction and is suitable for large projects, while LD is well known to technicians working in the industrial automation field. A typical situation found in many projects are programs developed with mixed languages, that is, LD for low-level functions and ST for more complex functions (it also acts as the *glue* for putting different subroutines together).

Unfortunately, open-source EMs seldom support IEC 61131-3-compliant programming languages. This drawback limits the code reusability and portability on other EMs and PLCs and requires, in general, longer training times. However, it may also lead to a better optimization of the produced code.

In most open-source solutions, the programming language used to develop the actual control application is the same as the one adopted for implementing the EM. This means that the EM code, which is freely released in the source form, can be possibly modified to meet specific design requirements. This often results in a more efficient EM (where, for instance, some unused features are disabled at compile time), which sometimes is able to show an improved determinism (the EM can be calibrated with respect to the hardware of the PLC, e.g., by modifying the driver of the network interface card or the operating system scheduler). By contrast, some particular features of EtherCAT may sometimes be unavailable in open-source EMs, and enhanced determinism can be achieved only with the aid of specific hardware components (i.e., selected network interface cards).

The documentation, as well as the availability of a large number of libraries (to ease the implementation of the algorithms used in industrial control systems), is certainly one of the most significant advantages of commercial EMs.

Finally, the ability to integrate a number of subnetworks at the cell level is important in large industrial plants, in order to coordinate the production processes. In this context, the availability of a well-engineered application programming interface (API) for enabling communications between PLCs can make the difference, especially when different industrial Ethernet technologies have to be integrated in the same plant. Open-source EMs often offer communication services that are limited to the field level segment of the network (i.e., in the case of EtherCAT, the segment between EM and its slaves), while no API is provided for communications at the cell level (i.e., between different EMs).

In conclusion, the choice of the master should not be underestimated, since it can strongly constrain the project, and in particular its maintainability and characteristics, in terms of both functionalities and performance of the control application. Since in most situations, applications cannot be easily migrated from one EM to another, requirements about performance, desired features, and costs must be carefully considered during the initial design phase.

18.6.3 EtherCAT Application Example

This section focuses on a simple example of a typical EtherCAT application, so as to offer the reader a flavor of EM configuration and programming. The same application has been implemented with both a commercial (TwinCAT) and an open-source (EtherLab) EM solution. In the case of TwinCAT, two programming languages have been employed, namely, the LD and ST.

The sample application considered here is a frequency divider. Its goal is switching the level of a digital output (DO) each time three rising edges are sampled from a digital input (DI)—for instance, by pressing three times a push button connected to DI. The EtherCAT network involved in this example consists of a single EM (a standard PC, which runs either TwinCAT or EtherLab), one DI slave (Beckhoff EL1202 [EL1202]), and one DO slave (Beckhoff EL2202 [EL2202]).

The EM, independently from its implementation (TwinCAT/EtherLab), must be properly configured before entering the application loop that executes the control algorithm. The configuration takes place by means of a graphical interface in TwinCAT or relies on specific textual commands in the case EtherLab is used. Each EtherCAT slave hosts an OD, containing a number of entries that are addressed through a 16 bit index and an 8 bit subindex. OD entries allow accessing the slave registers that need to be read/written by the program.

In the configuration phase, EM sets up process data in each slave to specify what information is transferred over the network during every iteration of the control loop, in agreement with the application requirements. These PDOs are embedded in EtherCAT datagrams and transferred by means of an EtherCAT frame. In practice, they are the process data that are used by the control application to perform its task.

The final step of the configuration phase is linking process data to the variables in the process image used by the application in the control loop. A detailed analysis of the configuration steps is not reported here: a thorough description of TwinCAT can be found in the public software documentation, while additional details about the configuration phase for both the EMs taken into account are included in [SCA12].

18.6.3.1 Ladder Diagram

A program written in TwinCAT LD consists of two parts: declaration of variables and code. Declaration of variables is performed by means of an IEC 61131-3-compliant textual programming language. For instance, the variables used in the frequency divider example are shown in Figure 18.9.

```
1 : VAR
2 :       (* Process variables *)
3 :       input AT %IX0.0 : BOOL;
4 :       output AT %QX0.0 : BOOL := FALSE;
5 :       (* Internal variables *)
6 :       switch : BOOL := FALSE;
7 :       enable : BOOL := TRUE;
8 :       (* Function block: CTU counter *)
9 :       counter : CTU;
10: END_VAR
```

FIGURE 18.9 Declaration of variables for the application example.

Process variables `input` and `output` (lines 3 and 4 in the picture) are two Boolean values representing the input DI and output DO; they are linked to the process data in the EtherCAT frame as defined in the configuration phase of the EM. Symbols `%I` and `%Q` are used for input and output process variables, respectively. `X` is one of the possible sizes for process variables (`X`: bit, `B`: byte, `W`: word, and `D`: double word), while the numbers in the format `B.b` specify the position of the variable in the process data memory. `B` and `b` stand for the byte and bit number, respectively (e.g., address `X12.3` selects bit number 3 of byte number 12). Such an addressing method is used to make control application variables largely independent of the underlying communication technology used to interconnect PLCs and remote devices (either a fieldbus or industrial Ethernet).

In Figure 18.9, `switch` and `enable` are two internal variables, while `counter` is a function block, that is, a complex *object* with its own attributes and static variables, which retain their values between successive executions. Variables can be initialized at the same time when they are declared: for instance, `output` is initialized to 0 (`FALSE`), which means that the initial value of the DO slave is reset.

Figure 18.10 shows the LD implementation of the application, as well as the most common symbols used in this language, namely, *contacts* and *coils*. In the LD model, current flows through the circuit from left to right. The basic components are contacts and coils, and each component is associated to a variable. When the variable associated to a contact is TRUE, the current flows through the contact; otherwise, when it is FALSE, the current flow is blocked. When current flows through a coil, the variable associated to that coil is set to TRUE; otherwise its value is FALSE. The symbol "/" over a contact or a coil inverts the behavior of that component; for example, in the case of a contact, the current flow is blocked if the variable associated with the contact is TRUE. An LD program is executed from top to bottom.

Other more complex components are also defined in the LD programming language, which are known as function blocks, the most important being *timers* and *counters*. A count up (CTU) counter is shown in the first row of Figure 18.10. Each time a rising edge is sampled by the CU input of the CTU counter, the CV value of the counter (an unsigned integer variable) is increased by one. When the value of CV is greater than or equal to PV, the output Q is set to TRUE (so that the current flows out from Q). When the current flows through the RESET input, the CTU counter stops counting, and the CV value is set to FALSE. Moreover, the output Q stops emitting the current flow.

The first row of the program in Figure 18.10 means that each time the value of the process variable `input` is TRUE, the current reaches the input CU of the CTU counter, and thus, the CV value is incremented by 1. When the value CV equals 3 (the value of PV), the current exits from output Q and crosses the two coils associated with the `switch` and `enable` variables (both variables are set to TRUE).

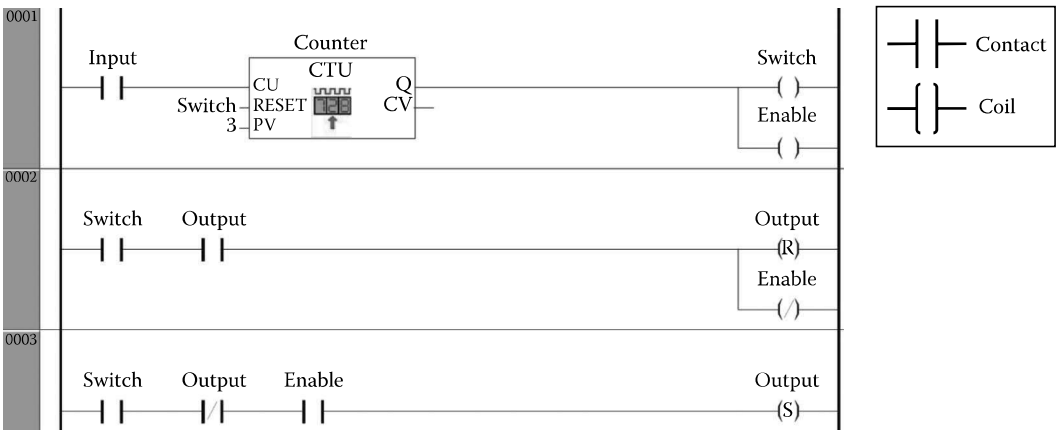

FIGURE 18.10 The example application in ladder diagram.

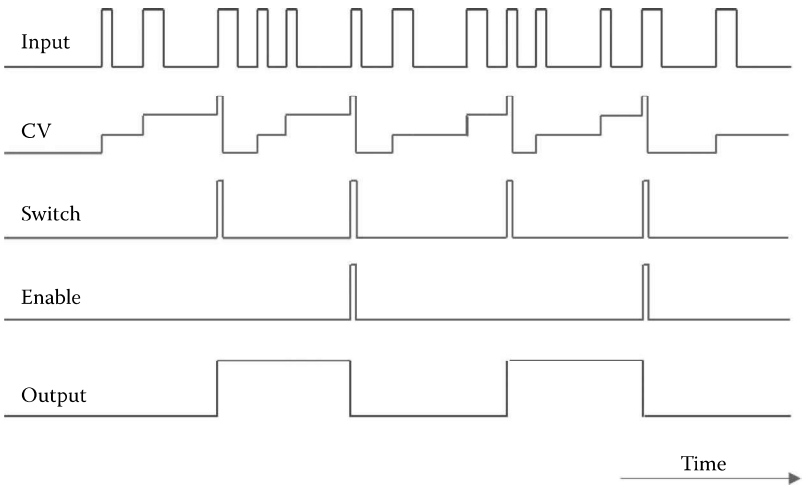

FIGURE 18.11 Timing diagram of the LD program.

The two contacts in row 2 build an AND operator. When both variables `switch` and `output` are TRUE, current flows through both coils associated to the `output` and `enable` variables. The idea behind this scheme is that, when the counter reaches the value of PV (i.e., when `switch` is TRUE), the program switches the `output` variable, and, consequently, also the DO output is switched.

The code in row 2 reverts the output from TRUE to FALSE. The coil with the "R" symbol has a specific behavior: when traversed by a current flow, it resets to FALSE the associated variable. The "S" coil in row 3 is needed to set the associated variable to TRUE. These behaviors are different when compared with classical coils, because actions are performed only in the presence of current flows. In summary, the second row of the LD code sets the variable `output` to FALSE and the `enable` variable to FALSE if the `switch` and `output` variables are both TRUE. The `enable` variable is set to FALSE in order to disable the third row of code in the case a change of output has just been performed in the same execution cycle.

The third row of the code is similar to the second and manages the transition of the `output` variable from FALSE to TRUE. Indeed, if a transition of the output is required (`switch` is TRUE), the `output` variable is FALSE, and the second row has not changed the output yet (`enable` is TRUE), the `output` variable is set to TRUE. In the following cycle, `switch` remains TRUE, the counter is reset, and, since no current exits from the contact Q, the `switch` and `enable` variables are set to FALSE.

A possible example of timing diagram for the variables involved in the example is shown in Figure 18.11.

18.6.3.2 Structured Text

The implementation of the same example in the TwinCAT ST programming language is shown in Figure 18.12.

From a lexical and syntactical point of view, ST resembles the typical high-level programming languages used in computer engineering. The part of the program reserved to variable declaration (lines 2–9) has exactly the same syntax as LD. Code lines 10–14 perform the detection of a rising edge in the `input` variable. If the current sampled value is TRUE and the value sampled in the previous cycle was FALSE (line 11), a rising edge is detected, and the variable `counter` is incremented by 1 (line 12). The `counter` variable has the same meaning as the CT value of the CTU counter in the LD program.

When three rising edges are detected (line 16), the output is switched (lines 17–21), and the `counter` variable is reset to 0. From this point on, the program is put into the initial state, and therefore, it can start counting rising edges again.

This example is useful to understand that, although LD and ST are actually different, their complexity is comparable. Usually, ST is more suitable for large projects since it is more expressive and complete

```
1 : PROGRAM MAIN
2 : VAR
3 :       (* Process variables *)
4 :       input AT %IX0.0 : BOOL;
5 :       output AT %QX0.0 : BOOL := FALSE;
6 :       (* Internal variables *)
7 :       old_input : BOOL := FALSE;
8 :       counter : INT := 0;
9 : END_VAR

10: (* Detection of a rising edge *)
11: IF input = TRUE AND old_input = FALSE THEN
12:       counter := counter + 1;
13: END_IF
14: old_input := input;

15: (* Output switching *)
16: IF counter = 3 THEN
17:       IF output = TRUE THEN
18:               output := FALSE;
19:       ELSE
20:               output := TRUE;
21:       END_IF
22:       counter := 0;
23: END_IF
```

FIGURE 18.12 Structured text of the sample application program.

than LD. Moreover, ST can be easily mastered by programmers that are used to classical high-level programming languages, such as C, C++, Java, Ruby, Python, etc.

18.6.3.3 EtherLab C

The same frequency divider has also been implemented by using EtherLab on a Linux-based PC. The EtherLab EM follows a completely different approach with respect to TwinCAT. Indeed, the programming language is C, and the configuration phase is performed directly in the control program by means of C functions and without any graphical interface aid. The configuration phase, not described for reasons of space and because not particularly interesting, consists in a sequence of commands to associate the control application with the EM and activate it and in a mapping between the required entries in the slaves' OD and a memory area in the PC (accessible by the control application).

At the end of the configuration phase, a memory pointer (`domain_pd`) allows the control application to access the memory areas associated to the input of the DI and to the output of the DO terminals. The offsets of these process data with respect to the memory address `domain_pd` are `DI_offset` and `DO_offset` for the input and the output, respectively. The EtherCAT control task must start after the configuration phase. The timing of the control task can be dealt with in EtherLab by using either POSIX system calls, in the case of not demanding applications, or specific services offered by the underlying hard real-time operating systems for increased accuracy.

The control task corresponds to the function `cyclic_task()`, whose body is reported in Figure 18.13. It is periodically executed on every T_{CYCLE}, and it is scheduled by using a timing function similar to `wait` in the pseudo-code in Figure 18.7. Inside the control task, two other functions, `ecrt_master_receive(master)` and `ecrt_domain_process(domain)`, at lines 7 and 8, are used to receive and extract process data from the EtherCAT frame. Data extracted from the datagrams are then stored in the process memory area pointed by the global variable `domain_pd`.

```
1 : int counter = 0;
2 : int old_input = 0;
3 : int input;
4 : int output = 0;

5 : void cyclic_task() {
6 :     /* Receive new process data */
7 :     ecrt_master_receive(master);
8 :     ecrt_domain_process(domain);

9 :     input = EC_READ_BIT(domain_pd+DI_offset, 0);
10:     /* Detection of a rising edge */
11:     if (input==1 && old_input==0) {
12:         counter = counter + 1;
13:     }
14:     old_input = input;

15:     /* Output switching */
16:     if (counter==3) {
17:       if (output==1) {
18:           output = 0;
19:       }else{
20:           output = 1;
21:       }
22:       counter = 0;
23:     }
24:     EC_WRITE_BIT(domain_pd+DO_offset, 0, output);

25:     /* Send new computed process data */
26:     ecrt_domain_queue(domain);
27:     ecrt_master_send(master);
28: }
```

FIGURE 18.13 C code of the sample application program.

In line 9, a bit is read from the process memory at position `domain_pd+DI_offset` and stored in the `input` variable. The new value of the variable `input` is then used by the control algorithm (lines 11 to 23) to check if the `output` variable has to be switched again. As for ST in TwinCAT, the `counter` variable keeps track of the number of `input` rising edges. When `counter` reaches the value 3 (line 16), the `output` variable, and thus the value of the DO output, is inverted and the new value is then written in the process memory (line 24). Process data are finally embedded in datagrams (line 26) and sent to the slave devices in the EtherCAT frame (line 27).

References

ETG12 EtherCAT Technology Group, EtherCAT—The Ethernet fieldbus, 2012. [Online]. Available: http://www.ethercat.org/.

ET1100 Beckhoff Automation GmbH, Hardware data sheet ET1100 slave controller, 2013. [Online]. Available: http://www.beckhoff.com/.

IE158 IEC 61158-3/4/5/6-12, Industrial communication networks—Fieldbus specifications—Part 3-12: Data-link layer service definition—Part 4-12: Data-link layer protocol specification—Part 5-12: Application layer service definition—Part 6-12: Application layer protocol specification—Type 12 elements (EtherCAT).

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

8023 IEEE 802.3-2008, IEEE standard for information technology-Specific requirements—Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.

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

CEN10 Cena, G.; Bertolotti, I.C.; Scanzio, S.; Valenzano, A.; Zunino, C., On the accuracy of the distributed clock mechanism in EtherCAT, *Proceedings of IEEE International Workshop on Factory Communication Systems (WFCS 2010)*, pp. 43–52, May 18–21, 2010.

CEN12 Cena, G.; Bertolotti, I.C.; Scanzio, S.; Valenzano, A.; Zunino, C., Evaluation of EtherCAT distributed clock performance, *IEEE Transactions on Industrial Informatics*, 8(1), 20–29, February 2012.

CEN10b Cena, G.; Scanzio, S.; Valenzano, A.; Zunino, C., Performance analysis of switched EtherCAT Networks, *Proceedings of IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2010)*, pp. 1–4, September 13–16, 2010.

CER11 Cereia, M.; Cibrario Bertolotti, I.; Scanzio, S., Performance of a real-time EtherCAT master under linux, *IEEE Transactions on Industrial Informatics*, 7(4), 679–687, November 2011.

MAR12 Maruyama, T.; Yamada, T., Hardware acceleration architecture for EtherCAT master controller, *Proceedings of Ninth IEEE International Workshop on Factory Communication Systems (WFCS 2012)*, pp. 223–232, May 21–24, 2012.

TWIN3 Beckhoff Automation GmbH, TwinCAT 3 | eXtended Automation (XA), 2013 [Online]. Available: http://www.beckhoff.com/.

ELAB Ingenieurgemeinschaft IgH, EtherLab—EtherCAT master, 2013. [Online]. Available: http://www.etherlab.org/.

SCA12 Scanzio, S., SoftPLC-based control: A comparison between commercial and open-source EtherCAT technologies, *Handbook of Research on Industrial Informatics and Manufacturing Intelligence: Innovations and Solutions*, IGI Global, pp. 440–463, 2012.

IEC13 International Electrotechnical Commission, Programmable controllers—Part 3: Programming Languages, IEC 61131-3:2013 (Ed. 3.0), 2013.

EL1202 Beckhoff Automation GmbH, EL1202 | 2-channel digital input terminal 24 V DC, TON/TOFF 1 µs, 2013 [Online]. Available: http://www.beckhoff.com/.

EL2202 Beckhoff Automation GmbH, EL2202 | 2-channel digital output terminal 24 V DC, TON/TOFF 1 µs, push-pull outputs, tri-state, 2013 [Online]. Available: http://www.beckhoff.com/.